Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming Languages

— THESIS DRAFT: DO NOT DISTRIBUTE —

Satoshi Matsuoka
Department of Information Science, The University of Tokyo [1]

April 8, 1993

[1]Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone 03-3812-2111 (overseas +81-3-812-2111) ex. 4108. E-mail: matsu@is.s.u-tokyo.ac.jp

Contents

1 Introduction **3**
1.1 Motivation and Overview of this Thesis 3
1.2 Contributions and Outline of this Thesis 5
1.3 Historical Retrospect of this Research 8

I Inheritance in Concurrent Object-Oriented Languages—Anomalies and their Solutions **13**

2 Introduction—Inheritance and its Anomalies in Object-Oriented Concurrent Programming Languages **14**
2.1 Motivation 14
2.2 Inheritance Anomaly in OOCP 16

3 Inheritance Anomalies in the Previous Proposals **18**
3.1 Simple Examples of Inheritance Anomaly — Caused by 'Body's, Explicit Message Reception within Methods, Path Expressions, Direct Key Specifications 18
3.1.1 Bodies 18
3.1.2 Explicit Message Receptions 20
3.1.3 Path Expressions 21
3.1.4 Direct Key Specifications 21
3.2 Problems with *Behavior Abstractions* 21
3.3 Problems with First-Classing of Accept Sets — *Enabled Sets* 22
3.4 Problems with Method Guards 28

4 Analysis of Inheritance Anomaly **31**
4.1 Partitioning of States 31
4.2 History-only Sensitiveness of States 32
4.3 Modification of Acceptable State 33
4.4 Examples of Analysis of Anomaly Occurrence—Synchronizing Actions . . 34
4.5 The Formal Analysis of Inheritance Anomaly—State Partitioning 35
4.5.1 Overview of Cook-Palsberg Inheritance Semantics and its Extensions 35
4.5.2 Concurrent Objects with Synchronization Constraints 39
4.5.3 Schemes for Synchronization Specification: DPSS and DKSS . . . 41

4.5.4 Previous Proposals and DKSS 42
4.5.5 Proof of the Anomaly in Inheritance with DKSS 44
4.5.6 The Main Cause of Anomaly in Inheritance 47

5 Recent Proposals for Solutions to the Inheritance Anomaly 49
5.1 Shibayama's Proposal 49
5.2 Caromel's Proposal 50
5.3 Frølund's Proposal 50
5.4 Meseguer's Proposal 52
5.5 Ishikawa's Proposal 52
5.6 Summary of Previous Proposals and Their Limitations 53

6 Our Proposed Solutions to Inheritance Anomaly 55
6.1 Early Proposal 1 — First-Class Guards with a Reflective Architecture . . 55
6.2 Early Proposal 2 — Eliminating Synchronization Code Syntactically . . . 55
6.3 Our New Proposal 57
6.3.1 Overview of Execution Model of Concurrent Objects 57
6.3.2 Method Sets 58
6.3.3 Synchronizers 61
6.3.4 Transition Specifications 62
6.4 Inheriting Synchronization Code in our New Proposal 65

7 Examples of Avoiding Inheritance Anomaly 67
7.1 State Partitioning Anomaly—Method `get2`: 69
7.2 History-Only Sensitiveness—Method `gget` in `class gb-buf`: 70
7.3 State Modification Anomaly—Methods `Lock / Write-Lock / Unlock`: . 71
7.4 Combining Synchronization Schemes 73

8 Efficient Implementation Architecture for our Proposal 75
8.1 Overview of ABCLonAP1000 75
8.2 Overview of Efficient Asynchronous Message Passing 76
8.3 Implementation of Synchronizers 78
8.4 Example of Synchronizer Execution 81
8.5 Implementation of Transition Specification 82
8.6 Example of Transition Execution 88
8.7 Preliminary Benchmarks 92

9 Discussions and Future Work 93
9.1 Summary of Part I 93
9.1.1 Why Method Sets are not Full First-class Entities? 93
9.1.2 Inheriting Synchronization Code with Encapsulation 94
9.1.3 Which Synchronization Scheme do we Use? 95
9.2 Conclusion and Directions for the Future 95

II Reflection in Concurrent Object-Oriented Languages—Architecture, Language, Implementation, and Semantics 97

10 Introduction—Reflection 98
10.1 Problems with Previous Computational Models of Concurrent Objects . . 98
10.1.1 Requirements in Parallel Computation with OOCP languages . . 98
10.1.2 Why Current Models of Concurrent Objects are not Sufficient . . 99
10.2 Introduction to Reflection and Reflective Architectures 100
10.3 Reflection in Object-Oriented Concurrent Systems — The Benefits 101
10.4 Contribution: ABCL/R2—A Language Based on Hybrid Group Architecture102

11 Object-Oriented Concurrent Reflective (OOCR) Architectures — Categorization of Previous Work 104
11.1 Previous Works in Classification of Reflective and Metalevel Architectures 104
11.2 Classification of OOCR Architectures 106
11.2.1 The Individual-Based Architecture — ABCL/R 107
11.2.2 The Group-wide Reflective Architecture — ACT/R 109
11.2.3 Limitations of Both Architectures 111

12 ABCL/R2: A Hybrid Group Architecture Language 113
12.1 ABCL/R2: a Language Based on Hybrid Group Architecture 115
12.1.1 Object Groups in ABCL/R2 115
12.1.2 Object and Group Creation in ABCL/R2 121
12.1.3 Non-reifying Objects 123
12.1.4 The Two Kinds of Reflective Towers 123

13 Efficient Implementation of ABCL/R2 127
13.1 Problems in Achieving Practical Efficiency in OOCR Languages 127
13.2 Implementation Issues of OOCR Languages 128
13.3 Implementation Scheme of ABCL/R2 129
13.3.1 The Low-Level Kernel 130
13.3.2 Partial Compilation of Scripts 131
13.3.3 Self-reification of Default System Objects 133
13.3.4 Light-weight Objects 134
13.3.5 Creation of Individual Tower via Dynamic Progression of Reflectivity135
13.3.6 Compilation of Non-Reifying Objects 138
13.3.7 Lazy Creation of Meta-Groups 139
13.4 Performance Measurements of the ABCL/R2 Compiler 140

14 Examples of Reflective Programming in ABCL/R2 144
14.1 Example 1—Controlling Explosion of Parallelism via Computational Resource Management 144
14.2 Example 2—Time Warp Scheduling in ABCL/R2 146

15 Formal Semantics of the Hybrid Group Architecture—ABCL/Rμ 153

16 Discussions and Future Work **167**
16.1 Summary of Part II 167
16.2 Directions for the Future 168

A Previous Work that Addressed Conflicts of Inheritance vs. Concurrency/Distribution in OOCP languages **183**
A.1 Conflict between Inheritance and Distribution 183
A.2 Conflict between Delegation and Synchronization 184

B Appendix: Definition of the Time Warp Group **186**

List of Figures

3.1 A Bounded Buffer Object 20
3.2 Definition of Bounded Buffer Class with Body (The code related to accessing the local array storage for insertion and removal is omitted for brevity.) 20
3.3 `B-buf` and `x-buf` with Behavior Abstractions 23
3.4 Inheritance Anomaly with Behavior Abstractions 24
3.5 `X-buf2` with Enabled-Sets 26
3.6 General Analysis of Enabled Set 27
3.7 `B-buf` and `x-buf2` with Method Guards 28
3.8 Inheritance Anomaly with Guards — the `gget` method 29
3.9 Inheritance Anomaly with Guards — the `Lock` class 30

4.1 Conceptual Illustration of the State Partitioning Anomaly 32
4.2 Conceptual Illustration of the State Modification Anomaly 34
4.3 Inheritance Anomaly in Synchronizing Actions 36
4.4 Method System Domains 37
4.5 Semantic Domains 37
4.6 Auxiliary Functions 38
4.7 Functions in the Denotational System of Inheritance 38
4.8 Object Behavior Functions 39
4.9 Relationships Between the Synchronization Constraints and the Synchronization Specifications 43
4.10 Auxiliary Function *slook* (in Cook's notation) 44
4.11 Proof of Anomaly in Inheritance 47

6.1 Object and its Metaobject in X0/R 56
6.2 Definition of `Lock` Class with X0/R 56

8.1 Structure of Concurrent Objects 77
8.2 Synchronizer Implementation for Bounded Buffers 79
8.3 Example of Synchronization Control via Synchronizers (1) 83
8.4 Example of Synchronization Control via Synchronizers (2) 84
8.5 Implementation of Transition Specification 86
8.6 Example of Synchronization Control via Transitions (1) 90
8.7 Example of Synchronization Control via Transitions (2) 91

11.1 The Individual-Based Architecture in ABCL/R 108

11.2 The Group-Wide Architecture in ACT/R 110

12.1 The Individual and Group Reflective Towers in ABCL/R2 114
12.2 Group Definition in ABCL/R2 116
12.3 Reflective Architecture of ABCL/R2 118
12.4 Primary Metaobject Generator in ABCL/R2 119
12.5 Primary Evaluator in ABCL/R2 120
12.6 Dynamic Group Creation and Bootstrapping 124

13.1 The Low-Level Kernel 130
13.2 Abridged Partial Compilation Rule of ABCL/R2 132
13.3 Compilation of `[x <= (* y 2) @ z]` 133
13.4 Self-Reification Mechanism 134
13.5 Snapshot of an Individual Tower 136
13.6 Comparison of the Naive Approach and the Forwarding Approach 137
13.7 Creation of a New Group 139
13.8 Object Definitions for Fibonacci Numbers 142
13.9 Performance Measurements for Fibonacci Numbers on ABCL/R2 and ABCL/1 143

14.1 Controlling Explosion of Parallelism with ABCL/R2 145
14.2 Definition of Meta-Evaluator-M 149
14.3 Controlled and Uncontrolled Parallelism for Quicksorting 150
14.4 Meta-level Encapsulation of the Time Warp Algorithm (1) 151
14.5 Implementation of the Time Warp Scheduling in the Meta-meta Level . . 151
14.6 The Result of Meta-meta-level Scheduling of the Time Warp Algorithm applied to the Car Wash Problem 152

15.1 Metaobject Definition in ABCL/Rμ 163
15.2 Primary Evaluator in ABCL/Rμ 164
15.3 Primary Evaluator in ABCL/Rμ (cont'd) 165
15.4 Group Manager in ABCL/Rμ 166

16.1 The Road Map of Our Previous and Current Works, Plus Future Directions 170

B.1 The Usage of Time Warp Group 187
B.2 Group Kernel Objects of the Time Warp Group (Metaobject Generator) 188
B.3 Group Kernel Objects of the Time Warp Group (Metaobject Generator)(cont'd) 189
B.4 Group Kernel Objects of the Time Warp Group (TimeWarp Evaluator) . 190
B.5 Group Kernel Objects of the Time Warp Group (Metaobject of Evaluator) 191

List of Tables

8.1 Benchmark of Intra-node Message to Dormant `b-buf` Object on AP1000. 92

Abstract

We investigate and propose two major language features to realize and enhance extensibility and re-usability in concurrent object-oriented (OO) languages. Part 1 addresses inheritance in concurrent-OO languages: re-use of synchronization code in concurrent OO-languages has been considered difficult due to *inheritance anomaly*, which we analyze and categorize extensively, and minimize with our new proposal. We propose language primitives with the following novel characteristics: (1) it allows multiple *synchronization schemes*—the language schemes for *programming* synchronization—to coexist and be integrated, (2) re-use of synchronization code is done similarly to sequential OO-languages for user familiarity, (3) it offers high degree of encapsulation—even synchronization schemes could be encapsulated in superclasses in many cases, and (4) it can be efficiently implemented on conventional MPPs. We demonstrate the effectiveness of our proposal with solutions to the analyzed inheritance anomaly cases. We also show that the proposed language features can be implemented efficiently in a fine-grain setting.

Part 2 addresses extending and re-using concurrency features of concurrent-OO languages with *reflection*. Reflection has been advocated as being beneficial in concurrent/distributed computing, where the complexity of the system is much greater compared to sequential computing. Unfortunately, attempts to formulate and encapsulate metalevel features provided in practical reflective systems have led to various difficulties, due to the lack the proper abstractions for group-wide object coordination. We present a new concurrent-OO reflective language with a *hybrid group architecture*, ABCL/R2, whose key features are the notion of heterogeneous object groups and coordinated management of *group shared resources*. We describe and give examples of how such management can be effectively modeled and adaptively modified/controlled with the reflective features of ABCL/R2. We identify that this architecture embodies *two* kinds of reflective towers, *individual* and *group*. We also present efficient implementation schemes for ABCL/R2, which include: efficient lazy creation of metaobjects/meta-groups, partial compilation of scripts (methods), dynamic progression, self-reification, and light-weight objects, all appropriately integrated so that the user-level semantics remain consistent with the meta-circular definition so that the full power of reflection is retained, while achieving practical efficiency.

Acknowledgements

It is customary for a thesis to start-off by acknowledging various individuals. It is, however, ever so difficult for me to write acknowledgements for this particular thesis, for the number of people that had had some contribution or another, either directly or indirectly during the course of this research, is just too many. More important to say is that, the research conducted is not merely worth its results alone; what are perhaps more important are the revelation of the value of friendship, discovery of enjoyment of fraternity, and the trust and faith in human nature and scientific truth I have almost lost but managed to regain.

Much of how the research had been conducted is covered in the historical retrospect section in the introductory chapter. Indeed, the three-year period during which this research had been conducted was a era of both personal leap as well as turmoil. It is not proper for me to go into any personal matters here; I will merely state here that, I am in deep acknowledgement of all the people for encouraging me to proceed with the golden road of computer science research, which I had the deepest love for but almost lost sight of. Now, upon completing this thesis, I have to say they were right in keeping 'on the road', and I cannot thank them enough.

I regret that I cannot name all the wonderful friends I have come to know through this research, but nevertheless I thank you all—appreciations to your are truly in my heart.

Nevertheless, there are individuals I cannot miss naming explicitly: Firstly, my deepest appreciation goes to Professor Akinori Yonezawa, the head of our laboratory as well as my colleague for the past four years. Ever since our dramatic meeting in OOPSLA'88, the first international conference I have ever attended, he has been ever so supportive in research as well as my professional career as a researcher, instances of just too many to mention here. Indeed, one could not wish for a better boss, and I consider myself most fortunate for being able to serve such a fine researcher as well as an individual.

Equally, I give my sincerest appreciation to Professor ('uncle') Mario Tokoro of Keio University; he too, has played a major role in the establishment of my career, and also many hours of discussions literally all over the world—in U.S., Canada, Mexico, Europe, and of course in Japan—significantly contributed towards improving this research.

Next, I must thank the individuals who have collaborated with me during the course of this research. The foremost thanks goes to my good friend and colleague Takuo Watanabe, whose joint work on ABCL/R2 and other areas on reflection and concurrent objects have been and still is one of the most fruitful research experience of my career (not to mention his fine taste of music). Also, million thanks to Hidehiko Masuhara, the wonderfully talented Lisp hacker, for making our 'pet' language come to true life. Also,

appreciation goes to Kenny Wakita, Makoto Takeyama, Kenjiro Taura, and Shigeru Chiba for their important contributions.

For Part 1 of the thesis, appreciations go to Gul Agha, Carl Hewitt, Jens Palsberg, José Meseguer, Lodewijk Bergmans, Etsuya Shibayama, Kohei Honda, , Shuichi Sakai, Ken'ichi Asai, and Masahiro Yasugi. Special thanks to Peter Wegner for motivating me to continue with the research during the difficult times when progress was slow.

For Part 2 of the thesis, I thank all the great friends at Xerox PARC—Gregor Kiczales (thanks also for all the Football and Volleyball games), John Lamping, Brian C. Smith (for his landmark thesis as well as delightful moments of discussions in his villa), Jim des Rivères, Ramana and Carlin Rao, and Luis Rodriguez. I also thank all the friends in Japan who work in the same area, and/or have contributed to this work—Yutaka Ishikawa, Yasuhiko Yokote, Hideyuki Nakashima, amongst countless others. I also appreciate the friends with whom the reflection workshop became true-to-life: Brian and Audrey Foote, Mamdouh Ibrahim, Jean-Pierre Briot, and Pierre Cointe. I also thank all the attendees of all the workshops on reflection I had been involved in organizing—discussions with you were all truly enjoyable as well as benefiting, and I am confident that as this technology become more prevalent in computing, we shall all be known as pioneers in the field.

I also cannot forget to thank all the former and the current students of our laboratory for their daily research discussions and assistance, and the same goes for other members of the department as well, especially Mrs. Shintani, the lovely librarian who has been so tolerant of my borrowed books being overdue, and helping me numerous times in finding the most obscure references. Also, Professor Kei Hiraki constantly motivated us to strive for utmost, practical efficiency with our implementations. Furthermore, I must say I enjoyed many hours of intense discussions with members of the Tokoro laboratory, including Ichiro Satoh.

Special appreciation goes to Professor Satoru Kawai, my former thesis advisor during my graduate student days, with whom I had started the research on concurrent objects, which got me launched on my current career. Also, I thank my parents and my dear sister Naomi for their support during my student days.

And finally, I dearly thank Miss Sumiko Ootsuka, our lovely secretary, for enduring and almost perfectly executing my day-to-day selfish requests without complaints, and supporting me in my work through the good as well as hard times.

Chapter 1

Introduction

1.1 Motivation and Overview of this Thesis

Recently, several research projects have focused on real-life implementation of Concurrent Object-Oriented (OO) languages on various platforms, such as a network of workstations, shared-memory UMA (Uniform-Memory Architectures) such as Sequent Symmetry and BBN Butterfly, and various distributed memory MPPs (Massively-Parallel Processors) such as Thinking Machine's CM-5, Intel Paragon, and Fujitsu AP1000. Concurrent-OO languages provide maximum computational and modeling power through (1) concurrency of objects, plus (2) familiar OO-software engineering concepts such as *encapsulation*, *code re-use*, and *application frameworks*[120], and can serve as a powerful platform for parallel applications[1]. Much work on establishing useful concurrent-OO language models have been quite fruitful: they include Actors[1], POOL[4, 5], Hybrid[85, 86], ABCL[127], and Maude[77].

Despite such background, plus promising results in impressive pioneering research systems such as POOL/DOOM[9], we do acknowledge that concurrent-OO languages have not yet seen widespread use in practice. We believe that the three major impediments are as follows:

'Myth' that 'Real'(Parallel) Message Passing is Slow: There is still a belief amongst the OO-community that message passing in a literal sense—parallel over processor interconnections—is a few orders of magnitudes slower compared to sequential procedure invocation. Since most programmers use sequential-OO languages that implement methods with plain procedures, the term 'message passing' has become synonymous to 'procedure-calling' in OO-programming. In parallel programming, performance is the key issue, and such unwarranted 'myth' has had negative effects.

Inheritance Anomaly—Difficulty in Re-using Synchronization Code: Several works have pointed out the conflicts between inheritance and concurrency in OO-languages[4, 56, 91, 109, 18], where attempts to inherit and re-use the code of

[1] To quote Robin Milner at ECOOP'91 Object-Based Concurrency Workshop panel, "I can't understand why objects are not concurrent in the first place (in OO-languages such as Smalltalk)."

concurrent objects results in extensive breakage of encapsulation. In this work, we have coined such a phenomenon as *inheritance anomaly* (See Part 1). One resulting critical drawback is that it becomes very difficult to construct a clean application framework—the most effective OO-software engineering discipline[120]—with concurrent-OO languages.

Difficulty in Controlling Concurrency and Distribution: Most models of Concurrent Objects adopt a PRAM-like model where objects exist in some 'pool' of uniformly accessible space. However, real-life concurrent computation in concurrent and distributed architectures involve many intricate factors of concurrency and distribution that do not manifest in these models, such as the routing mechanism of message delivery, memory management (garbage collection, etc.), object scheduling (computational resource allocation), object distribution policy management (for hot-spot resolution, etc.), and other cooperative resource management. That is to say, although these factors are usually hidden beneath the abstractions of the concurrent computational model, in practice they need to become manifest to achieve performance and even sometimes correctness. Although rather ad-hoc control schemes such as pragmas have been used in practice, this hinders re-use of programs because such pragmas usually assume some fixed abstraction of the underlying architecture.

Through our research, we have so far been successful in demonstrating that the slow-performance 'myth' can be overcome—here are some of our recent results: true remote asynchronous message passings in our proposed software/hardware architecture ABCL/onEM-4[121] achieves near sequential-OO 'message passing' performance (approximately 3μsec@25Mhz clock) with a combination of elaborate software/hardware. For MPPs based on conventional RISC processors, we have developed several novel implementation schemes with the language system ABCL/onAP1000[107], running on Fujitsu's AP1000 MPP (consisting of 512 SPARC chips@25Mhz clock interconnected by a 25Mhz torus network), where asynchronous message send to a local object takes 2.3μsec (about 20 SPARC instructions) and remote asynchronous message send takes 9μsec, which could be further improved with a better network interface.

This thesis tackles the remaining two problems that hinder the application of Concurrent OO-languages in practice: Part 1 of the thesis addresses the problem of providing efficient concurrent-OO language design that minimizes inheritance anomaly, promote code re-use, and aid the construction of parallel application frameworks. Although some number of proposals have been made in this regard, early attempts([56, 109]) were shown to be restrictive in [69]. The problem has sparked interest of several concurrent-OO researchers who have come out with various 'solutions' [98, 25, 63, 53, 78], especially after circulations of analysis papers such as [91, 69] and the early draft of [72]. However, we feel that they were not completely satisfactory with regards to practical applications. We propose new language constructs for concurrent-OO languages that solves the anomalies as well as allow high-performance implementations that could be integrated into ABCL/onAP1000.

To solve the second problem, the Part 2 of this thesis explores the use of *reflection*, or *reflective architectures* in Concurrent-OO language design. *Reflection* is the process

of reasoning about and acting upon the system itself[100, 64, 129]. A reflective system 'opens up' the system by providing an appropriate abstraction on the internal (often implementational) detail of itself to the user program. Contrary to the misconception that 'reflection' is some difficult-to-understand, not-too-useful philosophical jargon, it is a practical scheme that offers a new perspective in constructing a malleable, large-scale system such as programming languages[58], operating systems[124], and window systems[92]. In particular, we overcome both the design and implementation problems of previous reflective architectures, and propose a new reflective architecture for concurrent objects, called *hybrid group architecture*, and a language based on the architecture, ABCL/R2, and its efficient implementation scheme, which can be applied to other concurrent reflective languages as well.

1.2 Contributions and Outline of this Thesis

As stated above, the primary contribution of this thesis is to propose new solutions to the abovementioned two problems. We first outline the salient contributions of the thesis, then outline the technical contributions in more detail, along with the overviews of contents of each chapter.

- The primary contributions of Part 1 are:

  1. We conduct extensive survey and analysis of the inheritance anomaly not previously available. In particular, we point out the shortcomings of previous proposals that have claimed to 'solve' the inheritance anomaly problem.
  2. Based on the analysis, we propose language primitives to minimize the anomaly, which (1) allows multiple *synchronization schemes*—the language schemes for *programming* synchronization—to coexist and be integrated, (2) allow re-use of synchronization code similarly to sequential OO-languages for user familiarity, and (3) offers high degree of encapsulation—even synchronization schemes could be encapsulated in superclasses in many cases.
  3. We have also devised speed- and space-efficient incorporation into the software architecture ABCL/onAP1000, which supports fine-grain concurrent object execution. Preliminary benchmarks are shown to support our claim of efficiency.

- The contributions of Part 2 are:

  1. We investigate previous object-oriented concurrent reflective architectures, and establish categorizations into *Individual-Based Architecture (IBA)*, and *Group-wide Architecture (GWA)*. We then point out the shortcomings of the past architectural proposals with respect to practical application in parallel computing, such as scheduling (CPU resource), object allocation (memory resource), etc.
  2. Based on the investigation, we propose a new concurrent-OO reflective architecture called *hybrid group architecture*, and a language ABCL/R2, whose

key features are the notion of heterogeneous object groups and coordinated management of *group shared resources*, and the hybrid existence of *two* kinds of reflective towers, *individual* and *group*. We demonstrate how one can achieve *metalevel encapsulation* of concurrency features easily and lucidly using ABCL/R2.

3. We also have devised efficient implementation schemes for ABCL/R2, which are applicable to other reflective languages as well. They include: efficient lazy creation of metaobjects/meta-groups, partial compilation of scripts (methods), dynamic progression, self-reification, and light-weight objects. Benchmarks show comparable performance to the (non-reflective) base language ABCL/1; this is two orders of magnitude improvement over naive adaptation of the metacircular definition of ABCL/R2 on ABCL/1.

The detailed descriptions of the contributions of Part 1 of this thesis, along with the outline of its contents, are as follows:

1. We conduct extensive survey and analysis of the inheritance anomaly not previously available. In Chapter 2, we give an overview of inheritance anomaly phenomenon. We will then present non-trivial examples where the (rather simplistic) previous proposals for solutions are limited in their applicability in Chapter 3. Next we will analyze and categorize the cause inheritance anomaly more generally and formally in Chapter 4. We then examine some latest proposals by others for reusing synchronization code by controlling the inheritance anomaly problem, and point out their shortcomings in Chapter 5.

2. Based on the analysis, we then make a practical proposal that avoids the anomaly and thus allow flexible inheritance in concurrent-OO languages in Chapter 6. It extends and extensively refines the ideas in the past proposals to (1) separate and localize the synchronization schemes from the main bodies of methods, allowing fine-grained inheritance/overriding, and to (2) allow dynamic operations on the methods themselves, in order to control which messages are acceptable by an object. Furthermore, it has the following novel and favorable characteristics:

   - Our proposal allows multiple *synchronization schemes*—the basic language features such as guards for *programming* the synchronization of objects—to coexist and be integrated, so that the best scheme can be chosen to program given synchronization constraints.
   - The manner we re-use the synchronization code is syntactically similar to superclass method references in sequential OO-languages (e.g.,`super`). Thus, users with experience in OO-programming can readily adapt to our proposal. Inheritance rules are made to depend on each synchronization scheme, however, because the most 'natural' way of inheritance differs among the schemes.
   - We offer a high degree of encapsulation and re-use for synchronization code. Furthermore, even synchronization schemes could be encapsulated in superclasses in many cases by proper exporting of class information by the user.

Examples of how inheritance anomaly is resolved using our proposal, along with other programming examples, are presented in Chapter 7.

3. Expressiveness is not our sole concern—we have also devised speed- and space-efficient incorporation into the software architecture of the aforementioned ABCL/onAP1000, which we present in Chapter 8. In particular, all space/time-consuming data structure construction for object synchronization can be done at compile-time. Preliminary benchmarks support our claim of efficiency, allowing message passing overhead to be confined within factor of approximately 1.5 to 2 compared to optimized asynchronous message passing of ABCL/on AP1000.

The detailed descriptions of the contributions of Part 2 of this thesis, along with the outline of its contents, are as follows:

1. Although researchers have classified metalevel/reflective architectures from different viewpoints, so far to our knowledge no work has existed that generally discusses reflection in the context of concurrent/distributed object-oriented languages. Nor, has there been work that classifies architectures dependent on characteristics that are *particular to object-oriented concurrency*, i.e., not becoming manifest in sequential object-oriented languages. We investigate various aspects and object-oriented concurrent reflective architectures in Chapter 11, and establish the categorization of the previous architectures largely into *Individual-Based Architecture (IBA)*, where each object in the system has its own metaobject(s) which govern(s) its computation, and *Group-wide Architecture (GWA)*, where the collective behavior of a *group* of objects is represented as the coordinated actions of a group of meta-level objects, which comprise the *meta-group*. We then point out the shorcomings of the past architectural proposals with respect to the lack of the necessary abstractions to describe what could be described in terms of *coordinated resource management* in parallel computing, such as scheduling (CPU resource), object allocation (memory resource), etc.

2. In Chapter 12, we propose a new reflective architecture called *hybrid group architecture* that resolves the problem of IBA and GWA, and present a new object-oriented concurrent language with ABCL/R2, which incorporates heterogeneous object groups with group-wide object coordination and *group shared resources*. It also has other new features such as *non-reifying objects* for efficiency. By the design and implementation of ABCL/R2, we contribute feedback to the conceptual side of OOCR architectures and object groups by (1) showing that (heterogeneous) object groups are not ad-hoc concepts but can be defined uniformly and lucidly, and (2) identifying that hybrid group architectures embody *two* kinds of reflective towers, instead of one: the *individual tower* which mainly determine the structure of each object, and the *group tower* which mainly determine computation.

3. Next, in Chapter 13, we describe our novel schemes for actually implementing this (rather complex) reflective architecture, which are also applicable to other object-oriented (concurrent) reflective languages in general. In particular, for programs

that do not involve reflective computation, ABCL/R2 is shown to be just as fast as the original ABCL/1, and even for cases where reflective computation is involved, it is within a factor of 6–7. This is a considerable improvement over the previous ABCL/R, which is a factor of 1000 slower compared to ABCL/1 for all computations.

4. Then, in Chapter 14 we describe how these reflective features of ABCL/R2 allow coordinated resource management to be effectively modeled and efficiently controlled in the metalevel, achieving *metalevel encapsulation*—as an example, we study the scheduling problem of the Time Warp algorithm[55] used in parallel discrete event simulations.

5. Finally, we briefly explore the semantic foundations of our architecture, by the definition of Micro-ABCL/R2 in Chapter 15. In particular, we propose the technique of *structured mail address* which can describe the base-meta relationship more cleanly compared to other proposals.

1.3 Historical Retrospect of this Research

Before we go on to the main contents of the paper, which presents the research results pretty much 'as is', I shall compliment it by looking back at the historical retrospect of how the research had been conducted in the past years. Readers who are more interested in the technical contents should proceed to Part 1.

Ever since encountering the notion of objects back in 1985 during my undergraduate days, I was fascinated ever since on their concurrent behavior. Indeed, even those days, concurrent objects have been under extensive research, possibly dating back to Hewitt's Actors, or maybe one could unwind the history back to Nygaard's SIMULA, which already facilitated coroutines in 1967. By 1986 a number of early concurrent-OO languages had been proposed; it was also the year when ACM OOPSLA was first held. It was clear that concurrent objects were the 'wave of the future', given the parallel and distributed nature of future computing environment. I myself tried to extend the flexibility of inter-object communication by integrating Linda-style Tuple Space into Smalltalk for my Master's Thesis work, which was presented in OOPSLA'88.

However, the more I study both the theoretical and practical aspects of concurrent objects, I was faced with the gap between "what is good in theory" and "what is being used in practice". Despite the successes of sequential object-oriented (OO) languages such as Smalltalk and C++, concurrent OO languages were not being used at all in practical environments. In fact, even my personal day-to-day working environment did not have any practical incarnation of any concurrent OO-languages on real parallel machines. This situation persisted even after I was hired as a junior faculty during the midst of my doctoral studies by the Department of Information Science, the University of Tokyo, to join the research group of my present boss and colleague Professor Akinori Yonezawa, when he too moved to the current department in the fall of 1989.

The big question was "why?": Although a number of papers talked about the wonderful properties of concurrent objects, there were numerous proposals for language primi-

tives, etc., and there was even some sort of vague consensus within the OO community that concurrent objects are in fact important wave of the future, almost nobody was using it in practice. Of course, I could immediately come up with numerous answers—for example, parallel machines with fast network interconnect were not available then, and as a result, there were no good compilers or easy-to-use programming environments in the spirit of Smalltalk-80.

Still, that seemed to be not the only answer. What the researchers started to realize in the late '80s was that, inheritance and concurrency did not seem to meld well together. In fact, most concurrent OO languages were not really 'object-oriented' in the true sense of the word, because most did not provide any means of inheritance, including ABCL/1, the primary language of our research group. It goes without saying that currently, inheritance is the principle language feature for constructing application frameworks to facilitate maximum organized and well-disciplined re-use of code. It is not known whether the lack of inheritance in the early concurrent OO-languages was merely a careless failure to recognize its importance, or held back its incorporation due to vague realization that something does not go right had they incorporated inheritance.

In any case, in '89, I had encountered several work in resolving this conflict, so that inheritance would 'work right' in concurrent OO-languages. Initially, upon reading the paper, the proposals seemed reasonable; then, it struck me one day in late 1989 that the proposals were not working in some situations. I, in collaboration with Ken Wakita (now with Tokyo Institute of Technology) had then set out to create a canonical example of this phenomenon, and came up with the `get2` example, and coining of the phenomenon with the term *inheritance anomaly*. Next stage was the formalization and the attempt to resolve this anomaly, which was obviously avoidable with guarded methods; however, upon discussion with Makoto Takeyama in the Summer of 1990, then a graduate student in our lab, it became obvious that history-only sensitive behavior caused yet another kind of anomaly. Takeyama came up with the rudimentary form of the `gget` example in this paper.

It was clear then that categorization and analysis of the previous language proposals were necessary in order to highlight the essence of inheritance anomaly. Thus began a long series of work to survey and analyzing all known proposals of concurrent OO-languages with inheritance. To my expectations, they all suffered from one form of anomaly or other. The first half of Part 1 of this thesis presents the details of the categorization and analysis.

While this research was being pursued, it also became obvious that facilitating inheritance does not provide all the solutions inherent in re-using concurrency of objects. For example, consider the case where one wants to 'add' some scheduling to a existing application written with some concurrent-OO language. The ideal situation is to clearly encapsulate the scheduling code so that it little affects the existing code. However, re-use of existing code with simple inheritance alone would involve their extensive re-programming so that it would conform to the protocol of the application framework, usually requiring extensive re-configuration of classes, and intermixing of the scheduling code into the existing user-level application code.

Clearly, some facility that would provide such encapsulation orthogonal to inheritance was required. Then, in late 1989, Takuo Watanabe, who was then a Ph.D. Student for

Prof. Yonezawa at the Tokyo Institute of Technology, moved to our department in the merger of the research group. Takuo had been working on *reflection* with concurrent objects since 1988, and had already proposed the language ABCL/R which was presented at OOPSLA'88. At the time, he was working on language ACT/R, which tried to capture the group-wide behavior of a group of concurrent objects. Upon studying his work, and attending the Birds-of-a-Feather session on object-oriented reflection in OOPSLA'89, I immediately realized the potential power of the framework with respect to encapsulating concurrent behavior in the meta-level; thus started the fruitful collaboration with Takuo which is still continuing today. At the same time, a news arrived on the possibility of staging an in-conference workshop on object-oriented reflection at OOPSLA'90, and both Takuo and I were offered to be the far-east coordinator. Other organizers included Gregor Kiczales of Xerox PARC, famous for his pioneering work on practical reflective-OO systems with CLOS MetaObject Protocol (MOP).

During the spring-fall period, we prepared diligently for the workshop, soliciting as well as reviewing surprising number of papers. At the same time, Takuo was revising his work on ACT/R, which I helped out a little, and at the same time I undertook designing a toy reflective language called X0/R, which aimed at solving the inheritance anomaly problem, to which Takuo gave numerous valuable comments. After successful staging of the OOPSLA'90 workshop, we started reviewing the design of ABCL/R, X0/R, and ACT/R; there, the deficiencies of each architecture started to become obvious. Another problem was that none of the languages were implemented practically; The prototype of ABCL/R was extremely slow, and in fact it seemed impossible to implement ACT/R in any practical way, because the increase in the concurrency in the meta-level was drastic. X0/R was more practical, but its design was taylored for solving the inheritance anomaly, and did not provide sufficient metalevel abstractions for robust metalevel programming. We were clearly in need of a language with not only the combined features of ABCL/R and ACT/R, but also facilitated practical implementability.

We then started designing the new language, ABCL/R2. I first designed the overall architecture, and studied whether the computational process in the meta-levels and above would be consistent. After initial debugging, we next started writing the metacircular definition of ABCL/R2. After a few design iterations, and incorporation of non-reifying object I developed for X0/R, the language was complete. We wrote several examples, most notable being the hierarchical metalevel encapsulation of TimeWarp, which was adopted and modified from Takuo's earlier example. The resulting work was made into a paper, which was later accepted and presented at the ECOOP'91 conference next summer.

After writing the paper, the immediate need for implementation was apparent—after all, ABCL/R2 only had a metacircular definition as its only 'implementation', which obviously could not be run in practice. We first investigated extending Takuo's ABCL/R implementation which was done on top of ABCL/1, but it was too apparent that the execution efficiency would be hopeless; I wished for a reflective language that runs in a comparable range as ABCL/1. We thus decided to implement a subset based on ABCL/1, throwing away numerous features from the language. The biggest sacrifice was that the resulting language was not really reflective, for it did not run the metacircular definition. Moreover, much of the features we would have programmed using meta-level objects had been hardwired. The resulting prototype ran on TOP-1 Common Lisp, which is

a parallel dialect of Common Lisp. Although the implementation was sufficiently fast, we could not experiment much with reflective programming, and moreover, it was very unstable for several reasons. It was helpful, however, in establishing some of the early ideas for optimization.

Based on the experience of the first prototype, we started designing the efficient compiler and runtime of the full-set implementation of ABCL/R2 from the Summer of 1991. Joined by a young and gifted Lisp hacker, Hidehiko Masuhara, the project proceeded surprisingly smoothly. The first version of the implementation of ABCL/R2 described in this thesis became operational in early 1992. Due to various optimizations, the current ABCL/R2 matches the execution efficiency of publically distributed version of ABCL/1 for non-reflective computation, and even if reflection is involved, the overhead is typically within the factor of 10 for that part of the program. To our amazement, the examples we had written for the ECOOP'91 paper ran almost flawlessly, with the expected benefits of meta-level computation. The detailed description on the work on ABCL/R2 is given in Part 2 of this thesis.

In concurrence to the work on ABCL/R2, I had been involved in staging of several workshops on Reflection and Metalevel Architectures: the second workshop in OOPSLA'91; another small workshop at ECOOP'92; and served as a program committee member and also performed various local arrangement tasks for "IMSA'92 International Workshop on Reflection and Metalevel Architectures", the first major international workshop exclusively on metalevel architecture with emphasis on OO and practical computing, co-chaired by Prof. Yonezawa and Brian C. Smith of Xerox Palo PARC, who is one of the pioneers in the field. The intense workshop was quite a success according to most participants, which brought together many of the latest work in the field, as well as raised new research issues.

Getting back to the work on inheritance, in the late 1990 our research group had decided that real-life implementation on MPPs was a necessity for concurrent-OO languages, and formulated a strong implementation group (this was also motivated by the realization that direct multiprocessor porting of the Lisp-based ABCL implementation was obviously doomed to fail, based on the first ABCL/R2 subset prototype experience.). At the same time, Prof. Yonezawa initiated the collaboration with the computer architecture group of Electro-Technical Laboratory in Tsukuba, Japan on the possibility of implementing fine-grain concurrent OO-languages on their 'macro-dataflow architecture' MPP called EM-4. Although there were some initial difficulties in the learning process of the intricacies of the seemingly drastically different architecture, we eventually came up with a software implementation architecture, ABCL/onEM-4, which improved the remote message passing speed by orders of magnitude (approximately 7μseconds at 12.5Mhz clock).

We had also started collaboration with the Parallel Computing Center of Fujitsu Ltd., which allowed access to their AP1000 MPP, a 512-node SPARC-based architecture with a fast multi-feature interconnect architecture. The big challenge was to design an implementation that could compete with the EM-4 implementation. That challenge had been overcome with the implementation architecture ABCL/onAP1000, which incorporates several new techniques for implementing fine-grain concurrent-OO languages in general. The details of these implementations are outlined in [121, 107].

During those times, several new proposals for solving the inheritance anomaly problem had appeared in conferences such as ECOOP and OOPSLA. All had referenced the earlier draft of my analysis work. Unfortunately, although such work had made some progress, it was still unsatisfactory. Also the motivating force was to design a new version of ABCL more suitable for implementation on top of the abovementioned architectures, to which robust and proper support for inheritance was an absolute necessity. I had thus started the design of inheritance mechanism that avoided the anomalies in 1992, which was subsequently completed by the year's end. Special care was taken to incorporate the implementation of the proposal into AP1000, and not to sacrifice its execution efficiency, by close collaboration with Kenjiro Taura, the chief implementor of ABCL/onAP1000. The resulting proposal is described in detailed in the latter chapters of Part 1.

It is especially gratifying, as a researcher in programming language, to see the language of one's own design be acknowledged by the general public and used by someone outside your group. Indeed, the current version of the ABCL/R2 runs on a variety of Common Lisp platforms, and is now distributed publically via anonymous ftp throughout the Internet. This is probably due to the fact that ABCL/R2, both the language itself and its implementation, is at least at the current moment the one-of-its-kind with respect to usability. Hopefully, when ABCL/onAP1000 is completed, it will see widespread use in multiple MPP platforms as well.

Part I

Inheritance in Concurrent Object-Oriented Languages— Anomalies and their Solutions

Chapter 2

Introduction—Inheritance and its Anomalies in Object-Oriented Concurrent Programming Languages

2.1 Motivation

Inheritance is the prime language feature in *sequential* OO (*Object-Oriented*) languages, and is especially important for code re-use. Another important feature is concurrency; although many OO languages in use today (such as C++ and Smalltalk) are sequential, it is natural to consider objects as being a unit of concurrency. A recent breed of OOCP (*Object-Oriented Concurrent Programming*) languages attempt to provide maximum computational and modeling power through concurrency of objects; in particular, our current prototype ABCL/onEM-4 language exhibits a real-life message passing latency less than 10 μseconds for two concurrent objects located on a separate physical node of a multicomputer[121, 107].

Several researchers, however, have pointed out (albeit fragmentarily) the conflicts between inheritance and concurrency in OO languages[4, 56, 91, 109, 18]. More specifically, concurrent objects and inheritance seemingly have conflicting characteristics, thereby inhibiting their simultaneous use without heavy breakage of encapsulation. We have coined such a phenomenon as *inheritance anomaly* in OOCP. Its 'inauspicious' presence has persuaded OOCP languages *not* to support inheritance as a fundamental language feature. Some of the examples are families of Actor languages[61], POOL/T[4], Procol[111], and also, ABCL/1[128, 127]. There are other OOCP languages that do provide inheritance, yet are not concerned with the problems of conflicts — for those languages, we believe that the difficulties presented in this paper are unavoidable in practice.

Inheritance anomaly entails a severe drawback for the development of large-scale and complex systems in OOCP languages, because there, the greatest benefits of using the OO framework are inheritance and encapsulation. It is therefore essential that clean amalgamation of inheritance and concurrency be achieved in order for large-scale systems

to be constructed with OOCP languages. Unfortunately, previous work have largely neglected the proper analysis of the problem, and merely proposed ad-hoc solutions that are applicable for certain types of problems, but as we will see, are inapplicable for others. Instead, we argue that we must first analyze and categorize the conflicts, and based on the analysis, explore if an ideal solution is in fact possible.

Part I of this thesis is organized as follows: First, we give an overview of inheritance anomaly. We will then present non-trivial examples where the (rather simplistic) previous proposals for solutions are limited in their applicability. Next we will analyze and categorize the cause inheritance anomaly more generally. We then examine some latest proposals by others for reusing synchronization code by controlling the inheritance anomaly problem. Finally, we make a proposal in which not only the code for the synchronization but also the *synchronization scheme*—the basic language features such as guards for *programming* synchronization—could be encapsulated in the superclass. Our current proposal is designed with high practicality in mind. It extends and extensively refines the ideas in the past proposals to (1) separate and localize the synchronization schemes from the main bodies of methods, allowing fine-grained inheritance/overriding, and to (2) allow dynamic operations on the methods themselves, in order to control which messages are acceptable by an object. Furthermore, it has the following novel and favorable characteristics:

- Our proposal allows multiple *synchronization schemes*—the basic language features such as guards for *programming* the synchronization of objects—to coexist and be integrated, so that the best scheme can be chosen to program given synchronization constraints.

- The manner we re-use the synchronization code is syntactically similar to superclass method references in sequential OO-languages (e.g.,`super`). Thus, users with experience in OO-programming can readily adapt to our proposal. Inheritance rules are made to depend on each synchronization scheme, however, because the most 'natural' way of inheritance differs among the schemes.

- We offer a high degree of encapsulation and re-use for synchronization code. Furthermore, even synchronization schemes could be encapsulated in superclasses in many cases by proper exporting of class information by the user.

- Expressiveness is not our sole concern—we have also devised speed- and space-efficient incorporation into the software architecture of the aforementioned ABCL/onAP1000. In particular, all space/time-consuming data structure construction for object synchronization can be done at compile-time.

Throughout the thesis, we demonstrate the effectiveness of our proposal by resolving the representative inheritance anomalies. We also give an overview of the implementation architecture, and some preliminary benchmarks that support our claim of efficiency. The proposed language primitives are being incorporated into the MPP version of our concurrent-OO language ABCL, currently being implemented on AP1000 and CM-5.

2.2 Inheritance Anomaly in OOCP

One of the prime concerns in OOCP is *synchronization* of concurrent objects: when a concurrent object is in a certain state, it can accept only a subset of its entire set of messages in order to maintain its internal integrity. We call such a restriction on acceptable messages the *synchronization constraint* of a concurrent object. For example, consider a bounded buffer with methods `put()` and `get()`, where `put()` stores an item in the buffer and `get()` removes the oldest one; then, the synchronization constraint is that one cannot `get()` from a buffer whose state is *empty* and cannot `put()` into a buffer whose state *full* is likewise prohibited.

In most OOCP languages, the programmer explicitly programs the methods to control the set of acceptable messages for each object, in order to *implement* the object *behavior* that satisfy the synchronization constraint. *Synchronization code* is the term we use to refer to the portion of the method code where object behavior with respect to synchronization is controlled. The synchronization code must always be *consistent with* the synchronization constraint of an object; otherwise the object might accept a message that it really should not accept, resulting in a semantical error during program execution[1]. Here, in order to program the synchronization code, the programming language must provide some primitives for object-wise synchronization, such as semaphores, guards, etc.; we refer to the scheme for achieving object-wise synchronization using those primitives in the language as the *synchronization scheme* of the language.

Unfortunately, it has been pointed out that *synchronization code cannot be effectively inherited without non-trivial class re-definitions*. This conflict, which we have coined as *inheritance anomaly* in OOCP, has been identified by several researchers[56, 91, 109], although a comprehensive analysis has not been given yet to our knowledge. Inheritance anomaly is more severe than the violation of class encapsulation in sequential OO-languages that has been pointed out by Snyder[102], because in some of the schemes it is possible to create a general counterexample where NONE of the parent methods can be inherited. We will defer the more detailed analysis of inheritance anomaly until the latter chapters; here, we identify the following situations where the benefits of inheritance is lost:

1. Definition of a new subclass K' of class K necessitates re-definitions of methods in K as well as those in its ancestor classes.

2. Modification of a new method m of class K within the inheritance hierarchy incur modification of the (seemingly unrelated) methods in both parent and descendent classes of K.

3. Definition of a method m might force the other methods (including those to be defined at the subclasses in the future) to follow a specific protocol which would

[1]Such a distinction between the synchronization constraints as a specification versus the behavior of the actual code that implements it, have not been clearly addressed in the previous literatures to our knowledge; in fact, the term 'synchronization constraints' has been confusingly used to mean both in various contexts.

not have been required had that method not existed. Encapsulated definition of *mix-in classes* would thus be very difficult.

One notable fact is that the occurrence of inheritance anomaly *depends on* the synchronization scheme of the language; in other words, re-definitions would be required for classes in an OOCP language that adopted a certain synchronization scheme , while the (semantically identical) classes could be safely inherited in another language that provides an entirely different synchronization scheme . This implies that the heart of the problem is the semantical conflicts between the descriptions of object-wise synchronization and inheritance within the language, and not on how the language features are implemented underneath. Moreover, it is not immediately obvious whether previous techniques developed in concurrent/distributed languages and systems are applicable.

Chapter 3

Inheritance Anomalies in the Previous Proposals

Recently, several proposals have been made for effectively allowing synchronization code to be inherited based on various synchronization schemes (examples are [3, 24, 56, 109, 84], among others). Some (although not all) of these proposals emphasized strong control over the conflicts a.k.a. inheritance anomaly, effectively claiming that synchronization code can be inherited for all common and/or necessary cases. Unfortunately, it is possible to show that such proposals still suffer from inheritance anomaly — in this chapter, we fortify this claim by presenting the actual (counter)examples of anomaly occurrence.

Before we proceed, however, we make a point that the proposals selected here are considered to be representative of certain classes of synchronization schemes , and the intention of the (counter)examples is to illustrate what type of inheritance anomaly would occur for such schemes . We do NOT intend to claim that a particular proposal is useless — as a matter of a fact, some do embody good ideas that could be used as a basis of a more complete solution.

3.1 Simple Examples of Inheritance Anomaly — Caused by 'Body's, Explicit Message Reception within Methods, Path Expressions, Direct Key Specifications

In order to gain the reader's insight into the problem, we first present simple examples of inheritance anomalies occurring in OOCP languages. Some of these cases have already been pointed out by the previous researchers.

3.1.1 Bodies

Some OOCP languages allow each object to have a so called '*body*', an internal method with its own thread of control. The body thread remains active irrespective of the external message reception. The body is typically used to control message receptions, usually in

the fashion of Ada's **select** statement. After receiving a message, the body thread takes on the responsibility of invoking the method corresponding to the message. In some languages, the body thread suspends during method processing, while in others the body thread runs independently of the threads for message processing.

Some researchers refer to objects with body as *active objects*; by contrast, objects without body are called *passive objects*. There have been many variations of active objects in concurrent languages for distributed computing. The notable examples are SR[7], ADA, Mediators[44], and ALPS[113]. These languages have no generic support for inheritance; as a result, conflicts with inheritance naturally do not arise.

America[4] discusses the difficulty of integrating inheritance with languages that allow bodies: On defining a subclass from another class, the definition of the subclass usually require *total re-definition* of the body. This is rather obvious, because otherwise the newly added features cannot be used. America points out that this poses difficulty in programming because having a different body means that the dynamic behavior of such a new object may be totally different from the old ones, thus severely interfering with formal reasoning about the program. America states that, after initial experiments with inheritance in the OOCP language POOL/T, it was decided not to adopt inheritance as a primitive language feature[1]. Another related difficulty we point out is that such re-definitions require total knowledge of and access to the synchronization code of the ancestor classes. Thus, not only that they cannot be inherited, but also encapsulation of class implementation is broken with respect to synchronization constraints.

As an example of the 'body' anomaly, consider a first-in first-out bounded buffer class as illustrated in Figure 3.1. It has two public methods, `put()` and `get()`; `put()` stores an item in the buffer and `get()` removes the oldest one. Two instance variables `in` and `out` count the total numbers of items inserted and removed, respectively, and act as indices into the buffer — the location of the next item to be put is indexed by (`in` *mod* `size`) and that of the oldest item by (`out` *mod* `size`). Upon creation, the buffer is in the empty state, and the only message acceptable is `put()`; arriving `get()` messages are not accepted but kept in the message queue un-processed. When a `put()` message is processed, the buffer is no longer empty and can accept both `put()` and `get()` messages, reaching a 'partial' (non-empty and non-full) state. When the buffer is full, it can only accept `get()`, and after processing the `get()` message, it becomes partial again.

Figure 3.2 is a definition of class `b-buf` which implements the above described behavior, given with an extended syntax of C++ for reader familiarity. (Note that, some liberty is taken with the syntax and semantics — for instance, C++ does not provide the Smalltalk-80 style `super` pseudo variable, whose meaning should be obvious to those familiar in OO programming.). Explicit message reception is made within the body using the `select` and `accept` statements. The `get()` message is accepted by the first `accept` statement in the body if the buffer is not empty; then the actual `process_get()` method is invoked with the `start` statement. Upon its termination, the result of the method invocation is directly returned to the caller. Here, it is quite obvious that in any subclasses of `b-buf`, the entire `body()` must be re-defined in order to account for the

[1]The recent version of POOL called POOL/I incorporates inheritance. Proper body re-definition is left as the responsibility of the programmer.

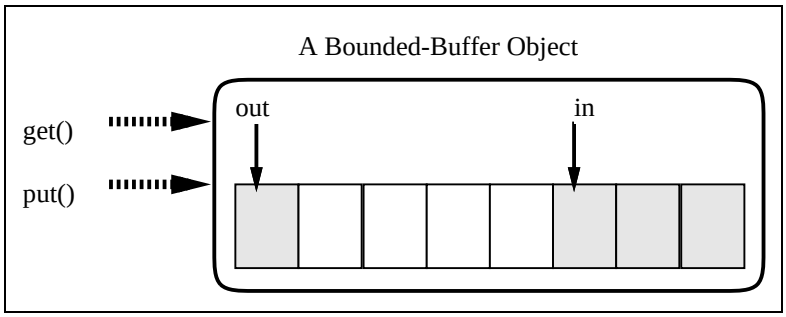


Figure 3.1: A Bounded Buffer Object

newly added method definitions.

```
Class b-buf: ACTOR {// b-buf is an Actor
   int in, out, buf[SIZE];
public:
   void b-buf() {
      in = out = 0;
   }
   void process_put() { //store an item
      in++;
   }        //the argument of the call is omitted
   int process_get() {  //remove an item
      out++;
   }        //the return value of the call is omitted
   void body() {
      loop {
         select {
            accept get() when (!(in == out))
               start process_get();
         or
            accept put() when (!(out = in + SIZE))
               start process_put();
}}}}
```

Figure 3.2: Definition of Bounded Buffer Class with Body (The code related to accessing the local array storage for insertion and removal is omitted for brevity.)

There are several languages that allow body within objects ([11, 24]. [34] also essentially allows bodies when the 'low level' scheme is utilized).

3.1.2 Explicit Message Receptions

An analogous situation occurs if a language allows explicit (interior) reception of messages within a method, in that the newly added method definitions cannot be entirely accounted for. Therefore it would be difficult to incorporate inheritance into languages that allow

interior message receptions. Examples of such languages are ABCL/1[128, 127] and CSSA[83].

There are also languages that extend existing sequential OO languages with explicit message reception statements in order to achieve inter-object concurrency, such as Concurrent C++[43], Buhr et. al.'s extension to C++[19], or Tuple Space Smalltalk[68]. For these languages, however, the messages explicitly received are not processed via the normal method dispatch mechanism of the base language. As a result, inheritance and communication are totally separated from the beginning, causing extensive breakage of encapsulation.

3.1.3 Path Expressions

Again, a similar problem occurs for languages with synchronization schemes expressed in variants of Path Expressions[23]. Additionally, the original path expression suffers the limitation that is imposed by the expressive power of Path Expressions with respect to complex synchronization constraints of objects. For instance, the textual length of the path expression of the above bounded buffer example would be enormous for a large `SIZE`, because one must account for every possible combinations of interleaved `put`s and `get`s; more specifically, the expressive power of the original Path Expressions is limited to the regular expression, whereas the bounded buffer require a more powerful language class for concise description. This can be resolved with augmenting the terms in the path expression with guards and thereby allowing conditional synchronization[8]. An example OOCP language with augmented Path Expression is Procol[111]. Nevertheless, the original problem is not resolved, because one still cannot account for the newly added methods in the subclass unless the entire path expression is re-defined.

3.1.4 Direct Key Specifications

One very important classification of inheritance anomaly is its occurrence in the synchronization schemes involving operations with message keys. We refer to this as the *direct key specification anomaly*. The primary reason for anomaly is that the newly added keys in the subclasses cannot be accounted for in the synchronization scheme of the methods inherited from the parent methods. Languages employing this type of synchronization schemes such as SINA[110][2] or OTM[46] would suffer from the inheritance anomaly if they were to be extended to incorporate inheritance. (For the example of the anomaly occurring with bounded buffers, see[56].)

3.2 Problems with *Behavior Abstractions*

Kafura et. al.'s proposal called the *behavior abstraction*[56] attempts to solve the above problems, especially the problem with direct key specifications, in the context of their

[2]Although SINA does not support inheritance, there is an extension called Sina/ST[3] which employs pattern matching of method names and arguments in the similar manner as the path expression. Inheritance and delegation are simulated using this scheme. The path expression anomaly we have discussed in Section 3.1.3 would occur for this scheme.

language ACT++. The essence of their proposal is to assign identifiers to *accept sets*, namely, the set of keys of messages that can be accepted by an object.

Figure 3.3 is the definition of the bounded-buffer object with behavior abstractions. We basically adopt a simple Actor-like language, whereby:

- Each object is single threaded i.e., an object can only accept one message at a time.
- Message passing is asynchronous, and pending messages are placed in the message queue.
- The next 'behavior' of the object is specified with the `become` primitive (see below).

The `behavior` statements declare three sets of keys named `empty`, `partial`, and `full` assigned to {`put`}, {`put`,`get`}, and {`get`}, respectively. The synchronization scheme employs the `become` statement to designate a set of method keys acceptable in the next state. We call such a set the *next accept set*. This set is not a first-class value; rather, another *key* is designated to each next accept set at the first part of a class definition.

Kafura describes in [56] how behavior abstraction serves as a clean solution to the anomaly exhibited in the `x-buf` example; there, `x-buf` has one additional method `last` that is similar to `get` — the difference is that it removes the last item previously put into the buffer instead of the first. In Figure 3.3, neither `put` nor `get` need to be re-defined in `x-buf`, whereas re-definitions of all the methods were necessary for the comparative language that could only specify the method keys.

Unfortunately, it is possible to create a non-trivial counterexample of inheritance anomaly with behavior abstractions. Consider creating a class `x-buf2`, a subclass of `b-buf`. `x-buf2` has one additional method `get2`, which removes the two oldest items from the buffer simultaneously. (Notice that this cannot be done with successive messages sends of `get`, because `get` messages from different objects may be interleaved.) The corresponding synchronization constraint for `get2` requires that at least two items exist. As a consequence, the partial state must be partitioned into two — the state in which exactly one item exists, and the remaining states. To maintain consistency with the new constraint, we need another accept set `x-one` that represents the former state (the `behavior` definitions in Figure 3.4). Then, the methods `get` and `put` *must be re-defined* (Figure 3.4). Here, notice that NONE of the methods (except the initializer) in `b-buf` can be inherited — the anomaly has occurred again[3].

3.3 Problems with First-Classing of Accept Sets — *Enabled Sets*

Tomlinson and Singh[109] propose a scheme that enhances Kafura's in their Actor-based reflective language called Rosette. In Rosette, the accept sets can be treated as first-class objects called *enabled sets*. We show that this difference is essential, because their

[3]Recently, they have proposed a more advanced scheme called *behavior sets*, which is similar in essence to Tomlinson and Singh's enable sets we discuss next.

```
Class b-buf: ACTOR {// b-buf is an Actor
   int in, out, buf[SIZE];
behavior:
   empty   = {put};
   partial = {put, get};
   full    = {get};
public:
   void b-buf() {
      in = out = 0;
      become empty;
   }
   void put() {
      in++; //store an item
      if (in == out + size) become full;
      else                  become partial;
   }
   void get() {
      out++; //remove an item
      if (in == out) become empty;
      else           become partial;
   }
}

Class x-buf: b-buf {// extends b-buf
behavior:
   x_empty   =                 renames empty;
   x_partial = {put,get,last} redefines partial;
   x_full    = {get,last}     redefines full;
public:
   void x-buf() {
   }
   int last() {
      in-- ; //remove the last item
      if (in == out) become x_empty;
      else           become x_partial;
   }
}
```

Figure 3.3: `B-buf` and `x-buf` with Behavior Abstractions

```
Class x-buf2: b-buf { // x-buf2 is a subclass of b-buf
behavior:
   x_empty   =                renames empty;
   x_one     = {put,get};
   x_partial = {put,get,get2} redefines partial;
   x_full    = {get,get2}     redefines full;
public:
   void x-buf2() { in = out = 0; become x-empty; }
   void get2() { out += 2; //definition of get2
      if (in == out)          become x_empty;
      else if (in == out + 1) become x_one;
      else                    become x_partial;
   }
   //The following re-defines the methods in b-buf.
   void get() { out++;
      if (in == out)          become x_empty;
      else if (in == out + 1) become x_one;
      else                    become x_partial;
   }
   void put() { in++;
      if (in == out + size)   become x_full;
      else if (in == out + 1) become x_one;
      else                    become x_partial;
   }
}
```

Figure 3.4: Inheritance Anomaly with Behavior Abstractions

proposal can localize (although not eliminate) the method re-definitions in some cases. We also show, however, that there are still other cases that would require a considerable amount of re-definitions.

Here is a brief overview of Rosette with respect to synchronization schemes : although its original syntax is based on S-expressions, we will continue to use our C++ based syntax with the following extensions:

- The `become` statement now specifies the next state and the next enabled set of the object:

  `become(`⟨*enabled-set*⟩`, (`⟨*new-state*⟩`))`

- An enabled set is an instance of class `Enable`; here the constructor adopts a special syntax whereby a set of message keys to be enabled are specified:

  `Enable(`⟨*message keys*⟩`)`

  There are several operations defined for the enabled set, such as union (`+`), intersection (`&`), etc.

- In order to specify the next enabled set for an object, we typically define a private method for each enable-set:

  `Enable` ⟨*method*⟩`() {return Enable(`⟨*message keys*⟩`))`

- There are two kinds of methods, *public* and *private*. The public methods are invoked as a result of a message reception from an external object. Message sending is asynchronous, and only those messages whose corresponding methods are currently 'enabled' by the enabled set can be accepted. On the other hand, the private methods are internal to the object and can be only invoked from within the public and private methods of the same object as a function call.

Now, consider defining, in addition to `get2`, method `empty?` which checks whether the buffer is empty or not. The method is in effect stateless, that is, it does not affect the state of the buffer. Thus, this message should always be acceptable irrespective of object state (as long as other methods are not executing). Then, in principle its definition should be independent of definitions of other methods, since the effects on the object state by other methods are irrelevant to `empty?`. But this is not the case — in the definitions of `b-buf` and `x-buf2` (Figure 3.5), we can observe the followings:

- We must override every single private methods that returns an enabled set so that it enables the `empty?` method (all the private methods of `x-buf2` in Figure 3.5).

- We must perform extensive case analysis of object state for the newly added method — this is necessary even if the method itself does not affect the state of the object.

The advantage of enabled-sets over behavior abstractions is that re-definition of the *parent methods*, although unavoidable, can sometimes be *encapsulated* within private

```
Class b-buf: ACTOR { // b-buf is an Actor
  int in, out, buf[SIZE];
private:
  Enable empty() { return enable([put]) };
  Enable partial() { return enable([put,get]) };
  Enable full() { return enable([get]) };
public:
  void b-buf() { in = out = 0;
    become(empty(),(in,out,buf));
  }
  void put() {
    if (in == out + size) become(full(),(in,out,buf));
    else                  become(partial(),(in,out,buf));
  }
  void get() { // Similar to put()...
}

// The entire private methods must be re-defined
Class x-buf2: b-buf {
private:
  Enable empty() {
    return Enable(empty?) + super empty()};
  Enable one() {
    return Enable(get,put,empty?) };
  Enable partial() {
    if (in == out + 1)
       return super partial + Enable(empty?);
    else
       return super partial() + Enable(get2,empty?) };
  Enable full() {
    return super full() + enable(empty?) };
public:
  void x-buf2() { in = out = 0; become x-empty; }
  void get2() { out += 2; // addition of get2()
    if (in == out)          become(empty(),(in,out,buf));
    else if (in == out + 1) become(one(),(in,out,buf));
    else                    become(partial(),(in,out,buf));
  }
  // Painstaking case analysis is necessary
  int empty?()[] { // addition of empty?()
    if (in == out)          become(empty(),(in,out,buf));
    else if (in == out + 1) become(one(),(in,out,buf));
    else if (in == out - 1) become(full(),(in,out,buf));
    else                    become(partial(),(in,out,buf));
  }
}
```

Figure 3.5: `X-buf2` with Enabled-Sets

methods by inheritance. This is seen in Figure 3.5, where only the private methods such as `empty` and `full` are re-defined. This is due to the first-class nature of enabled sets derived naturally from the reflective language architecture of Rosette; by allowing first-class operations on the enabled sets, an instance of a subclass can extend its enabled set to include the methods included in the subclass, supplemented with necessary synchronization code. We feel that reflective architecture is essential in OOCP languages[129], and this is one example of how it can be used to enhance the descriptive power of OOCP languages. Still, as we can observe in this case, all the synchronization code needs to be re-defined, as opposed to method guards, in order to achieve the same functionality.

Here, let us illustrate this by generalizing the method re-definitions of the enabled-sets. The private methods returning an enable set correspond to the 'states' distinguished at class K. On defining a method m at class *K-sub*, a subclass of K, the user needs to check, for each 'state', whether addition of m incurs partitioning of that state. If so, the predicate which determines the state may need to be partitioned. In Figure 3.6, this is done for the private methods `state_1` through `state_n`. In our `empty?` example, since the method was always acceptable, `Enable(...,empty?,...)` had to be added to ALL the private methods of `x-buf2`. Furthermore, on specifying the next behavior of m, the programmer must judge which of the states among those labeled `state_1` through `state_n` is appropriate, depending on the current state of the object (Figure 3.6).

```
Class K-sub: K { //K-sub is a subclass of K
  ⟨Instance Variable Definitions⟩
private:
  Enable state_1() {
    if (⟨method⟩  is acceptable)
      return Enable(⟨method⟩) + super state_1()
    else
      return super state1_() };
  //Repeat for state_2 through state_n
      ...
public:
  ⟨type⟩ ⟨method⟩(⟨args⟩...) {
    return ⟨value⟩;
    if      (Object is in state 1) become(state_1(),⟨new state⟩);
    else if (Object is in state 2) become(state_2(),⟨new state⟩);
          ...
    else if (Object is in state n) become(state_n(),⟨new state⟩);
  }
}
```

Figure 3.6: General Analysis of Enabled Set

Another practical limitation with Rosette is that it involves first-class operations on enable sets each time a message is received. Although the cost of such operations on Rosette could be kept relatively low for coarse-grained concurrency, it is nevertheless unavoidable, and could incur substantial overhead in fine-grain parallel computing (see

[73] for discussions).

Despite its limitations, we do acknowledge the significance of Rosette in pointing out that some first-classing of elements of synchronization scheme provides the possibility of enlarging the class of synchronization code that can be safely inherited. In Chapter 6, we will describe a more elaborate synchronization scheme intended for resolving the inheritance anomaly. Part of our scheme is to automate the inheritance of synchronization code for accept-set based specification of synchronization.

3.4 Problems with Method Guards

A natural synchronization scheme is to attach a predicate to each method as a guard, thus making each object a conditional critical region (for example, [39, 69] and indivisible objects in [62]). We illustrate this for `b-buf` and `x-buf2` in Figure 3.7. Here, we employ the following syntax:

⟨*method name*⟩(⟨*formal arguments*⟩) `when` (⟨*guard*⟩) `{` ⟨*body of method definition*⟩ `}`

where *guard* is a boolean expression whose terms are either constants or instance variables bound to primitive values. Method `m` is invoked only when *guard* evaluates to `True`. For instance, in class `b-buf`, the guard (`in` < `out` + `size`) attached to `put()` assures that `put()` is not invoked when the buffer is full. As shown in Figure 3.7, all the methods defined at `b-buf` are inherited by `x-buf2` without any changes to the methods or the guards.

```
Class b-buf: ACTOR {
  int in, out, buf[SIZE];
public:
  void b-buf()  { in = out = 0; }
  void put() when (in < out + size) { in++; }
  void get() when (in >= out + 1)   { out++; }
}

// x-buf is a subclass of b-buf
Class x-buf2: public b-buf {
public:
  void x-buf2()
  void get2() when (in >= out + 2) { out += 2; }
  void empty?() when (true) { return in == out; }
}
```

Figure 3.7: `B-buf` and `x-buf2` with Method Guards

This scheme does provide an elegant solution to the `get2`/`empty?` example. Furthermore, although a naive implementation of guards is not usually very efficient, it can be

improved with the use of program transformation[69] and other optimization techniques; and since they are usually invisible to the programmer, the full benefit of inheritance can be attained without sacrifices in efficiency.

However, the problem is that the occurrence of inheritance anomaly still cannot be prevented. This is a different kind of anomaly from the ones we have so far discussed in this paper. We will give two examples: one is the definition of the `gget()` method, and the other is the definition of the class `Lock` as a *mix-in class*.

First we consider defining `gb-buf`, a subclass of `b-buf`, adding a single method, `gget()`. The behavior of `gget()` is almost identical to that of `get()`, with the sole exception that it cannot be accepted immediately after the invocation of `put`. Such a condition for invocation cannot be distinguished with method guards and the set of instance variables available in `b-buf` alone; we need to define an extra instance variable `after-put`. As a consequence, both `get()` and `put()` must be re-defined as in Figure 3.8. We note that the analogous situation also occurs for accept set based schemes.

The reason for the anomaly occurrence is that we cannot judge the state for accepting the `gget` message with the guard declarations in `b-buf`. To be more specific, `gget` is a *trace-only or history-only sensitive* methods with respect to instances of `b-buf`; we will defer the discussion until the next chapter.

```
// gb-buf is a subclass of b-buf with gget()
Class gb-buf: b-buf {
   bool after-put;
public:
   void gb-buf() { after-put = False};

   // Definition of gget()
   void gget() when (!after-put && (in >= out + 1))
       { out++; after-put = False; }

   // The following methods must be re-defined
   void put() when (in < out + size) { in++; after-put = True; }
   void get() when (in >= out + 1) { out++; after-put = False; }
}
```

Figure 3.8: Inheritance Anomaly with Guards — the `gget` method

We next consider the `Lock` class, which is an abstract *mix-in* class[17]. Direct instances of `Lock` are not created; rather, the purpose of `Lock` is to be 'mixed-into' other classes in order to add the capability of locking an object. In `Lock`, a pair of methods `lock` and `unlock` have the following functionality: an object, upon accepting the `lock` message, will be 'locked', i.e., will suspend the reception of further messages until it receives and accepts the `unlock` message. Its synchronization constraint is *localized* i.e., it is not affected by methods of the class it is being mixed into.

When `Lock` is 'mixed-into' the definition of `b-buf` to create the class `lb-buf`, we are likely to assume that it would not affect the definition of other methods, since the state

of the object with respect to `lock` and `unlock` is totally orthogonal to the effect of other messages. However, this is not the case — first, we must add an instance variable `locked` which indicates whether the object is currently 'locked' or 'unlocked'; this is obviously necessary since it is impossible to distinguish between the two states otherwise. Then, the inherited methods such as `put` or `get` must be overridden in order to account for `locked` (Figure 3.9). Furthermore, all methods which would be defined in the subclasses of `lb-buf` must also account for `locked`. This would not have been necessary if we were to be defining exactly the same methods in the subclass of `b-buf`. To summarize, the effect of mixing-in `Lock` cannot be localized in `b-buf`.

Why has anomaly occurred here? Again, `lock` and `unlock` are history-only sensitive methods. In addition, although neither of them cause partitioning of states, they modify the synchronization constraints of the methods that are already defined, in this case both `put` and `get`. Thus, method guards of `b-buf` had to be modified in order to maintain consistency with the new constraints.

```
Class Lock: ACTOR {
   bool locked;
public:
   void Lock() {locked = False};
   void lock() when (!locked} {lock = True};
   void unlock() when (locked) {lock = False};
}

// lb-buf is a subclass of b-buf with Lock mix-in
Class lb-buf: b-buf,Lock {
public:
   void lb-buf();
   // The following methods must be re-defined
   void put() when (!locked && (in < out + size)) { in++; };
   void get() when (!locked && (in >= out + 1)) { out++; };
}
```

Figure 3.9: Inheritance Anomaly with Guards — the `Lock` class

Chapter 4

Analysis of Inheritance Anomaly

We have seen through examples that the previous proposals are not sufficient for avoiding the inheritance anomaly. We believe that their shortcomings are due to insufficient analysis of the situation; that is to say, the conflict we treat here is deeply rooted in the semantics of synchronization constraint/schemes verses semantics of inheritance, and analysis is first necessary for achieving a sufficiently clean solution.

There are three reasons why inheritance anomaly occurs, depending on what the subclass definition entails on how the *state* of the object upon which the messages are acceptable are modified:

- **Partitioning of Acceptable States** — `get2`, `gget`
- **History-only Sensitiveness of Acceptable States** — `gget`, `lock`
- **Modification of Acceptable States** — `lock`

The three causes are relatively independent; for example, the `gget` partitions the states as well as being history-only sensitive.

4.1 Partitioning of States

The `x-buf2` example in Figure 3.2 is a anomaly caused by *partitioning of acceptable states*. In object-oriented languages, an object is said to have some 'state'. Then, one can consider the 'set of states' an object can have. This set can be partitioned into disjoint subsets according to the synchronization constraint of the object; in the bounded buffer examples in Chapter 3, there are three distinguishable set of states, under which respective sets of acceptable messages can be defined: *empty*, *partial*, and *full*. This is conceptually illustrated by the left rectangle of Figure 4.1.

Now, when a new method is added in the definition of the subclass, the partitioning of the set of states in the parent class may need to be further partitioned in the subclass; this is because the synchronization constraint of the new method may not be properly be accounted for in the partitioning of the parent class. In our example, when the `get2()` method was added in `x-buf2`, a partitioning of `x-partial` into `x-one` and `x-partial` was necessary in order to distinguish the state at which only one element is in the buffer.

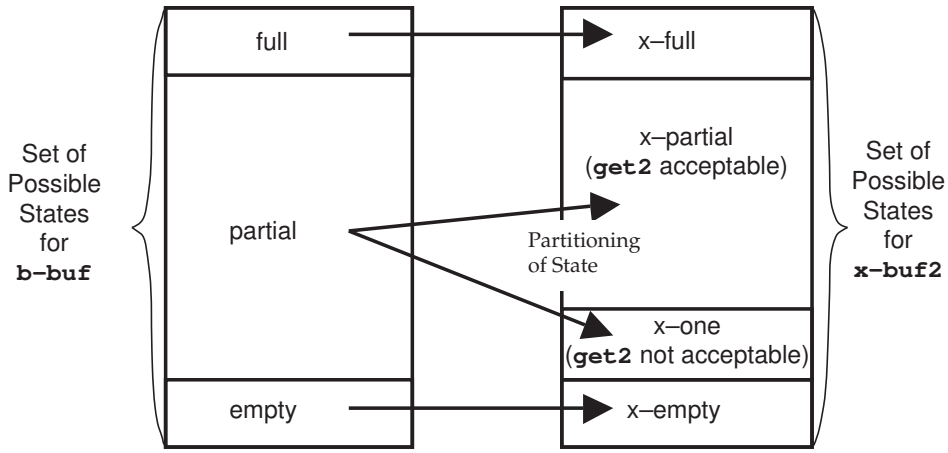


Figure 4.1: Conceptual Illustration of the State Partitioning Anomaly

For accept set based synchronization schemes, this state partitioning is usually distinguished at the termination of the methods with some conditional statements, upon which the objects 'become' that state. This is seen for example in the definitions of `put` and `get` methods in Figure 3.3. Requirement for method re-definitions follows naturally, as we have illustrated in Figure 3.4, because the new partitioning must be accounted for in all the methods. Note that, this is not resolved by making accept sets first-class values, because this partitioning cannot be affected by the operations upon the accept-set data.

This partitioning is not a problem for method guards, because they are able to directly judge whether the message is acceptable or not under the current state. Thus, even if the new methods were added, the guards would not need be re-defined, provided that it would not affect the partitioning of the methods in such a way that the condition denoted by a guard in a certain method would no longer be valid; this certainly holds for most cases of inheritance.

Localization of the synchronization schemes is possible via the use of private (localized) method definitions and first-classing of accept sets. This would in turn allow the sharing of synchronization schemes. The advantage here is that the re-definitions of methods that share and/or re-use the synchronization schemes could be localized to within a single method. However, it is not clear if there are situations where such schemes would be advantageous over method guards in the first place.

4.2 History-only Sensitiveness of States

When two different views in modeling the 'state' of objects. One is the *external view*, where the state is captured indirectly by the external observable behavior of the object. This view is taken by the models of parallelism based on process calculi, such as CCS[79] and Actors[1]; there, the equivalence of two objects are determined solely with how they respond to external experiments, and not with how their internal structures are

constructed[1]. Another is the *internal view*, where the state is captured by the valuation of the state variables in the implementation of the object; for example, a Cartesian point object can have a valuation such that its x-coordinate is 3, and its y-coordinate is 5. (The actual semantics is more complicated by the fact that the valuation could be another object, and that objects have methods with self and super references (see [93]).)

The two views on state are not identical; there are set of states whose elements can be distinguished under the external view, but is indistinguishable under the internal view. With method guards, in particular, only the latter states are distinguishable, because guards are usually boolean expressions consisting of constant object values, instance variables of the object, and various arithmetic/logical operators (other syntactic categories such as message keys are usually not allowed). Then, it follows that there exist some synchronization constraint that cannot be specified with a given set of instance variables and method guards: this is precisely the history information that do not manifest itself in the values of the instance variables.

When such a distinction becomes necessary, the state of the object under the internal view must be 'refined' in order to match the state of the external view. For this purpose, the methods in a parent class must be modified; that is to say, the state of the object is *history-only sensitive* with respect to the internally distinguishable ones. This is illustrated in our previous `gget` example in Figure 3.8, where the state "immediately after accepting `put`" cannot be distinguished with the set of instance variables available in `b-buf`, requiring the addition of an instance variable `after-put`. Since the proper valuation of this variable must be done in all the methods, the requirements of method modification arose (The situation is similar for accept set based schemes in this respect, in that the `gget` example would require considerable re-definitions.). Also notice that `gget` partitions the state as well.

4.3 Modification of Acceptable State

The methods in the `Lock` example in Figure 3.9 are *history-only sensitive* in a similar manner as `gget`. The difference from `gget` is that the execution of the methods in `Lock` modifies the set of states under which the methods inherited from the parent could be invoked (Figure 4.2). That is to say, mixing-in of `Lock` introduces finer-grained distinction for the set of states under which `get` (or `put`) in `lb-buf` can be invoked. This would naturally require the modification of the method guards to account for the new synchronization constraint.

In addition, the `lock` method in class `Lock` is *orthogonally restricting* in the following sense: `Lock` imposes restriction to the set of states in which each method of the parent class can be invoked; this restriction is determined orthogonally to whether that method can be invoked when that instance is regarded as an instance of the parent class. For example, whatever the state of the object (i.e., whether the state is an element of full, partial or empty in the parent class), the question whether the instance is locked or not

[1]To be more precise, the equivalence relation of objects are typically defined by the *bisimulation* relation. One could define several classes of bisimulation relations, yielding weaker or stronger equivalences according to his requirements, e.g. whether object congruence is required, etc.

solely depends on the state of the variable `locked`. This may not always be the case in general; nevertheless, *orthogonally restricting* is an important subclassification in which the first-class accept set based schemes exhibit good characteristics.

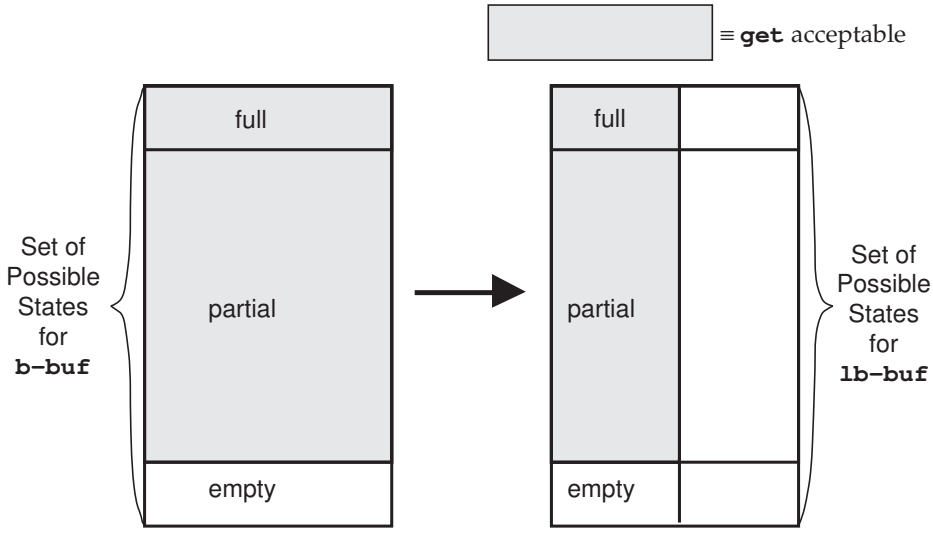


Figure 4.2: Conceptual Illustration of the State Modification Anomaly

4.4 Examples of Analysis of Anomaly Occurrence—Synchronizing Actions

Given the above categorizations, we can now analyze the effectiveness of the synchronization schemes, and create an example of anomaly occurrence. Below is the brief analysis on *Synchronizing Actions* which was recently proposed[84]. The primary synchronization scheme in Synchronizing Actions is essentially the same as behavior abstractions, but it also supports method guards in the form of preconditions. Figure 4.3 is the definition of a bounded buffer in Synchronizing Actions. The four keywords in the method definition are as follows: `matching`, `action`, `pre`, and `post` specify the guard, the pre-actions, the method body, and the post-actions, and respectively. Synchronizing Actions supports intra-object concurrency, and behavior abstractions is used to exclude mutually interfering operations.

Since Synchronizing Actions utilize behavior abstractions, one could conjecture that the anomalies presented in Section 3.2 could occur. It is a little bit difficult, because most of the partitioning is absorbed in the guards; it is however, possible to create a mutual exclusion condition that cannot be reflected to the guards, thus requiring re-definitions. As an example, we define a class `extended-bounded-buffer2`, which extends the `bounded-buffer` class with a method `read-middle`, which returns the middle elements of the buffer excluding the head and tail. Thus, `read-middle` should

not be invoked when the buffer consists of less than three elements. Furthermore, suppose that the implementation details require that there exist five or more elements for `read-middle` to be mutually independent from both `put` and `get` (this alternative partitioning cannot be reflected to the guards). Figure 4.3 is the resulting subclass definition of `extended-bounded-buffer2`. Here, notice that not only the `concurrency-control` part have to be extended, but also the methods themselves must also be re-defined (the precondition part). The required re-definition occurs for the same reason as the original behavior abstractions — the partitioning of states.

4.5 The Formal Analysis of Inheritance Anomaly— State Partitioning

In the previous section, we have given categorizations and intuitive overviews of reasons for inheritance anomaly occurrence. To make our claim more general, we will analyze the inheritance anomaly problem more formally; in particular, we show that, for accept-set based schemes, it would always be necessary for the synchronization specification (or, code) in the parent classes to be modified in order to maintain consistency with the synchronization constraints. This is shown by employing a simple operational model of concurrent objects with inheritance and synchronization constraints. The outline of the proof is as follows: First, we make a minor extension to the Cook-Palsberg inheritance semantics[33] to incorporate object states. Then, restricting our attention to state transitional behavior objects, we define functions that characterize the synchronization constraints of objects. Next, synchronization specifications are categorized into two schemes, one using predicates and the other using accept sets, and functions that characterize both schemes are given; consistency is then defined in terms of extensional equivalence of the functions. Finally, we prove the occurrence of the anomaly by constructing the characterization function for the scheme using accept sets, and showing that on creation of a subclass, the synchronization specifications in the superclasses must be modified in order to maintain consistency with the synchronization constraints.

4.5.1 Overview of Cook-Palsberg Inheritance Semantics and its Extensions

We first employ the semantics of inheritance defined by Cook and Palsberg[33]. We make minor extensions to incorporate object states, just simple enough for our purpose. Also, slight restrictions are made so that classes are *well-formed*. Other notations used here follow their paper.

Domains and Functions in the Method System

The definitions of method system domains are identical to Cook's (Figure 4.4). The definition of semantic domains is identical, except for the addition of **Stat**, the domain of states of instances (Figure 4.5). Intuitively, the state of an instance can be given with the values bound to its instance variables. (We could define it more precisely as is done

```
class bounded-buffer;
private:
   const SIZE = 64;
   int in = 0, out = 0, buf[SIZE];
concurrency-control:
   int N = 0;          //counts queued elements
   behavior-abstraction
      op-on-head = { get }
      op-on-tail = { put }
public:
   method put(int elem);
      matching   ( N < SIZE );
      pre        { exclude op-on-tail; }
      action     { in++; /* add element to tail of buf */ }
      post       { N++; }
   method int get();
      matching   ( N > 0 );
      pre        { exclude op-on-head; }
      action     { /* return element from head of buf */ out++; }
      post       { N--; }
end bounded-buffer;

class extended-bounded-buffer2 inherits bounded-buffer;
concurrency-control:
   behavior-abstraction // new exclusion sets for read-middle
      op-on-head-and-tail-and-middle = { get, put, read-middle-elements }
      op-on-head-and-middle = { get, read-middle-elements }
      op-on-tail-and-middle = { put, read-middle-elements }
public:
   method int[] read-middle-elements();
      matching   ( N >= 3 );
      pre        { exclude op-on-head-and-tail-and-middle; }
      action     { /* return the middle elements of buf excluding two
                elements from both head and tail */ }
      post       { N = 2; }
   // re-definitions of both put and get methods in bounded-buffer
   method put(int elem);
      matching   ( N < SIZE );
      pre        { if (N >= 5) exclude op-on-tail
                   else exclude op-on-tail-and-middle; }
      action     { in++; /* add element to tail of buf */ }
      post       { N++; }
   method int get();
      matching   ( N > 0 );
      pre        { if (N >= 5) exclude op-on-head
                   else exclude op-on-head-and-middle; }
      action     { /* return element from head of buf */ out++; }
      post       { N--; }
end bounded-buffer;
```

Figure 4.3: Inheritance Anomaly in Synchronizing Actions

Instances:	ρ, φ	$\in$	**Ins**
Classes:	κ	$\in$	**Cls**
Message Keys:	m	$\in$	**Key**
Primitive Functions:	f	$\in$	**Prim**
Method Expressions:	e	$\in$	**Exp**

Figure 4.4: Method System Domains

Undefined:			?		
Numbers:			**Num**		
Values:	α	$\in$	**Val**	$=$	**Beh** + **Num** + **Stat**
Behaviors:	σ, π	$\in$	**Beh**	$=$	**Key** $\rightarrow$ (**Fun** + ?)
Functions:	ϕ	$\in$	**Fun**	$=$	**Val** $\rightarrow$ **Val**
States:	ζ, ξ	$\in$	**Stat**		
Generators:			**Gen**	$=$	**Beh** $\rightarrow$ **Beh**

Figure 4.5: Semantic Domains

for standard denotational semantics by introducing local locations for each instance, but for our purpose, the simple definition we give here suffices.) Introducing **Stat** also affects the domain of values, **Val**, as shown in Figure 4.5.

Two auxiliary functions are added to handle states. To specify the set of all possible states of instances of a class, we use the function *States* for each class κ. It has the property that for any class κ, $States(\kappa) \subset \mathbf{Stat}$. The auxiliary function *state* returns the current state of a given instance (Figure 4.6). We also extend Cook's definition of *par* (standing for 'parent') as follows:

$$par^n(\kappa) = \underbrace{par(par(\ldots par(\kappa)))}_{n} \quad (n \geq 1)$$

Henceforth, we adopt the notations $\kappa^\rho \stackrel{\text{def}}{=} class(\rho)$ and $\zeta^\rho \stackrel{\text{def}}{=} state(\rho)$ for brevity.

We also employ a portion of the functions in Cook's denotational system of inheritance, as shown in Figure 4.7. For convenience, we define $bec = \lambda\kappa.fix(gen(\kappa))$, which is the common behavior for all instances of a given class.

Well-Formed Classes (WFC)

Now that we have established the domains and functions in the method system, we need to define the notion of *well-formed classes (WFC)*. When a class is well-formed, all the methods available for the class are *well-defined*, that is, invocations of all methods that was defined at that class or at its superclasses do not result in `messageNotUnderstood` or infinite looping in the method lookup. The reason for the well-formedness requirement is that we need the exact notion of what method keys are available for a given class. For

class	:	$\mathbf{Ins} \rightarrow \mathbf{Cls}$	the class of an instance
par	:	$\mathbf{Cls} \rightarrow (\mathbf{Cls} + ?)$	the parent class of a class
meth	:	$\mathbf{Cls} \rightarrow \mathbf{Key} \rightarrow (\mathbf{Exp} + ?)$	non-inherited methods for a class
root	:	$\mathbf{Cls} \rightarrow \mathbf{Bool}$	*true* if root class, *false* otherwise
States	:	$\mathbf{Cls} \rightarrow 2^{\mathbf{Stat}}$	the set of possible states for instances of a class
state	:	$\mathbf{Ins} \rightarrow \mathbf{Stat}$	the current state of an instance

(Abbreviations: $\kappa^\rho \stackrel{\text{def}}{=} class(\rho)$ and $\zeta^\rho \stackrel{\text{def}}{=} state(\rho)$)

Figure 4.6: Auxiliary Functions

beh	:	$\mathbf{Ins} \rightarrow \mathbf{Beh}$	the behavior of an instance (to be extended in Sect. 4.5.2)
gen	:	$\mathbf{Cls} \rightarrow \mathbf{Gen}$	the *generator* of a class
bec	:	$\mathbf{Cls} \rightarrow \mathbf{Beh}$	the behavior common to all instances of a class
bec	=	$\lambda\kappa.fix(gen(\kappa))$	(to be extended in Sect. 4.5.2)

Figure 4.7: Functions in the Denotational System of Inheritance

instance, a method becomes unavailable with the addition of an overriding method that is not well-defined, invalidating that method key. There could be inverse cases where well-defined methods might be associated with all keys acceptable for a certain class, but somewhere up its inheritance path, a non well-defined method (which was hidden by an overriding well-defined method) might surface.

For our purpose, we assume that if a function is given a value in an illegal domain, it would return $\perp_?$. This allows us to define the following:

Definition 1 (Valid Domain Function) *Let f be a function. Then, $vd(f)$ is the subset of the domain of the function for which the mapped value is not $\perp_?$. To be more precise,*

$$vd(f) = \{x \mid x \in \text{domain of } f,\ f(x) \neq \perp_?\}.$$

On defining a method, we need a pair consisting of a method key ω and a method expression e_ω. We use a notation $\omega \mapsto e_\omega$ for such a pair. Note that both ω and e_ω are syntactic entities. We also need to specify exactly at which class in the inheritance tree a given method is defined.

Definition 2 *A method $\omega \mapsto e_\omega$ is* defined at *class κ iff $meth(\kappa)\omega \in \mathbf{Exp}$.*

The function *beh*, given a key, should return an element of **Fun**. However, an analysis reveals two possible cases which may prevent this:

- The function is undefined — for example, there exists a method with key m, whose associated expression contains a `super` message send with key ω, but the corresponding method is undefined in the superclasses.

$$
\begin{array}{rcll}
M & : & \mathbf{Ins} \to \mathbf{Key} \to \mathbf{Stat} \to (\mathbf{Ins} \times \mathbf{Key})* & \text{(Message Dispatch)} \\
M(\rho)m\zeta & = & [[\varphi_1, \omega_1], \ldots, [\varphi_n, \omega_n]] & \\
 & & & \\
S & : & \mathbf{Ins} \to \mathbf{Key} \to \mathbf{Stat} \to \mathbf{Stat} & \text{(State Transition)} \\
S(\rho)m\zeta & = & \zeta' &
\end{array}
$$

Figure 4.8: Object Behavior Functions

- The function evaluation does not terminate — for example, there is a mutual recursion among the methods defined at the same class.

We then restrict ourselves to *well-defined* methods, whose associated expression evaluate to a function in **Fun**, and not to the $\perp_?$ element:

Definition 3 (Well-Defined Method)
A method $\omega \mapsto e_\omega$ *is* well-defined at class κ *iff A method* $\omega \mapsto e_\omega$ *is defined at class* κ *and* $bec(\kappa)m \in$ **Fun**

We now define *well-formed class*, whose available methods are all well-defined.

Definition 4 (Well-Formed Class (WFC)) *A class* κ *is* well-formed *iff* κ *is a root class, or:*

1. $\forall m \in vd(meth(\kappa))$, m *is well-defined at* κ,
2. $par(\kappa)$ *is well-formed, and*
3. $\exists n$ *such that* $root(par^n(\kappa)) = true$ *(i.e., the message lookup does not 'loop').*

Some of the properties of WFCs, which we give without proof, are as follows:

- The well-formedness property is inherited; that is, when one creates a subclass of a WFC by adding only well-defined methods, then the created subclass is a WFC.
- A set of acceptable method keys of an instance, $(vd(beh(\rho)))$, is equivalent to the union of all keys of its class and of its ancestor classes.

4.5.2 Concurrent Objects with Synchronization Constraints

We will next define the operational behavior of concurrent objects with synchronization constraints. Since our prime intention is to characterize a concurrent object ρ with its behavior, $(beh(\rho))$, its state, (ζ^ρ), and transition of states, we use an extension of the model given by Shibayama[97], which introduces the notion of states, as given in Figure 4.8. For our purpose, we only need S, for we will only employ state transition behavior of a single object. So we let $beh(\rho)$ be S by restricting the domain of primitive functions to

Stat $\rightarrow$ **Stat**; for a function[2] that just returns a behavior or performs some calculations, we define it as $id_{\mathbf{Stat}}$. Then beh : **Ins** $\rightarrow$ **Key** $\rightarrow$ **Stat** $\rightarrow$ **Stat** as required. Since our system has classes, we also employ bec : **Cls** $\rightarrow$ **Key** $\rightarrow$ **Stat** $\rightarrow$ **Stat**.

Accept Function

Our next step is to characterize the 'synchronization constraints'. For this, we introduce the *accept function*, which denotes the abstract condition of message acceptance. As we have indicated earlier, a message is acceptable by an object if the invocation of the associated method does not violate the assertion (concerning the synchronization constraints) that must hold for the current state of the object, in order for the internal consistency of the object to be maintained. This is a natural and powerful way to characterize the synchronization constraints, as object states can be partitioned by arbitrary predicates regarding its state. The predicate may even include terms regarding implicit states, such as history of method invocations.

The intuitive meaning of the accept function $\mathcal{A}$, which is defined below, is as follows: for an instance of a particular class, upon receipt of a message, when in a particular state, $\mathcal{A}$ decides whether it is safe to accept the message or not. For instance, the `get()` message in class `b-buf` should not be acceptable if the instance is in the empty state, i.e., informally $\mathcal{A}$(`b-buf`)(`get()`)(`in == out`) = *false* (Note: `in == out` stands for the empty state).

Definition 5 (Accept Function) Accept function $\mathcal{A}$ *is defined as follows:*

$$\mathcal{A} \quad : \quad \mathbf{Cls} \rightarrow \mathbf{Key} \rightarrow \mathbf{Stat} \rightarrow \mathbf{Bool}$$

$$\mathcal{A}(\kappa)m\zeta \quad = \quad \begin{cases} \mathit{true} & \text{if } m \in vd(bec(\kappa)) \text{ and message } m \text{ is acceptable at state } \zeta \\ \mathit{false} & \text{if } m \in vd(bec(\kappa)) \text{ and message } m \text{ is not acceptable at state } \zeta \\ \perp_? & \text{otherwise} \end{cases}$$

Accept Sets

Alternatively, given an object and its state, we can enumerate the methods whose invocation do not violate the assertion that must hold for the current state of the object. By enumerating the keys of the methods, we obtain a finite set of method keys, which we call the *accept set*.

Definition 6 (Accept Set) *The* accept set b *is a finite set of method keys for an instance* ρ *of a WFC* κ^ρ*, for which the associated methods are acceptable for its current state* ζ^ρ*.*

Now, given $\mathcal{A}$, we can derive the *accept set function* $\mathcal{B}$ as a function returning, for a given state, the accept set of an instance of a class:

$$\begin{aligned} \mathcal{B} \quad &: \quad \mathbf{Cls} \rightarrow \mathbf{Stat} \rightarrow 2^{\mathbf{Key}} \\ \mathcal{B} \quad &= \quad \lambda\kappa\zeta.\{m \mid m \in vd(bec(\kappa)), \mathcal{A}(\kappa)m\zeta = \mathit{true}\} \end{aligned}$$

[2] Well-termination of functions is made even more complex with the synchronization constraints, such as the possibility of a deadlock. Here, for the sake of simplicity, we do not consider such ill-formed cases; see [79] and others for these and other issues relevant to model of concurrency.

Each accept set naturally satisfies the following condition: for any element key in the accept set, the method corresponding to the key is already defined at that class or its superclasses, *not those methods that are going to be defined at the subclasses.* This condition, which is a significant property of accept sets, is stated formally below; it is trivial to prove that it holds for any WFC κ.

$$\forall \zeta \in States(\kappa).(\mathcal{B}(\kappa)\zeta \subset vd(bec(\kappa)) \in 2^{\mathbf{Key}})$$

4.5.3 Schemes for Synchronization Specification: DPSS and DKSS

Before proceeding, we stress the point that $\mathcal{A}$ (and its derivative $\mathcal{B}$) merely characterize the object's synchronization constraints that must be *satisfied* to maintain its internal integrity, and does not characterize the *behavior* of an object with respect to message acceptance. The program description for controlling message acceptance, i.e., the *synchronization specification*, must be given within the textual definition of each class. In other words, $\mathcal{A}$ is a function derived from the synchronization constraints, and does not characterize the synchronization specifications.

For example, in `b-buf` of Section 2.2, the synchronization constraint for an empty buffer is that only message `put()` can be accepted. Then, the synchronization specification is given in the definition of `b-buf` with **become** statements so that it would be *consistent* with the synchronization constraint i.e., the constraint is always satisfied. If we would mistakenly write **become** `full` for **become** `empty` in the method `get()`, then the synchronization specification would not be *consistent* with the synchronization constraint for `b-buf`, causing an error. In such a case, the synchronization constraint is said to be *broken.* Here, for a more precise treatment of the notion of consistency, we categorize schemes for synchronization specification into those using predicates and those using accept sets. Then, for each scheme, we define a characteristic function that represents the behavior of objects with respect to message acceptance.

Direct Predicate Specification Scheme (DPSS)

The first category specifies a predicate per each method indicating the condition under which the message that invokes the method can be accepted. For example, one could attach a predicate as a guard within the text of the method expression, or give a separate description in the class definition[3]. We categorize such a scheme of synchronization specification as the *Direct Predicate Specification Scheme (DPSS).* An example of DPSS will be given in Section 4.

The behavior of an object whose synchronization specification is given with DPSS is represented by a function $\mathcal{A}_P$: **Cls** $\rightarrow$ **Key** $\rightarrow$ **Stat** $\rightarrow$ **Bool**. In order for the synchronization specification of the object to be consistent with its synchronization constraint, $\mathcal{A}$ and $\mathcal{A}_P$ must be *extensively equivalent*, that is, for a given key, and state, they must evaluate to the same value; otherwise, the

[3]We need not question whether the non-accepted message is either discarded or placed somewhere in the message queue, for it is irrelevant to the current argument.

synchronization constraints would be broken. To present an informal example, $\mathcal{A}$(`b-buf`)(`get()`)(`in == out`) = $\mathcal{A}_P$(`b-buf`)(`get()`)(`in == out`) = *false* must hold. Otherwise, if $\mathcal{A}_P$(`b-buf`)(`get()`)(`in == out`) were *true*, an attempt would be made to `get()` from an empty buffer, which would result in an error.

Definition 7 *The accept function $\mathcal{A}$ and the DPSS function $\mathcal{A}_P$ are said to be* consistent for class κ *iff $\mathcal{A}(\kappa)$ and $\mathcal{A}_P(\kappa)$ are extensively equivalent for all $m \in vd(bec(\kappa))$ and $\zeta \in States(\kappa)$.*

Direct Keyset Specification Scheme (DKSS)

Alternatively, one comes up with an idea of directly specifying the accept sets as first-class entities within the program descriptions (which, as we shall see later, is the root of the anomaly). Again, these can be given within method definitions, or as separate descriptions within class definitions. We categorize this scheme as the *Direct Keyset Specification Scheme (DKSS).*

The behavior of an object whose synchronization specification is given with DKSS is represented by a function $\mathcal{B}_K : \mathbf{Cls} \rightarrow \mathbf{Stat} \rightarrow 2^{\mathbf{Key}}$. As was with $\mathcal{A}$ and $\mathcal{A}_P$, $\mathcal{B}$ and $\mathcal{B}_K$ must be extensively equivalent; otherwise synchronization constraints would be broken. Another requirement derived from the equivalence is that the keys appearing in the DKSS program description must only be those of the methods that were defined at the class or at the ancestor classes — that is, methods defined at the subclasses cannot appear in the description. In more formal terms, it must satisfy for each class κ, the predicate $\forall \zeta \in States(\kappa).(\mathcal{B}_K(\kappa)\zeta \subset vd(bec(\kappa)))$. Here, one must be careful — this condition was naturally satisfied with $\mathcal{B}$, but in this case the keys given by a program could be arbitrary; for instance, one could refer to a method key of a subclass by mistake. Such cases must be detected by the interpreter/compiler and reported as an error.

Definition 8 *The accept set function $\mathcal{B}$ and the DKSS function $\mathcal{B}_K$ are said to be* consistent for class κ *iff $\mathcal{B}(\kappa)$ and $\mathcal{B}_K(\kappa)$ are extensively equivalent for all $\zeta \in States(\kappa)$.*

The relationship between $\mathcal{A}, \mathcal{A}_P, \mathcal{B}$, and $\mathcal{B}_K$ is illustrated in Figure 4.9. Notice that, although $\mathcal{B}$ is derived from $\mathcal{A}$, there is no direct connection between $\mathcal{A}_P$ and $\mathcal{B}_K$.

4.5.4 Previous Proposals and DKSS

DKSS is a generalization of the synchronization specification employed in many of the previous proposals. Although those proposals seem superficially different, they are in fact variations of DKSS. We will explain how such variations can be captured within our framework:

- Some proposals represent our notion of accept sets as first-class identifiers[56]. In our framework, this is equivalent to systematically assigning a unique first-class identifier `a`$_i$ to each accept set. This is possible, as any set of methods $vd(bec(\kappa))$

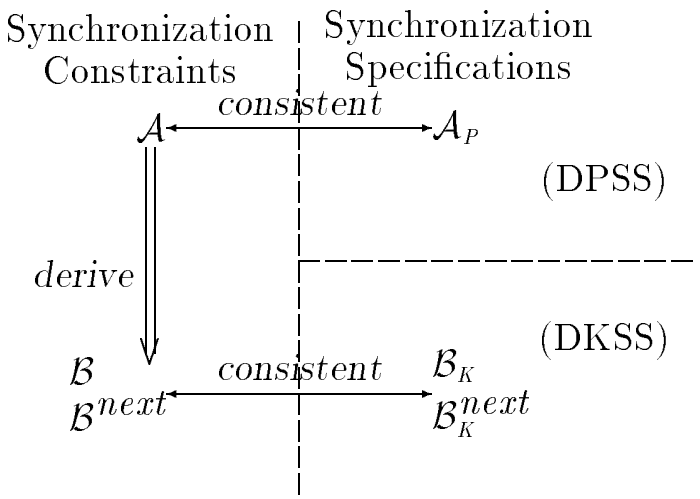


Figure 4.9: Relationships Between the Synchronization Constraints and the Synchronization Specifications

is finite:

$$\begin{aligned} \mathtt{a_0} &\stackrel{\text{def}}{=} \emptyset \\ \mathtt{a_1} &\stackrel{\text{def}}{=} \{m_1\} \\ &\vdots \\ \mathtt{a_k} &\stackrel{\text{def}}{=} \{m_{k_1}, m_{k_2}, \ldots\} \\ &\vdots \end{aligned}$$

Then, for the synchronization constraints, we can regard the function $\mathcal{B}(\kappa)$ as returning an identifier, i.e., $\mathcal{B}(\kappa)\zeta \longmapsto \mathtt{a_i}$. For synchronization specification, we employ $\mathtt{a_i}$ within the method expression e to denote an accept set; thus $\mathcal{B}_K(\kappa)$ also returns an identifier as a result, and the equivalence can be defined in terms of the identifier equivalence.

- Many proposals specify the *next accept set*, that is, the accept set for the next method invocation. In our framework, this corresponds to the accept set for the state after a state transition. The characterization of next accept sets induced by the synchronization constraints is given with the *next accept set function* $\mathcal{B}^{next}$, derived from $\mathcal{A}$ as follows:

$$\begin{aligned} \mathcal{B}^{next} &: \mathbf{Cls} \rightarrow \mathbf{Key} \rightarrow \mathbf{Stat} \rightarrow 2^{\mathbf{Key}} \\ \mathcal{B}^{next} &= \lambda\kappa m\zeta.\{m' \mid m' \in vd(bec(\kappa)),\ \mathcal{A}(\kappa)m\zeta = true,\ \mathcal{A}(\kappa)m'(bec(\kappa)m\zeta) = true\} \end{aligned} \tag{4.1}$$

The behavior of an object whose synchronization specification given with the scheme of this variation is represented by a function $\mathcal{B}_K^{next} : \mathbf{Cls} \rightarrow \mathbf{Key} \rightarrow \mathbf{Stat} \rightarrow 2^{\mathbf{Key}}$. We define the consistency requirement between $\mathcal{B}^{next}$ and $\mathcal{B}_K^{next}$ analogously to those of $\mathcal{B}$ and $\mathcal{B}_K$.

$$
\begin{array}{lcl}
\mathit{slook} & : & \mathbf{Cls} \to \mathbf{Key} \to (\mathbf{Stat} \to 2^{\mathbf{Key}}) \\
\mathit{slook} & = & \lambda\kappa m.\ [\lambda e \in \mathbf{Exp}.\mathit{SyncSpec}(\kappa) \\
 & & \quad \lambda v \in ?.\ \mathbf{if}\quad \mathit{root}(\kappa) \\
 & & \qquad\qquad\quad \mathbf{then}\ \lambda\zeta \in \mathbf{Stat}.\emptyset \\
 & & \qquad\qquad\quad \mathbf{else}\ \mathit{slook}(\mathit{par}(\kappa))m \\
 & &](\mathit{meth}(\kappa)m)
\end{array}
$$

Figure 4.10: Auxiliary Function *slook* (in Cook's notation)

We prove that with DKSS, anomaly always occurs in inheritance. This corresponds exactly to occurrence of state-partitioning anomaly in naive accept-set based synchronization schemes.

4.5.5 Proof of the Anomaly in Inheritance with DKSS

We now prove that the anomaly in inheritance always occurs for DKSS. Our proof is done with $\mathcal{B}_K^{next}$ for convenience, but a similar proof can be formulated for $\mathcal{B}_K$. For simplicity, we only consider single inheritance, and that only a single method is defined at each class.

Let $\kappa_0, \kappa_1, \ldots, \kappa_{n-1}$ be well-formed classes, where $\mathit{root}(\kappa_0) = \mathit{true}$, and $\mathit{par}(\kappa_i) = \kappa_{i-1}$ for $1 \leq i \leq n-1$. Let $\omega_i \mapsto e_{\omega_i}$ be the well-defined method at class κ_i. We next refine the formulation of $\mathcal{B}_K^{next}$: with each definition of method $\omega_i \mapsto e_{\omega_i}$ at κ_i, we associate a function $\beta_i : \mathbf{Stat} \to 2^{\mathbf{Key}}$ satisfying the condition we described in Subsection 4.5.3, $\forall\zeta \in \mathit{States}(\kappa_i).(\beta_i(\zeta) \subset \mathit{vd}(\mathit{bec}(\kappa_i)))$. β_i is an abstraction of the program description of synchronization specification made with DKSS for the method $\omega_i \mapsto e_{\omega_i}$ at class κ_i. Such a description can either be given within the method expression e_{ω_i}, or separately within the class specification[4]. In both cases, β_i is said to be *defined at class* κ_i *for* ω_i. To present an informal example, if the class `x-buf` only were to add a single method `get2()`, then $\beta_{\texttt{x-buf}}(\texttt{in == out}) = \{\texttt{put()}\}$.

Next, we construct $\mathcal{B}_K^{next}$ from $\{\beta_i\}_{i=1\ldots n-1}$. Our construction becomes slightly subtle if methods are overridden; given κ_i and ω_i, we cannot simply use the corresponding β_i — rather, starting from κ_i, we must search up the class hierarchy until the definition of the method is found at some class; then the β_i defined at that class is the synchronization specification for ω_i. To be more precise, let $\mathit{SyncSpec}(\kappa_i)$ denote β_i that is defined at κ_i. We define an auxiliary function *slook*, which, given a class and a key, searches up the class hierarchy and returns β_i for the key defined at the class or the 'closest' ancestor class in the class hierarchy (Figure 4.10). By using *slook*, we can then construct $\mathcal{B}_K^{next}$ as:

$$\mathcal{B}_K^{next} = \lambda\kappa m\zeta.\mathit{slook}(\kappa)m\zeta \tag{4.2}$$

Function $\mathcal{B}_K^{next}$ constructed as above needs to be consistent with $\mathcal{B}^{next}$ for each class κ. When the consistency requirement is satisfied, we call each $\{\beta_i\}_{1,\ldots,n-1}$ the *class-specific accept set specification*:

[4]It does not matter whatever first-class status the accept set is given.

Definition 9 (Class-Specific Accept Set Specification) *Each function* $\{\beta_i\}_{i=1\ldots n-1}$ *such that* β_i : **Stat** $\rightarrow 2^{\mathbf{Key}}$ *is called the* class-specific accept set specification for class κ_i *if* $\mathcal{B}_K^{next}$ *constructed in the manner of equation (4.2) is consistent with* $\mathcal{B}^{next}$ *for each class* κ.

We also distinguish a special case of method addition for our proof: The situation is as follows: immediately after the invocation of a particular method defined at one of the parent classes, the message for the added method in the subclass can never be accepted. This is generalized in the following manner: we say that *method (key)* ω_i *prohibits method (key)* ω_j if the next accept set for ω_i never contains ω_j:

Definition 10 (Prohibition of a Method) *For class* κ, *let* $\omega_i, \omega_j \in vd(bec(\kappa))$. *Then* ω_i prohibits ω_j *in class* κ *iff* $\forall \zeta \in States(\kappa).(\omega_j \notin \mathcal{B}^{next}(\kappa)\omega_i\zeta)$

On defining a subclass, the synchronization specification for a ancestor method need not be re-defined if the ancestor method prohibits the added method. In practice, however, it is not often for a method to prohibit another method — methods that do are special in the sense that they are not affected by the state of the object prior to their invocation, and also tend to have an 'absolute' effect on the state of the object. In the bounded buffer example, we could define the method `clear()` which would clear the entire contents of the buffer; then `clear()` would prohibit `get()`, as the buffer would be in the empty state no matter what state it had been in prior to the invocation of `clear()`. Furthermore, for all methods in class κ, we can assume that there is at least one method in $vd(bec(\kappa))$ by which it is not prohibited. Otherwise, the method would be useless, as it would never become acceptable.

Now we are ready for our proof. First, let us examine the ideal case in which the anomaly does not occur. Consider the situation where $\mathcal{B}^{next}$ is consistent with $\mathcal{B}_K^{next}$ for all classes $\kappa_1, \ldots, \kappa_{n-1}$, and we are defining κ_n to be a subclass of κ_{n-1} by having a well-defined method $\omega_n \mapsto e_{\omega_n}$ at κ_n. By all means κ_n is now a WFC. Then, we would like to give a new program description of synchronization specification at κ_n with DKSS, so that none of the synchronization specification given in the superclasses need be modified. This means that every $\{\beta_i\}_{1\ldots n-1}$ remain unchanged, for any modification in the synchronization specification in the superclasses would reflect in the change of β_i's. At the same time, $\mathcal{B}_K^{next}$ constructed as in equation (4.2) must be consistent with $\mathcal{B}^{next}$ for all κ_i $(1 \leq i \leq n)$. Unfortunately, this is not possible, as we state and prove below.

Theorem 1 (Anomaly in Inheritance with DKSS) *Let* $\kappa_0, \kappa_1, \ldots, \kappa_{n-1}$ *be well-formed classes, where* $root(\kappa_0) = true$, *and* $par(\kappa_i) = \kappa_{i-1}$ *for* $1 \leq i \leq n-1$. *Let* $\omega_i \mapsto e_{\omega_i}$ *be the well-defined methods at class* κ_i. *Consider creating class* κ_n, *a subclass of* κ_{n-1}, *by defining a method* $\omega_n \mapsto e_{\omega_n}$ *which would be well-defined at class* κ_n. *Also, assume that:*

1. $\{\beta_i\}_{i=1\ldots n-1}$ *are class-specific accept set specifications, and*
2. β_n *is a characterization of synchronization specification given with next accept set variation of DKSS at class* κ_n.

Then, in order for $\mathcal{B}_K^{next}$ constructed as in equation (4.2) to maintain consistency with $\mathcal{B}^{next}$, every $\{\beta_i\}_{i=1...n-1}$ must be modified if the corresponding ω_i (i) is not overridden, and (ii) does not prohibit ω_n in class κ_n.

The outline of the proof is as follows: in the `b-buf` example, the next accept set `partial` corresponds to all states such that 0 < `out - in` < `size`. But when a new method `get2()` is added, set of states can be partitioned by the synchronization constraints for `get2()` in such a way that in one set, the new method is acceptable, and in the other, it is not. Correspondingly, the accept set must also be partitioned (`x-one` and `x-partial`). So, the program descriptions of synchronization specifications that referred to the accept sets in all parent methods need to be re-written (`get()`, `put()`). This is presented more formally below:

PROOF:
First, without the loss of generality[5] we can assume that $States(\kappa_i)$ are equivalent for all i. Now, we can partition $States(\kappa_n)$ into disjoint subsets $S_{\omega_n}, \overline{S_{\omega_n}}$ in such a way that $S_{\omega_n} \cup \overline{S_{\omega_n}} = States(\kappa_n)$, and ω_n is acceptable iff the state of the object is an element of S_{ω_n}, and vice-versa. More formally for all $\zeta \in States(\kappa_n)$,

$$\begin{cases} \zeta \in S_{\omega_n} & (\textbf{if } \mathcal{A}(\kappa_n)\omega_n\zeta = true) \\ \zeta \in \overline{S_{\omega_n}} & (\textbf{if } \mathcal{A}(\kappa_n)\omega_n\zeta = false) \end{cases}$$

For the parent class κ_{n-1}, consider ANY method $\omega_i \mapsto e_{\omega_i}$, where $1 \leq i \leq n-1$ and ω_i is not overridden, and ω_i does not prohibit ω_n in class κ_n. We can assume in general that for any state $\zeta \in States(\kappa_{n-1})$, the next accept set is not an empty set, i.e., $\exists b \subset vd(bec(\kappa_{n-1}))$ such that $b = \mathcal{B}^{next}(\kappa)\omega_i\zeta$ and $b \neq \emptyset$. This holds because if $b = \emptyset$, the object will no longer be able to accept any messages — and since state changes are only possible when an object accepts a message, such a state is a deadlock. (This is analogous to the agent bisimilar to **0** in CCS.)

Now, for ω_i, consider a state $\xi \in States(\kappa_n)$ where the next accept set contains ω_n; namely, $\omega_n \in \mathcal{B}^{next}(\kappa_n)\omega_i\xi$. We can easily show that such a state ξ is guaranteed to exist, as ω_n is not prohibited by ω_i in κ_n. Let b_Q denote the next accept set for ξ in the parent class κ_{n-1}, that is, $b_Q \stackrel{\text{def}}{=} \{\omega_{Q1}, \omega_{Q2}, \ldots, \omega_{Ql}\} = \mathcal{B}^{next}(\kappa_{n-1})\omega_i\xi = \mathcal{B}_K^{next}(\kappa_{n-1})\omega_i\xi$. There may be other states which also map to b_Q with $\mathcal{B}^{next}(\kappa_{n-1})\omega_i$. Let $Q \subset States(\kappa_{n-1})$ be a set of all such states; in other words, $Q = \{\zeta \mid \mathcal{B}^{next}(\kappa_{n-1})\omega_i\zeta = b_Q\}$. By the definition of $\mathcal{B}^{next}$ and the construction of $\mathcal{B}_K^{next}$, the corresponding $\beta_i(\zeta)$ defined for ω_i at κ_i must likewise be equal to b_Q for all $\zeta \in Q$ in order for $\mathcal{B}_K^{next}$ to maintain consistency with $\mathcal{B}^{next}$.

Next, partition Q into disjoint subsets $q, \overline{q}$ so that $q = Q \cap S_{\omega_n}$ and $\overline{q} = Q \cap \overline{S_{\omega_n}}$. Then, ω_n is acceptable only if $\zeta \in q$. So, by the definition of $\mathcal{B}^{next}$ in (4.1), we derive the following for each $\zeta \in Q$ (Figure 4.11):

$$\mathcal{B}^{next}(\kappa_n)\omega_i\zeta = \begin{cases} b_Q \cup \{\omega_n\} & (\zeta \in q) \\ b_Q & (\zeta \in \overline{q}) \end{cases} \tag{4.3}$$

[5]The proof for monotonically increasing domain would be essentially the same, albeit a little more complicated.

Here, we can guarantee that q is non-empty by the construction of Q (since ξ was chosen so that the next accept set would contain ω_n in the first place) and thus $\mathcal{B}_K^{next}$ must be modified so that it is consistent with $\mathcal{B}^{next}$ for class κ_n. But we can easily show from the construction of $\mathcal{B}_K^{next}$ in equation (4.2) that this requires modifications to β_i, so that $\beta_i(\zeta) = b_Q \cup \{\omega_n\}$ if $\zeta \in q$, and $\beta_i(\zeta) = b_Q$ if $\zeta \in \overline{q}$. Since this applies to any ω_i such that $1 \leq i \leq n-1$ as the selection of ω_i was arbitrary, every β_i must be modified, if both (i) and (ii) hold for ω_i, in order to maintain consistency.

As an additional note, careful readers might be concerned with the case $\overline{q} = \emptyset$; actually, this is the only case where indirect re-naming of accept sets by the previous proposals might work. Although we have deliberately omitted the treatment here for brevity, an analysis reveals that occurrence of such a case is just as rare as the prohibition of methods (every $\overline{q}$ must be $\emptyset$).

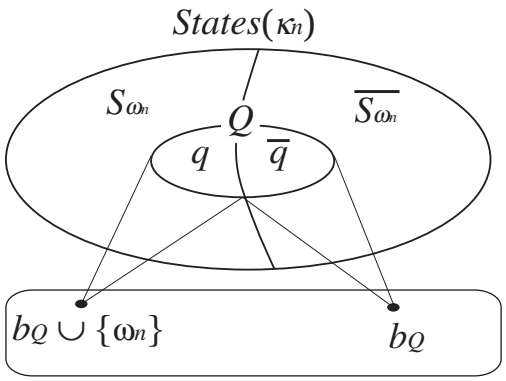


Figure 4.11: Proof of Anomaly in Inheritance

4.5.6 The Main Cause of Anomaly in Inheritance

What then, is the main cause of the anomaly? It is due to the properties of the class hierarchy with respect to accept sets. In DKSS, the accept sets are treated as first-class entities within the program description. Then, as we have seen in the proof, the synchronization specifications of the parent classes must be modified on creation of a new subclass. The only way to avoid this in DKSS is to allow reference to the method keys of the child classes in the synchronization specifications of the parent classes. But this is not allowed, as *one-way references* of method keys from child classes to their parents is one of the general properties of inheritance. If we were to allow inverse references from parents to their children to be predetermined, that would mean that a class *would already know* by which classes it would be inherited and would know the entire contents of the each. In such a situation, inheritance would be almost useless — once the class hierarchy is created, not only the hierarchy itself, but the methods contained therein cannot be modified.

It is easy to see that DPSS does not exhibit the anomaly of DKSS, since method keys are not (usually) treated as first-class entities within the predicates. As a result, the synchronization specification for a particular method is totally independent of the definitions of any other methods. But it cannot cope well with history-only sensitive behavior, as we have seen.

Chapter 5

Recent Proposals for Solutions to the Inheritance Anomaly

Recently, there has been much research that have proposed to minimize the effect of inheritance anomaly in OOCP languages, allowing inheritance of synchronization code in various situations. We review them in this chapter, discussing their advantages as well as their limitations.

5.1 Shibayama's Proposal

Shibayaba first proposed a scheme based on fine-grained inheritance of synchronization schemes , so that the amount of code that must be re-defined can be minimized. In the proposed extension of ABCL/1 to incorporate inheritance[98] , methods are categorized into *primary*, *constraint*, and *transition* methods. A method of one category may have its counterparts with identical method names in other categories, and each of them can be separately defined/inherited/overridden. The categorization of methods is as follows:

- A *primary method* is responsible for the task other than object-wise synchronization.
- A *constraint method* acts as a method guard. The difference from the guard in the previous chapters is that the guards can be re-defined independently of the primary methods; thus, only the constraint methods need to be overridden when the guards of the methods of the parent class must be changed (the corresponding primary methods are unaffected).
- A *transition method* determines how the messages are delegated among the objects. Its re-definition allows dynamic modification of the delegation path. By encapsulating the state transitions of the object in the delegated object, re-definitions involving history-sensitive state transitions can be localized.

By separating the synchronization code from other parts of method definitions, the amount of re-definitions is minimized. Shibayama also shows in [98] that locking behavior of an object can be treat with a modest amount of code re-definitions in the concurrent implementation of a 2-3 tree.

The limitations of Shibayama's proposal include: (1) there was no way to operate upon *sets* of methods as abstract synchronization states of objects, resulting in lack of encapsulation as well as other drawbacks such as cumbersome re-definitions when multiple methods are affected, (2) almost identical inheritance rules were applied to each category, which made code re-use awkward, and (3) no implementation schemes nor benchmarks were given—in particular, Shibayama's proposal requires a number of successive method delegations to describe state transitions, whose execution could be a considerable overhead.

5.2 Caromel's Proposal

Caromel's proposal[25], based on the Eiffel II language, in effect provides different synchronization schemes as a set of library routines implemented in terms of more lower-level primitives. For this purpose, methods, bodies, messages, message queues, and guards (dubbed respectively as routines, LIVE routines, requests, request lines, and blocking conditions, in their terminology) are manipulated as first-class entities in the synchronization code. A concurrent object, by inheriting from a class called `ABSTRACT_PROCESS`, attains the ability to dynamically associate synchronization code to individual methods. By overriding this association in the `synchronization` routine of the child classes, it is often possible to localize the effect of inheritance with respect to object-wise synchronization.

Caromel has shown that (1) first-classing of elements of synchronization schemes, and (2) adopting different synchronization schemes according to the requirements of different synchronization constraints, promote code re-use of concurrent objects.

The proposal, unfortunately, does not sufficiently resolve the inheritance anomaly in general. The limitations are that, once a synchronization scheme is adopted in the superclass by concrete instantiation of the abstract class, it is often difficult to override it in a consistent way to adopt a different synchronization scheme. This is in a sense breakage of encapsulation, because once a user 'commits' by defining concrete classes, their subclasses must assume such commitments in their programming, and in the worst case, the entire synchronization scheme must be re-programmed as we have seen for the 'body' anomaly. Another problem is the overhead of the execution synchronization code itself: since the synchronization scheme is not built-in as a primitive of the language, and no special optimizations are performed for integrating user-level code, execution of the user-defined code would be a substantial overhead. This cost was not an issue in [25], since Eiffel II seems to be intended for coarse grained concurrent computing in an distributed environment. But for achieving medium to fine-grain concurrency, the cost is definitely intolerable.

5.3 Frølund's Proposal

Frølund proposes a simple framework in which concentrates on re-use of synchronization code for the derived (i.e., overridden) methods[63]. The synchronization scheme is based on method guards, in which synchronization constraints get increasingly restrictive in

subclasses. Basically, one specifies a guard that gives the condition under which the method *cannot be accepted*, i.e. a *negative* guard. Furthermore, the guard expressions are accumulated along the inheritance chain for a given method, so that, given a method with the name `m`, all the guards for the methods in the ancestor classes with the name `m` and were thus overridden must evaluate to *false* in order for the message `m` to be accepted. Thus, the re-use only works in the way to restrict the conditions under which the messages are acceptable. Frølund points out that this is reasonable given that it should be possible for superclass operations to work on (all) subclass state, i.e., if an ancestor operation is not enabled in a particular state, then a derived operation with extended behavior will also incur inconsistency in that state. In addition, one could refer to other methods within the guard expressions; in this case, the method itself is not invoked, but instead, its guard(s) are evaluated and the resulting boolean value is returned. This is similar to the abovementioned proposal by Shibayama.

Although the proposal works to avoid the state partitioning anomaly because it is based on guards, and also allows re-use of guards to some degree, the scheme seems limited in practice:

- The paper only focuses on the re-use of guards, and avoidance of other anomalies is not given enough consideration. Re-use and associated anomalies regarding history sensitiveness (represented by `gb-buf`) is not mentioned at all. State modification is considered to some degree with the `Lock` example, but the solution is with ad-hoc construct that is non-extensible in the subclass: one can only specify the synchronization constraints that should hold uniformly for all methods to be defined in subclasses except for a single exception method with the `all-except(`*method-name*`)` construct. This will allow for a very simple `lock` case, where only one particular method (i.e., `lock`) needs to be excluded, with a declaration such as:

  ```
  (lock_var == 1) disables all-except(lock);
  ```

  However, is not possible to add a method which COULD be invoked under the same or more restrictive synchronization constraint as `lock`, because `all-except(lock)` prohibits any further extensions regarding the constraint on the `lock_var`, including the future definitions in the subclasses. For example, it is impossible to extend the class by adding a method `inquire-lock` which (1) inquires and returns the state of the lock, and (2) can be invoked irrespective of the condition of the lock itself.

- Another problem is that the scheme never seems to have been implemented. Although the proposed syntax for the separation of guards and their associated method is clean, it is not immediately obvious how one might implement it in an efficient way, because the guards and methods no longer have one-to-one correspondences.

Nevertheless, we feel that Frølund's work is noteworthy for pointing out issues of encapsulation and consistency in re-use.

5.4 Meseguer's Proposal

Meseguer proposes a new formalism[77] for modeling concurrent systems, and an OOCP language called Maude, which is based on this formalism. The language possesses the flexibility to provide cleans solutions for (some of the) anomalous examples we have presented in this paper[78].

Meseguer's formalism is a logic called the (concurrent) rewriting logic, which (Meseguer states that) most models of concurrent computation can be regarded as its special instantiations. A concurrent system is derived from (instantiations of) *modules*, that are composed of *terms* and rewrite rules. Computation proceeds by simultaneous simplification of terms when there are applicable rewrite rules. There are two types of modules, *functional* and *system*. The rewriting in system modules are not equational, i.e., does not exhibit the Church-Rosser property. This allows the modeling of phenomenon specific to concurrent computations, such as non-deterministic choice. The Maude language[77], based on this framework, provides *object-oriented modules* for ease of programming in concurrent object-oriented style. For actual execution, object-oriented modules are first translated into system modules; then, computation proceeds with concurrent rewriting according to the rewrite rules of the translated module. Inheritance is also supported in object-oriented modules directly with Maude's order-sorted type structures.

Inheritance anomaly is avoided in Maude in the following way[78]: the side conditions placed on the rewrite rules can serve as a guard; thus, state-partitioning anomaly does not occur. In addition, rewrite rules can be flexible, operating on the term structures as first class values. Thus, there is (albeit implicit) reflective capability in Maude, which allow history information to be encoded within the term structure of the class definition in a straightforward way. For example, it is simple to define a class which adds the locking capability to arbitrary classes. As an example, suppose we mix class `Lock` and class `A` to create class `Lockable-A`. The definition of the `Lock` class would "masquerade" the class identifier of `A` into a quoted one, `A'`. This quoted class can only accept the `unlock` message, and when it does, it restores the class identifier to `A`.

Although Maude does provides powerful language features to support several synchronization schemes, more work needed to be done to exploit the the extent of applicability of Maude to other classes of inheritance anomaly. One problem with proposal is that, because the solutions are based on the powerful pattern-matching capabilities of Maude, practical application of the proposal could be inefficient as a result in a fine-grain setting. Unless some good implementation scheme is devised, a single message send could take at the least several hundreds of instructions.

5.5 Ishikawa's Proposal

Ishikawa proposed a communication mechanism between concurrent objects that allows re-use of communication protocols more complex than the standard client-server protocol[53]. The proposal is similar to those based on enable sets; each class maintains a set of *method set*, a set of bindings between the method name and the method body itself. The programmer specifies which method set the method definition will belong to,

allowing multiple definition of methods with the same name belonging to different method sets. Each object dynamically maintains a *visible set*, which dictates which methods are acceptable to the object. The user specifies operations on the visible set `include` and `exclude` with a method set name as an argument. In addition, *intensive sets* of an object dictates which methods could be subject to immediate (unqueued) execution.

The proposal has significance in pointing out that first class operations on method sets could be one important feature the language must support in order to allow re-use of synchronization code; however, it too has its limitations. Although the original intent of the proposed mechanism was not on solving the inheritance anomaly in the first place, Ishikawa nevertheless presents a solution to the `Lock` anomaly problem in the paper. However, the solution seems to be quite limited, as it works only for very simple locks, and not for more complex locks such as the `WRITE-LOCK` example which incorporate a hierarchy of locks, as is described in Section 7.3. As a matter of a fact, by our understanding the presented solution is not correct in a sense that the locking interferes with the synchronization behavior of the buffer—according to his definition, the visible set of an object, which holds the methods that are acceptable, is substituted with the `scope` construct in the combined subclass of `Buffer` and `Lock` without any regard to the previous state of the object. In other words, whatever information the object had on the synchronization information that corresponds to the object state prior to locking, is thrown away. This presents a serious problem: for example, if the buffer was full prior to locking, the visible set holding the information is thrown away because the after method of `unlock` in class `LockBuffer` replaces the visible set of the object to be the initial one of the object, namely, `#unlocked` and `#free` by the use of the construct `scope #visible` (See [53], Example 5.2 for details of the original solution).

Other problems with the proposal is that (1) the state partitioning anomaly easily occurs for the same reason as accept sets; secondly, (2) it seems that the mechanisms have never been implemented in practice; the basic mechanisms for the `import/export` seems quite inefficient to implement compared to simple guards or enable sets, with no immediate obvious means of optimizations.

5.6 Summary of Previous Proposals and Their Limitations

The previous proposals reviewed here have identified to some degree that (1) separation of synchronization codes to localize the changes in the synchronization code, and (2) first-class construction of synchronization schemes (possibly with reflective capability of a language), could potentially provide the necessary flexibility to keep code re-definitions to the minimum.

However, when one considers practice, we feel that they are not satisfactory for the following reasons:

- We have seen in the previous chapters that no single synchronization scheme would serve as a panacea for all the cases of inheriting synchronization codes for satisfying the new synchronization constraints of the subclass; in particular, accept sets and

guards each have their advantages as well as drawbacks. Instead, our premise is that the user should be able to choose whatever synchronization scheme deemed appropriate for particular synchronization constraints. With objects, the emphasis would then be on orthogonality/encapsulation property, that is, the principle that not only the synchronization code but also the synchronization scheme should be encapsulated, e.g., the scheme employed in the parent class is an implementation detail *that should not be exposed to the subclasses*. The proposals, however, restrict the user to a single synchronization scheme, thus limiting the expressiveness, and/or has not addressed the encapsulation issue at all. As a result, re-use that could lead to inheritance anomalies, which could otherwise be avoided using different synchronization schemes, become difficult.

- A related problem is that none of the schemes have properly considered avoiding the necessity of recompilation of parent methods, or encapsulation of their source code. In practice, it is strongly desirable to be able to separately compile the subclasses with a restricted set of static information exported from the superclasses. This is both due to (1) speed (avoiding long recompilation delays) and (2) propriety issues (many commercial class libraries comprising an application framework does not come with a source code). Although some of the simpler proposals in the previous chapters could satisfy this requirement, the issue nevertheless has not been properly considered.

- Some proposals allow inheritance of synchronization code separately from main method bodies. The manner the synchronization code of the superclass is referenced, however, is done in a syntactically and conceptually different way compared to standard method inheritance. Thus, users are faced with two different inheritance systems, possibly with no obvious conceptual model of how they interact.

- The final and the most important issue is that, for all proposals, efficiency issues of the implementation of their schemes have been given little consideration, if implementation were considered at all. Implementation of efficient concurrent OO-languages is not at all trivial— in our recent work in implementing language ABCL on various architectures, developing software technologies for efficient implementation has been one of the primary focus of our research: for example on ABCLonAP1000[107], we have trimmed down the local *asynchronous* message sending overhead down to about 20 SPARC instructions in the best cases, nearly matching the (procedure calling) method invocation time of C++. Remote asynchronous invocation requires less than 10 μseconds, competing favorably with architectures that facilitate special message-passing hardware. On analyzing the previous solutions, none considers whether their implementation could be done with a comparable efficiency; in fact, the overhead for some of them could easily be hundreds of instructions per each method invocation, which would prohibit their usage in practical situations.

Chapter 6

Our Proposed Solutions to Inheritance Anomaly

During the course of our research, we have proposed several solutions to minimize redefinitions and promote code re-use. Although the early ones have been totally superseded by our new proposal outlined in Section 6.3, they have nevertheless been precious steps worthy of introduction here.

6.1 Early Proposal 1 — First-Class Guards with a Reflective Architecture

Our first proposal was based on reflection and first-classing of method guards[71]. It facilitated guarded methods, since it avoids the state-partitioning anomaly. In addition, we attempted to minimize the anomaly for history-only sensitive cases by granting first-class status to a set of method guards. A OOCP language X0/R with a reflective architecture as shown in Figure 6.1 achieves this purpose. The *metaobject* of x, denoted as x`.meta`, is a meta-level representation of the structure and computation of x. Here, the metaobject is itself a (concurrent) object — this is basically the *individual-based* model of reflection as proposed in [117]. Given x, we can manipulate the guards as a first class-object via x`.meta`. Figure 6.2 illustrates how the `Lock` mixin class can be programmed with this strategy. (Note that '*object* `<--` *message*' denotes a message send.)

Although further application of X0/R to other solutions were not investigated, the idea of 'replacing' the set of guards dynamically was valuable[1].

6.2 Early Proposal 2 — Eliminating Synchronization Code Syntactically

In our prototype OOCP language HARMONY[115], we took another approach of *eliminating* the need for synchronization code with automated concurrency control feature of

[1]Some other ideas of X0/R, such as *dynamic progression of degree of reflectivity*[71], was employed in the actual efficient implementation of our OOC-Reflective language, ABCL/R2[67].

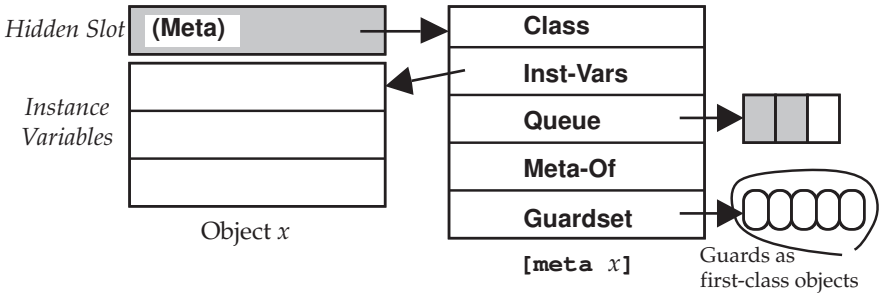


Figure 6.1: Object and its Metaobject in X0/R

```
class lock {
    Guard_set saved_guards; //  saved set of first-class guards
  methods:
    // replace the guards of the metaobject so that only unlock can be accepted.
    void lock() {
      saved_guards = self.meta <-- get_all_guards();
      self.meta <-- set_all_guards(FALSE_GUARDS);
      self.meta <-- set_guard("unlock", TRUE);
    }
    // not invoked until the guard is set to TRUE with :lock
    void unlock() when FALSE {
      self.meta <-- set_all_guards(saved_guards);
      // The guard for unlock is reset to  FALSE as a result.
    }
}
```

Figure 6.2: Definition of `Lock` Class with X0/R

the system, instead of inventing various new synchronization schemes which might cause yet another anomaly. Transactions supplemented with method guards are the basic synchronization scheme of objects, of which guards are usually only necessary for essential methods, because inter-object synchronization for maintaining integrity is now implicit in the transaction facility of the system. With such a strategy, inheritance anomaly is much less likely to occur, since there is little requirement for the methods whose synchronization code cause the anomaly. With bounded buffer, for example, only the essential (guarded) `put` and `get` methods are necessary; compound methods known to cause anomalies, such as `get2` which removes two items from the buffer atomically, are no longer necessary in the first place. This is because with Harmony one can perform successive `get`s to the buffer with guarantee of atomicity without any programming of synchronization code such as locking. The details are found in a [115].

Application of HARMONY to fine-grain computing, however, was difficult due to exten-

sive overhead of distributed transaction management among the objects. Nevertheless, the idea of making the synchronization scheme implicit, was valuable.

6.3 Our New Proposal

We have recently proposed a set of language OOCP languages primitives that provide high degree of efficient and encapsulated re-use of synchronization code[73]. The primitives are being incorporated into the new version of ABCL we are currently designing/implementing. The proposal is designed with high practicality in mind—it extends and extensively refines the ideas in the past proposals to (1) separate and localize the synchronization schemes from the main bodies of methods, allowing fine-grained inheritance/overriding, and to (2) allow dynamic operations on the methods themselves, in order to control which messages are acceptable by an object. Furthermore, it has the following novel and favorable characteristics:

- Our proposal allows both guard-based and accept-set based synchronization schemes to coexist and be integrated, so that the best scheme can be chosen to program given synchronization constraints.
- The manner we re-use the synchronization code is syntactically similar to superclass method references in sequential OO-languages (e.g.,`super`). Thus, users with experience in OO-programming can readily adapt to our proposal. Inheritance rules are made to depend on each synchronization scheme, however, because the most 'natural' way of inheritance differs among the schemes.
- We offer a high degree of encapsulation and re-use for synchronization code. Furthermore, even synchronization schemes could be encapsulated in superclasses in many cases by proper exporting of class information by the user.
- Expressiveness is not our sole concern—we have also devised speed- and space-efficient incorporation into the software architecture of the aforementioned ABCL/onAP1000. In particular, all space/time-consuming data structure construction for object synchronization can be done at compile-time.

6.3.1 Overview of Execution Model of Concurrent Objects

The execution model of our concurrent object is an extension of ABCM[127]. A rough overview is as follows: an object sends messages asynchronously, either *past type* ("asynchronously send and no-wait", syntactically denoted by *Receiver* `<-` *Msg*) or *now type* ("asynchronously send and wait for a reply", syntactically denoted by *Receiver* `<--` *Msg*). The transmission order of messages between two objects are preserved (*transmission order preservation law*). The now-type is similar to procedure call, except that (1) the receiver can continue its execution even after it has returned a reply message and (2) the *reply destination* (reply message box) of a now-type message is a first-class object, which can be passed around as message arguments.

Upon message reception, an object executes messages in a mutually exclusive manner. An object is *dormant* if it is not processing a message, and *active* if it is. All the messages received during active mode are placed in its message queue. When in dormant mode, the object scans its message queue from its head, and accepts the first message that satisfies its synchronization specification, which is the combination of *method sets*, *synchronizers*(extended form of guarded methods), and *transition specifications*(state transition directives of *accept-sets*): First, the object evaluates its set of *synchronizers* to determine, within its current *accept set* (i.e., the set of currently acceptable methods for the object) which messages are acceptable according to its current state. The object scans the message queue for an acceptable message, and executes the corresponding method. After its execution is completed, the object evaluates the *transition specification* associated with the method, and alters its accept set with the method sets specified in the transition.

Synchronization schemes specify operations on the accept sets with *method sets* to designate the set of methods to be enabled/disabled. Method sets are inherited and re-defined in subclasses. The programmer can choose and combine the synchronization schemes in the synchronization code to best express his synchronization constraints (i.e., with various forms of synchronizers and transition *types*). Altogether, the set of synchronization code is localized within the *synchronization specification* of a given class. The constituent primitives of synchronization specification can be individually inherited and re-used in a 'fine-grain' manner, using a similar syntax to standard method inheritance but with separate inheritance rules for each synchronization scheme. The main part of method that do not contain synchronization code (i.e., the method body) is totally separated from the synchronization specification, and are inherited and overridden separately from synchronization code.

6.3.2 Method Sets

A *method set* is a set of methods (identifiers) bound with the corresponding method bodies. It is worthy to note in the outset that method sets are designed NOT to be full first class objects, in that they cannot be assigned to variables and such, and are only subject to restricted run-time extensions. This allows the *Virtual Function Table (VFT)* (the table that holds pointers to the compiled methods code according to their method identifiers) to be determined at compile time for efficient execution and low storage space.

Primitive method set constructor has the form `#{`*method name*`,...}`. There are also some primitive set operations, such as '`|`'(union). Other than within the method definitions, the method sets in the program must be assigned to explicit identifiers to be referenced, disallowing arbitrary run-time set operations. The basic definition form is:

```
mset name #{method name, ...};
```

Example 11 Consider the `b-buf` class with the two methods, `put` and `get`:

```
class b-buf {
method_sets:
    mset    EMPTY    #{put}         // only 'put' to empty buffer
    mset    FULL     #{get}         // only 'get' from full buffer
    mset    PARTIAL  EMPTY | FULL   // both possible otherwise
        :
}
```

A *method qualifier* is a limited form of qualifier expression to denote a constructor for a set of methods. The restriction on the qualifier is that the resulting set must be computable at compile time. Currently, the supported form is as follows:

- `all`—all the methods of the class including the inherited methods,
- `defined`—the defined methods of the class, excluding the inherited ones, and
- `all_except(`*method set,...*`)`—all the methods of the class except the ones of the specified method sets.

Example 12 The following constructs a pair of mutually exclusive method set: `LOCKED` is a singleton set containing `unlock`, while `UNLOCKED` contains all the methods defined at the class, except `unlock`.

```
class Lock {
method_sets:
  mset    LOCKED        #{unlock}
  mset    UNLOCKED      all-except(LOCKED)
        :
}
```

Method set definitions can be re-defined in subclasses. If the overridden definitions of the superclass needs to be referenced, the subclasses may refer to them using the `super` operator followed by the method set identifier[2] (see Example 13). All the method sets are recomputed in the subclass to account for re-definitions, so that any changes are propagated to the derived method sets.

One special case is as follows: when the name of a method is not syntactically manifest in any of the method set definitions, then the method is called a *synchronization free* method. Such a method is added implicitly and uniformly to all the method sets of the class. This is to allow non-constrained methods to be freely invokable by default. When

[2]In practice, matters are more complicated due to multiple inheritance.

the method is later explicitly used in a construction of a method set in a subclass, this implicit addition is nullified for that subclass and its siblings.

Also, the method qualifier is (re)computed when one defines a new method in the subclass: for example, if new methods are added in a subclass of `Lock`, they are added to the `UNLOCKED` method set for that class, because `UNLOCKED` is defined as `all-except(LOCKED)`.

Example 13 Consider defining a class `x-buf` as a subclass of `b-buf` of Example 11 by adding the following two methods: (1) `last`, which removes the last element that was `put`, and (2) `empty?` which checks whether the buffer is empty or not. Since the synchronization constraint for `last` is identical to that of `get`, it is added to the method set where `get` was a member, namely, `FULL`. Method `empty?` is a synchronization-free method, and is automatically added to all the method sets. Furthermore, the method sets that were derived from `FULL`, i.e., `PARTIAL`, are also automatically updated in `x-buf` as well.

```
class x-buf: b-buf{
method_sets:
    mset    FULL     super FULL | #{last}
    // PARTIAL  is automatically redefined.
        :
}
```

Altogether, method set `PARTIAL` in `x-buf` is `#{put, get, last, empty?}`.

A method set can be statically bound to a guard expression, so that a method set in the superclass can be partitioned into multiple method sets, depending on runtime object states. Such *guarded method set* is introduced to cope with the state partitioning anomaly, allowing transitions to have (full) flexibility of guards. Below is the syntax:

`mset` *name* `#{`*method name*`, ...}` `when` *guard expression*`;`

Note, however, that we do not allow arbitrary run-time set operations on the method sets; the purpose of the guarded method set definitions is to maintain the encapsulation property without method sets being full first-class entities for efficiency reasons.

Example 14 The following definition of `PARTIAL` allows partitioning of the state in the subclass:

```
class x-buf2: b-buf {
method_sets:
    mset    PARTIAL     super PARTIAL           when (size == 1)
    mset    PARTIAL     super PARTIAL | #{get2} when (size > 1)
        :
}
```

6.3.3 Synchronizers

A *synchronizer* is a combination of *guard expressions*, *enabling specifier*, and a list of method sets. In essence, it is similar to a guarded method, but is more flexible in that a single guard can be assigned to multiple methods in method sets.

A guard expression is basically a side-effect free boolean expression involving instance variables and method arguments. Local variables of individual methods are not allowed in the guards to maintain encapsulation. The named method argument must exist in all the methods associated with the guard (both for synchronizers and transitions) or compile-time error will result. The guard expression also can contain *acceptance inquiry* function `enabled()` and `disabled()`, which takes either a method name or a method set as an argument. The guard expressions can be assigned symbolic names with `guard` definitions, so that they can be re-used and possibly redefined in subclasses.

`guard` *name* *guard expression*

The synchronizer thus becomes:

`synchronizers:`
guard `enables` *method set name* `,` ...`;`

A special keyword `initially` can be specified in place of a guard in order to indicate which method set is enabled initially upon object creation. Also, a synchronizer can be chosen NOT to be inherited for methods in a given method set with `override` *method set*.

Example 15 Synchronization specification for `b-buf` can be made with synchronizers in the following way:

```
guards:
    guard  empty_guard   (size == 0)
    guard  full_guard    (size == MAX_SIZE)
    guard  partial_guard (!empty_guard && !full_guard)
synchronizers:
    initially      enables EMPTY;
    partial_guard  enables PARTIAL;
    empty_guard    enables EMPTY;
    full_guard     enables FULL;
        :
```

6.3.4 Transition Specifications

Transition specification can be used as an alternative synchronization scheme to synchronizers. A *transition* corresponding to a method is executed immediately after the completion of the method body. Its purpose is to specify the transitional behavior of an object's accept set, that reflects the synchronization constraint dictated by the internal state of the object. The transitions are specified on a method-by-method basis, via *transition specifications* for the method. Each transition of an object is associated with a *transition type*, which designates the effect of the method set upon the current accept set, and an optional *guard*, which governs the condition under which the transition is executed. The basic syntax of transition specification for a given method is as follows:

```
transitions:
  transition method name1() {
    transition-type-1  method-set {when guard-expression 1};
                :
    transition-type-n  method-set {when guard-expression n};
  }
  transition method name2() {
                :
```

Each line of the transition specification is simply called a *transition*. The guard expression of each transition is evaluated sequentially from transitions 1 to n, and the transition of the first guard to evaluate to `true` is executed exclusively. This not only automates the disambiguation of multiple possible transitions, but also allows for finer control of inherited transitions.

The currently available *transition types* are as follows: `become`, `push`, `enable`, `disable`, `restore`, `wait_once`, `enable_once`, `disable_once`, and `is`. Below is their brief description:

1. `Become`: Replaces the accept set entirely.

2. `Push:` Replaces the current accept set, but also 'pushes' it so that it can be restored with a subsequent `restore`.

3. `Enable:` Enables the methods in the method set in addition to the ones in the current accept set. (Effectively, the new accept set is the set union of the old accept set and the specified method set).

4. `Disable:` A complement to `enable`, disables the methods that are elements of the argument method set and the current accept set (effectively, the set difference is taken).

5. `Restore:` Restores the method set to the one prior to performing `push`, `enable`, or `disable`.

6. `Wait_once:` The current accept set is 'pushed' and replaced as is with `push`, but is subsequently restored with a forced `restore` transition in the next accepted message, just prior to execution of the real transition. This effectively allows handshaking-type protocol to be programmed easily, as is with the `wait-for` construct of ABCL/1.

7. `Enable_once`, `disable_once:` A combination of `enable` or `disable` with automatic `restore`, as is with `wait_once`.

8. `Is:` A special-purpose keyword to inherit the transitions of the parent class.

The transition specifications of the parent class are inherited, but overriding them can be done in a more sophisticated manner compared to overriding of method sets and synchronizers:

- Each class can have its *default transition specification*, indicated by a special keyword `default`. If the method does not have any transition specification in its class or its superclasses, the default transition specification is used *if one is defined.* The default transition specification can also be overridden along the inheritance chain.

- The transition `is self` *method-name* refers to the entire transition specification of that method, allowing sharing among the methods. The expression `super` *method-name* is the similar, except that the search for the corresponding reference starts from the immediate superclass as is with Smalltalk-80 (with disambiguation rules for multiple inheritance).

- There is an automatic inheritance rule to relieve the programmer from explicit declaration of inheritance of transition specifications: when one defines a transition for a method, if there are no lines with the `is` specification, an implicit `super` *method-name* is assumed to exist as the *last* transition in the specification, effectively inheriting the entire transition specification of the superclass for the method. (In order to prohibit this automatic inheritance, the last line of the transition specification can made into a special form `override`.)

- The guard expression can be substituted for `otherwise` to be the *otherwise transition*. This transition is selected when there are no guards that become true. When there are multiple otherwise transitions, the one in the most specific subclass supersedes the others in the superclasses.

Note that, combined with the ordered evaluation rule of the transitions, the above inheritance mechanism of transition specifications allows the programmer the freedom to define transitions that either precede or succeed the superclass transitions with arbitrary placement of the `super`. Default transitions can be referenced by `self` and `super` as well, so that individual methods can customized upon the default transitions of its superclass.

Example 16 Here is an alternative definition of `b-buf` using transitions.

```
transitions:
transition get() {
    become EMPTY when (size == 0);
    become FULL  when (size == BUFSIZE);
    become PARTIAL otherwise;
}
transition put() {
    is get();  // transition of put the same as get
}
        :
```

6.4 Inheriting Synchronization Code in our New Proposal

When the programmer creates a new subclass, he (re)defines the new methods, and also (re)defines the method sets, synchronizers, and transitions to satisfy the new synchronization constraint, re-using much of the synchronization code of the superclasses. This is achieved in principle in a syntactically similar way as the re-use of normal methods in sequential object-oriented languages: syntactic references to method sets and other values in the superclasses are modified to be those of the inherited classes when they are re-defined, and superclass definitions can be referenced with `self` and `super`. However, inheritance rules are customized for each synchronization scheme according to their characteristics, as we have seen in the descriptions. The required updating of synchronization code is encapsulated within the synchronization specification of the class, i.e., the method sets, synchronizers and transition specifications. The main bodies of the methods in the superclasses, by contrast, are unaffected in the subclasses, avoiding the inheritance anomaly and achieving encapsulation.

In general, inheritance anomaly is avoided in the following way: state portioning anomaly does not occur when using synchronizers. For transition specifications, there are two ways of expressing the state partitioning in the subclasses: first is to augment the sets of transitions with the additional partitioning required. By appropriate placement of `super`, only the states that are partitioned for the new methods in the subclass need to be modified, and majority of the parent transitions are re-used automatically by the implicit inheritance rule of transition specifications. Second is to employ method sets bound with (dynamically computed) guards. By describing the new partition of the method sets with the guards, the transitions in the superclasses can be refined with re-definitions only for the relevant method sets. For state modifications such as `lock`, synchronizers can be re-defined, or alternatively, transitions can also be used effectively to 'switch' the method sets according to the state. We do not employ (expensive) higher-order term structure encodings and expensive pattern matchings as in Maude[78], but still provide comparable descriptive power.

Furthermore, as we exemplify in the next chapter, encapsulation is possible not only

for the synchronization code of the superclass, but also possible to for the *synchronization scheme* of the superclass in many cases, if the user exports the appropriate information of the synchronization specification to the subclass; therefore, by careful design, the user can maintain the freedom of selecting the appropriate synchronization scheme irrespective of the scheme employed in the superclasses.

Chapter 7

Examples of Avoiding Inheritance Anomaly

The inheritance anomaly examples are now programmed using our proposal. We show that (1) the synchronization code is encapsulated in the synchronization specifications and does not manifest in the main body of the methods, and (2) separate inheritance (rules) for method sets, synchronizers, and transitions allow fine-grain re-use of superclass synchronization code, keeping re-definitions very small. Furthermore (3) synchronization schemes are also encapsulated; separate solutions for the same problem are programmed using either synchronizers or transitions (except `gget`). We also emphasize that synchronization scheme of the superclasses are also encapsulated, i.e., the solutions would work irrespective of the choice of the synchronization scheme in the original `b-buf` first presented below:

- **Bounded Buffer with Synchronizers:**

```
Class b-buf {
    int size = in = out = 0; int item[MAX_SIZE];
  method_sets:
    mset    EMPTY    #{put}         // only 'put' to empty buffer
    mset    FULL     #{get}         // only 'get' from full buffer
    mset    PARTIAL  EMPTY | FULL   // both possible otherwise
  synchronizers:
    initially enables EMPTY;
    (0 < size && size < MAX_SIZE) enables PARTIAL;
    (size == 0) enables EMPTY;
    (size == MAX_SIZE) enables FULL;
  methods:
    void put(int item){
       size--;
       out = (out + 1) % max_size;
       return item[out]; }
    int get(){
       size++; in = (in + 1) % max_size;
       item[in] = x; }
}
```

- **Bounded Buffer with Transition Specifications:**

```
Class b-buf {
    int size = in = out = 0; int item[MAX_SIZE];
  method_sets:
    mset    EMPTY    #{put}         // only 'put' to empty buffer
    mset    FULL     #{get}         // only 'get' from full buffer
    mset    PARTIAL  EMPTY | FULL   // both possible otherwise
  methods:
    void put(int item){
       size--;
       out = (out + 1) % max_size;
       return item[out]; }
    int get(){
       size++; in = (in + 1) % max_size;
       item[in] = x; }
  transitions:
    transition default {  //  the default transition specification
        become EMPTY when (size == 0);
        become FULL  when (size == BUFSIZE);
        become PARTIAL otherwise;
    }
}
```

7.1 State Partitioning Anomaly—Method `get2`:

The method obtains two elements atomically from the buffer. State partitioning in the subclass is trivially satisfied with guards. With accept sets, there are two possible solutions: One is to augment the transition specifications of each method to realize the partition. The other is to use method sets bound with guards—method sets are 'refined' with guards to dynamically add `get2` as an element depending on the internal state of the object.

- **Solution with Synchronizers:**

```
Class x-buf2: b-buf { // x-buf2 is a subclass of b-buf
//  Optional re-definitions of method sets,
//    necessary if the future subclasses are to use transitions.
  method_sets:
    mset    FULL    super FULL | #{get2}
    mset    PARTIAL super PARTIAL | #{get2}
    mset    ONE     super PARTIAL
  synchronizers:
    (size > 1) enables get2
  methods:
    int get2(){ // code to operate and return two elements; }
}
```

- **Solution with Transition Specifications (1):**

```
Class x-buf2: b-buf { // x-buf2 is a subclass of b-buf
  method_sets:
    mset    FULL    super FULL | #{get2}
    mset    PARTIAL super PARTIAL | #{get2}
    mset    ONE     super PARTIAL
  methods:
    int get2(){ // code to operate and return two elements; }
  transitions:
    transition default { // account for new state partitioning
        become ONE when (size == 1);
        // implicit 'is super default()'
    }
}
```

- **Solution with Transition Specifications (2):**

```
Class x-buf2: b-buf { // x-buf2 is a subclass of b-buf
  method_sets:
    mset    FULL    super FULL               when (size == 1)
    mset    FULL    super FULL | #{get2}     when (size > 1)
    mset    PARTIAL super PARTIAL            when (size == 1)
    mset    PARTIAL super PARTIAL | #{get2}  when (size > 1)
    mset    ONE     super PARTIAL
  methods:
    int get2(){ // code to operate and return two elements; }
  // Transitions need not be re-defined
}
```

7.2 History-Only Sensitiveness—Method `gget` in `class gb-buf`:

Although not all history sensitiveness can be resolved with our scheme, cases where only the previous message affects the accept set can be handled gracefully with the family of `_once` transitions. Here, we present a solution to the `gget` method in `gb-buf`; notice that the solution works irrespective of whether `b-buf` is specified with synchronizers or transitions. A more elaborate inter-object protocol can be designed by the combined use of both synchronizers and transitions.

- **Solution with Transition Specifications:**

```
class gb-buf: b-buf {
  method_sets:
    mset    AFTER-PUT     #{gget};
    mset    FULL          super FULL | AFTER-PUT
  methods:
    // gget identical to get except for synchronization constraint
    int gget() { return super get(); }
  transitions:
    transition put() {
        disable_once AFTER-PUT;  // Only once immediately after put
    }
    // gget automatically handled by the default: transition,
    //    if b-buf were specified with transitions.
}
```

7.3 State Modification Anomaly—Methods `Lock / Write-Lock / Unlock`:

The `write-lock` class defines a two-level lock where the method `lock` locks the object exclusively so that no other methods can access it until it receives a corresponding `unlock` message, whereas the `write-lock` message allows side-effect free methods to be invoked. `Write-lock` is to be used as a mixin, so that the invokable methods under write-locked state can be extended in the subclasses. It is also straightforward to refine the locks by constructing a hierarchy of locks.

- **Solution with Synchronizers:**

```
typedef lock_state = {unlocked_lock,write_locked_lock,locked_lock};
class write-lock {
    lock_state lock_var = unlocked_lock;
    lock_stack lstatck = new(lock_stack);
  method_sets:
    mset    LOCKED        #{unlock}
    mset    WRITE-LOCKED  self LOCKED  // redefined in subclass
    mset    UNLOCKED      all-except(LOCKED)
  synchronizers:
    (lock_var == unlocked_lock) enables UNLOCKED;
    (lock_var == write_locked_lock) enables WRITE-LOCKED;
    (lock_var == locked_lock) enables LOCKED;
  methods:
    void lock() {
       lstack.push(lock_var);
       lock_var = locked_lock;}
    void write-lock() {
       lstack.push(lock_var);
       lock_var = write_locked_lock;}
    void unlock() {
       lock_var = lstack.pop;}
}
```

- **Solution with Transition Specifications:**

```
typedef lock_state = {unlocked_lock,write_locked_lock,locked_lock};
class write-lock {
    lock_state lock_var = unlocked_lock;
    lock_stack lstatck = new(lock_stack);
  method_sets:
    mset    LOCKED        #{unlock}
    mset    WRITE-LOCKED  self LOCKED  // redefined in subclass
    mset    UNLOCKED      all-except(LOCKED)
  methods:
    // Locking can be handled entirely with transitions,
    //  but we define the methods to maintain encapsulation
    void lock() {
       lstack.push(lock_var);
       lock_var = locked_lock;}
    void write-lock() {
       lstack.push(lock_var);
       lock_var = write_locked_lock}
    void unlock() {
       lock_var = lstack.pop;}
  transitions:
    transition lock() { push LOCKED; }
    transition write-lock() { push WRITE-LOCKED; }
    transition unlock() { restore; }
}
```

- We give two examples of the use of `lock`: One is to add a method to the `lock` class itself which inquires the status of the lock, which is not possible with Frølund's proposal[63]. Notice how the inheritance of method sets allows easy re-use of existing lock code.

```
class lock2: lock {
  method_sets:
    mset    ALWAYS        #{inquire-lock}  // Always invokable.
    mset    LOCKED        super LOCKED | ALWAYS
    // WRITE-LOCK is automatically updated
    mset    UNLOCKED      super UNLOCKED | ALWAYS
  methods:
    lock_state inquire-lock() { return lock_var; }
}
```

As an example of use of `write-lock` as a mix-in, we define a class `lb-buf`, which is a bounded buffer that allows locking; in particular, if the buffer is write-locked, then only the `empty?` method can be invoked. Further extensions to `lb-buf` is possible by augmenting the method set `READ-ONLY`. We note that this type of encapsulated extensibility is much more cumbersome with other proposed schemes:

```
class lb-buf: b-buf,write-lock {
  method_sets:
    mset    READ-ONLY     #{empty?}  // Can be extended in subclasses
    mset    WRITE-LOCKED  super WRITE-LOCKED | READ-ONLY
  // No other definitions are necessary
  // Note that empty? is in both UNLOCKED and WRITE-LOCKED
}
```

7.4 Combining Synchronization Schemes

Recall that a message is accepted as a combined effect on the accept set by the synchronizers and transitions: for instance, after the accept set has been determined by the transitions, the guard in the synchronizer may further restrict the messages that could be received. This is illustrated in the following stream-communication example:

Example 17 *Here, is a two-phase handshaking protocol for stream communication: the receiver waits for a* `stream_open` *message from an anonymous sender; when it receives the message, it sends an acknowledgement, and waits for subsequent* `stream_put` *messages only from the same sender—messages from other senders are not accepted. When it receives the* `stream_close` *message from the sender, it returns to the initial state.*

```
class stream-receiver {
    Object opener = NULL;

  method_sets:
    mset  STREAM_CLOSED   #{stream_open}
    mset  STREAM_OPEN     #{stream_put, stream_close}

  synchronizers:
    initially enables CLOSED;
    // Stream_put and stream_close received
    //   only if it matches the opener
    (message_sender == opener) enables STREAM_OPEN;

  methods:
    int stream_open(Object message_sender) {
      opener = message_sender;  // subsequent messages from opener
      opener <- ack(true);  // sends acknowledgement to sender
    }
    int stream_put(Object message_sender, int data) {
      // consume data
      message_sender <- ack(true);   // sends acknowledgement to sender
    }
      int stream_close(Object message_sender) {
      message_sender <- ack(true);   // sends acknowledgement to sender
      opener = NULL; // close stream
    }

  transitions:
    transition stream_open() {
      become STREAM_OPEN;
    }
    transition stream_close() {
      become STREAM_CLOSED;
    }
}
```

Chapter 8

Efficient Implementation Architecture for our Proposal

As stressed earlier, it is very important that the synchronization scheme must be implementable in a highly efficient manner, if we expect applicability to true practice. Our language proposal is carefully crafted not to sacrifice run-time execution efficiency, despite its expressive power. The run-time architecture of the implementation is a non-trivial extension to the highly-efficient run-time architecture of ABCLonAP1000[107]. We first briefly overview the novel architectural characteristics of ABCLonAP1000, and then describe how the extension that support our proposal can be seamlessly integrated with it.

8.1 Overview of ABCLonAP1000

Most work in high performance concurrent OOPLs have focused on combination of elaborate hardware and highly-tuned, specially tailored software[49, 121] to drastically improve upon the two key factors in achieving high-performance—efficient inter-node message passing and efficient intra-node multithreading—allowing 'message passing' speed to approach those of sequential OO languages. We have shown with our work ABCL/onAP1000 that, even without such special hardware support, we can achieve comparable performance on conventional machines such as AP1000 and CM-5 which lack special hardware supports such as message send instructions or hardware scheduling queues, via the use of a suitable *software* architecture (compiler + runtime environment)[107].

The key software technologies in our implementation of ABCL/onAP1000 include the followings:

Integration of Stack-Based and Queue-Based Scheduling: Employing the standard queue-based multiprocessing suffers from the high overhead of queue manipulation. Our scheduling mechanism avoids queue manipulation in many cases by employing efficient stack-based scheduling as much as possible.

Multiple Virtual Function Table: Our computational model assumes that objects synchronize on message reception by processing messages under mutual exclusion,

and enqueing any messages received during processing. Furthermore, transmission order of messages must be preserved. We reduce the run-time checking overhead of such concurrent OOPLs with a scheme employing *multiple* Virtual Function Tables (VFT), integrating the the runtime checking of message acceptance (whether the object is dormant or not) into standard method look-up procedure through a VFT.

Active Message-based Inter-node Message Sending: The standard remote communication on multicomputers is to 'wrap' a message with various tags, and a fixed message handler of the receiver node interprets the tags one by one, causing significant overhead. We reduce this overhead by employing the *Active Messages*[114] concept, which by only specifying the message handler address specific to each message, eliminates the tags and the associated overhead caused by their interpretation. Our compiler crafts the methods so that the active message first invokes the prolog that unbundles the remainder of the message solely based on the static type information of the message.

Other key technologies include low latency remote object creation via object-address prefetching/fault function tables, integrated long-range load-balancing/distributed garbage collection, etc. The actual implementation of ABCL/onAP1000 is undergoing on Fujitsu's multicomputer AP1000 that consists of 512 SPARC nodes interconnected with a 25MB/s torus network. The preliminary measurements of asynchronous message passing between two objects on different nodes is 9 μseconds in the best case, comparing favorably with fine-grain machines. Furthermore, the minimal cost of asynchronous method invocation of local objects is 2.3 μseconds, including locality checking, dynamic method dispatching, and scheduling.

8.2 Overview of Efficient Asynchronous Message Passing

In our implementation, representation of an object is similar to that of C++, with the following additional slots:

A Message Queue: a list of arrived, unprocessed messages.

Temporary Box (Saved Context) List: Execution Contexts (i.e., temporary variables and continuation address) of objects are saved into *temporary boxes* when the object is blocked, e.g., when reply of the now-type (RPC-style) message has not arrived. (Note that an object is not blocked when it is dormant.)

Virtual Function Table Pointer (VFTP): points to *multiple* VFTs. Basically, a method is compiled and its pointer is placed in the virtual function table as is with C++. Other virtual table entries are created for object synchronization.

As mentioned above, the key idea is the extended use of the virtual function table: each class has *multiple* virtual function tables, each of which roughly corresponds to

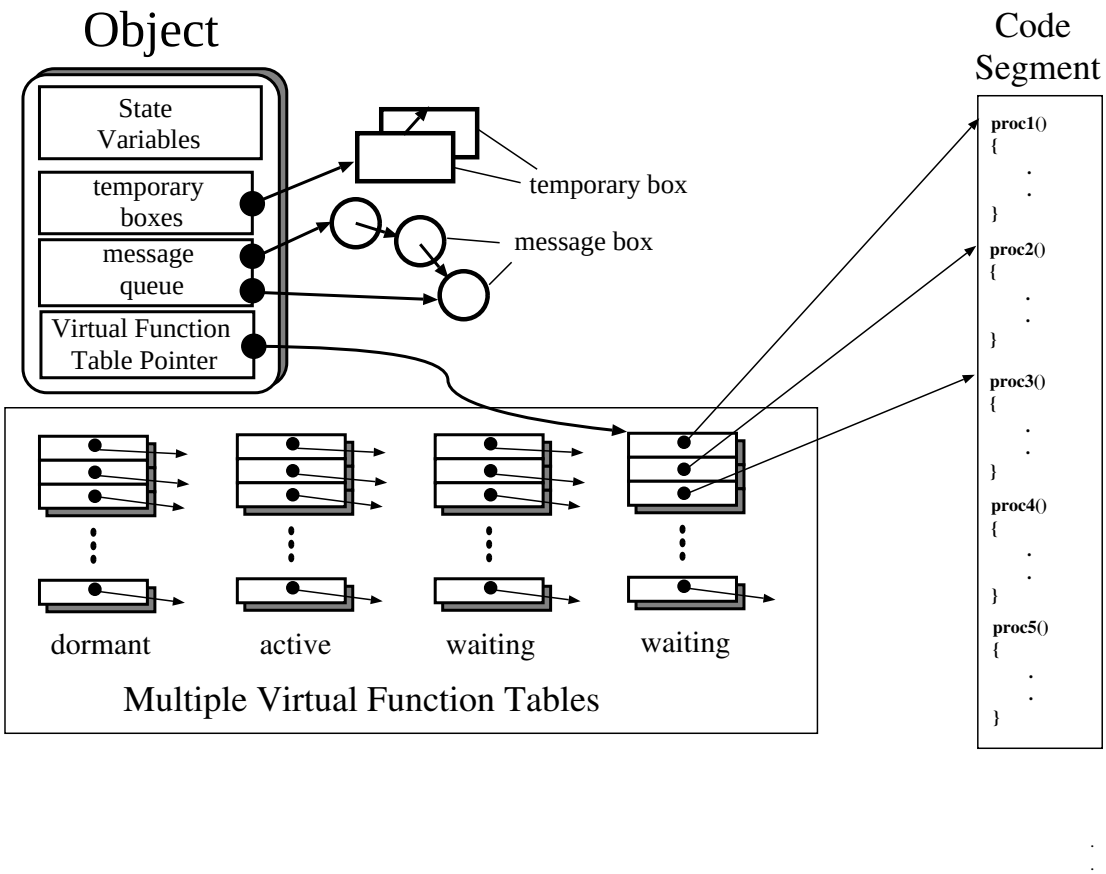


Figure 8.1: Structure of Concurrent Objects

the mode of an object. The object can only receive messages when in dormant mode, and any messages received in active mode i.e., when the object is processing a message, must be enqueued in the message queue. After the locality check, which takes 3∼4 SPARC instructions, we simply look-up the virtual function table with the statically-determined index number of the message and call the indexed procedure with the message as arguments. If the object is in dormant mode, the procedure will be the method body itself, which is executed immediately. Otherwise, if the object is in active mode, the procedure will be a tiny *queuing procedure (QP)*, which stores the message into a heap-allocated message box from the stack, and returns execution to the sender. Each queueing procedure is customized according to the type of the corresponding message. We thereby eliminate the runtime checking of whether or not the receiver object is dormant, by incorporating it into the virtual function table look-up, which is already a necessary cost in OOPL[1]. Local message passing takes approximately 25 SPARC instructions in the unoptimized case, and can be reduced to 8 without inlining in the best cases. (As a comparison, virtual function invocation in C++ takes 8-10 instructions.)

8.3 Implementation of Synchronizers

The implementation of synchronizers become more complex compared to ordinary guarded methods, since multiple synchronizers can enable different method sets. More precisely, a set of guards determined by a set of synchronizers serve as disjunctive invocation conditions for the methods who are elements of the intersection of the method sets. For example, for `BoundedBuffer`, the synchronizer definitions:

```
synchronizers:
    (size == 0) enables EMPTY;
    (0 < size && size < MAX_SIZE) enables PARTIAL;
```

dictates that for `#{put}` (which is equal to `EMPTY` $\cap$ `PARTIAL`), it is invokable if (`size` $==$ `0`) $\vee$ (`0` $<$ `size` $\&\&$ `size` $<$ `MAX_SIZE`) (which in this case is equivalent to (`size<MAX_SIZE`).

Our implementation strategy seamlessly and efficiently integrates the synchronizers with message passing mechanism developed for ABCLonAP1000. The entire set of synchronizer definitions are collected and ordered along the superclass chain, with synchronizers in the subclass having precedence over those in the superclasses in the ordering. An indexable table, called the *Guard Table (GT)* is initialized and assigned to each invokable method. For each synchronizer definitions, a boolean function of its guard is compiled, and its pointer is registered in the Guard Tables of the methods it enables. After all the synchronizers have been processed, the entries in the dormant mode VFT are modified in the following way: For each method whose guard table is not empty, the VFT entry is replaced with a *Guard Procedure (GP)* which (1) evaluates the guards in the Guard Table one by one, then (2) when a guard evaluates to true, then it invokes the original

[1] Additionally, we utilize the VFTP for other optimization purposes, such as lazy object initialization and fast remote creation of object: see [107] for details.

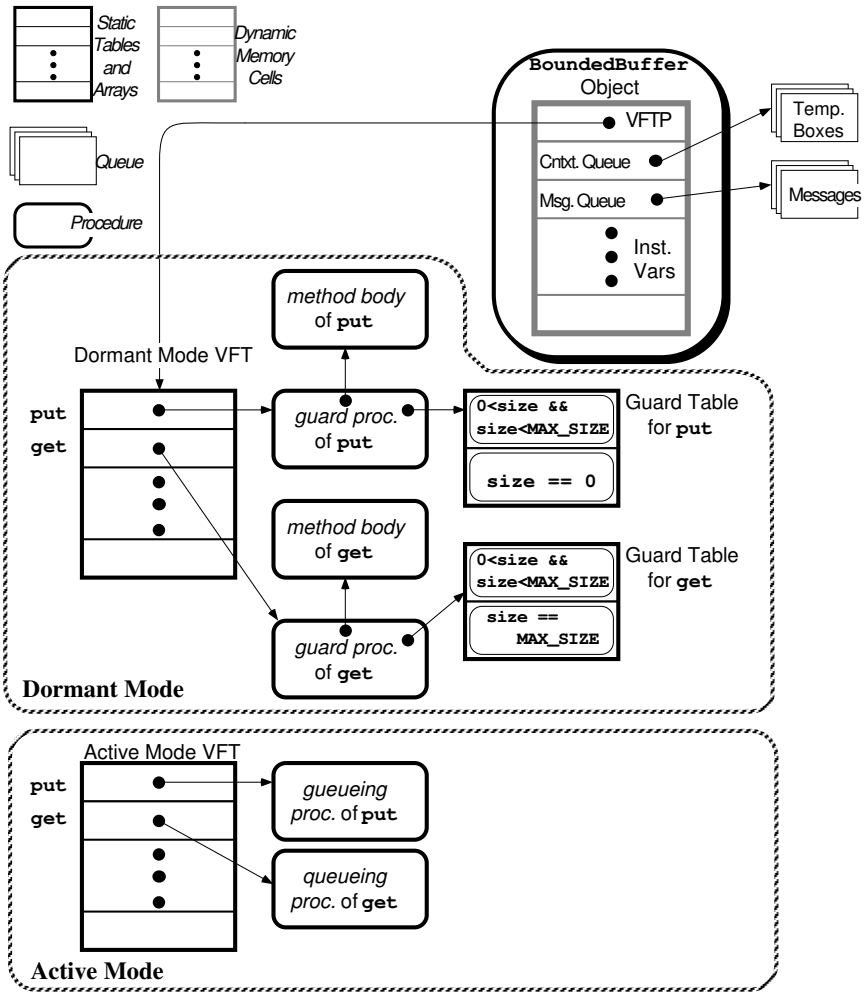


Figure 8.2: Synchronizer Implementation for Bounded Buffers

procedure for the method body (*Method body Procedure, or MP*), else (3) when all the guards fail, then it acts as a queuing procedure. Figure 8.2 illustrates the implementation for the bounded buffer object—notice that (1) the guard procedure and the guard table are customized for each method of the object, and (2) The pointer references (the arrows in the figure) that originate from compile-time entities only point to other compile-time entities; in particular, VFT entries only point to compiled procedures.

Seamless integration of the mechanism cannot be done naively— there are several subtleties to considered, and we have made enhancements and modifications to the original execution mechanism of ABCLonAP1000, while preserving the execution efficiency. They are as follows:

- *Context Preservation of Guard Procedures*— The evaluation of guard procedure is more subtle compared to straightforward MP execution of ABCLonAP1000: since the sender object only knows the argument type information of the receiver object,

and does not know what procedure it is invoking at run-time, the local calling context of the procedure would be that of the method body procedure, under which the guards must be evaluated, and furthermore, *MP* must also be executed under the same context. Thus, we must avoiding the expensive replication of the arguments, for a single method corresponds to multiple set of guards. One approach is to inline all the guards in the guard table and also the *MP* itself into *GP*, but this would cause expensive replication of compiled method code in OO-languages due to inheritance, unless robust dynamic compilation and caching scheme were exploited as in SELF[28]; furthermore, such an approach would conflict with our goal of separate compilation. Instead, we take an approach where (1) the guards are inlined (since they are static and relatively small), and (2) *MP* is jumped into, carrying over the called context of the *GP* into *MP*. Here, *GP* need not refer to the source code of *MP*, in particular, the local variables need not be allocated on the stack in *GP*, thanks to the requirement that the guards be side-effect free.

- *Integration of Synchronization-free Methods*— For synchronization-free methods that were added implicitly to all the method sets, the guard table is not constructed in the implementation; rather, the default method body procedure is directly referenced by the VFT as is with the original ABCLonAP1000.

- *Dormant Object Method Invocation*— Another major difference with the original ABCLonAP1000 architecture is the invariant on the message queue and object execution state: On ABCLonAP1000, the message queue of a dormant object is empty; for our new proposed language, a dormant object could have messages that was queued in step (3) of the above procedure. Still, the above mechanism safely invokes the next incoming message properly, without wasteful checking of messages in the queue known to be unacceptable, because the state of the object will not change for a dormant object until another method is executed.

- *Active Object/Non-acceptable Method Invocation*— A message may be queued either because (a) the object is active, or (b) all guards failed. Such queued messages are invoked via global queue scheduling: When another (later incoming) method completes, the efficient stack-based scheduling mechanism inherited from ABCLonAP1000 performs the following post-processing: the object checks its message queue for pending messages, and if there are, it places itself in the global scheduling queue, and returns to the caller. Later, the object and its method is scheduled from the global queue. We have shown that this scheme makes most use of the efficiency and locality of inherent processor scheduling mechanism (i.e., the processor stack), without sacrificing the necessary weak-fairness property in concurrent processing[107]. The following changes must be made, however:

  - (A) The entries in the scheduling queue— since the guard procedure must be (re-)invoked for messages that were enqueued due to guard(s) failure, the entry is made a pair <*object, non-queueing VFT*>, instead of the <*object, continuation address*> of the original ABCLonAP1000. *Non-queueing VFT* is a special version of the dormant VFT (see next item).

– (B) Scanning of the message queue— due to the presence of guards, the message queue must be scanned for an acceptable message, without re-enqueing. This is achieved by having another version of the dormant VFT called the *non-queueing VFT*, whose entries are Guard Procedures that do not (re-)enqueue the message. The message queue is scanned, invoking the corresponding methods via the non-queuing VFT. When the execution returns, the global scheduler checks if the invocation was to a failed guard procedure: if so, the scanning continues, until the end of the message queue is reached.

8.4 Example of Synchronizer Execution

Let us see how the presented scheme consistently and efficiently preserves the synchronization constraint of an object, as well as the required properties of the language, e.g., transmission order of messages. In other words, we show that our message passing/scheduling mechanism preserves the order of message reception when they are acceptable.

Example 18 *Suppose object B is the* `BoundedBuffer` *object with buffer size 1, and object A executes the following when B is empty (*`size == 0`*) and dormant:*

```
int x,y;
B <- put(1);       Message-1
B <- put(2);       Message-2
x = B <-- get();   Message-3 (A waits for and B returns a value)
y = B <-- get();   Message-4 (A waits for and B returns a value)
```

By the transmission order preservation law, messages are transmitted to B *asynchronously* in the order sent; thus, the computational model dictates that (1) Message-1 is accepted and fills the buffer with value 1, (2) Message-2 is enqueued because the buffer is full, but object A is not blocked, (3) Message-3 is accepted and returns 1 to A, at which point Message 2 becomes acceptable and fills the buffer with value 2, and (4) finally, Message-4 is accepted and returns 2 to A. As a result, `x` $= 1$ and `y` $= 2$ must hold after the completion of the above sequence. Let us see how this semantics is efficiently realized in our implementation for the case of local message send (Figures 8.3 and 8.4):

Step 1: `B <- put(1)` is executed at A. Since B is dormant and (`size == 0`), the first guard in the Guard Table for `put` evaluates to true; thus, the body method of `put` is invoked, filling the buffer with `1`. Execution returns to A, and B becomes dormant again.

Step 2: `B <- put(2)` is executed at A. Now that B is full (`size == MAX_SIZE`), all the guards for `put` become false and thereby Message-2(`put(2)`) is enqueued into the message queue by copying from the stack. Execution returns to A, and B becomes dormant again.

Step 3: `x = B <-- get()` is executed at A. Since B is full, a guard for `get` evaluates to true and its method body procedure is executed, obtaining value `1` from the

buffer. Prior to returning, B discovers a pending Message-2 in its message queue; by the virtue of our scheduling scheme, B changes its VFT to active, places itself in the global scheduling queue to ensure fair scheduling, and execution returns to A via normal procedure return, which assigns `1` to `x`.

Step 4: `y = B <-- get()` is executed at A. Since B is active, Message-3 is enqueued in the message queue of B by a queueing procedure. Then, since A expects a reply but cannot get one, it will suspend itself with by saving the continuation that expects a reply from B from the stack. Notice that this continuation is associated with the enqueued message, and is NOT placed in the scheduling queue.

Step 5: After other entries of the scheduling queue are executed, eventually, B is scheduled, and the first element of the message queue, Message-2(`put(2)`), invokes its guard procedure. This time, the buffer is empty, and method body procedure of `put` is invoked storing `2` into the buffer. Upon return B again finds pending Message-4(`get`) in its message queue, so again B places itself in the scheduling queue.

Step 6: Eventually, B is scheduled again, and Message-4 is invoked with success. Upon return, its VFT is restored to dormant. Furthermore, the method discovers that A has been suspended with a continuation waiting for a value; as a result, the continuation is invoked, which returns control to A and thereby assigns `2` to `y`.

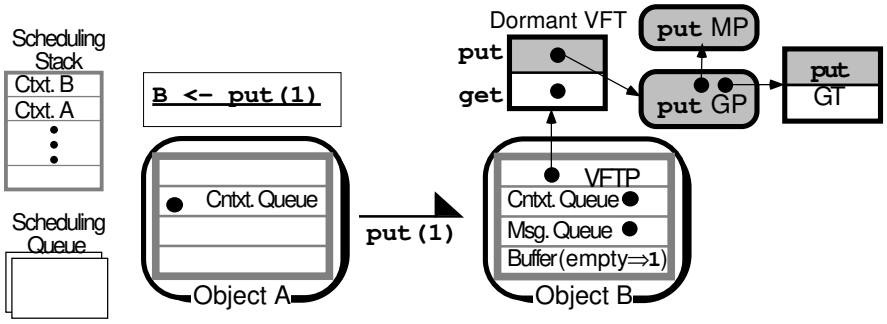
(Implementation of remote messages also preserves the semantics, but it would involve description of the scheduler invoked by active messages, but the space prohibits us from describing the detailed mechanisms.)

Compared to Frølund's proposal, let us briefly mention the advantage of 'enabling' the method instead of 'disabling' it with guards. Since disabling is done all along the inheritance chain, the enabling condition would be a *conjunctive* condition, i.e., the dual must become false, whose form would be the negated *conjunct* of all the guards associated with the method. As a result, the only general way to determine if a method is invocable is to evaluate the entire set of guards that is associated with the method (Conversely, if a particular guard evaluates to false, the message is queued.) This is the dual of our situation, in which only one guard needs to be true in order to accept the message (i.e., the condition is *disjunct*).

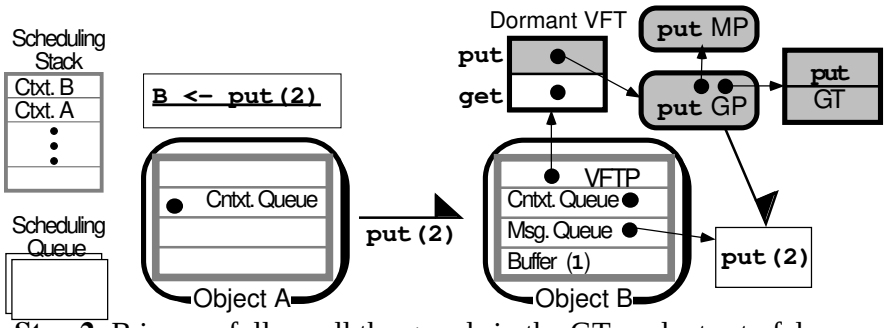
The question then is, which is more advantageous in general? Here is an observation: If one considers an object to be a critical region, then concentration of messages to a single object would certainly cause a bottleneck and thus loss of parallelism; as a result, efficient parallel algorithms would usually avoid such situation (i.e., messages being queued) as much as possible in their programming. Thus, it would be better to optimize for the non-queued case, which is in fact what we do by 'enabling' the methods in the method set.

8.5 Implementation of Transition Specification

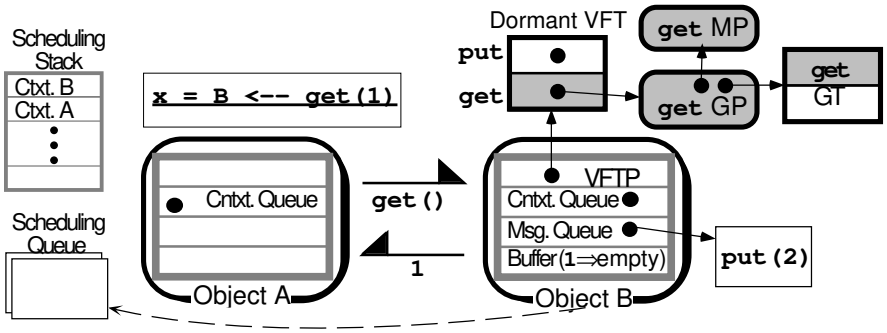
The transitions might seemingly require first-class operations on the method sets, resulting in excess run-time overhead: however, that is not the case. In order to support such



Step 1: Since B is dormant and empty, the first guard in the GT for **put** evaluates to true, invoking the MP of **put**, filling the Buffer with **1**.

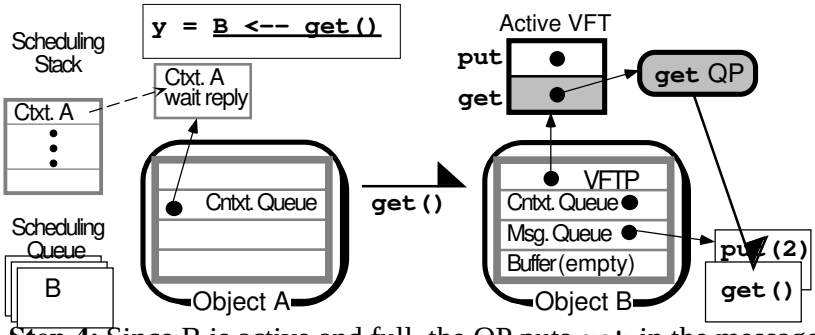


Step 2: B is now full, so all the guards in the GT evaluates to false; thus GP enqueues the message **put(2)** into the message queue of B.

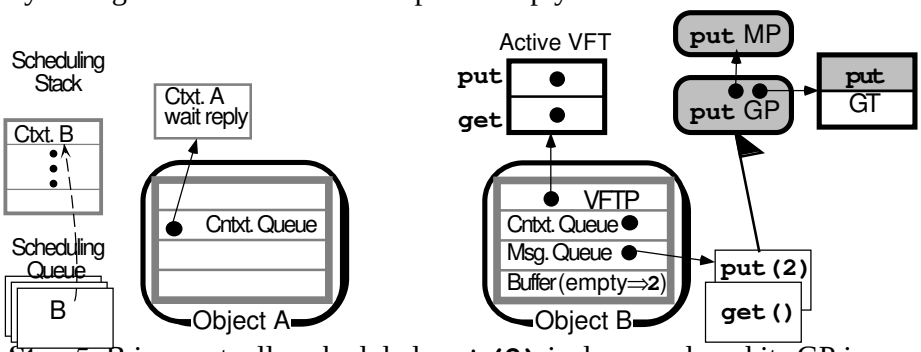


Step 3: Since B is dormant and full, the MP of **get** is executed, returning **1** to A. Prior to returning, finding **put(2)** in its message queue, it sets its VFT to active and puts itself in the scheduling queue.

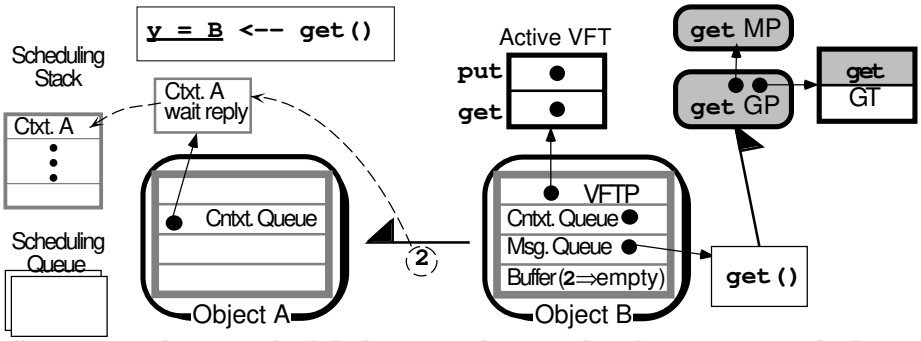
Figure 8.3: Example of Synchronization Control via Synchronizers (1)



Step 4: Since B is active and full, the QP puts **get** in the message queue. Then, since A cannot immediately get a reply, A suspends itself by saving its continuation that expects a reply from B.



Step 5: B is eventually scheduled, **put(2)** is dequeued, and its GP is invoked. Since the buffer is empty, **2** is stored, and upon return since **get** is in the message queue, B enqueus itself again into the scheduling queue.



Step 6: B is later rescheduled, **get** is dequeued and its GP is invoked, at which VFT is restored to dormant and **2** is returned. The continuation of A waiting for this value is invoked, assigning **2** to **y**.

Figure 8.4: Example of Synchronization Control via Synchronizers (2)

dynamic properties, we have devised a following runtime structure: *transition stacks*. Again, the implementation is integrated with the ABCLonAP1000 message passing implementation and the synchronizer implementation above, so the programmer is free to "mix and match" the synchronization schemes, and the overhead for the invocation of a method that need not synchronize are kept to the minimum.

First, the compiler looks for the transition specifications for each method in the class definitions, and expands away the inheritance. Then, for each transition type, it generates customized virtual tables that realizes its intended semantics. The entries in the generated VFT are categorized as follows:

- *Method Body Procedure/Guard Procedure (MGP)*: Either the Method Body Procedure if there are no synchronizer definition for the method, or the Guard Procedure, if there are. MGPs are employed as entries for methods that do not have the associated transition specifications.

- *Method Transition Procedure (TP)*: a 'wrapper' of *MGP* for methods with transitions. In order to execute both the *TP* itself and the wrapped *MGP* under the same call context without duplicating the arguments, here is what we do: *TP* first transfers control directly by jumping to the *MGP* of the corresponding method, except that the return address to the sender of the message on the stack is saved and modified so that the control returns to the point of jump in *TP* upon execution of return sequence on *MGP*, instead of to the original sender. Then, *TP* 'restores' the execution context of *MGP* by simple stack/frame pointer manipulation, then executes the compiled code of the transition under the restored called context, then returns to the saved address. (In practice, matters are more complicated when arguments are passed via registers.)

- *Queuing Procedure (QP)*: The standard queuing procedure.

- *Pass-through procedure (PP)*: Transfers control to the corresponding slot of the previous VFT in the transition stack (see below).

Some transition types replace the VFT for the active mode with its own; others 'mask' the current VFT; in this case, the MGP and QP entries are directly invoked from the mask VFT, whereas the PP entries invoke the corresponding entries in the 'masked' VFT. This is achieved efficiently using a *transition stack* as illustrated in Figure 8.5. When an object employs transitions, an extra level of indirection is introduced in the VFTP reference: VFTP first points to an entry in the transition stack of the object, from which the VFTs are referenced as shown in the figure. Transitions that 'mask' the previous VFTs are pushed onto the transition stack, with VFTP serving as the stack pointer. The size of the stack is kept relatively small, (a few entries will suffice), because each element in the stack could introduce an extra cost of one procedure invocation and two pointer references for PP entries.

For each transition type (except `is`), the followings describe: (1) The VFT that is generated at compile time for the designated method set, (2) the operations that are performed on the transition stack upon executing the transition, and (3) the resulting effect:

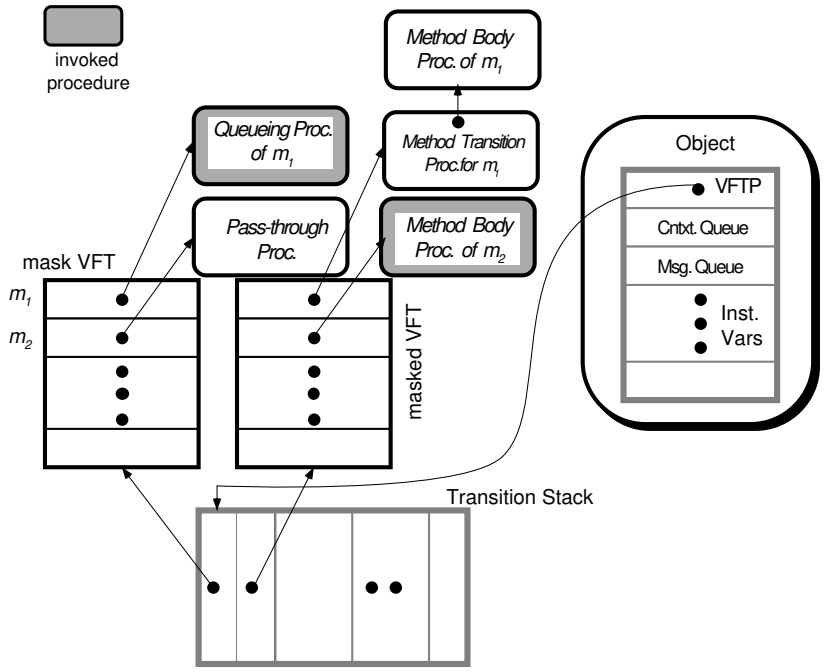


Figure 8.5: Implementation of Transition Specification

- `Become`:

  1. **Constructed VFT:** Method-set elements $\Rightarrow$ *MGP* or *TP*; Non-Element $\Rightarrow$ *QP*.
  2. **Operation:** The top entry of the stack is replaced with the constructed VFT.
  3. **Effect:** The messages with methods that are members of the method set are invoked; other methods are queued.

- `Push`:

  1. **Constructed VFT:** Method-set elements $\Rightarrow$ *MGP* or *TP*; Non-Element $\Rightarrow$ *QP*.
  2. **Operation:** The constructed VFT is 'pushed' onto the transition stack.
  3. **Effect:** Same as `become`, except that the old accept set can be restored with a subsequent `restore` transition.

- `Enable`:

  1. **Constructed VFT:** Method-set elements $\Rightarrow$ *MGP* or *TP*; Non-Element $\Rightarrow$ *PP*.
  2. **Operation:** The constructed VFT is 'pushed' onto the transition stack.

3. **Effect:** The members of the method set can be invoked, while non-member methods depend on the state of the masked VFT; the net accept set is the 'union' of the old accept set and the designated method set.

- `Disable:`

  1. **Constructed VFT:** Method-set elements $\Rightarrow$ *QP*; Non-Element $\Rightarrow$ *PP*.
  2. **Operation:** The constructed VFT is 'pushed' onto the transition stack.
  3. **Effect:** The members of the argument method are (temporarily) disabled until subsequent transition, while non-members are unaffected.

- `Restore:`

  1. **Constructed VFT:** none
  2. **Operation:** The topmost element of the transition stack is 'popped' off.
  3. **Effect:** The previous accept set prior to `push` or `enable` is restored.

- `Wait_once:`

  1. **Constructed VFT:** Method-set elements $\Rightarrow$ *TP* with unconditional `restore` transition (including methods without transition specifications); Non-Element $\Rightarrow$ *QP*.
  2. **Operation:** The constructed VFT is 'pushed' onto the transition stack.
  3. **Effect:** Same as `push` for the duration of the next accepted message. Upon completion of the next message, the subsequent unconditional `restore` transition of *TP* restores the previous accept set, realizing the intended semantics. (Note that queued messages do not invoke the *TP*.)

- `Enable_once:`

  1. **Constructed VFT:** Method-set elements $\Rightarrow$ *TP* with unconditional `restore` transition (including methods without transition specifications); Non-Element $\Rightarrow$ *PP*.
  2. **Operation:** The constructed VFT is 'pushed' onto the transition stack.
  3. **Effect:** Same as `enable` for the duration of the next accepted message. Upon completion of the next message, the subsequent unconditional `restore` transition of *TP* restores the previous accept set, realizing the intended semantics. (Note that queued messages do not invoke the *TP*.)

- `Disable_once:`

  1. **Constructed VFT:** Method-set elements $\Rightarrow$ *QP*; Non-Element $\Rightarrow$ customized *TP* that does not invoke the *MGP*, but rather, (1) acts as a *PP* but expects a return, and (2) with unconditional `restore` transition (including methods without transition specifications);

2. **Operation:** The constructed VFT is 'pushed' onto the transition stack.
3. **Effect:** Same as `disable` for the duration of the next accepted message. Upon completion of the next message, the subsequent unconditional `restore` transition of *TP* restores the previous accept set, realizing the intended semantics. (Note that queued messages do not invoke the *TP*.)

As is with synchronizers, for each VFT constructed, a corresponding non-queueing VFT is constructed. The difference in the entry is the *QP*, which returns indicating immediate guard failure.

When combined with synchronizers, a subtle problem arises when the object becomes dormant from active: when only the synchronizer is present, the compiler can statically determine the address of the dormant VFT since there is only one <dormant VFT, non-queueing VFT > pair. But, when synchronizers are used in combination with transition specifications, the current VFT address must be saved when the object becomes active, and later restored when it becomes dormant again, because of the dynamics of the transition stack. Fortunately, only one VFT address need to be saved, because the object never executes another transition before its VFTP is restored. An extra field could be allocated within an object, but a simpler alternative is to save the data above the topmost element on the transition stack, since it will never be overwritten.

For method sets bound with guards, multiple versions of the method set and the associated VFTs corresponding to each bound guard is constructed at compile time. The guard table similar to the synchronizer is also generated to the compiler per each method. Then, upon installing the method set onto the transition stack, the guards are evaluated (in the context of the invoked method) in order to select the appropriate method set.

8.6 Example of Transition Execution

As an example, consider the behavior of the `lb-buf` in Chapter 7. Assume that the Synchronizer was used in the synchronization specification of `b-buf`, and Transitions were used in the synchronization specification of `write-lock`:

Example 19 *Suppose object B is the* `lb-buf` *object with buffer size 1, and object A executes the following when B is empty and dormant:*

```
int x,y;
B <- write_lock;     Message-1 // Write-locks buffer
B <- put(2);         Message-2 // Not accepted because of write-lock
x = B <-- empty?()   Message-3 // Returns 1
B <- unlock;         Message-4 // Unlocks buffer, put accepted.
y = B <-- get();     Message-5 // Should return 2
```

Again, the computational model dictates that (1) Message-2 'write-locks' the buffer so that only the message in the method set `WRITE-LOCKED`—`unlock` and `empty?`—can be accepted, thus (2) Message-2 is enqueued but object *A* is not blocked, (3) Message-3 accepted and returns true (`1`) to *A*, (4) Message-4 is accepted and 'unlocks' the buffer,

at which point Message-2(`put`) becomes acceptable, and fills the buffer with value `2`, and finally, (5) Message-5 is accepted and returns 2, which is assigned to `y` in A. As a result, `x` = 1 and `y` = 2 must hold after the completion of the above sequence. Let us again examine the behavior of our efficient implementation scheme (Figures 8.6 and 8.7):

Step 1: `B <- write_lock` is executed at A. The constructed VFT for `lock` is:

- entry for `unlock` $\Rightarrow$ *TP* for `unlock` (the `restore` transition),
- entry for `empty?` $\Rightarrow$ *MP* for `empty?`,
- other entries $\Rightarrow$ *QP*.

The `write_lock` method invoked at B has a null *MP*, so the `push` transition in its *TP* is immediately executed, which 'pushes' this VFT onto the transition stack.

Step 2: `B <- put(2)` is executed at A. This invokes the queueing procedure, which enqueues it(Message-2) into the message queue of B. Execution returns to A, and B becomes dormant again.

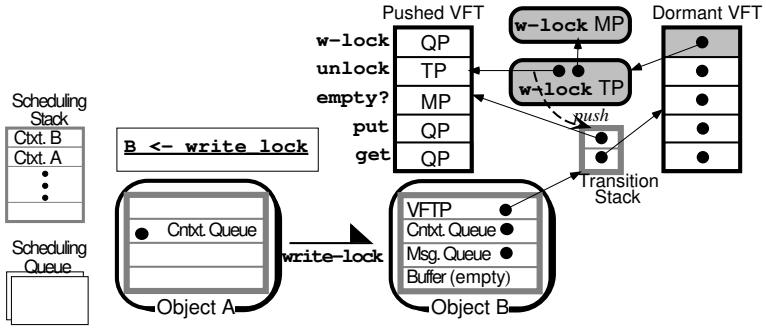
Step 3: `y = B <-- empty?()` is executed at A. This invokes the *MP* of `empty?`. Upon return, B discovers the pending Message-2(`put`) in its message queue; so as is with Step 3 of Example 18, B changes its VFT to active, places itself in the global scheduling queue, and execution returns to A via normal procedure return, which assigns `1` to `x`. (We note that the enqueued VFT is the non-queuing VFT corresponding to the one described in Step 1.)

Step 4: `B <- unlock` and `y = B <-- get()` are executed at A. Since B is active, Message-4(`unlock`) is enqueued in the message queue of B, and execution returns to A. The same happens for Message-5(`get`), except that, since A expects a reply but cannot get one, it will suspend itself with by saving the continuation that expects a reply from B.

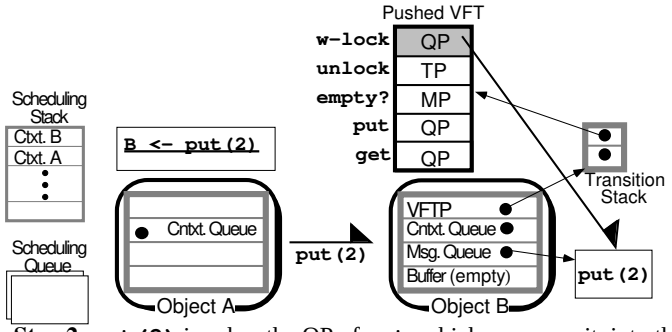
Step 5: Eventually, B is scheduled, and the first entry of the message queue, Message-2(`put(2)`) is invoked via the non-queueing VFT. This results in an immediate guard failure, so the second entry, Message-4(`unlock`) is invoked. This time, it succeeds: the resulting effect of the `restore` transition 'pops' the transition stack. Upon return, B again finds messages in the message queue, so again B places itself in the scheduling queue, this time with the 'standard' dormant VFT.

Step 6: Eventually, B is scheduled again, and Message-2 is invoked with success via the dormant VFT, storing `2` into the buffer, followed by an entry into the global scheduling queue.

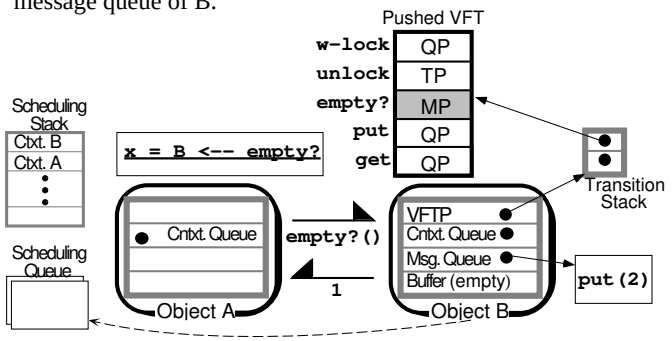
Step 7: After another invocation of Message-5(`get`) via global scheduling, upon return the method `get` discovers that A has been suspended with a continuation waiting for a value; as a result, the continuation is invoked, which returns control to A and thereby assigns `2` to `y`.



Step 1: The **push** transition in the TP of the **write_lock** method pushes the customized VFT onto the transition stack.

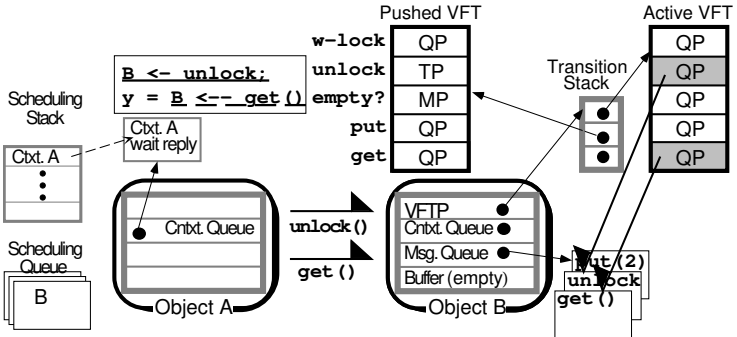


Step 2: **put(2)** invokes the QP of **put**, which enqueues it into the message queue of B.

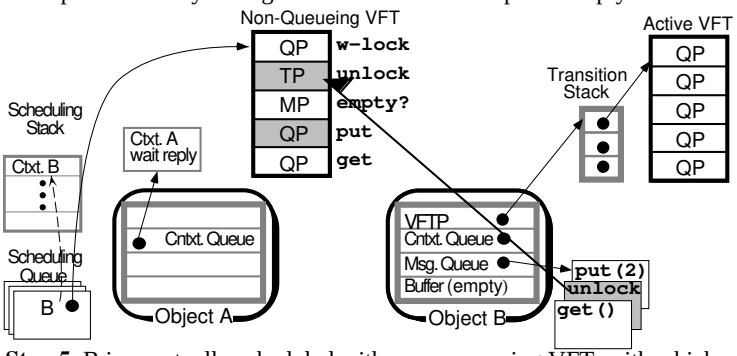


Step 3: Since B is dormant, the MP of **empty?** is executed, returning **1** to A. Prior to returning, finding **put(2)** in its message queue, it sets its VFT to active and puts itself in the scheduling queue.

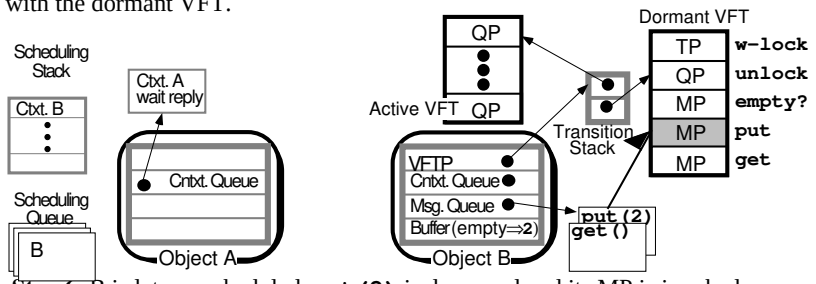
Figure 8.6: Example of Synchronization Control via Transitions (1)



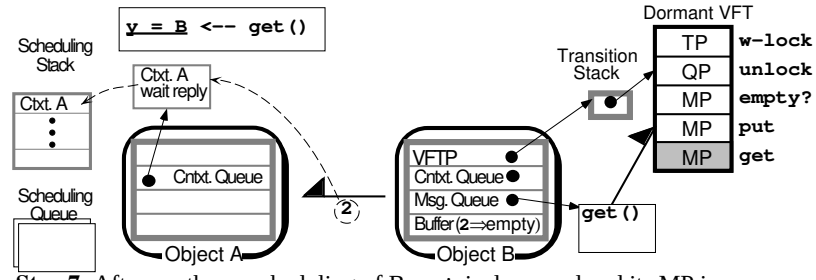
Step 4: Since B is active, the QP puts **`unlock()`** in the message queue, and likewise for **`get()`**. Then, since A cannot immediately get a reply, A suspends itself by saving its continuation that expects a reply from B.



Step 5: B is eventually scheduled with a non-queueing VFT, with which **`put(2)`** is invoked resulting in guard failure; next, unlocked is invoked, which succeeds and its associated transition 'pops' the transition stack. B again makes itself active and enqueus itself into the scheduling queue with the dormant VFT.



Step 6: B is later rescheduled, **`put(2)`** is dequeued and its MP is invoked via the (non-queueing) dormant VFT, storing **2** into the buffer. B again makes itself active and enqueus itself into the scheduling queue.



Step 7: After another rescheduling of B, **`get`** is dequeued and its MP is invoked, and **2** is returned. The continuation of A waiting for this value is invoked, assiging **2** to **`y`**.

Figure 8.7: Example of Synchronization Control via Transitions (2)

Activity	*SPARC Instructions*				
	`b-buf` Synch. (Unoptim.)	`b-buf` Synchrnzr. (Optimized)	`b-buf` Trans. (Unoptim.)	`b-buf` Trans.Opt. (Optimized)	ABCL/ onAP1000 (Null Mess.)
Check Locality	4	4	4	4	3
Lookup and Call	6 (w/arg)	6 (w/arg)	6 (w/arg)	6 (w/arg)	5 (wo/arg)
Guard Evaluation	15	7	0	0	0
Switch VFTP to Active Mode	3	0	3	0	3
Execution of Method Body	8	8	7	7	– (empty)
Check Message Queue	3	0	3	0	3
Switch VFTP to Dormant Mode	3	0	3	0	3
Polling of Remote Message	5	0	5	0	5
Transition Execution	11	3	22	14	0
Stack Pointer Adj. and Return	3	3	3	3	3
Total Instrs. (`put()`)	61	31	56	34	25
Message Instrs. (incl. Guards)	52	22	48	26	25
Total Time (μsec, AP1000@25Mhz)	5.7	3.4	5.5	3.4	2.3

Table 8.1: Benchmark of Intra-node Message to Dormant `b-buf` Object on AP1000.

8.7 Preliminary Benchmarks

We have performed preliminary benchmarks on intra-node asynchronous message passings to the `b-buf` objects described in Section 7 on AP1000, and also compared the result with that of the original ABCL/onAP1000. A local `put/get` message pair was sent repeatedly to a dormant object, and averaged out. For the original ABCL/onAP1000, a message with null arguments was sent repeatedly to invoke an empty method. Table 8.1 shows the results: the cost of unoptimized message passing (including guard evaluation for synchronizers and transitions) is approximately twice in comparison to a null message passing on ABCL/onAP1000. The compiler can optimize down to ABCL/onAP1000 message passing speed in ideal cases, however: the optimized code inlines the guard and the method body into *MP*, and furthermore does not perform VFTP switching or remote message polling, owing to the fact that the method is a small, leaf method in the message chain.

The overhead incurred by our synchronization scheme is significantly smaller compared to other cost of concurrent execution in our implementation architecture. For example the cost of intra-node message to an active mode object takes approximately 10μseconds on ABCL/onAP1000, and inter-node message passing to a dormant object takes approximately 9μseconds[107]. Integration of our scheme merely tacks on the small difference in the overhead for intra-node messages given in Table 8.1. Although larger-scale benchmarks are still undergoing, we believe the above figures, plus the observed speedups in larger-scale ABCL/onAP1000 benchmarks, strongly support our claim of efficiency.

Chapter 9

Discussions and Future Work

9.1 Summary of Part I

The prime objective of OOCP languages is to provide maximum computational power through concurrency of objects. At the same time, OOCP languages allow the system to be dynamically extensible and configurable. This effectively captures the essential properties of concurrent computational systems, which are highly complex and must change and evolve to adapt to the requirements of the user. A concurrent object-oriented language, thus, must provide robust and efficient language mechanisms for allowing high degree of extensibility and re-use. Ideas that have flourished in the sequential OO world for this purpose, particularly inheritance, provides a powerful solution for this objective, allowing construction of large-scale application frameworks that a user customizes to create his own application. This 'differential' programming is fundamentally different from the compositional approach to programming; in fact, one could be adventurous to say the former style might be more suitable in a concurrent setting, because compositionality is often hard to achieve in a concurrent system.

Unfortunately, as we have shown, synchronization constraints and inheritance have conflicting characteristics, resulting in inheritance anomaly, and thus it is difficult to simply combine them in a clean way. We have analyzed various types of inheritance anomaly and discussed several approaches to its solution. Our main proposal in particular was to have multiple synchronization scheme constructs which allow proper encapsulation as well as efficient implementation. Below, we discuss the implications of our language construct design.

9.1.1 Why Method Sets are not Full First-class Entities?

As mentioned earlier, our proposal is carefully crafted to allow the structure of all the VFTs to be determined statically at compile time. This is to avoid the necessity of constructing or operating upon the method sets at run-time in the implementation, because such run-time operations are costly: due to extensive subclassing, the number of methods for user-defined classes for large application frameworks exceed one hundred on the average in real-life settings[13]. Thus, operations that involve run-time copying of VFTs must be avoided.

There are two potential operations that break the requirement of static generation of VFTs: (1) copying of VFTs from the 'prototype' VFT upon individual object creation, and (2) first-class operations that does not allow in-place updating of VFT entries (this is a similar situation to the difficulties of updating large data structures in single-assignment languages). In particular, if a VFT was constructed per each message reception, the execution overhead would be prohibitive. Even relatively conservative proposals such as enable sets would incur the allocation and dynamic management of a data structure whose size is not bound by a constant. Given that objects are created in a fine-grain manner, such overhead should be avoided.

Another concern is the storage space; if each concurrent object were to have a customized VFT, it would quickly overwhelm the available memory space, if each VFT had approximately hundred entries. It is typical for the system to have thousands to millions of objects—for example, our simple N-queens benchmark alone created 4,000,000 objects for $N = 13$, most of which is simultaneously alive at some point in execution. This means that the VFT would consume $4 \times 10^6 \times 4 \times 10^2 = 1.6 \times 10^9$, or 1.6 Gigabytes! in the benchmark.

By contrast, our scheme avoids the necessity of dynamic construction and direct memory operations upon the virtual tables altogether. Even method sets bound with guards can be case-analyzed at compile time, and multiple instances according to the cases can be statically prepared. Since the maximum number of VFTs generated is bound by a constant factor of the number of classes multiplied by the size of the synchronization specifications (i.e., number of transition types), the number should remain manageable in practice. Furthermore, objects and methods that do not require synchronization suffers zero or a slight overhead of an indirection, because invocation via virtual tables would be identical to that of C++ via VFT, and transition procedures will not be unnecessarily invoked for such methods.

The use of multiple virtual table has been independently proposed by [95] for efficient implementation of supporting fine-grained, object-wise protection capabilities. Unfortunately, the proposal was not implemented at the time of the writing of the paper, and as a result, contains a serious drawback for fine-grained execution: the described implementation would involve run-time creation of customized virtual table per each object, exactly what we avoid in our scheme as described above.

9.1.2 Inheriting Synchronization Code with Encapsulation

One of the primary principles of object-orientation is the encapsulation of *methods* by abstracting their operations and hiding their underlying implementation. Analogously, in our proposal, *method sets* encapsulate the 'abstract state' of a concurrent object with respect to its synchronization constraints, and realization of transitional behavior between the abstract states is realized by the underlying implementation using synchronization code. By properly defining a class so that method sets precisely describe such abstract states, general subclassing/code re-use can be done in the following way: (1) If no new states/transitional behavior need to be identified in the subclasses, then re-definitions can be confined to set-operations on the method sets. (2) If the re-definitions affect the abstract states (such as state partitioning), then the user chooses the appropriate

synchronization scheme that best describe the new set of abstract states from the existing ones, and performs (local) modification to the affected states. In common cases, re-using existing synchronization code is transparent, as we have demonstrated in Section 7. Even if not, users can still re-use much of the synchronization code of the superclasses in a familiar way using `super` references, etc. This in turn allows definitions of abstract classes that serve as basis of application frameworks for parallel programs.

Still, language mechanisms alone are not omnipotent; we stress the importance of both the detailed user knowledge of the synchronization constraints, and good OO-design disciplines in re-use. When inheriting methods, it is impossible to re-use or refine them in subclasses, unless the user has a good understanding of their behavior. Analogously, the user has to have a good understanding of the abstract state that each method set represents plus their transitional behavior when re-using synchronization code. In fact, one can abuse our mechanism and create obscure situations to break encapsulation deliberately, just as one can break encapsulation with sequential OO-languages. Still, we claim that we have achieved higher balance of encapsulation, expressive power, and practical implementability compared the previous proposals.

9.1.3 Which Synchronization Scheme do we Use?

The user has the liberty of using the 'appropriate' synchronization scheme depending on the synchronization constraint he has to satisfy, and in many cases either scheme can be used, just as there are many ways to implement the functionality of a method. There are, however, situations where one is more preferable, depending on (1) semantical and/or (2) efficiency issues. For example, cases involving simple state partitioning are better programmed using synchronizers, whereas cases where a certain set of methods needs to be enabled/disabled in protocols are better programmed using transitions. As for efficiency, the use of transitions adds an extra overhead of indirection and transition stack management, but there are cases where it is better, e.g., when the message queue of an object becomes long, synchronizers have to evaluate the guards for each message in the queue, whereas transitions need not[106]. Establishing a concrete guideline is a subject of our future work.

9.2 Conclusion and Directions for the Future

Throughout the paper, inheritance was used as a means of code re-use. Relationship with another aspect of inheritance, subtype classifications, might be required. One related research we should note is the event types for synchronization as first class values, in the concurrent version of language SML[94]. Another work towards the type theory for active objects[87] by Nierstrasz et. al. and also by Honda[47] are also interesting—although our work has not addressed the issues of types in OOCP languages and adhered to inheritance as means of reusing code, the relationship between the type theory and the semantics of inheritance is an active area of research for sequential OO languages. Recent work by America et. al. to separate the subtyping hierarchy from the inheritance hierarchy in the POOL family of concurrent-OO languages such as POOL/I and POOL/S[6, 5] could bring

the two aspects together: in their formalism, subtyping relationship is determined solely by the observed external behavior and not the internal structure. Using this formalism, one can determine if the inheritance is actually correct in a sense that the behavior of the parent class is entirely preserved; as a result, it could be possible to 'type' the behavior of a concurrent object whose synchronization was specified using our scheme.

In conclusion, in order for OOCP languages to be usable for large-scale programming we still need to strive on the followings with respect to inheritance:

- Establishing a more precise and formal definition of classification of inheritance anomaly. Although we have shown a result of one formal analysis, it was not complete in that it only treats the anomaly that occurs with state partitioning. The work on active object types mentioned above could serve as a basis of more comprehensive formalism.

- Further identification of a general class of synchronization schemes with respect to anomaly classifications. Although we tried to be as comprehensive as possible by categorizing and selecting representative synchronization schemes, formal analysis might provide more insights for further sub-categorization.

- Continuous development of languages features which respect the encapsulation and efficiency criteria, based on the above two analysis.

- Completing the implementation of the compiler and the language system. There are many other implementation issues that we have investigated, such as concurrent OO-specific optimizations, distributed garbage collection and long-term load-balancing[108], etc.

- Construction of practical parallel applications on top of those systems. Already applications for N-body simulation and Genome RNA secondary structure prediction are being ported, whose preliminary measurements have shown promising speedups.

Part II

Reflection in Concurrent Object-Oriented Languages— Architecture, Language, Implementation, and Semantics

Chapter 10

Introduction—Reflection

10.1 Problems with Previous Computational Models of Concurrent Objects

10.1.1 Requirements in Parallel Computation with OOCP languages

Object-Oriented Concurrent Programming (OOCP) is mostly recognized to have been initiated by the proposals of the Actor concept by Hewitt[45]. As we have already covered, the emphasis of the work in the area is shifting towards their effective use and implementations on parallel architectures. However, real-life concurrent computation in concurrent and distributed architectures involve many intricate factors of concurrency and distribution that do not manifest in these models, such as the routing mechanism of message delivery, memory management (garbage collection, etc.), object scheduling (computational resource allocation), object distribution policy management (for hot-spot resolution, etc.), and other cooperative resource management. That is to say, although these factors are usually hidden beneath the abstractions of the model, in practice they need to become manifest in situations such as described below:

- The underlying architecture sometimes become significant in the analysis of concurrent programs; for example, sometimes the topology of the underlying network architecture affects the execution efficiency of parallel algorithms.

- The efficiency of parallel programs depend heavily on the resource allocation and distribution policies. Parallel programs sometimes exhibit drastic difference in execution time depending on different scheduling algorithms to allocate the computational resource. With distributed-memory architectures, execution efficiency changes drastically according to how data are mapped to memory because of the difference in the access cost of local and distributed memory.

- High-level language features such as object migration, homogeneous and heterogeneous object groups distributed among multiple nodes, object persistence, clock

consistency management (e.g. the Time-Warp algorithm for discrete event simulation), distributed class management, etc., require significant underlying support to preserve the required semantics of the language. For example, in a class-based OOCP language, even a slight change in the definition of a class must be propagated throughout the system, not only for the instances of that class itself but also for the instances of the subclasses of that class. This might involve significant implementation cost depending on the inheritance rule of the language.

Another important point is that, future parallel architectures should not be a standalone computational back-end; rather they should be parts of heterogeneous computing systems interconnected via a high-speed network. This requires the parallel architectures to be *open systems*, that is, to have the ability to dynamically adapt to the new problems and the changes in the external environment. In order for the system to support such dynamic evolution, the language and the model should be able to describe such evolution. These often involve changes in the underlying implementation; for example, the number of nodes in the system may increase, or the network latency may decrease due to evolution of the architecture.

Overall we would like to model the implementation details in a tractable way when necessary. The above aspects are still considered to be mostly outside the scope of programming languages, and their control is only available in fixed, ad-hoc fashion, with little possibility for user extensibility. But we want to do this in a portable way, within the confines of some consistent language model.

10.1.2 Why Current Models of Concurrent Objects are not Sufficient

Ever since the Actor concept was proposed, there has seen several works in attempting to establishing it as a formal model of concurrent computation[31, 1]. Recently, there have been other works that attempts to understand the Actor model in terms of other models of concurrent computation, such as the ν-calculus[48, 47], Rewriting Logic[77], and CC[57]. In addition, there have been other works that explored the semantics of particular OOCP languages[6, 130, 99].

The ultimate goal of these models can be regarded as establishing the foundations of concurrent computations in general, just as the Lambda Calculus is intended to give the foundations of the (untyped) sequential computation. Indeed, since the models abstract out the unnecessary implementation details of the underlying support of the computation — the programming language, the operating system, and the hardware architecture — they are significantly useful for formally proving the properties of concurrent programs.

When applied to practical parallel computation with objects, however, their programming involves many intricate details stemming from the diverse elements of parallelism and distribution, such as scheduling and load balancing, as mentioned above. These intricacies are not present in sequential programs, and moreover, are usually *not* part of the abovementioned models of concurrent. For example, in the ν-calculus, the effect of the communication is considered to be immediate between each interaction of objects, and all the transmission delays incurred in real-life is abstracted out (or rather, absorbed) in

its structural congruence rules[48], so the model *cannot* describe the cost of inter-object communication.

By all means such abstractions are not unwarranted, and in fact its importance must be fully acknowledged: incorporation of the implementational details is contradictory to the usual requirement that the models be as abstract as much as possible. By 'covering-up' the implementation as unnecessary details, one can use such models to prove equivalence properties of programs, for example.

Nevertheless, when building a system based on these models, most of the intricacies confront us, and cannot be 'swept under the rug'. As we have stated earlier, even for analysis, implementation details sometimes become significant. For example, it is well known that the topology of the network architecture affects the complexity of the parallel algorithm. Thus, we would like to model the implementation details in a tractable way when necessary.

One approach to solve this dilemma is the to treat the intricacies as entities that do not appear in the computational model, and add ad-hoc meta-level extensions to the language. This has been successfully employed in Prolog: the usual Prolog systems add numbers of meta-level predicates to control the computational behavior of the system, such as the '`!`' (cut) and `assert` predicates. These meta-level predicates destroy the declarative purity of Prolog, but have proven to be necessary in practice. In the context of parallel programming languages, one often embeds special directives in the programming language to control data mappings and scheduling policy.

Although such approaches are effective, they usually destroy the pureness of the underlying computational model. Thus, substantial effort that went into constructing a consistent model is (partially) lost, because most formal properties that are derivable from such models require absolute purity of the model.

10.2 Introduction to Reflection and Reflective Architectures

In order to extend the computational model to incorporate the 'implementation details' so that the benefits of the model can be enjoyed as much as possible, one could add, in an ad-hoc way, the intricacies as new elements into the model. However, this would introduce unnecessary complexity into the model for other purposes.

As for re-use, one could force the user to always follow some inheritance protocol. But this is not favorable for future extensions. In principle, elements of parallel computations that are 'meta' should be orthogonally encapsulated in the meta-level of application-level (i.e., the base-level) computation.

To solve the above two dilemmas, the effective solution is to employ *reflection*, or *reflective architectures*.

Reflection is the process of reasoning about and acting upon the system itself[100, 64, 129]. In a conventional computing system, the *subject matter* of computation is external to the system, i.e., the computation is performed on data that represent or model entities that are external to the system. A *reflective system* takes a step further by allowing the subject matter be the system itself, i.e., computation can be performed on the system

itself in a uniform way[64]. A reflective system thus 'opens up' the system by providing an appropriate abstraction on the internal (often implementational) detail of itself to the user program. Contrary to the misconception that 'reflection' is some difficult-to-understand, not-too-useful philosophical jargon, it is a practical scheme that offers a new perspective in constructing a malleable, large-scale system such as programming languages[58], operating systems[124], and window systems[92]. One can make an analogy of reflection to *inheritance*, which was derived from AI and incarnated into object-oriented programming languages, and now serves important roles in methodologies for system construction of user interfaces, operating systems, and database systems. We believe that, in the same manner, reflection can offer another clear architectural perspective, especially for the dynamic aspects of the system where inheritance is known to be somewhat ineffective.

To expose, or *reify* its internals, a reflective system contains data that represents or models the metalevel structural and computational aspects of itself within itself. Such data is said to be *reified*, and must be dynamically self-accessible and self-modifiable by the user program. Furthermore, when the user modifies such data, the modification must be 'reflected' to the actual computational state of the user program. This requirement is termed as *causal-connection*, and is satisfied by all reflective systems. When the requirement is satisfied, the meta-level data that provides the abstraction is called the *Causally-Connected Self-Representation(s) (CCSR)*. For practical purposes, the meta-relation is often that of implementation. In such systems, causal connection is automatically guaranteed. Although modifying the implementation may seem like a dangerous endeavor, it has been shown that modification can be confined to the local portion of the user program by some appropriate *Metaobject Protocol*, a scheme pioneered in CLOS[58].

Another requirement for reflective systems is that self-access and self-modification must be possible in a way that is little different from the base-level programming. In other words, appropriate abstraction of the internal program state must be provided by the system so that the user can perform the meta-programming of the system in a clean, concise manner that does not deviate significantly from 'normal' programming. For example, in 3-Lisp[100], one can program reflective computation merely by defining *reflective procedures* whose means of definition differ little from that of normal procedures.

The approach of using reflective computational model is more favorable compared to the ad-hoc extensions approach, because the lucidity of the base model is maintained. In addition, it allows natural layering of the implementation; when the semantics of the base level is basically unaffected, one can easily switch back and forth between the 'implementation view' to the 'abstract view', depending on his requirements.

10.3 Reflection in Object-Oriented Concurrent Systems — The Benefits

The benefits of *computational reflection* in sequential languages have been identified by many researchers: Previously, programming language functionalities such as debugging facilities, interfaces to external world, and exception handling were added to languages in an ad hoc and inflexible manner. In contrast, languages based on reflective execution models are powerful in providing linguistic mechanisms to use the above functionalities

in a uniform and flexible manner via clean access of the abstraction on how the system is implemented[64, 58].

We claim that reflection is more beneficial in the construction of concurrent computational models, for a concurrent system embodies multitudes of aspects that do not arise in sequential computing as mentioned earlier: scheduling, communication, and load-balancing, among numerous others. Here, as were pointed out in [129] and [101], constructing a concurrent system with a reflective architecture could be beneficial to encompass such aspects within the programming language framework for the reasons mentioned above. The dynamic optimization is especially effective in a concurrent system because (1) as mentioned, the reflective architecture allows the programmer to control the computational aspects of concurrent system dynamically[1], and (2) both the reflective computation and the base-level computation can be performed in parallel[129, 70, 117].

10.4 Contribution: ABCL/R2—A Language Based on Hybrid Group Architecture

The primary contribution of the second half of this thesis in the establishment of a new model/architecture and language of reflection in the context of concurrent objects to allow flexible use of parallel code. As we describe in the next section, past proposals of reflective architectures for OOCP languages lacked the necessary abstractions to describe what could be described in terms of *coordinated resource management* in parallel computing, such as scheduling (CPU resource), object allocation (memory resource), etc. The primary reason is that previous architectures only had partial ingredients for realizing group-wide coordination of objects. This chapter demonstrates the language system ABCL/R2, which realizes *linguistic lucidity* (the ability of reflective systems to be describable solely in terms of itself). First, in Chapter 11, we survey and discuss the limitations of the previous OOCR architectures, especially with respect to group-wide coordination of objects. Then in Chapter 12, we present an OOCR language with a *hybrid group architecture*, ABCL/R2, which incorporates heterogeneous object groups with group-wide object coordination and *group shared resources*. It also has other new features such as *non-reifying objects* for efficiency. By the design and implementation of ABCL/R2, we contribute feedback to the conceptual side of OOCR architectures and object groups by (1) showing that (heterogeneous) object groups are not ad-hoc concepts but can be defined uniformly and lucidly, and (2) identifying that hybrid group architectures embody *two* kinds of reflective towers, instead of one: the *individual tower* which mainly determine the structure of each object, and the *group tower* which mainly determine computation. Next, in Chapter 13, we describe our novel schemes for actually implementing this (rather complex) reflective architecture. In particular, for programs that do not involve reflective computation, ABCL/R2 is shown to be just as fast as the original ABCL/1, and even for cases where reflective computation is involved, it is within a factor of 6–7. This is a considerable improvement over the previous ABCL/R, which is a factor of 1000 slower

[1] One of the interesting applications we will note is to change the scheduling policy of object execution dynamically [20] in distributed simulation based on the Time Warp Scheme[55].

compared to ABCL/1 for all computations. Then, in Chapter 14 we describe how these reflective features of ABCL/R2 allow coordinated resource management to be effectively modeled and efficiently controlled in the metalevel, achieving *metalevel encapsulation* — as an example, we study the scheduling problem of the Time Warp algorithm[55] used in parallel discrete event simulations. Finally, we briefly explore the semantic foundations of our architecture, by the definition of Micro-ABCL/R2 in Chapter15. In particular, we propose the technique of *structured mail address* which can describe the base-meta relationship more cleanly compared to other proposals.

Chapter 11

Object-Oriented Concurrent Reflective (OOCR) Architectures —Categorization of Previous Work

Recently, several proposals were made in the use of reflection in the framework of *Object-Oriented Concurrent Programming (OOCP)*, with various *Object-Oriented Concurrent Reflective (OOCR)* architectures: languages such as ABCL/R[117] and X0/R[71], AL-1/D[88] were based on the *individual-based architecture*, ACT/R[118] was based on the *group-wide architecture*[1]. Each language differs in what meta-lingual features can be realized through reflection, i.e., what features particular to concurrent OO computing, such as object monitoring and migration, could be cleanly encompassed within the programming language framework. There, the evolution of the languages can be understood for the difference in the characteristic of their reflective architectures. This chapter concentrates on categorizing and pointing out the advantages and weaknesses of each reflective/metalevel architecture.

11.1 Previous Works in Classification of Reflective and Metalevel Architectures

First, before starting the categorization, let us review the previous work in classifying the various metalevel and reflective architectures:

Maes[65] gives an overview of computational reflection in programming languages. She defines reflective architectures as recognizing "reflection as fundamental programming concept", and providing explicit treatment of reflective computation:

1. being able to *reify* the running aspects of the system, and
2. guaranteeing the *causal connection*.

[1]The original idea of the group-wide architecture was mentioned in the last sections of [117] and [129]

She proceeds to exemplify the reflective architectures for various language paradigms: procedure-based, logic-based, rule-based, and object-oriented. Unfortunately, the paper does not address any concurrency issues.

Harmelen[112] classifies meta-level architectures of logic-based and rule-based systems. He places the emphasis on locus of actions as being resident in the object level and meta-level, and distinguishes *object-level inference*, *mixed-level inference*, and *meta-level inference*. The meta-level inference architecture is further sub-classified based on the linguistic relationships among both levels: *mono-lingual*, *amalgamated*, and *bi-lingual*. The conclusion of the paper is that bilingual, meta-level inference architecture is the most promising for logic-based systems due to its flexibility and modularity among other reasons. The paper does not go deeply into the issue of reflection other than to say that being able to reason about and to modify one's metalevel are vital, nor mention concurrency.

Ferber[41] concentrates on classifying metalevel architectures of class-based object-oriented languages. He attempts to explore the relationships of metaclasses (as in *ObjVLisp*[32]) and metaobjects (as in 3-KRS[64]), and identifies three models for computational reflection in (sequential) object-oriented languages, namely,

- the *metaclass model*, where the metaclass object serves as the metaobject,
- the *specific meta-object model*, where there is a unique metaobjects of class `Meta-Object` per each object, and
- the *meta-communication model* where messages are reified.

The work also does not address concurrency, but is important in identifying the conceptual characteristics of metaobjects.

Smith[101] briefly discusses the issue of reflection in concurrent systems. He identifies two cases, one in which the causal connection is the one of implementation, and the other where the disconnection is achieved with a single autonomous meta-level monitor process. Tanaka's reified/reflected message scheme in an actor language[2][104] is an example of the former; the reification/reflection is purely that of structure of object implementation.

Although the above works discusses various aspects of reflection and classifies metalevel/reflective architectures from different viewpoints, so far to our knowledge no work has existed that generally discusses reflection in the context of concurrent/distributed object-oriented languages. Nor, has there been work that classifies architectures dependent on characteristics that are *particular to object-oriented concurrency*, i.e., not becoming manifest in sequential object-oriented languages. It is our premise that the need for such a discussion and classification is becoming omnipresent — for as we argued above, the power of reflection can be best utilized in concurrent computation. Furthermore, we claim that in order for reflection to be put to practical use in concurrent/distributed systems, we need to analyze and evaluate the merits and weaknesses of each, and synthesize a new architecture suitable for practical use.

[2]Unfortunately, this language has no official name; henceforth, we will refer to it as *T-Actor* for convenience

11.2 Classification of OOCR Architectures

First, the two aspects of CCSR in OO languages are the (1) *structural aspect*, indicating how objects or group of objects in the base-level and the meta-level are constructed and related, and the (2) *computational aspect*, indicating how meta-level objects represent the computation of the base-level objects[41]. Altogether, appropriate construction of the *meta-level architecture* dictates what structural and computational aspects can be reflected and operated upon — this we believe is more prevalent in concurrent systems, for factors such as scheduling must be given appropriate meta-level abstractions in order for them to be effectively and efficiently accessed from the base-level.

The OOCR architectures can be categorized from the structural aspect, chiefly dependent on whether the architecture has the notion of *metaobjects*, which Pattie Maes introduced for sequential object-oriented language 3-KRS in [64]. It is often argued that providing explicit metaobjects in the architecture is conceptually natural with respect to the uniformity principle, "everything is an object." But, what do we really mean by a 'metaobject'? It is pointed out that there has been great confusion on the usage of the term[42] — we first attempt to provide conceptual distinction in the terminologies.

A *meta-level object* is any object that resides in the meta-level of the object-level computation. *Metaobjects* are objects that reflects the structural and computational aspect of a *particular* base-level object; thus, a metaobject is a meta-level object, while the converse is not necessarily so. The relationship between the metaobjects and the base-level object they represent could either be one-to-one or many-to-one, i.e., there could be multiple metaobjects collectively 'representing' a single base-level object[50, 80]. The metaobjects must be unique to a single object in the base-level, that is, they are not shared.

There are several important criteria that metaobjects must satisfy that distinguish it from other meta-level objects:

1. *Identifiability of Metaobjects* — a base-level object must be able to identify its metaobjects intrinsically.

2. *Uniqueness of Metaobject Denotation* — metaobjects must be able to intrinsically and uniquely identify the base-level object it denotes.

3. *Causal Connection* — a base-level object and its metaobjects must be *causally connected.*

4. *Governing of Computation* — the metaobject must dynamically govern the computation of the object it denotes or represents.

By these criteria, the 3-KRS and ABCL/R 'metaobjects' are metaobjects as we describe later on, whereas the CLOS 'metaobjects' are meta-level objects in our terminology. Class objects in class-based languages such as ObjVLisp[32] are not metaobjects either, for (1) the denotation is not direct and/or not unique, and (2) classes are not "meta" in a computational sense[41], that is, they do not actually govern the dynamic aspect of object computation.

Given the distinction of metaobjects and meta-level objects, a reflective object-oriented system can basically be constructed in two ways:

- **Individual-Based Architecture:** Each object in the system has its own metaobject(s) which govern(s) its computation. By 'individual-based' we mean that an individual object is the unit of base-level computation that has a meaningful CCSR at the meta-level.

- **Group-wide Architecture:** In this architecture, a group of meta-level objects comprises the meta-level. The behavior of an object is not governed by metaobjects; rather, the collective behavior of a *group* of objects is represented as the coordinated actions of a group of meta-level objects, which comprise the *meta-group*. By 'group-wide' we mean that the entire object group is the unit of base-level computation that has a meaningful CCSR at the meta-level (as a meta-group); thus, there are *NO* intrinsic meta-relationships between a base-level object and a particular metaobject at the meta-level.

For sequential OO-languages, the representative of the individual-based architecture is 3-KRS, and the one representative of the group-wide architecture is CLOS. Below, we will give examples of each architectures for object-oriented concurrent languages in our research.

11.2.1 The Individual-Based Architecture — ABCL/R

An example of this architecture is ABCL/R[117]. Each object has its own unique metaobject, $\uparrow x$ that can be accessed with a special form `[meta` x`]`. Conversely, given a metaobject $\uparrow x$, `[den` $\uparrow x$`]` denotes the object it represents. The correspondence is 1-to-1, i.e., `[meta [den` $\uparrow x$`]]` $\stackrel{\text{def}}{=} \uparrow x$ and `[den [meta` x`]]` $\stackrel{\text{def}}{=} x$. The structural aspects of x — a set of state variables, a set of scripts, a local evaluator, and a message queue — are part of state variables of $\uparrow x$. A metaobject has its own metaobject $\uparrow\uparrow x$ as well and so on, conceptually forming an infinite tower of metaobjects (Figure 11.1).

The *arrival* of a message M at object x is represented as an *acceptance* of the message `[:message` M R S `]` at $\uparrow x$, where R and S are the *reply destination object* and the *sender object*, respectively. The interpretation of the script of x by $\uparrow x$ is carefully designed so that the concurrent activities allowed within an object in the computational model are preserved. Causal connection and total governing are guaranteed as $\uparrow x$ implements x. Figure 11.1 illustrates this. As are most individual-based architectures, reflective computation by an object in ABCL/R is basically performed by sending messages to its metaobject (or any other metaobjects in the tower); alternatively, the metaobject can immediately identify and send messages to its denotation.

Note that the individual-based architecture is independent from the issue of inter-level concurrency. As stated earlier, In a sequential OO-reflective architecture, there is only a single computation thread in the tower of metaobjects. This thread performs the interpretation of a certain level, and a reflective operation causes a 'shift' of this level. By contrast, in ABCL/R, there is internal concurrency within a single object as defined by the computational model of ABCL/1[127]: message reception/queuing versus

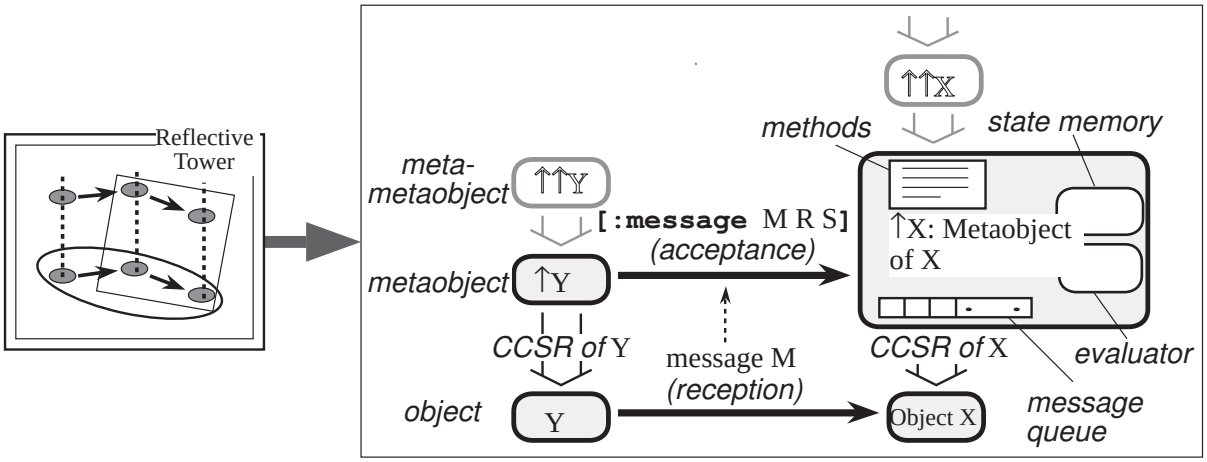


Figure 11.1: The Individual-Based Architecture in ABCL/R

the execution of the user script. The interpretation of x by $\uparrow x$ is designed so that the former is performed as a meta-level task while the latter is performed as a base-level task. Concurrency between the tasks is then preserved, because ABCL/R supports inter-level concurrency between the base-level object and its metaobject.

Other examples of individual-based architecture are Rosette[109], Merling III[40], Tanaka's actor language[3][104], and X0/R. The distinction here is the relationship between the object and its metaobject — whether the meta-object has a separate identity from the object it denotes[4]:

- **'Unified' Identity:** when the meta relationship is that of total implementation as in ABCL/R. There is actually only one unique object in the tower; whether to regard it as an object or its metaobject (or metametaobject, ...ad infinitum) depends on from which level the object is being 'viewed' in the computation.

- **Partially Self-Reifying Identity:** when there are no explicit metaobjects, but the snapshot of object state can be reified. We can consider the reified structure to be obtained from the implicit metaobject. T-Actor is an example of this.

- **Separate Identity:** the metaobject has its own separate identity. Metaobjects in 3-KRS, Merling III, and X0/R are of this kind. In a class-based architecture, a metaobject is often an instance of class `MetaObject` or its subclasses.

The advantage of the separate identity over the first two is that the metaobject could be interchanged dynamically in order to change the basic behavior of the object, such as

[3]Unfortunately, this language does not have an official name; henceforth, we will refer to it as *T-Actor* for convenience.

[4]Both Merling III and Tanaka's Actor Language do not employ the term 'metaobject'. However, they do have the notion of some meta-level structure representing the structure and the computation of each object, which we regard as metaobjects in our terminology.

message reception. Although unrestricted alteration of metaobjects could be catastrophic to the integrity of computation, it nevertheless can be a powerful tool if employed in a controlled manner[71]. Dynamic alteration is more difficult for the first two, although possible for the unified identity by changing the behavior of the metaobject through its meta-metaobject. One major drawback of the separate identity particular to concurrency is the possible cost for maintaining causal connection. This becomes prevalent if the only way for the objects to communicate is via messages, because there is a natural delay for the transmission to occur. Then, the object and its metaobject must explicitly synchronize their activities in order to maintain the causal connection.

The metaobjects can be extended to be *multiple*, that is, each object could have multiple meta-objects that captures the certain 'view' of its computation. For example, in language AL-1[52] and AL-1/D[88], each base-level objects has associated with it six different meta-objects, depending on the 'models' they provide—they are: *operation*, *resource*, *DE*, *migration*, *statistics*, and *system*. Each model can also be considered to provide different restricted and consistent 'views' in the entire space of functionality the metaobject can provide, so that the user can program the metaobjects consistently and more safely under a certain view. For example, the metaobject operation model provides the view of the object it is governing similar to the one of ABCL/R, whereas the resource model provides a more primitive view of the base-level object, reifying the object in the form of *segment*, which is an abstraction of the memory it occupies, and *context*, which is an abstraction of its execution environment. Altogether, the different models are coordinated within the *Multi-Model Reflection Framework (MMRF)* [88].

In MMRF, the identities of the metaobjects can be unified or separate. Here, the issue is whether to allow concurrent execution of metaobjects of different models. If the identity is unified, then unless the object is multi-threaded, one need not worry about the coordination of different models, as long as each model is guaranteed to produce consistent state of objects for all models per each meta-level method execution. However, it has the drawback of prohibiting coordination of different models. If the identity is separate, such coordination would be possible, but maintaining the consistency of base-level object becomes difficult. In particular, metaobjects might require excessive message-passings to coordinate their behavior. In AL-1/D, the identity of the metaobjects are separate, but a extrametalinguistic control feature is provided for coordination: metaobject methods that are affixed with '!' can be run in parallel with methods of other metaobjects, while methods affixed with '!!' must be executed atomically.

11.2.2 The Group-wide Reflective Architecture — ACT/R

ACT/R[118] is an Actor-based language that supports the notion of groups under this architecture. It is based on Gul Agha's Actor formalism[1] which models the state of *system* S of Actors as its *configuration*, C (a pair consisting of a set of Actors and a set of *tasks* in the system), and defines transitional semantics between the configurations, each transition being the computation of a task. Now, given a configuration C, We define the *metaconfiguration* $\uparrow C$ as being a metalevel representation of a configuration by a system of Actors. A task $< t, m, k >$ in C where t is the *tag* of the task, m is the *target mail address*, and k is the *message value*, is represented in the metaconfiguration as a

meta-task $< u, m_\theta,$ `[:task` $\uparrow t \uparrow m \uparrow k$ `]` $>$, where u is the task tag of the meta-task, m_θ is the mail address of the *task handler Actor* θ, and $\uparrow$ is the *handle function*[5]. Actors in the base-level have their specific information stored at the meta-level *database entry Actors*, which are accessed via the meta-level *database Actor* δ^S. The framework is conceptually illustrated in Figure 11.2. We formally prove that $\uparrow S$ is a correct representation of S by showing that the transitions in $\uparrow S$ faithfully represent transitions in S. The difficulty is that the atomicity of a transition in the base level no longer holds at the meta-level; we overcome this with a technique called *normalization*. For details, see [118].

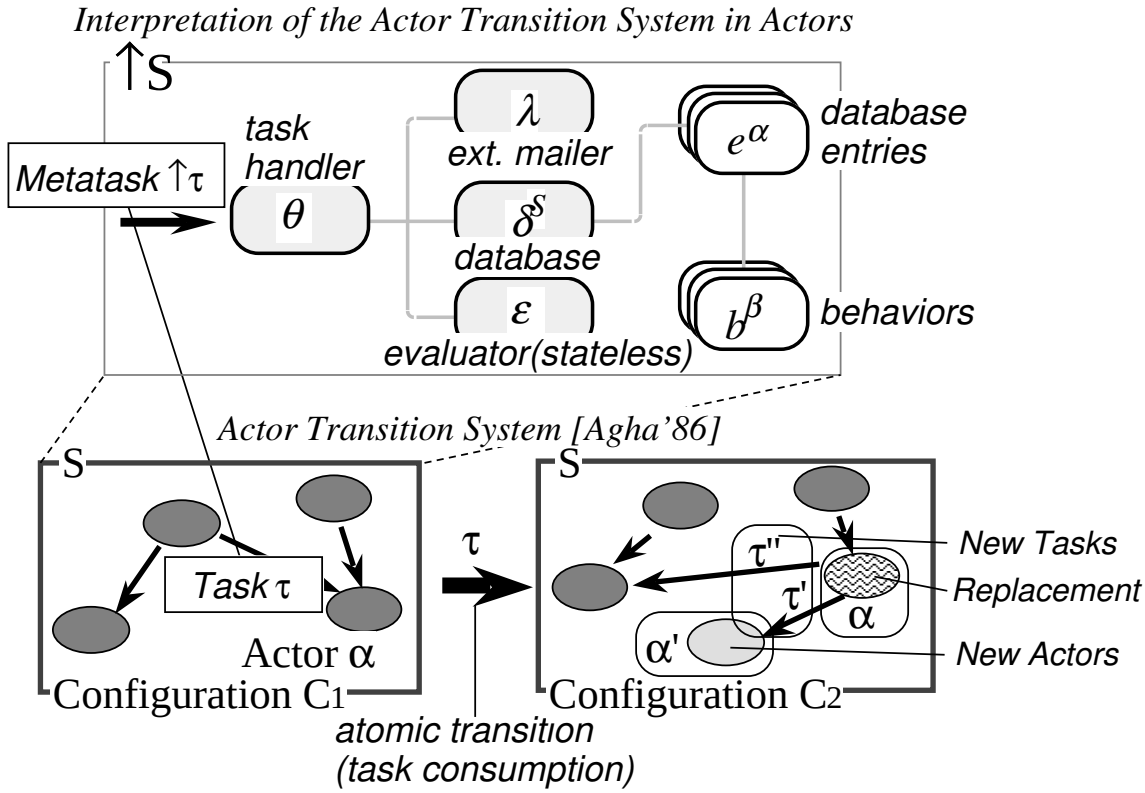


Figure 11.2: The Group-Wide Architecture in ACT/R

The essence of ACT/R is the lack of metaobjects, for the behavior of a single Actor is realized at the meta-level by coordinated action of multiple meta-level Actors. All reflective operations are performed solely via message sends, which are interpreted at the meta-level concurrently with interpretations of Actors in the base level. We claim that there are no metaobjects in ACT/R, although one could argue that the database entry Actors are so, for there is one entry Actor for each handle of the base-level Actors. They do not, however, satisfy our criteria, for (1) the denotation is not direct, and (2) the database entry Actor is not solely responsible for the computation of the corresponding Actor in the base level, e.g., the message reception is the responsibility of the task handler Actor.

[5]Not to be confused with the metaobject notation in ABCL/R.

Another example of the group-wide architecture, is the Muse (now Apertos) distributed operating system[124, 123, 122], although it has some features of the hybrid group architecture we discuss below. In Muse, the meta-level objects in the 'meta-space' 'support' the activity of the objects of the base-level. There are specific meta-level objects responsible for various operating system tasks, such as message delivery, scheduling, memory management, etc. The reflective operation is performed by communicating with the meta-space via communication ports called *reflectors*. Muse is not a programming language, nor has an underlying monolithic base language, as is true for Genera[103] (which is Lisp-based). The system is nevertheless reflective, in that Muse has its own object model that can be realized by multiple object-oriented programming languages. Because of its language independence (being a general-purpose operating system), features that are highly language dependent, e.g., environments, are not handled in the Muse object model. However, Muse does support primitives for constructing some types of language features that are essential for the manipulation of the Muse object system, such as classes.

11.2.3 Limitations of Both Architectures

We have seen that the the structural aspect determines the computational aspect of the architecture to some degree; so let us focus on the structural aspect. We have identified two distinct architectures: the individual-based architecture (ABCL/R, etc.) and the group-wide reflective architecture (ACT/R, etc.). Both architectures are lucid — both have clear metacircular definitions, and the latter in particular has a clean operational semantics based on Actor-based transition systems. The question is, can the architectures remain linguistically lucid in practice, e.g., when applied to resource management? We identify their limitations and claim that an architecture that combines their benefits (i.e., their *hybrid*) is necessary for practical use:

Limitation of Individual-based Architectures

Individual-based architectures are effective for introspecting or altering the behavior of a single, designated object. The limitation of individual-based architectures is that it lacks the 'global view' of computation. Each metaobject is self-contained in a sense that it only controls the computation of its denotation; other objects can only be indirectly introspected or affected through their respective metaobjects. Thus, implicit coordination among the group of objects become difficult.

Limitation of Group-wide Reflective Architectures

One of the purposes of group-wide reflection is to overcome the limitations of individual-based architectures, supporting

1. object group and group communication,
2. dynamic modification of message delivery semantics, and
3. implementation modeling (e.g., machine boundary inherently forms a group).

The group-wide reflective architectures resolve the limitation of individual-based architecture to some degree; However, the limitation of group-wide architectures is that the identity of a base-level object is lost at the meta-level, i.e., identity is not intrinsic to the meta-level, but is implicit. To perform a reflective operation on a particular object, the identity of the object must be constructed explicitly from dispersed objects of the meta-system. As a consequence, what is natural with the individual-based architecture become difficult, for (1) explicit programming is required, and (2) causal connection is expensive to maintain, because it is difficult to obtain the true representation of the current state of the object in the meta-level due to the time delays in the message sends and the concurrent activities of the other parts of the meta-system.

Limitation of Both Architectures

Furthermore, both architectures lack the inherent notion of bounded resources, that is, computation basically proceeds in the presence of an unbounded number of objects representing resources, which would be bounded in real-life. For example, for the individual based, the infinite reflective tower can be constructed for all the objects in the system; for the group-wide, the amount of computation increases by the order of magnitude for the meta-level interpretation, but this is absorbed in the increased parallelism inherent in the basic Actor formalism.

To summarize, we need an architecture that combines the benefits of both architecture, and allow implementation in a practical way. In the next Chapter, we propose such an architecture, and a language based on the architecture, called ABCL/R2.

Chapter 12

ABCL/R2: A Hybrid Group Architecture Language

In order to overcome the limitations, we propose an amalgamation of both architectures, called the *Hybrid Group (Reflective) Architecture*, and a language based on the architecture, ABCL/R2.

In order to preserve the explicit identity and structure of objects at the meta-level, we maintain the tower of metaobjects in the same manner as the individual-based architecture. For coordinated management of system resources such as computational resource, we introduce object groups, whose meta-level representation is a group of meta-level objects that are responsible for managing the collective behavior of its member objects. The conceptual illustration of the resulting architecture is given in Figure 12.1. Note that there are two kinds of reflective towers, the *individual tower* for individual objects and the *group tower* for groups: the details will be described in the ensuing sections.

The hybrid group architecture does not merely combine the benefits of both architectures; the key benefit is that it is possible to model coordinated resource management which were otherwise difficult for previous OOCR architectures. Another merit is that it is more readily implementable as compared to pure group-wide architectures, because much of the execution of individual objects can be confined within each metaobject, and only parts of computation that are necessary for group-wide behavior need to be sent to the meta-level objects of the group. Thus, the hybrid architecture (and ABCL/R2) is more applicable to practical problems as we will show. Finally and of most importance is that our architecture maintains the linguistic lucidity of reflection—both the individual-based behavior and the group-wide behavior can be described in terms of its own language (we define a metacircular interpreter, which is executable in real-life).

In order to combine the two architectures, there are two possible approaches, depending on which architecture to employ as a base. In ABCL/R2, we choose the individual-based architectures as a base, and incorporate features of group-wide reflective architectures. The inverse approach is also currently under study.

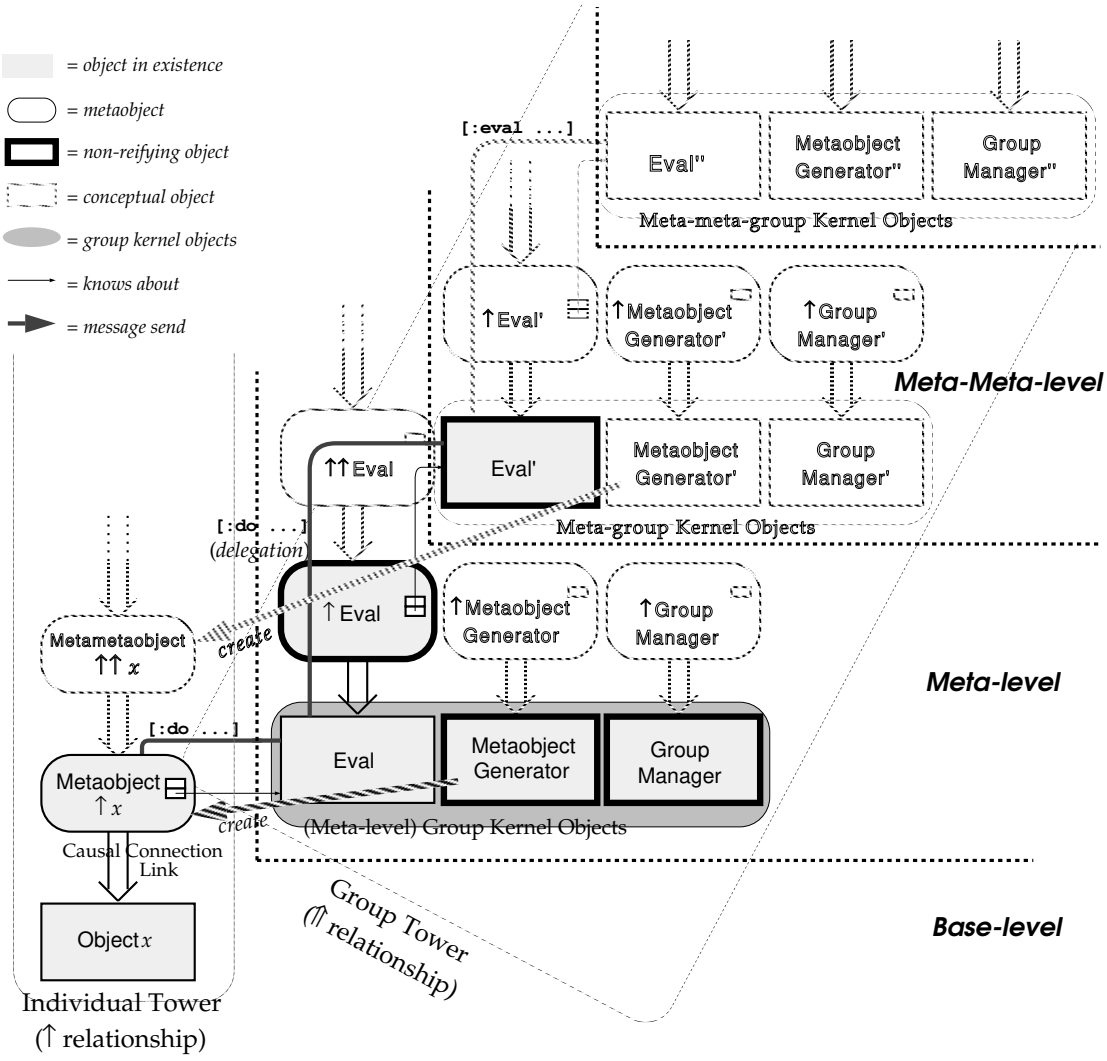


Figure 12.1: The Individual and Group Reflective Towers in ABCL/R2

12.1 ABCL/R2: a Language Based on Hybrid Group Architecture

ABCL/R2 is our prototype OOCR language with the hybrid group architecture. It is a direct descendent of ABCL/R: each object x has its own meta-object $\uparrow x$, i.e., the unit of CCSR of an object is its metaobject, in which the collective behavior of a set of objects(actors) is implemented by a group of objects at the meta-level. Also, as in ABCL/R, (1) the message reception and evaluation may proceed concurrently, preserving the ABCL/1 semantics, and (2) conceptually, there is an infinite structural (object-metaobject relationship) reflective tower per each object; the infinite meta-regression is resolved with lazy creation of the metaobject on demand.

12.1.1 Object Groups in ABCL/R2

The prime new feature of ABCL/R2 is the *heterogeneous object group*, or *group* for short. Members of a group participate in group-wide coordination for the management of system resources allocated to the group. Of special importance is the management of sharing of computational resources, which corresponds to scheduling in concurrent systems.

An object in ABCL/R2 always belongs to some group, with `Default-Group` being the default. A newly created object automatically becomes a member of the *same* group as its creator by default. The restriction is that an object cannot belong to multiple groups simultaneously. At the base-level, the structure of a group is flat in the sense that there are no base-level member objects which perform tasks specific to the group. Rather, analogous to the group-wide reflection, the structure and the computation of a group is explicitly defined at the meta- and higher levels of the group, by the objects called the *group kernel objects*. The group performs management of resources by coordinating among the metaobjects of the members and the group kernel objects.

Groups can be created dynamically with the group creation form as shown in Figure 12.2. The creation process of a group is not intrinsic, but is given given a concrete metacircular definition with ABCL/R2. As a result, not only that we have the tower of metaobjects, but we also have the tower of *meta-groups* as in ACT/R. We defer the details of group creation until Section 12.1.2.

The Group Kernel Objects

The group kernel objects are the CCSR of the group and its management. They are identified with a dark shaded area in Figure 12.3:

- The *Group Manager* — represents and 'manages' the group. When a group is created, the identity of the group is actually that of the group manager object. It has two state variables holding the mail addresses of the primary evaluator and the primary metaobject generator of the group, but there are no inverse acquaintances. It also embodies the definition of itself for the creation of new groups. Its definition will be described in Section 12.1.2.

```
[group Group-Name   ;;; Group Definition.
  ;; a metaobject generator (required)
  (meta-gen Metaobject-generator)
  ;; a primary evaluator (required)
  (evaluator Evaluator)
  ;; additional resources (optional)
  (resources
     [name := expression]
     ;; example: a scheduler
     [scheduler := [scheduler-gen <= :new]]
              :
     )
  ;; extra (user definable) scripts definitions (optional)
  (script
    ;; Example: reflective operation to allocate more computational power on request
    (=> [:give-me-more-power Priority]
        ;;; Compute how much computational power can be given to the object.
        ;;;   Assume that the evaluator is extended to have a scheduler.
        [scheduler <= [:give-more-power-to sender computed-amount]])
              :
    )
  ;; initialization expressions
  (initialize
     Initialization-Expressions...
     ;; example: define an initial member of the group
     [object Root ...]
              :
    )
  ;; initialization for metalevel actors (set-up purpose)
  (initialize-meta
    Initialization-Expressions-for-Metalevel...
    ;; example: notify the initial scheduler
    [[meta evaluator] <= [:set-scheduler scheduler]]
              :
    )]
```

Figure 12.2: Group Definition in ABCL/R2

- The *(Primary) Metaobject Generator* — serves as the primary generator of the metaobjects for each member of the group. When a new object is created in the group, its metaobject is always created at the same time. The default metaobject generator of the system, `Metaobject-Generator`, is shown in Figure 12.4.

- The *(Primary) Evaluator* — represents the shared computational resource of the group. It purpose is to evaluate the methods of member objects. It is no longer a stand-alone, private object as in ABCL/R, but interacts with other group kernel objects and metaobjects for group management. The default definition is given in Figure 12.5. There could be multiple evaluators per group for parallelism.

Each metaobject of a group member object has state variables in its metaobject containing the mail addresses of group kernel objects for group membership; one is (the address of) the group manager, and the other is the primary evaluator (Figure 12.3). The group manager object can be accessed from the base-level with the special form `[group-of ...]`. Conversely, the metaobject generator is not directly known to the metaobjects of the group.

The Meta-Group

The group kernel objects are not members of the group they manage, because they reside at the meta-level of the member objects. But since the requirement that all objects belong to a group holds for group kernel objects as well (thus be able to compute in the first place), the *Meta-group* must exist to maintain the linguistic lucidity of reflective architectures. This is similar to ACT/R, where meta-actors comprised the meta-level group. In Figure 12.1, the metagroup of group G is identified as $\Uparrow G$. The group kernel objects are members of group $\Uparrow G$, while the metaobjects of the group kernel objects are members of $\Uparrow\Uparrow G$, etc. Here, the $\Uparrow$ tower forms a *group tower* distinct from the metaobject towers; we will discussed this in detail in Section 12.1.4.

In addition to the group kernel objects, there could be other meta-level objects that are members of the meta-group (or higher). In Figure 12.3, for example, the metaobjects of the evaluator and the group manager, in addition to the *scheduler* object, are members of the meta-meta-group of the base group.

Group Shared Resources

The member objects of a group share *group shared resources*, which are the CCSR of system resources, such as computational resource. The group-wide coordination of resource sharing by the member objects is controlled by the metaobject of each member object, with the aid of the group kernel objects. Coordination of sharing is thus done at the meta-level, and is basically invisible at the base-level. The homogeneity of metaobjects with respect to such coordinated behavior is guaranteed by the metaobjects being generated by the (primary) metaobject generator, which is unique to a group.

By default, the evaluator object is the CCSR of the shared computational resource of the group. By coordinating the evaluation with the metaobjects and the *scheduler* object, we can allocate more computational resources to certain objects in order to achieve

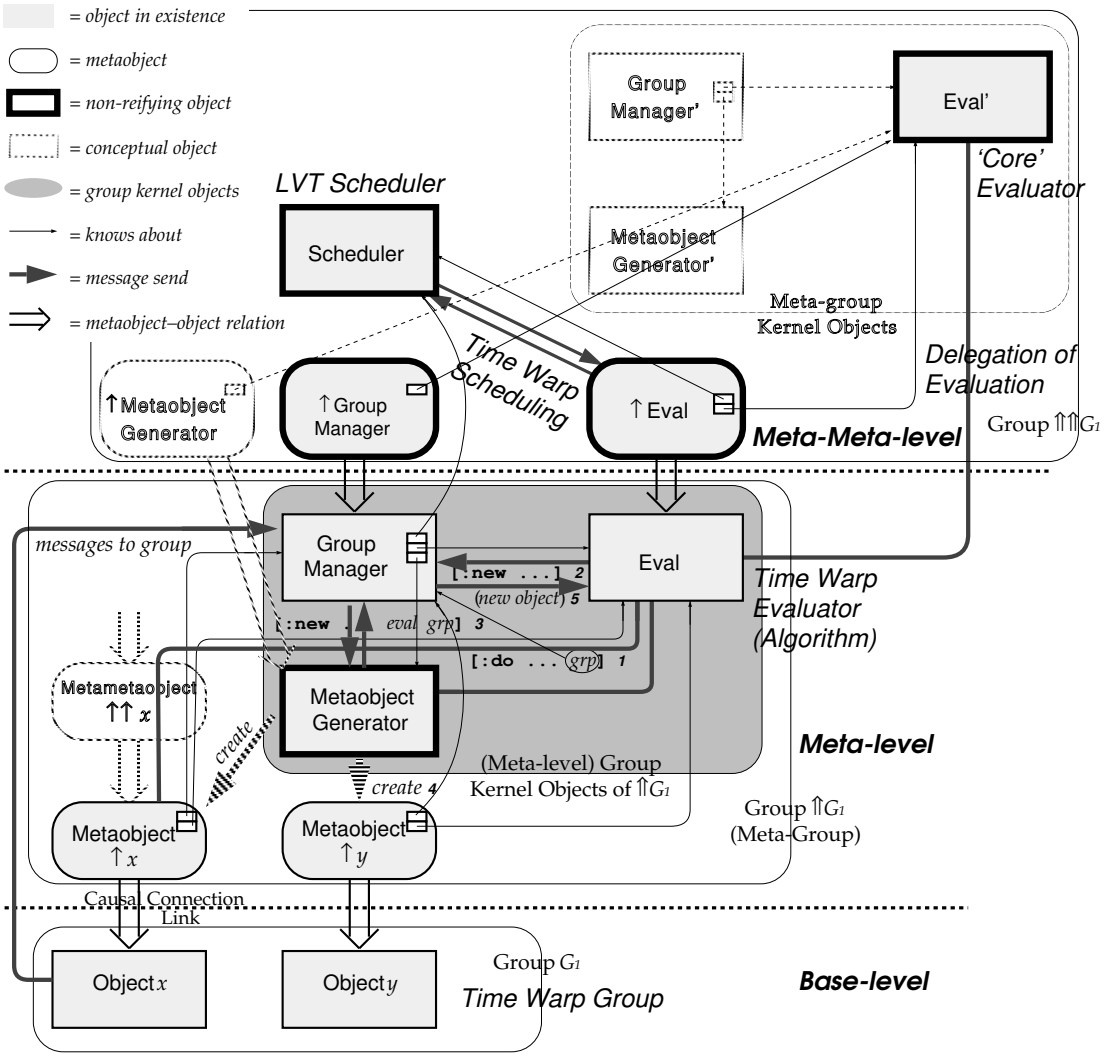


Figure 12.3: Reflective Architecture of ABCL/R2

```
[object Metaobject-Generator   ;; The 'Vanilla' Primary Metaobject Generator
 (script
  (=> [:new StateVars LexEnv Scripts Evaluator GMgr]
      ![object Metaobject   ;; The name 'Metaobject' is local to this method.
        (state [queue := [queue-gen <= :new]]
               [state := [env-gen <= [:new StateVars LexEnv]]]
               [scriptset := Scripts]
               [evaluator := Evaluator]
               [Group := GMgr]
               [mode := ':dormant])
        (script
         (=> [:message Message Reply Sender]
             [queue <= [:enq [Message Reply Sender]]]
             (when (eq mode ':dormant)
               [mode := ':active]
               [Me <= :begin]))
         (=> :begin
             (match [queue <= :deq]
              (is [Message Reply Sender]
                  (match (find-script Message Reply scriptset)
                    (is [Bindings ScriptBody]
                        [evaluator <=
                          [:do-progn
                              ScriptBody [env-gen <== [:new Bindings state]]
                              [den Me] GMgr] @
                          [cont _
                           [Me <= :end]]])
                    (otherwise
                     (warn "~S cannot handle the message ~S"
                           [den Me] Message))))))
         (=> :end
             (if (not [queue <== :empty?])
               [Me <= :begin]
               [mode := ':dormant]))
         ;; Methods implementing reflective operations.
         ;; see [117, 127] for details.
         (=> :queue
             !queue)
                 :
         )])
  )]
```

Figure 12.4: Primary Metaobject Generator in ABCL/R2

```
[object Eval    ;; The Primary Evaluator — computational resource for the group.
 (script
  (=> [:do Exp Env Id Gid] @ C    ; Evaluation for a single expression.
      (match (parse-exp Exp)
       ;; Variables
       (is [:variable Var]
           (match Var
            (is 'Me ![den Id])    ; pseudo variable Me
            (is 'Group !Gid)      ; pseudo variable Group
            (otherwise [Env <= [:value-of Var] @ C])))
                          :
       ;; Past-Type Message Transmission
       (is [:send-past Target Message Reply]
           [Me <= [:do-evlis [Target Message Reply] Env Id Gid] @
            [cont [target* message* reply*]
             [C <= nil]
             (if (not (null target*))
               [[meta target*] <= [:message message* reply* [den Id]]])]])
       ;; Now-Type Message Transmission
       (is [:send-now Target Message]  (similar to above, omitted) )
                          :
       ;; Object Creation
       (is [:object-def Name Meta-gen-spec State Script]
           [Me <= [:do Meta-gen-spec Env Id Gid] @
            [cont meta-gen*
             (if (null meta-gen*)
               ;; if a metaobject generator is not explicitly specified, use default.
               [Gid <= [:new State Script] @
                [cont object
                 (if Name
                   [Env <= [:set Name object] @ C]
                   !object)]]
               [meta-gen <= [:new State Env Script Me Gid] @ C])]])
             :
       )
  ;; composite evaluation messages
  (=> [:do-progn (FirstExp . RestExps) Env Id Gid] @ C
      [Me <= [:do FirstExp Env Id Gid] @
       [cont first* [Me <= [:do-progn RestExps Env Id Gid] @ C]]])
  (=> [:do-progn (LastExp) Env Id Gid] @ C
      [Me <= [:do LastExp Env Id Gid] @ C])
        :
  )]
```

Figure 12.5: Primary Evaluator in ABCL/R2

higher execution efficiency. Other shared resources could be defined for the purpose of either adding new functionalities to the group, and/or making the implementation details manifest in order to alter some existing behavior. For this purpose, the user specifies a specialized metaobject generator. Examples currently under study include class database objects for distributed class management.

12.1.2 Object and Group Creation in ABCL/R2

In ABCL/R2, the dynamics of object and group creation are manifest in the language; otherwise, the user would not be able to define more sophisticated groups tailored for his particular program and/or hardware architecture. Such extensibility via reflection distinguishes our work from previous works in object or process groups (see [59] for a survey), in which the functionalities of the group, especially its creation, were hard-wired and inalterable.

Object Creation Process

The default group membership of a newly created object is the same as its creator. This can be overridden, however, by explicit designation of a group in the object creation form. Similarly, a specialized metaobject for a particular object can be specified, overriding the primary metaobject generator of that group:

```
[object  object-name
    (meta  metaobject generator)      ;; optional, specify alternative metaobject
    (group  group)                    ;; optional, specify alternative group
    ...
```

For example, the user might specify that the new object z be created in group G_2. This is simply achieved by the evaluator sending the `[:new ...]` message to the group manager of G_2, ignoring the default G_1 passed in the parameter of the `[:do ...]` message in Step 1. Again, this is a (intended) consequence of the reflective architecture, and not a predefined group behavior.

We show that the behavior results from our reflective architecture. Rather than to force the reader to follow through the details of the code, we give intuitive descriptions of the process of object x creating a new object y in group G_1, as illustrated in the lower part of Figure 12.3. The labeled message sends in the figure is numbered in correspondence to the explanations below:

1. Object x receives a message, and attempts to evaluate the corresponding script of x which contains an object creation form `[object ...]`. At the metaobject level, the script, the new environment, and the group (represented by the group manager) are sent to the evaluator object in the `[:do ...]` message (labeled 1 in Figure 12.3).

2. When the evaluator encounters the `[object ...]` form in the script, it sends a `[:new ...]` message to the target group manager object. If there is an explicit group specification with `[group ...]`, then the target becomes the group manager

object of the specified group; otherwise, the target is the group manager that was passed as a parameter in the `[:do ...]` message in Step 1, causing the group of the new object y to be the same as x (i.e., G_1). In both cases, the evaluator passes the specialized metaobject generator to the target metaobject generator in the message if and only if the object creation form has the `(meta ...)` option (labeled 2).

3. The group manager sends the `[:new ...]` message to the metaobject generator. If a specialized metaobject generator is passed in the message from the evaluator, that becomes the target; otherwise, the target is the primary metaobject generator of G_1. In both cases, the group manager passes the evaluator and itself in the message so that the new object would become the member of G_1 (labeled 3).

4. The metaobject generator creates the metaobject of the new object y, and returns it to the group manager of G_1. This metaobject creation would be interpreted as a creation of an object at the base-level due to the causal-connection property (labeled 4).

5. The group manager returns the new metaobject $\uparrow y$ to the evaluator, which in turn returns it to $\uparrow x$; this is interpreted at the base-level as x receiving the new y. In the default case as illustrated in Figure 12.3, y belongs to G_1 — as a result, it shares the computational and other resources with x and other members of G_1 (labeled 5).

Dynamic Group Creation and Bootstrapping

A group is created dynamically at run-time with the evaluation of the group creation form in Figure 12.2. The name of the group is global; it actually refers to the group manager object. The two required objects are the primary metaobject generator and the primary evaluator of the group. Other shared resources are optional and are defined in the `(resources ...)` form. Next are the optional user-definable scripts of the group; they become the scripts of the group manager upon group creation. In Figure 12.2, the user defines a reflective method `:give-me-more-power` which allocates more computational resource to an object upon request. Finally, there are two initialization forms of the group: the former is the object-level initializer, whose prime purpose is the creation of the initial *fixed members* of the group; the latter is the meta-level initializer for initializing the meta-level objects of the group. Due to the dependency between the meta-level and the base-level objects, the latter is evaluated prior to the former.

The initial bootstrapping of a group is achieved in a manner similar to object creation: when the group creation form is detected, the evaluator object sends the group creation message to the group manager object. The group manager object then evaluates the object creation form (outlined in Figure 12.6) for the group manager of the new group. It roughly proceeds as follows: The mail addresses of the group shared resources are stored in the state variables, and the user-defined scripts of the group is merged with the default scripts of the new group manager. The generated initialization script includes the base-level and the meta-level initialization forms; the former is to be sent to the evaluator of the new group, while the latter is to be executed directly by the new group manager

(and is thus evaluated by the meta-group evaluator). The newly created group manager object is then sent the `:initialize` message to start the above initialization sequence.

When an object of group G_1 creates a group G_2, the $\Uparrow G_1$ is identical to $\Uparrow G_2$; in other words, the group kernel objects of G_2 becomes a member of the same group as those of G_1.

12.1.3 Non-reifying Objects

Another new feature of ABCL/R2 is the *non-reifying* object, whose purpose is to attain higher efficiency at the sacrifice for the loss of reflective capabilities. It is created with the following form:

```
[object object-name
    (meta non-reifying-meta)
    ...
```

The behavior of a non-reifying object is almost the same as that of a standard object. The difference is that reflective operations are disallowed — an attempt would result in an error.

The non-reifying object does not have a metaobject, i.e., the metaobject is only of 'hypothetical' existence, prohibiting actual reference to it within the script. (It is possible to have references to metaobjects of other standard objects.) Extensibility, as a consequence, is lost; however, non-reifying objects execute much more efficiently compared to the standard ones. In Figures 12.1 and 12.3, non-reifying objects are illustrated with thick borders; notice that they do not have metaobjects that actually exist. In our prototype implementation, a non-reifying object is actually an ABCL/1 object that mimics the interface of metaobjects in ABCL/R2. It runs faster compared to native ABCL/R2 objects due to optimized message handling.

As we see in the figures, some of the meta-level objects are not fully instantiated. This is because the members of the meta-group usually assume default behaviors identical to the ABCL/R objects, and the system thus can avoid the infinite meta-regression using standard techniques. However, the metaobject of the evaluator, for example, is instantiated; this requires that the evaluator of the meta-group be instantiated in order to process *its* evaluation. Other meta-level objects, such as the metaobject generator of $\Uparrow\Uparrow G$, is instantiated lazily at run-time when a new group is created.

In a standard group, the the group manager and the metaobject generator are defined as non-reifying for (1) efficiency of group-wide coordination management, and (2) they usually do not require reflective capabilities. We make a note that, reflection *IS* possible when required for meta-level objects of the group; an example of this is the evaluator object in the next section.

12.1.4 The Two Kinds of Reflective Towers

Before proceeding, let us discuss the relationship between the individual tower and the group tower. The term 'tower' implies that there are some kind of structural relationships

```
[[object Group-Manager-Name
  (state [meta-gen := Metaobject-Generator]
         [evaluator := Evaluator]
         [name := expression]                ;; additional resources
         [scheduler := [scheduler-gen <== :new]]  ;; the scheduler example
                :
        )
  (script
   ;; initialization (bootstrap) method (automatically generated)
   (=> :initialize
       ;; initialization for metalevel can be directly executed by itself
       Initialization-Expressions-for-Metalevel...
       [[meta evaluator] <= [:set-scheduler scheduler]] ;; example
                :
       ;; initialization codes must be sent to the new primary evaluator
       ;; Id is unspecified, (for there are no members)
       [evaluator <= [:do-progn '(Initialization-Expressions) global-env NIL Me]]
       )
   ;; default group manager methods (automatically generated)
   (=> :meta-gen
       !meta-gen)
   (=> :evaluator
       !evaluator)
   (=> [:new StateObj Env Script] @ C
       [meta-gen <= [:new StateObj Env Script evaluator Me] @ C])
   (=> [:new StateObj Env Script SpecialEvaluator] @ C
       [meta-gen <= [:new StateObj Env Script SpecialEvaluator Me] @ C])
                :
   ;; user-defined methods: (the :give-me-more-power example)
   (=> [:give-me-more-power Priority]
       ;;; Compute how much computational power can be given to the object.
       [scheduler <= [:give-more-power-to sender computed-amount]])
                :
   )]
<= :initialize]
```

Figure 12.6: Dynamic Group Creation and Bootstrapping

between the computations at each level. For example, the reflective tower in LISP is the tower of evaluation[100]. The state of the computation at level n can be given as a triplet data structure $< \mathit{expression}, \mathit{environment}, \mathit{continuation} >$ at level $(n-1)$, which can be reified/reflected at each level.

The reflective tower of an ABCL/R object is an individual tower of object-metaobject relationship $\uparrow$, where each metaobject solely determines the structure of the object it denotes. This is also the case for ABCL/R2as we have seen.

In addition, in ABCL/R2, the meta-group relationship $\Uparrow$, which is structural, forms the group tower as illustrated in Figure 12.1. This tower parallels the meta-evaluation relationship, which is computational, in the following way: Let G_x be the group of a given object x, and E_x (labeled Eval) be the primary evaluator of G_x, which is a member of $\Uparrow G_x$. Since E_x is an object itself, it needs some computational resource, provided by the primary evaluator E_{E_x} (labeled Eval') of the group G_{E_x} to which it belongs. This group is the meta-group of $\Uparrow G_x$, i.e. $\Uparrow\Uparrow G_x$. Now, E_x has a metaobject, $\uparrow E_x$ (labeled $\uparrow Eval$). In our current architecture, we define this object to be also a member of $\Uparrow\Uparrow G_x$, so that the evaluations of both E_{E_x} and $\uparrow E_x$ are performed by the evaluator of the group $\Uparrow\Uparrow G_x$ (labeled Eval"). This forms a homogeneous tower-like structure of meta-groups as seen in the figure.

The above indicates that the reflective towers might not be solely in the direction of the $\uparrow$ relationship, but also in the direction of the $\Uparrow$ relationship. Since this was not manifest in the previous OOCR architectures, we attempt to place some distinctions between them:

- The individual tower mainly determines the structure of the object, including its script. Thus, reflective operations to alter the script is in the domain of the individual tower.

- The group tower mainly determines the group behavior, including the computation (evaluation) of the script of the group members. Thus, changes to have different *interpretations* of the same script are in the domain of the group tower.

The above distinctions correspond to the issue whether a reflective operations would affect the program itself, or affect the *interpretation* of the program. However, we cannot merely say that the individual tower only represents structure, and the group tower only represents computation; for example, we could modify the metaobject via the meta-metaobject so that it would suddenly deadlock after receiving k messages. We need to establish more sound conceptual distinctions, and develop the model into a formal one, as has been done for single reflective towers for LISP in the works by Friedman and Wand[116] and by Danvy and Malmkjær ([36], etc.). We establish a formal model for groups for the language ABCL/Rμ, which is a similar but a much smaller language, in Chapter 15.

Now, there is a choice in the construction of the individual tower of member object x: it can be made either distinct or parallel to the group tower. This corresponds to the issue of the *group membership of metaobjects*. We have deliberately avoided the discussion up to this point, because the membership is dependent on the scheme whereby the meta-circularity is broken. In the current implementation of ABCL/R2, the lazy creation of an

metaobject is achieved with *self-reifying metaobject*. An regular ABCL/R2 object x (i.e., not non-reifying), upon evaluation of the form `[meta` x`]`, does not cause the creation of a new meta-metaobject $\uparrow\uparrow x$. Rather, the self-reifying metaobject is essentially 'raised' to become the meta-metaobject, and it is properly initialized so that it would serve as the metaobject of $\uparrow x$. This behavior is unfortunately not currently manifest as CCSR in the current implementation. As a result, the entire tower becomes a member of $\Uparrow G_x$, except for the base level x, as shown in Figure 12.3. Here, the individual tower is distinct from the group tower, in the sense that there are no correspondences between the meta-levels.

Can the architecture be practically implemented? Before outlining the efficient implementation scheme of the compiler and the runtime system in Chapter 13, we note the relevant topics here: In order for the two towers to be parallel, the metaobject of level n needs to be created by the metaobject generator of the corresponding meta-group of level n. This requires a lazy creation scheme for the group kernel objects upon metaobject creation, which again is not manifest as CCSR in the current implementation. The infinite regression of the meta-group manifests itself, for example in some of the scheduling examples we will describe in Chapter 14: there, the metaobject of the primary evaluators delegates all the evaluation tasks to the evaluator of the meta-group. This regression is supposed to terminate as it was in the example, although it could conceptually continue further on, upon which nothing would ever be computed! Thus, by lazy creation some 'ground' meta-group must be created in the real implementation, and there (at least conceptually) the evaluator on termination would correspond the native CPU hardware; in fact, since it receives the delegated computations from all levels, it would in effect be the sole computational resource in the group[1]. This is a generalization of the conceptual model of practical reflective systems such as Apertos[124, 123, 122].

[1]This would hold even for a set of evaluators.

Chapter 13

Efficient Implementation of ABCL/R2

13.1 Problems in Achieving Practical Efficiency in OOCR Languages

So far, we have described the language model based on the operational semantics via a metacircular definition. Of course, this is insufficient if we expect to have real-life model and language.

The major challenges of reflective languages and systems in general have been (1) how to implement it, and (2) how to do so efficiently. In order to maintain the causal-connection, the semantic model of a reflective language is usually given with an infinite tower of metacircular interpreters, as we have also done. Since it is impossible to literally implement the 'infinite' tower, most implementations 'bottom out' at some level with non-reflective version of the language, but this scheme is inefficient when the higher levels are merely interpreted, so the increase in the number of level of interpretation would result in prohibitive execution cost for the entire execution of the base-level language.

The novel idea in the initial work of 3-Lisp by Rivières and Smith[38] was to employ a single interpreter and 'shift up' only when reflective procedures are invoked, instead of always performing metacircular interpretations. When the metalevel computation of the reflective procedure finishes, the execution 'shifts down' back to the base-level. Since then, there have been few efforts on efficient implementation, which have been (1) only in the context of sequential languages, and/or (2) drastically limited the reflective capabilities to simple ones that could be efficiently implemented, e.g., message dispatch, and/or (3) have only proposed the overall idea of how it might be possible, but without concrete methodologies or actual implementations [96, 26, 21, 35].

As far as we know, there have been no study on efficient implementation of concurrent reflective languages; this is probably due to the difficulty in obtaining the true CCSR of the current state computation due to inter-level concurrency and the time delays incurred in inter-level communication[101]. Thus, for example, Rivières-Smith approach is not immediately applicable, because it does not account for concurrency, i.e., it assumes that at any given time only one level is running. Furthermore, because their scheme

is still interpretive, the resulting language will not be able to compete with compiled non-reflective languages when running standard programs. Thus, an alternative way of efficiently breaking the meta-circularity in OOCR languages without sacrificing the lucidity of the reflection must be devised.

This chapter presents an efficient implementation scheme of ABCL/R2 which is applicable to other OOCR languages as well. Contrary to its predecessor, ABCL/R[117], whose implementation was extremely slow due to interpretation that bottomed out with ABCL/1[127], the current implementation of ABCL/R2 is compiler-based, is independent from ABCL/1, and in fact competes with the ABCL/1 compiler for speed. ABCL/R2 now runs on various Common Lisp platforms on Workstations and PCs, and a true parallel version runs on top of a parallel Common Lisp on OMRON LUNA-88K, a shared-memory Mach machine. Benchmarks have shown that the execution speed of ABCL programs on ABCL/R2 (1) is nearly or over two-orders of magnitude faster than that on ABCL/R, and (2) closely compares with or even exceeds the speed on the publically distributed version of our ABCL/1 compiler[127] and also C + Sun Light-Weight Processes (LWP) library, neither of which support reflection.

The schemes we have developed for efficient implementation of OOCR languages are: (1) Efficient Lazy Creation of Metaobjects and Meta-groups, (2) Partial Compilation of Scripts (Methods), (3) Dynamic Progression of Degree of Reflectivity, (4) Self-Reification of Group Kernel Objects, (5) Non-reifying Objects (User-level) and Light-weight Objects[1]. We have also managed to *integrate* the optimized components of the system, so that the system remains faithful to the meta-circular definition of ABCL/R2[70]. For example, compilation coexist with reification, and objects of various degrees of reflectivity can send messages to each other. The obvious test is whether the meta-circular definition of ABCL/R2 in Chapter 12 runs on top of our system, and in fact it does.

13.2 Implementation Issues of OOCR Languages

The first version of implementation of ABCL/R2 only supported a small subset of the features described in Chapter 12; in particular, the two biggest drawbacks were that (1) its semantics was not properly meta-circular, and (2) it was extremely slow, due to meta-interpretation. Based on this initial implementational experience, we had set out to investigate efficient implementation schemes for ABCL/R2 and other OOCR languages in general.

As introduced above, the two issues in efficient implementation of reflective languages are: how to break the meta-circularity safely, and how to do so efficiency. Since ABCL/R2 is based on the hybrid group architecture, the implementation must always guarantee the proper maintenance of the following causal-connection properties:

- Reflective operations on the individual tower, i.e., message sends to a metaobject to invoke reflective operations, and/or dynamic customization of a metaobject (via the meta-metaobject), must be reflected properly and solely to the object it represents (*localization*).

[1] We make a brief note that, in this study, we limit the scope of "efficient implementation" to reflection; traditional compiler optimizing techniques were also incorporated but are not presented here.

- Reflective operations on the group tower, i.e., message sends to the group manager object and/or dynamic customization of the group kernel objects, must be reflected properly to all members of group (at the base-level) of the group tower.

Moreover, the actual system must be finitely constructible and executable in real-life. Thus, the problem is, how do we safely break the meta-circularity of the two towers. Previous works have resolved the finiteness issue with lazy creation, but it becomes non-trivial when two towers intertwine; for example, one could reify the one of the group kernel objects, say, the evaluator. The problem here is that of meta-circularity, i.e., the metaobject of the evaluator is itself an object, and is thus a member of the *metagroup* of the base-level group. As a result, not only that the metaobjects along the individual tower of the evaluator must be created lazily, but the same reification must properly trigger the lazy creation of the group kernel objects of the metagroup as well.

Efficiency issues are more problematic. In order to achieve practical efficiency, the implementation cannot remain solely interpretive. Instead, it must include a *compiler* which emits code that allows reflection, thereby achieving practical efficiency. Too much compilation, however, would not allow for reflection to occur. Interfacing of reflective and non-reflective portions in compiled code is another problem; although mixture of interpretive and compiled code has been done on a per-function basis in Lisp compilers, our case is more intricate due to the existence of both (1) reflection and (2) concurrency. Finally, the object-level problem is how to integrate the mixture of group kernel objects and individual metaobjects and meta-metaobjects...etc., that have been subject to various optimizations such as lazy creation of the individual tower, so that they are able to send messages to each other and remain consistent with the metacircular definition of ABCL/R2.

The obvious optimization is to 'delay' full-fledged reflection as much as possible. For this purpose, we employ the notion of *dynamic progression of degree of reflectivity*[71], which extends the simple lazy creation technique. Unnecessary creation of metaobjects that would result with naive implementation is avoided; and when the time comes when a metaobject needs to actually exist due to explicit reference, we employ the *light-weight* metaobject, which allows more efficient execution at the sacrifice of reflective features. Only when full-fledged reflective operations become necessary, the light-weight metaobject is replaced by a regular object. Although such progression is performed automatically by the system, it nevertheless must be integrated with other parts of the system as well as the compiled code.

13.3 Implementation Scheme of ABCL/R2

Currently, a new version of ABCL/R2 is implemented on top of the multi-threaded version of Common Lisp, running on Omron Luna-88k, a shared-memory computer. As mentioned earlier, the execution speed of this version compares closely with, and sometimes exceeds that of the original ABCL/1 compiler, with appropriate user intervention.

Our implementation scheme is structured as follows: First there is an underlying *low-level kernel* that provides the basic primitives for (non-reflective) object execution. Specifically, it maps the underlying Lisp threads to objects. Secondly, the compiler

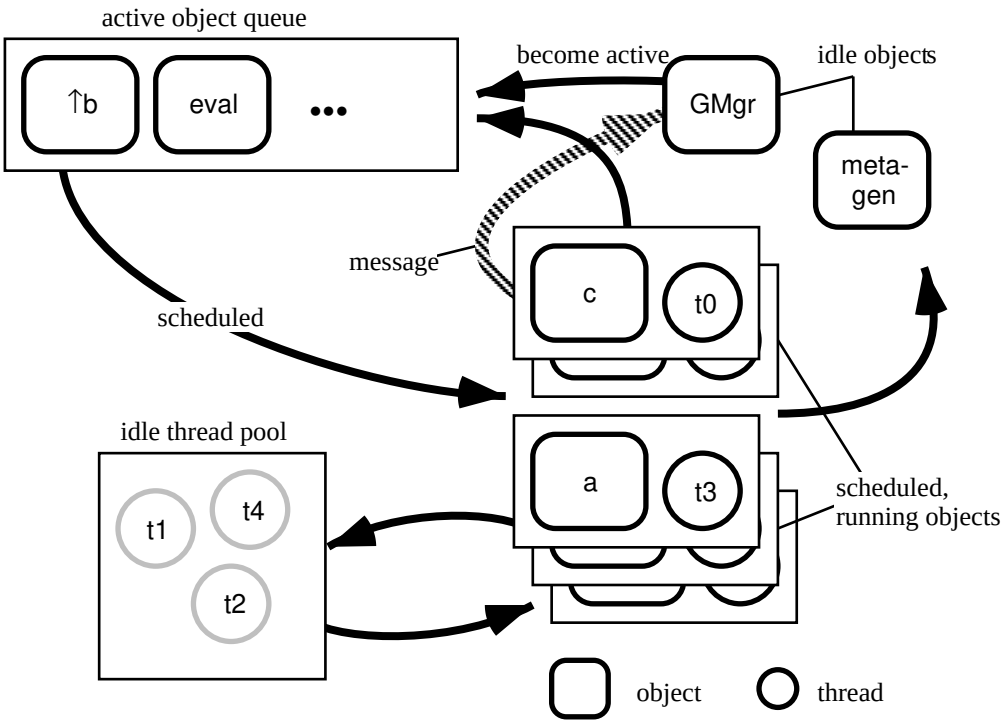


Figure 13.1: The Low-Level Kernel

performs *partial compilation* of scripts (i.e., methods). The compiled code is such that it allows co-existence of reflective and non-reflective code, so that execution efficiency is maintained while retaining reflective capabilities. Default system objects are further optimized by providing a pair of non-reflective and reflective compiled codes for *self-reification*. The compiled code is augmented with *light-weight objects*, that serves as fast replacements for normal objects in special roles such as *continuation objects* and *light-weight metaobjects*. The individual tower is constructed lazily with *dynamic progression* techniques, in which full-fledged creation of metaobjects is avoided until its capabilities are absolutely necessary. Interfacing of various objects with different degrees of reflectivity (including non-reifying objects, which are essentially fully compiled) become possible with *inter-level message forwarding*, which has the effect of avoiding unnecessary reification. Finally, lazy creation of meta-groups along the group tower is made possible with lazy creation of group managers and self-reification techniques.

13.3.1 The Low-Level Kernel

The *low-level kernel* has the task of providing the basic primitives for (non-reflective) object execution. Based on the underlying multi-threaded Lisp environment, Lisp threads are mapped to objects to provide primitive computational resources. It can be regarded as a simplified operating system kernel in that it provides primitive message transmission and scheduling capabilities. Figure 13.1 illustrates the structure of the low-level kernel.

For primitive message transmission, when a message is sent to an object, a triplet$\langle$*message body*, *reply destination*, *message sender*$\rangle$, is placed into the message queue of the receiver object. Then, if the object has been 'dormant', the object is made 'active' and placed into the *active object queue*.

The kernel employs a standard technique of creating a pool of fixed number of Lisp threads, from which a thread is acquired and assigned to objects for execution. A thread dequeues an object from the active object queue, then executes the compiled Lisp lambda-closure associated with the object. The lambda-closure describes the basic behavior of the object — namely, remove one message from its message queue, then execute the script matching the message pattern. After the execution of the script terminates, the object returns to the active object queue if it is still active (i.e., the message queue is non-empty), or becomes idle when it becomes dormant (i.e., the message queue is empty). The thread then starts the execution of another active object in the active object queue. The lambda-closures are predefined for the default group kernel objects and the default metaobjects, and produced by the compiler for non-reifying objects.

13.3.2 Partial Compilation of Scripts

The primary focus of reflection in ABCL/R2 is the ability to perform coordinated resource management, such as object scheduling and group consistency maintenance. For this purpose, 'what is to be reflected' concentrates on the concurrency aspect of the object-oriented concurrent computation; in other words, primitives in intra-object sequential computations — say, primitive arithmetic expressions — would serve little purpose if it could be reflected upon. (This is similar in spirit to CLOS Metaobject Protocol[58], where only the features added by CLOS, such as generic function dispatch, could be reflected upon, whereas Common Lisp features are basically hard-wired).

From such a perspective, we categorize operations into *reflective operations* and *non-reflective operations*. In ABCL/R2, the following operations can be reflected upon:

- Reference to the variables (including the references to lexical variables, and pseudo variables '`Me`' and '`Group`').
- Message sending (both 'past type' (asynchronous) and 'now type' (RPC-style)).
- Object creation and group creation.

As indicated above, other primitive operations such as arithmetic operations are not subject to reflection, and are thus compiled. Here, in order for reflective and non-reflective operations to co-exist, the following scheme is adopted:

- Consecutive expressions representing non-reflective operations are compiled into a simple Lisp lambda-closure (which is, in turn, compiled into native code by the Lisp compiler).
- Other expressions are also similarly compiled into a lambda-closure. The difference is that, when reflective operations appear within expressions that are non-reflective,

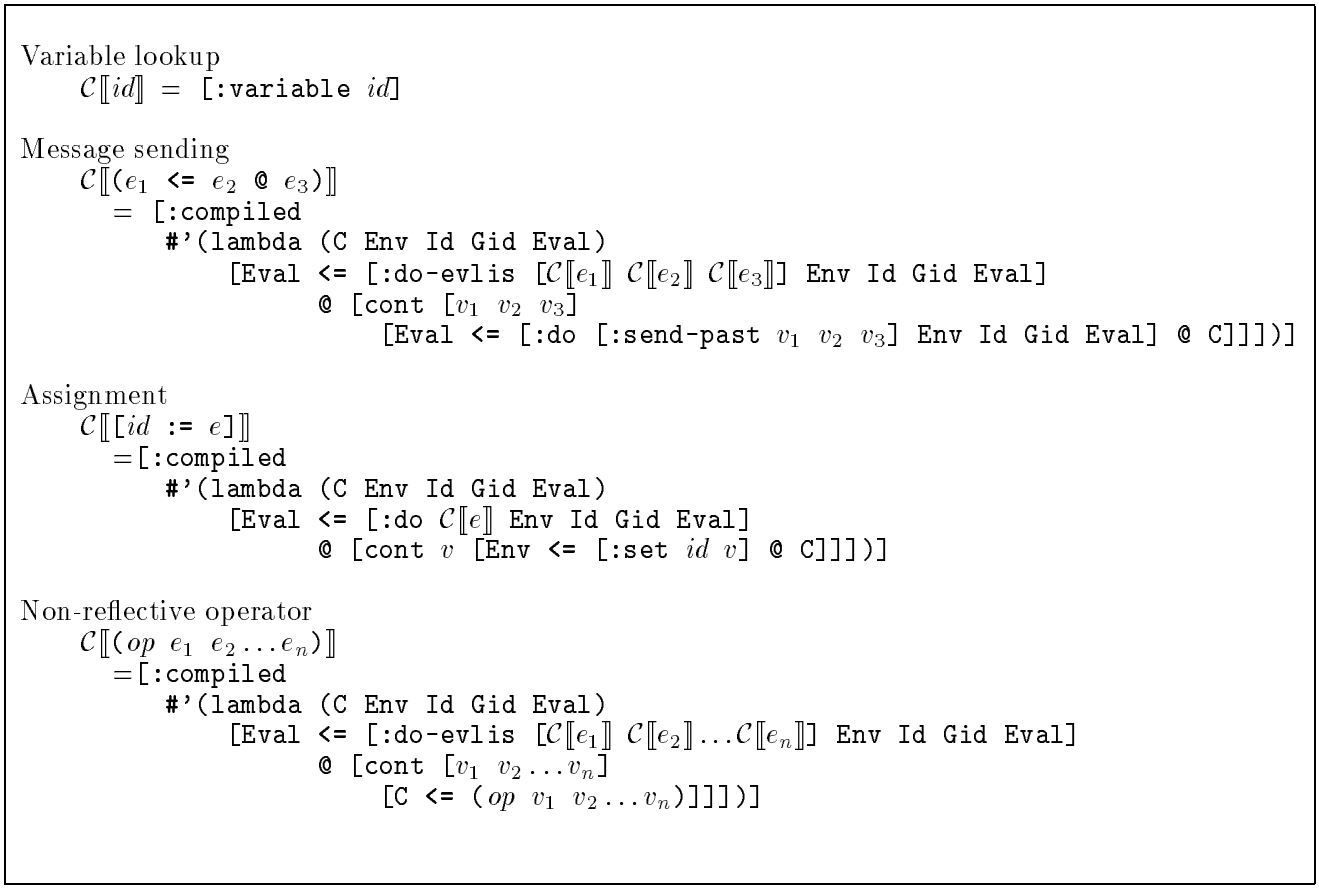


Figure 13.2: Abridged Partial Compilation Rule of ABCL/R2

a code to send a message to evaluator explicitly is embedded into the lambda-closure. The evaluator in turn receives the message and executes the reflective operations.

- Conversely, the evaluator can also receive a compiled lambda-closure as a message for direct execution. Thus, non-reflective operations embedded within reflective ones can be initiated by sending a message to evaluator.

Abridged compilation rules are shown in Figure 13.2 Function $\mathcal{C}$ compiles an ABCL/R2 expression to a Lisp form which is processed by the evaluator. The form consists of a pair of tag and data. When the evaluator receives a form `[:compiled` f`]`, the evaluator calls f with the following arguments: the continuation `C`, the environment `Env`, the object id `Id` and the group id `Gid` of the object being processed, and the `Eval` itself (i.e., this is equivalent to `(funcall` f `C Env Id Gid Eval)`. The description does not describe various minor optimizations, such as constant folding. In addition, the notations used in the RHS, `[`e_1 `<=` e_2 `@` e_3`]` and `[`e_1 $e_2 \ldots e_n$`]`, are syntactic shorthand for `(send-message` e_1 e_2 e_3`)` and `(list` e_1 $e_2 \ldots e_n$`)`, respectively.

```
#'(lambda (C Env Id Gid Eval)
    [Eval
     <= [:do-evlis
         [[:variable 'x]
          [:variable 'y] [:variable 'z]]
         Env Id Gid Eval]
     @ [cont [x-value y-value z-value]
         [Eval
          <= [:do [:send-past x-value
                              (* y-value 2) z-value]
                  Env Id Gid Eval]
          @ C]]])
```

Figure 13.3: Compilation of `[x <= (* y 2) @ z]`

As an example of compilation, let us consider the compilation of an expression `[x <= (* y 2) @ z]`, meaning "send the value of `(* y 2)` to the value of `x` where the reply destination is the value of `z`." Due to the lack of side effects, we evaluate the expression in the following order: (1) Reference the values of `x`, `y`, and `z`. (2) Compute the value of `(* y 2)`. (3) Send the value of `(* y 2)` to the value of `x`, with the value of `z` as the *reply destination*. Since the variable references and the message sending are reflective, the complied code has expressions explicitly requesting those operations to the evaluator; more specifically, the resulting compiled code is as follows: (1) send a message for variable reference to the evaluator, (2) receive the values of the variables by a newly created continuation, (3) compute the value of the sub-expression `(* y 2)`, then (4) send a message for message sending to the evaluator. Following is the Lisp code generated by the ABCL/R2 compiler (For reader clarity, we write `[`x `<=` y `@` z`]` for the actual Lisp expression generated `(send-message` x y z`)`, meaning a message send to x with y, with reply destination z.):

In the compiled code shown in Figure 13.3, arguments `C`, `Env`, `Id`, `Gid`, and `Eval` denote the continuation, the environment, the object ID, the group ID of the object, and the evaluator for the sub-expressions, respectively. `:Do-evlis`, `:variable`, and `:send-past` are message tags that request the evaluator to evaluate a list of expressions, reference a variable, and perform the past-type message transmission. The form `[cont ...]` creates a *continuation object* (explained later), to which the reply from the evaluator is sent. We make a note that the multiplication `(* y 2)` reflective operation, is embedded into the compiled code, while the reference to the variable `y` is achieved by sending an expression `[:variable 'y]` to the evaluator.

13.3.3 Self-reification of Default System Objects

In addition to the partial compilation of normal objects, the default group kernel objects (the group manager, the metaobject generator, and the evaluator) and metaobjects of regular objects each has a pair of scripts that are pre-compiled; one is the bottomed-out

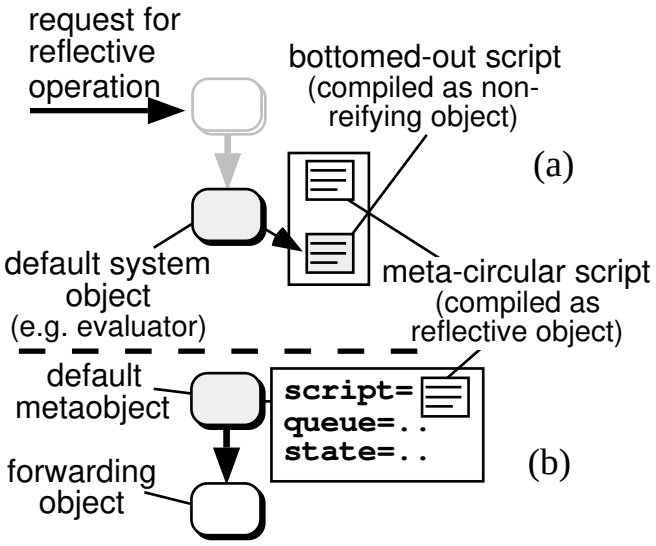


Figure 13.4: Self-Reification Mechanism

script and the other is the meta-circularly defined script for *self-reification*: Such objects initially execute with their bottomed-out (compiled) script (Figure 13.4(a)). Then, when their metaobjects are accessed, to which some reflective operation is requested, the former script creates a (default) metaobject that embodies the set of compiled scripts for that particular object.

When the metaobject created, it is initialized so that the metaobject could continue execution from the point where the reflective operation was requested in the bottomed-out script (this can only be realized with explicit argument passing rather than environment reification because the latter is not possible with bottomed-out, compiled script). Then the execution is delegated to the newly created metaobject, which starts the execution of the requested reflective operation (Figure 13.4(b)).

13.3.4 Light-weight Objects

Normal objects are heavyweight in the sense that it must support every conceivable operations allowed for an object. Here, the optimization strategy we employ is to sacrifice some of the capability of normal objects in trade for efficiency, and use them where appropriate. For this purpose, we introduce *light-weight objects*. The light-weight object does not have a message queue, but has a compiled script as a lambda-closure which is directly executable. When an ordinary object sends a message to a light-weight object, the sender executes the script of the receiver by directly invoking the lambda-closure of the object. It also does not have state variables; thus, it cannot have its own internal state. Execution overhead is reduced for two reasons: (1) its script is executed without scheduling overhead, and (2) its creation cost is smaller compared to normal objects due to the lack of state variables and the message queue.

Because of its limited functionality, objects which could be light-weight must satisfy

the following requirements:

stateless, because it does not have state variables,

receive a single message at a time, because it does not have a message queue

only perform simple operations, because complex operations would otherwise lead to deadlocks.

Light-weight objects are used extensively in ABCL/R2 to reduce the overhead of execution: one major use common to all levels is the *continuation object*; in the meta-level, it is also employed as the *light-weight metaobject*. Here, we defer the description of the latter until Section 13.3.5, and concentrate on its general use as a continuation object.

OOCP languages typically employ a programming style whereby continuation objects are created for delegating the result to, or synchronizing controls[2]. The meta-circular definition of metaobjects is a typical example: when a metaobject sends an expression to the evaluator for execution, the metaobject creates a continuation object, which receives a reply from the evaluator, and in turn notifies the end of evaluation by sending a `:end` message to the metaobject. This allows script execution and message reception to occur simultaneously[117]. In this manner, efficient implementation of continuation objects is crucial in achieving higher overall performance.

In ABCL/1, creation of continuation object was actually a syntactic macro that created a normal object. An expression:

`[cont` *Pattern Expressions*`]`

was equivalent to the following expression:

```
[object
  (script (=> Pattern Expressions))]
```

In ABCL/R2, continuation objects are implemented with light-weight objects, since the ways in which continuation objects are employed satisfy the abovementioned requirements: (1) stateless (temporal), (2) receives one and only one message, (3) its script is simple and terminates within finite steps, usually delegating the computed result to the next continuation object with a past type message send. The effectiveness of the light-weight objects is shown in Section 13.4.

13.3.5 Creation of Individual Tower via Dynamic Progression of Reflectivity

To construct the individual tower without unnecessary reification (i.e., creation of metaobjects), we employ the *dynamic progression* technique, which is an extension of lazy creation.

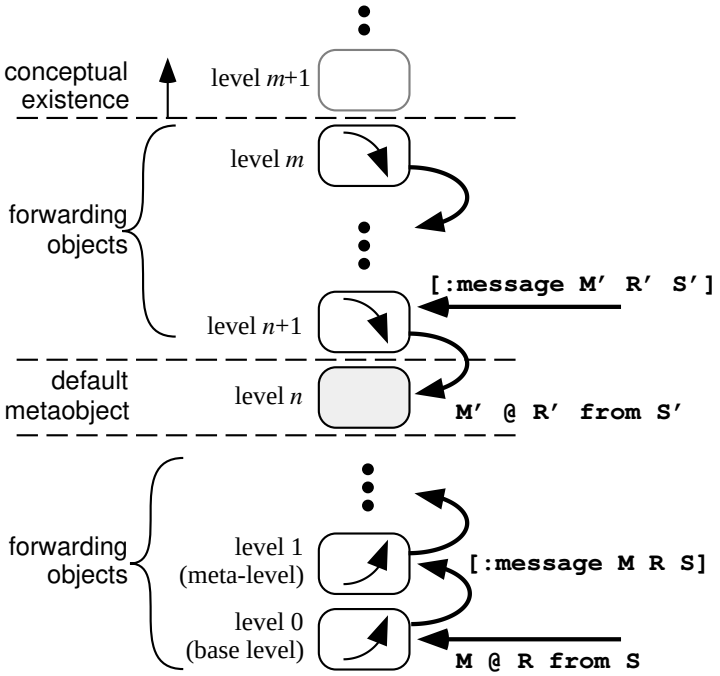


Figure 13.5: Snapshot of an Individual Tower

Figure 13.5 illustrates the implementational structure of an individual tower. To maintain the causal-connection, at any given time an tower has one and only one active object, the *default metaobject*, that could be directly executed by the low-level kernel (the shaded object at level n). All the other objects in the tower are either (1) *forwarding objects* (the white objects with solid border), or (2) of 'conceptual' existence i.e., does not physically exist within the system (the white object with hatched border). The level at which the default metaobject resides indicates the level of reflective operations that has been requested: in the figure, the reflective operation has been requested at level $(n-1)$ since the default metaobject resides at level n. Metaobjects above level n up to level m have been accessed in the past with the `[meta ...]` form. Objects above level m do not exist as they have not been accessed yet.

When the evaluator evaluates a message sending expression of a metaobject at level n, it attempts to send the message to the metaobject of the destination object at the same level, n. When the level of the metaobject of the receiver does not match that of the sender, the message is *forwarded* either up or down the tower to the default metaobject. For example, when the base-level object receives a message `M`, it forwards the message to its metaobject by sending a message `[:message M R S]` (`R` and `S` are the reply destination and the sender of `M`, respectively). Conversely, when the forwarding object at level $(n+1)$ receives a message `[:message M' R' S']`, it forwards the message by sending `M'` to the object at level n.

Since forwarding is a simple form of delegation within the tower, the forwarding metaobjects are implemented using the light-weight objects. We call such metaobjects

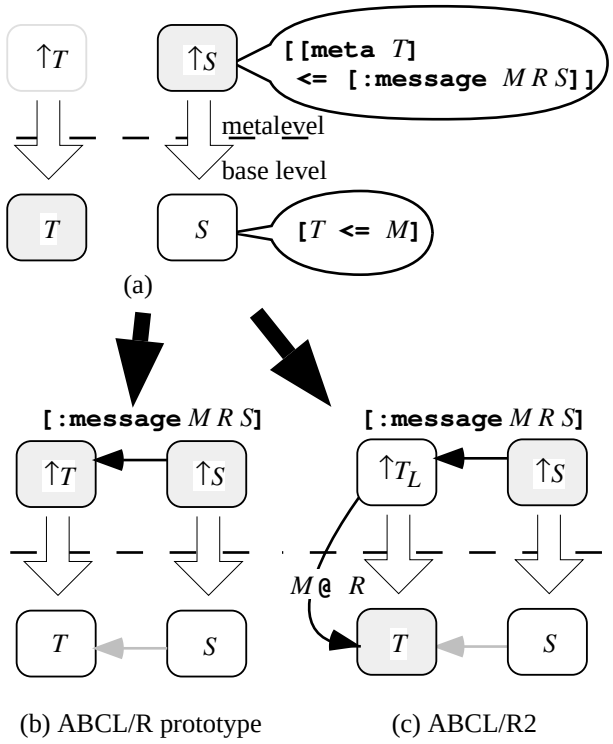


Figure 13.6: Comparison of the Naive Approach and the Forwarding Approach

the *light-weight metaobjects*. By employing the light-weight objects, forwarding along the tower need not be true message passing, which would incur great overhead; rather, the message is correctly delivered to the default metaobject by successive invocations of the lambda-closures of the light-weight metaobjects.

Here, the reader may become concerned with the fact that multiple messages could arrive at the object simultaneously, which could be a problem because light-weight objects do not have message queues. Fortunately, it is not a problem since forwarding is programmed to be pure functional, and multiple incoming messages are eventually buffered by the message queue of the default metaobject.

Forwarding augmented with light-weight objects is much more efficient compared to naive implementation of lazy creation of metaobjects. Let us contrast the two approaches and see why: in the latter approach, a metaobject $\uparrow x$ is created when the access to $\uparrow x$ occurs, that is, when the evaluator first evaluates an expression `[meta x]`[117]. Here, suppose that object S is sending a message M to object T, where S already has a metaobject while T does not. (Figure 13.6(a).)

Since the execution of S is already governed by its metaobject $\uparrow S$, $\uparrow S$ tries to send a message `[:message` M R S`]` (R is a reply destination of the message) to the metaobject of T. Here, the access to the metaobject of T by $\uparrow S$ causes the lazy creation of $\uparrow T$; T then becomes an object that is indirectly executed by $\uparrow T$ (Figure 13.6(b)). As a consequence, execution of T becomes comparatively slower. In our current approach, messages are

directly forwarded to T for faster execution (Figure 13.6(c)).

We refer to the following notion as the *dynamic progression of degree of reflectivity*[71] — the facility to realize the reflective functionality is progressively made more elaborate as more powerful ones are requested. This is a generalization of the lazy creation mechanism, in a sense that not only unnecessary reification is avoided, but also reflective features are restricted, allowing for efficient execution until full-fledged reflective operations become necessary. Such progression is performed automatically by the system in the following way:

- Initially, an executable object does not have its metaobject. (i.e., its metaobject(s) exist only conceptually.)
- When an expression `[meta` x`]` is evaluated, where x does not have its metaobject, a light-weight metaobject $\uparrow x_L$ is created as the metaobject of x. X remains directly executable as mentioned above (Figure 13.6(c)).
- When a message requesting some reflective operations (a message that does not match to the pattern `[:message` M R S`]`) arrives at $\uparrow x_L$, x is automatically reified, and $\uparrow x_L$ becomes the default metaobject $\uparrow x$. Hereon, the reflective operation is executed by $\uparrow x$.
- The access to the metaobject of $\uparrow x_L$ (i.e., the meta-metaobject of x) merely creates a light-weight metaobject $\uparrow\uparrow x_L$. X still remains to be directly executable by the forwarding mechanism.

The alternative strategy would have been to perform the appropriate reification/reflection of the message at the sender's end (actually, the evaluator), and let it decide the proper level of the receiver to which the message should be sent. This is not desirable, however, for several reasons: (1) it would break the object encapsulation (in the current scheme, the sender does not need to know the status of the individual tower of the receiver), (2) if such reification/reflection was manifest in the evaluator code, it would be complicated, (3) the code for dynamic progression, which requires concurrency control due to simultaneous message arrival, would be inefficient because the sender must explicitly 'lock' the receiver object, and (4) in a distributed implementation, it would incur several round-trip message sends to check the status of the receiver, and then sending the message, whereas our current scheme only requires a single message transmission.

13.3.6 Compilation of Non-Reifying Objects

Since non-reifying objects sacrifice reflective capabilities for faster execution, the script of non-reifying objects can be almost entirely be compiled into a lambda-closures of Lisp, and be executed without interpretation. The compiled code accepts messages of the form `[:message` M R S`]`, and executes the compiled script matching M. Expressions that are internal to the object and execute sequentially are directly translated into Lisp expressions. Expressions relevant to OOC-computing, such as message sending or object creation, are converted into function calls to the low-level kernel. For example, the

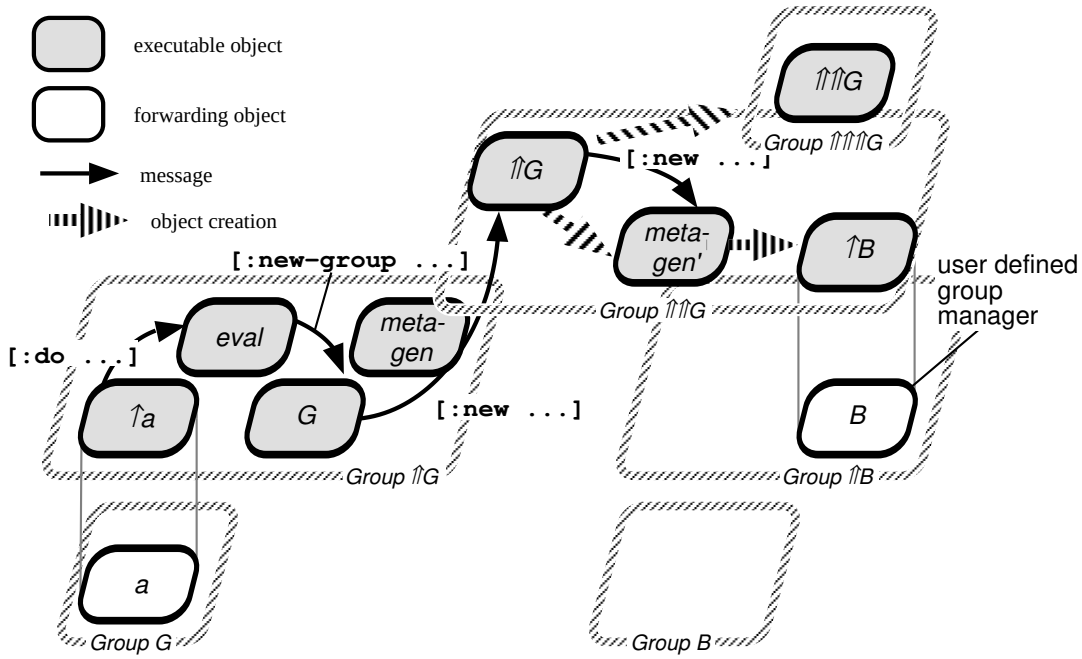


Figure 13.7: Creation of a New Group

expression `[x <= (+ 1 y) @ z]` is converted into a Lisp expression `(send-message x (+ 1 y) z)` where `send-message` is a primitive for message transmission provided by the low-level kernel.

Furthermore, since our system assumes multi-thread environment, a non-reifying object does not switch its context during the execution of its script. That is, when a non-reifying object sends a now-type message, a thread executing the object actually waits for the reply of the message.

Even though the script of non-reifying object is not subject to reification, the problem is that some messages may be sent to the (non-existing) metaobject(s) of the non-reifying objects due to the abovementioned encapsulation. For this purpose, a message to a metaobject of a non-reifying object automatically creates a light-weight metaobject in the same manner as is with the implementation of individual tower for normal objects. This solves the problem of interfacing non-reifying objects with reflective parts of the system.

13.3.7 Lazy Creation of Meta-Groups

The construction of the group tower is achieved in finite steps via lazy creation of group managers. In addition, the self-reification code of group kernel objects contains the necessary reification code for the group tower. Below is the overview of how the group tower is constructed:

1. All objects except the group manager object is created with a reference to the group manager object of the group of which it is a member.

2. The group manager is initially created without a reference to *its* group manager, i.e., the group manager of the meta-group. The latter is created lazily by the system when it is referenced.

3. Upon creation of the group manager, other group kernel objects (evaluator, metaobject generator, etc.) are not initially created. Their creation occurs when they are referenced via the group manager; at the same time, the group manager G references its own group ID, which results in the creation of $\Uparrow G$ via 2 (provided it is non-existent). The created group kernel object becomes a member of $\Uparrow G$ by receiving the reference to $\Uparrow G$ from G.

4. Acquiring a reference to the group kernel objects is initially possible only via the group manager (The reference can be freely passed around once it is acquired). Thus, all objects (including the members of user defined groups) of all levels satisfy the condition 1 (except the group manager).

Figure 13.7 illustrates the creation process of a new group. When the object a creates a new group B with the `[group ...]` form, the evaluation of this form is sent to the evaluator (the `[:do ...]` message). The evaluator, in turn, sends the `[:new-group ...]` message to the group manager, which is forwarded to the group manager of the meta-group, $\Uparrow$G (the `[:new ...]` message). Since the group kernel objects of the new group B become members of $\Uparrow$G, the metaobject generator of $\Uparrow$G must be created, which successively results in the creation of $\Uparrow\Uparrow$G as explained in 3. (Note that this object exists for consistency purpose only, and does not execute any code unless higher-level reflective operations are requested.) The newly created metaobject generator meta-gen' in turn creates the group kernel objects of group B.

13.4 Performance Measurements of the ABCL/R2 Compiler

For evaluation of basic performance of our ABCL/R2 system, We ran several benchmark programs for performance evaluation of ABCL/R2. Specifically, we tested performance of (1) non-reflective features, (2) the light-weight objects, and (3) the maximum cost of reflective computation.

For comparative purposes, we have performed the same benchmark on our ABCL/1[127] compiler (which does not support reflective features), which is currently being publically distributed. (For details, contact abcl@camille.is.s.u-tokyo.ac.jp.) We have also programmed the same algorithm in C with the Sun Light-Weight Processes (LWP) library, assigning a thread per object. The ABCL/R2 system we employ in this section is not the version on Luna-88k, but instead the pseudo-parallel version running on top of KCL (Kyoto Common Lisp) on SparcStation1+ (except for the lowest-level thread scheduler, the two implementations are identical.). The reason for this is that the current ABCL/1 system only supports pseudo-parallel execution on standard Common Lisp.

Although the benchmarks were performed for several programs, for brevity the one we present in this paper is the computation of Fibonacci numbers. In the parallel version `parallel-fib`, for each computation of $fib(n)$, two sub-objects that compute $fib(n-1)$ and $fib(n-2)$ are created. In the recursive (sequential) version, for each computation of $fib(n)$, the object `recursive-fib` creates two continuation objects to receive the values $fib(n-1)$ and $fib(n-2)$.

Figure 13.9 shows the result of the measurement. Some of the implications of these results are as follows:

- For the parallel version, the two corresponding lines in the middle indicate that, ABCL/R2 exhibits comparable performance to ABCL/1 for non-reifying objects.
- When light-weight objects are employed (as continuation objects), execution in ABCL/R2 is consistently faster (by approximately 30%). As we have noted earlier, this fact is important because the light-weight objects are heavily created and employed by the evaluator during the execution of scripts, and are also employed (transparently) in user programs.
- Normal objects still pay some cost of reflective execution, but by a factor of less than 10. This is astonishingly smaller compared to ABCL/R, by nearly or over two orders of magnitude: the computation of parallel $fib(12)$ takes over *12 minutes* on ABCL/R executing on an identical software/hardware platform, whereas it takes only 22 seconds for normal objects and 4 seconds for non-reifying objects on ABCL/R2.
- The C + Sun LWP version become drastically slow when the number of object increases, probably due to the overhead of stack allocation and context switching. (In fact, it was not possible to compute beyond $n = 16$ due to lack of memory. We also make a note that the measurement does not include the paging overhead.) Furthermore, non-reifying objects of ABCL/R2 exhibit comparable performance even when the number of objects is small.

Furthermore, by our programming experience thus far, the execution speed of actual programs on ABCL/R2 is often quite comparable to that on ABCL/1, even if reflective operations are employed. The reason for this is that, in practice, the user would employ the mixture of non-reifying objects and normal objects, and attempts to localize the reflective portion of his programs to normal (reflective) objects. As a result, a large portion of the user code runs with non-reifying objects (and light-weight objects), and meta-level execution would be localized to where it is really needed. Even if execution is with normal objects (which have metaobjects), the meta-level execution overhead is at a comparable level to ABCL/1 due to our optimization schemes (unlike ABCL/R, which is more than two orders of magnitude slower compared to ABCL/1).

```
[object parallel-fib-gen  ; the parallel version
 (meta-gen non-reifying-meta)
 (script
   (=> :new              ; '!' returns the evaluated
    ![object fib         ; expression
       (meta-gen non-reifying-meta) ; ABCL/R2 only
       (state [reply := nil] [sub-value := nil])
       (script
        (=> [:ans x]
            (if sub-value
               [reply <= [:ans (+ sub-value x)]]
               [sub-value := x]))
        (=> 0 ![:ans 0])
        (=> 1 ![:ans 1])
        (=> n @ R
            [reply := R]
            [[parallel-fib-gen <== :new]
              <= (- n 1) @ Me]
            [[parallel-fib-gen <== :new]
              <= (- n 2) @ Me]))]))]

[object recursive-fib
  (meta-gen non-reifying-meta)  ; ABCL/R2 only
  (script
   (=> 0 !0)
   (=> 1 !1)
   (=> n @ R
       [recursive-fib <= (- n 1)
        @ [cont fib-n-1
             [recursive-fib <= (- n 2)
              @ [cont fib-n-2
                   [R <= (+ fib-n-1 fib-n-2)]]]]]))]
```

Figure 13.8: Object Definitions for Fibonacci Numbers

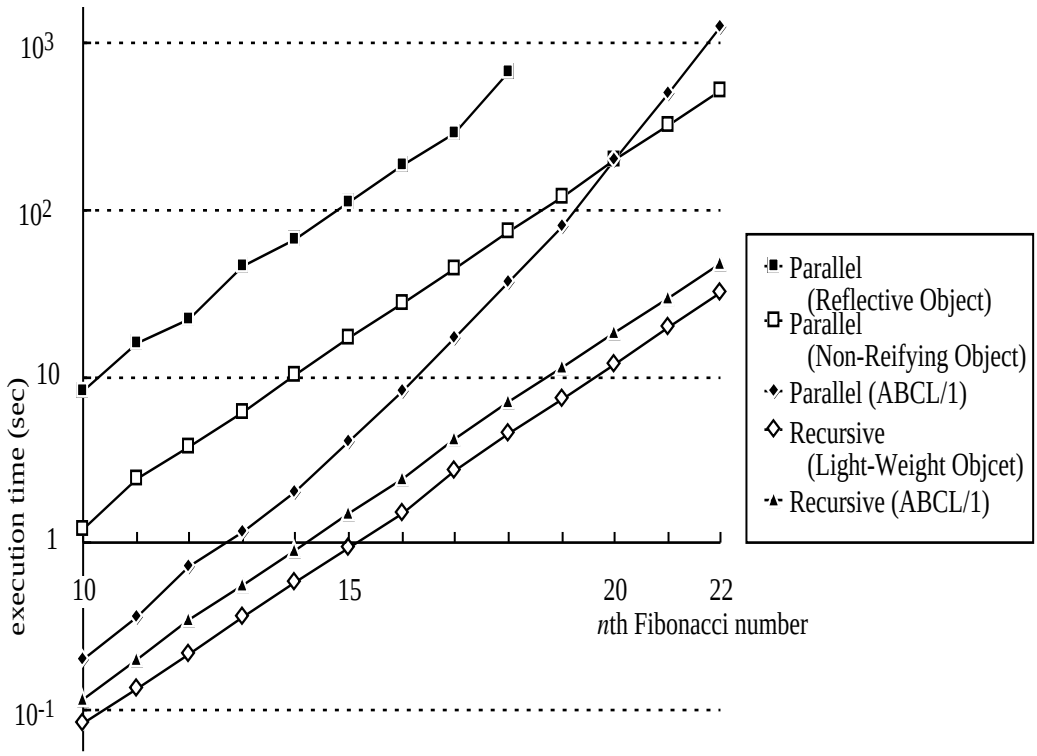


Figure 13.9: Performance Measurements for Fibonacci Numbers on ABCL/R2 and ABCL/1

Chapter 14

Examples of Reflective Programming in ABCL/R2

The most significant benefit in programming with OOCR languages is the ability to achieve *metalevel encapsulation* of control of parallelism, that is, the ability to orthogonally encapsulate the user-level programming of metalevel concerns in parallel execution—scheduling, load distribution, etc.—easily and transparently into the metalevel of the reflective tower. By this orthogonality, it would be possible to construct a generic application framework for parallel object. We will present an example of controlling the scheduling of objects to avoid explosion of parallelism. Furthermore, it would be possible to add such framework to *preexisting* applications with little modifications; we present an example of this by adding metalevel scheduling to the time-warp algorithm presented in [117].

Reflective programming in ABCL/R2 is performed in two ways. One is to utilize its metaobject in the same way as ABCL/R, which were described in [117]. Another is to introspect and affect the group-wide coordinated behavior of the group the object is a member of. This is performed with a message to its group, `[group-of` x`]` (delivered to its group manager object). The two schemes are not contradictory; in practice, a combination of both schemes is effectively used.

For the remainder of the section, we present an example of scheduling modeled in terms of computational resource management using reflective programming in ABCL/R2. Management of other resources can be performed analogously.

14.1 Example 1—Controlling Explosion of Parallelism via Computational Resource Management

The first example is controlling of explosion of parallelism via object scheduling.

Programs in Actor-like languages are usually written in a style such that the maximum available parallelism in the algorithm is exploited. However, too much parallelism wastes system resources, and as a result, has a negative effect on performance, e.g., wastes

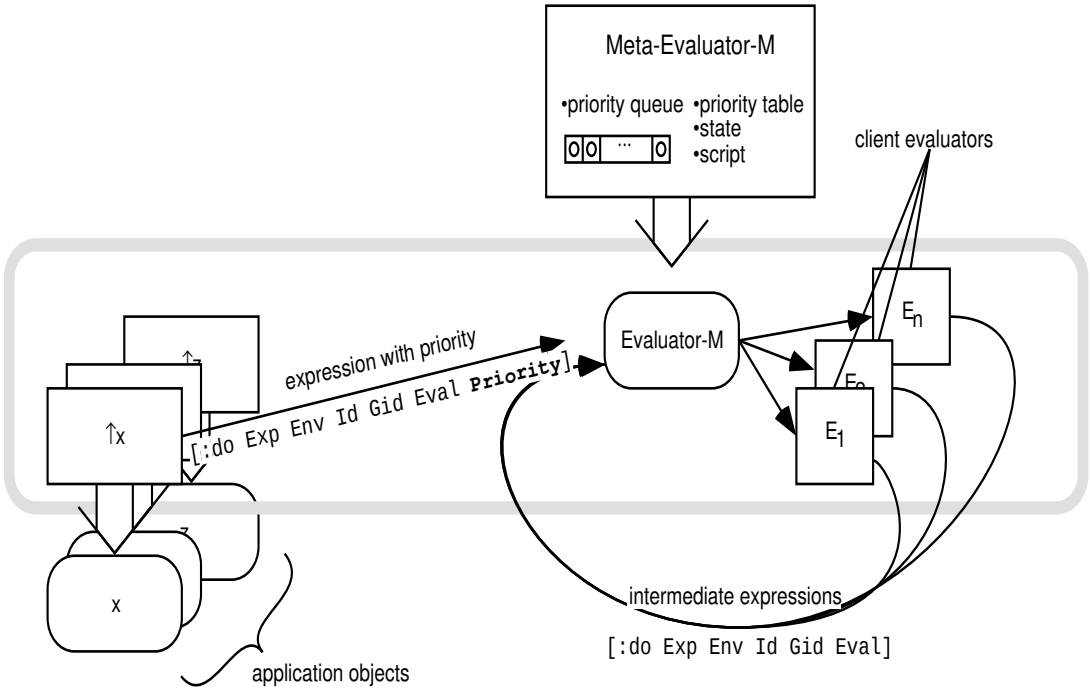


Figure 14.1: Controlling Explosion of Parallelism with ABCL/R2

more memory, incur more cost in scheduling, etc. By all means, one could perform user-level programming in order to control the number of objects created, etc.[2], but the resulting user code would have the base-level algorithm and the control algorithm heavily intermixed without proper encapsulation, and as a result, hampers program development, portability, and re-use. Rather, if such control could be encapsulated in the meta-level, not only that the user code need not contain provisions for control, but the same meta-level code could be re-used for a variety of concurrent programs in a portable manner.

In order to control the parallelism, we limit the number of objects created in the system. In particular, if the number of executable objects could always be suppressed so that it is comparable to or little higher than the total number of processing resources in the system, we obtain an ideal balance.

The system is organized as in Figure 14.1. Each application object embodies within itself a parameter indicating the "degree of progress" in its computation.[1] Such the object is created by the following form:

```
[object object-name
 (group pcontrol)
 (priority p)
 (state ...)
 (script ...)]
```

When an evaluator accepts this form, it sends an object creation message, which con-

[1] For example, in quicksort it is the length of the list to be sorted.

tains a "degree of progress" parameter p as a priority value in addition to the standard creation messages, to a customized metaobject generator through the group manager `pcontrol`. Then the metaobject generator creates a customized metaobject with that priority value. As indicated earlier, the computation of an application object at the base-level is performed by the evaluator at the meta-level, which is triggered by a message from the metaobject. This message contains the priority value. Hence, the metaobjects created by the customized metaobject generator send the priority value to the evaluator in addition to the standard parameters. Here, in order to exploit the meta-level programmability of our system, we let the metaobject of the evaluator (Meta-Evaluator-M) prioritize the message according to this parameter. Figure 14.2 shows the outline of the definition of Meta-Evaluator-M. By prioritizing (evaluation requests of) objects that are expected to terminate their computation faster, the number of executable objects can be suppressed. The evaluator (Evaluator-M) then distributes the task to the client evaluators ($\mathsf{E}_1 \ldots \mathsf{E}_n$), which represent actual processors.

We applied the above of scheme of parallelism control to two different parallel programs: the parallel version of computation of Fibonacci numbers in the previous section, and a parallel quicksort algorithm. To adapt to different algorithms, the only customization is to declare what to employ as the parameter. We created four client evaluators to coincide with the number of processors that LUNA-88K has (which is 4). Figure 14.3 is the graph of the executable object count for parallel quicksorting of 1,000 elements (the graph for Fibonacci is essentially the same in shape). The solid line indicates the object count when the parallelism is controlled, and the dashed line indicates no control. The horizontal axis indicates the number of expressions processed by the evaluator. As we can see, parallelism is appropriately controlled via suppression of excessive object creation (which is a user policy) with appropriate meta-level encapsulation (which is a system mechanism).

14.2 Example 2—Time Warp Scheduling in ABCL/R2

The ABCL/R paper [117] presented an example of how an OOCR architecture can be cleanly implement the *Time Warp* algorithm[55] (also known as the Virtual Time scheme) employed in parallel discrete event simulation. When objects model the entities and the message transmission/reception model the events in the simulation, the Time Warp algorithm serves to maintain the temporal consistency among the events. Consistency management is distributed and optimistic; each object has its own *Local Virtual Time (LVT)* (i.e., there is no global clock), and the messages are timestamped to be compared with the LVT of the recipient. When a conflict is detected, the object performs automatic *rollback* by sending *anti-messages* until it reaches the time just prior to the conflict occurrence.

In the implementation, the entire Time Warp algorithm was successfully encapsulated in the within the metaobject of each object, since Time Warp algorithm was meta-level to the execution of the simulation itself. One thing that was not addressed was the performance issues affected by different scheduling policies of objects. That is to say, the

algorithm was still programmed under the assumption that infinite parallelism was available with concurrent objects. However, a recent work by Burdorf and Marti[20], however, compared ten non-preemptive scheduling algorithms for the Time Warp algorithm, and discovered that there were orders of magnitude difference in their execution speed for some problems. Thus, we cannot ignore scheduling issues in practice when we implement the Time Warp algorithms with OOCP languages.

Burdorf and Marti made some simple assumptions in their performance measurement; for example, they did not allow interprocessor communication between the schedulers, which is necessary for inter-group load balancing. This is too restrictive for OOCP languages, where inter-scheduler communication would be simple. Also, they did not attempt any adaptive scheduling, that is, to alter the scheduling algorithm dynamically to adapt to better algorithms when excessive rollbacks occur during the simulation. For programming efficiency, we would like our language to be able to model these in an orthogonal way, so that the algorithmic concerns of Virtual Time (which is already encapsulated in the metalevel) is independent of the how the objects are scheduled by the available computational resource (thus, scheduling is meta- to the Time Warp algorithm, or meta-meta- to the simulation application.) Unfortunately, it was not easy to achieve such hierarchical and orthogonal encapsulation with previous OOCR languages for the reasons we discussed in Chapter 11.

With ABCL/R2 we can obtain a clean solution: We define a *TimeWarp group*, whose members are specialized with their individual metaobjects so that they coordinate in running the Time Warp algorithm. This is similar to the ABCL/R example, except that group membership automatically dictates Time Warp behavior, not requiring explicit metaobject specification. (The actual definition of the Time Warp group are given in Appendix B). Messages sent within the group or to destinations within other Time Warp groups must be of the form:

[*target* `<=` *message* `@` *reply-destination* `:vrt` *virtual-send-time*]

Since scheduling is meta-level to the execution of the Time Warp algorithm, we would want to encapsulate it in the *meta-level* of the Time Warp algorithm (i.e., meta-meta-level of the actual simulation algorithm), in the same manner that the algorithm itself was encapsulated in the meta-level of the simulation. The conceptual illustration of the encapsulation is given in Figure 14.4. For implementation, we utilize the group-reflective features of our architecture. We introduce the *TimeWarp scheduler* object labeled Scheduler in Figure 12.3. It is responsible for controlling the allocation of the computational resource within a Time Warp group. For meta-meta-level encapsulation of scheduling, the scheduler does not interact with the evaluator of the Time Warp group; rather, it interacts with the *metaobject* of the evaluator. The metaobject of the evaluator is specialized so that the evaluation request to the evaluator sent from an object in the group is not directly executed, but instead sent to the scheduler. The metaobject then asks the scheduler for the next evaluation job as determined by the algorithm of the scheduler. This behavior is outlined in the abridged code of the evaluator below:

```
[object TW-Eval-meta  ;;; metaobject of the evaluator of the Time Warp group
   (meta non-reifying-meta)
   (state [scheduler := Scheduler])
   (script
    ;; meta-level reception of message to the evaluator
    (=> [:message [KeyWord Expr Env Id Time] R S]
           where (member KeyWord '(:do :do-evlis :do-progn))
        [scheduler
           <= [:schedule [KeyWord Expr Env Id Time] :with Id Time]]
        (if (eql mode 'dormant)
          [Me <= :begin]))
    (=> :begin
        (match [scheduler <== :next]
            :
        ))]
```

Aside from the behavior specific to Virtual Time, the behavior of the evaluator is almost identical to that of a standard evaluator. The `TW-Eval-meta` (labeled ↑Eval in Figure 12.3) delegates most of the scheduled evaluation requests directly to the evaluator of the meta-group (labeled Eval'). Since Eval' is a non-reifying object, the delegation would terminate there. In effect, Eval' is the sole computational resource for all the members of the group as well as the objects that comprise the group, including the group kernel objects[2]. So, in a sense, Eval' is the native CPU hardware in an operating system; this is a generalization of the conceptual model of reflective operating systems such as Muse[124].

With this framework, the same Time Warp algorithm runs irrespective of the presence of/difference in the scheduling algorithm at the meta-meta-level. Thus, dynamic change of the scheduler can easily be accommodated as given in Appendix B. Furthermore, it would be easy to extend the Time Warp group to add inter-scheduler communication, and/or to have scheduler controlling multiple meta-group evaluators to adapt to growth of computational resource in hardware.

Figure 14.6 is the result of the car-wash simulation[81, 126] on ABCL/R2: cars in the incoming queue are washed by multiple attendants with different washing speed. When (1) no scheduling was performed, and (2) meta-meta-level scheduling was performed in FIFO order, the number of rollbacks increased proportionally to the number of cars. When the scheduling algorithm was changed to (3) 'lowest LVT first', the rollbacks were eliminated. (This seemingly 'ideal' result is probably due to centralized scheduling; when ABCL/R2 is extended to a distributed system, rollbacks should occur (albeit few in number)) Due to this, the execution speed of (3) was consistently several % faster than (1), despite the overhead of execution at the meta-meta-group level.

[2]The situation would be analogous for multiple evaluators.

```
[object Meta-Evaluator-M
 (meta-gen non-reifying-meta)
 (state [pqueue := aPriorityQueue]
        [state := newEnvironment]
        [scriptSet := givenScriptSet]
        [evaluator := anEvaluator]
        [Group-manager := aGroupManager]
        [mode := ':dormant])
 (script
  ;; arrival of a message with priority
  (=> [:message [Tag Exp Env Id Gid Eval Priority]
                Reply Sender]
            ;; enqueues the message in the
   [pqueue ;; prioritalized message queue
    <= [:enq [Priority [[Tag Exp Env Id Gid Eval]
                        Reply Sender]]]]
   (when (eq mode ':dormant)
     [mode := ':active]
     [Me <= :begin]))
  ;; start processing of a message
  (=> :begin
   ;; a message with the highest priority is taken out
   (match [pqueue <== :deq]
     (is [_ [Message Reply Sender]]
       (match (find-script Message Reply scriptSet)
           :
```

Figure 14.2: Definition of Meta-Evaluator-M

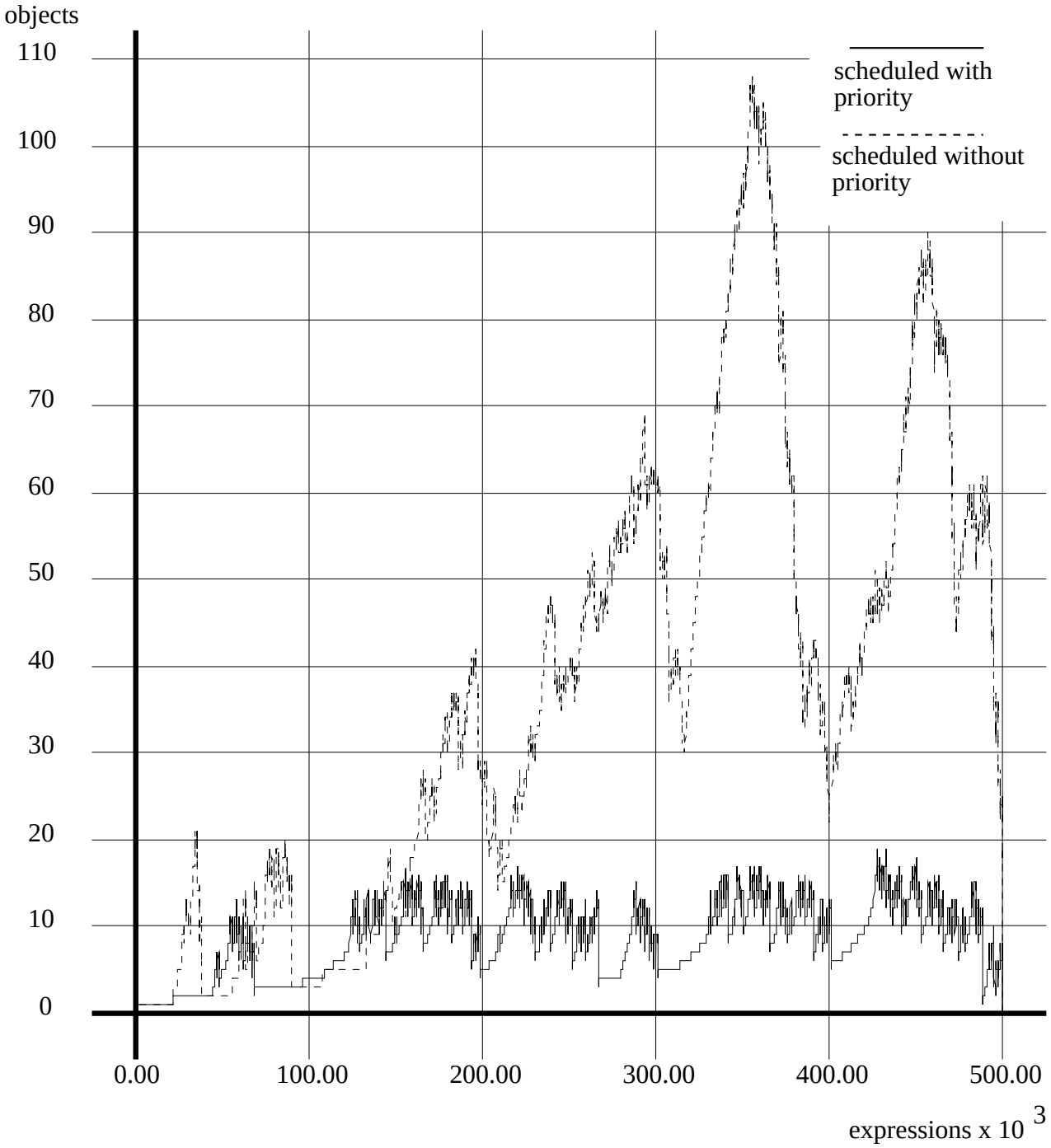


Figure 14.3: Controlled and Uncontrolled Parallelism for Quicksorting

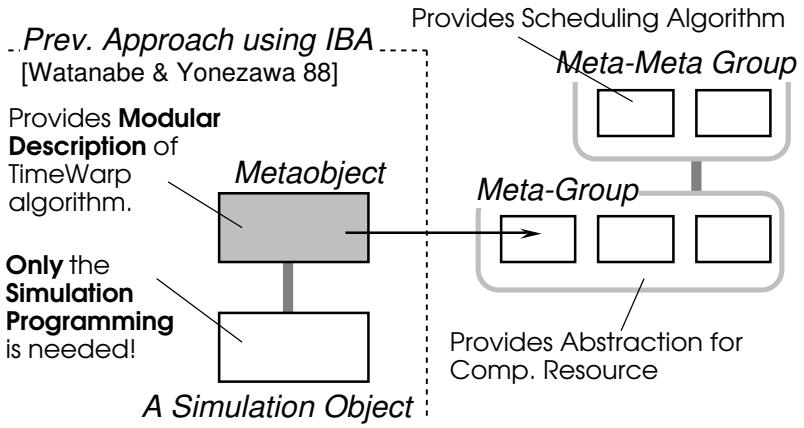


Figure 14.4: Meta-level Encapsulation of the Time Warp Algorithm (1)

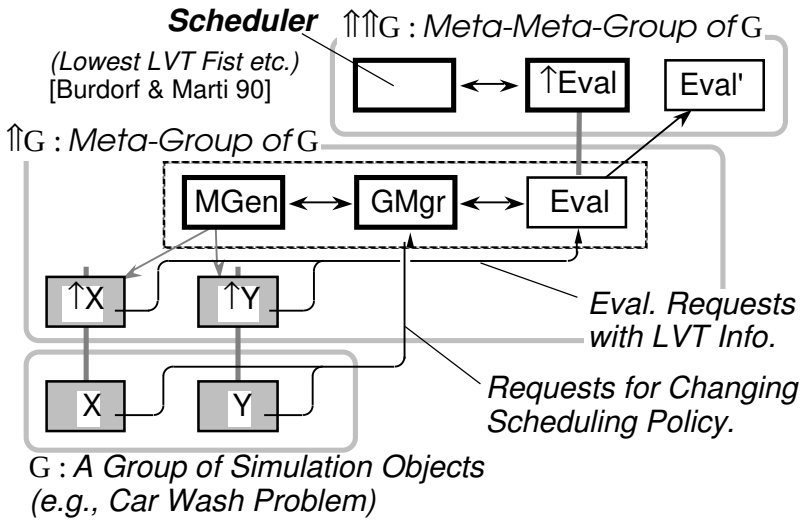


Figure 14.5: Implementation of the Time Warp Scheduling in the Meta-meta Level

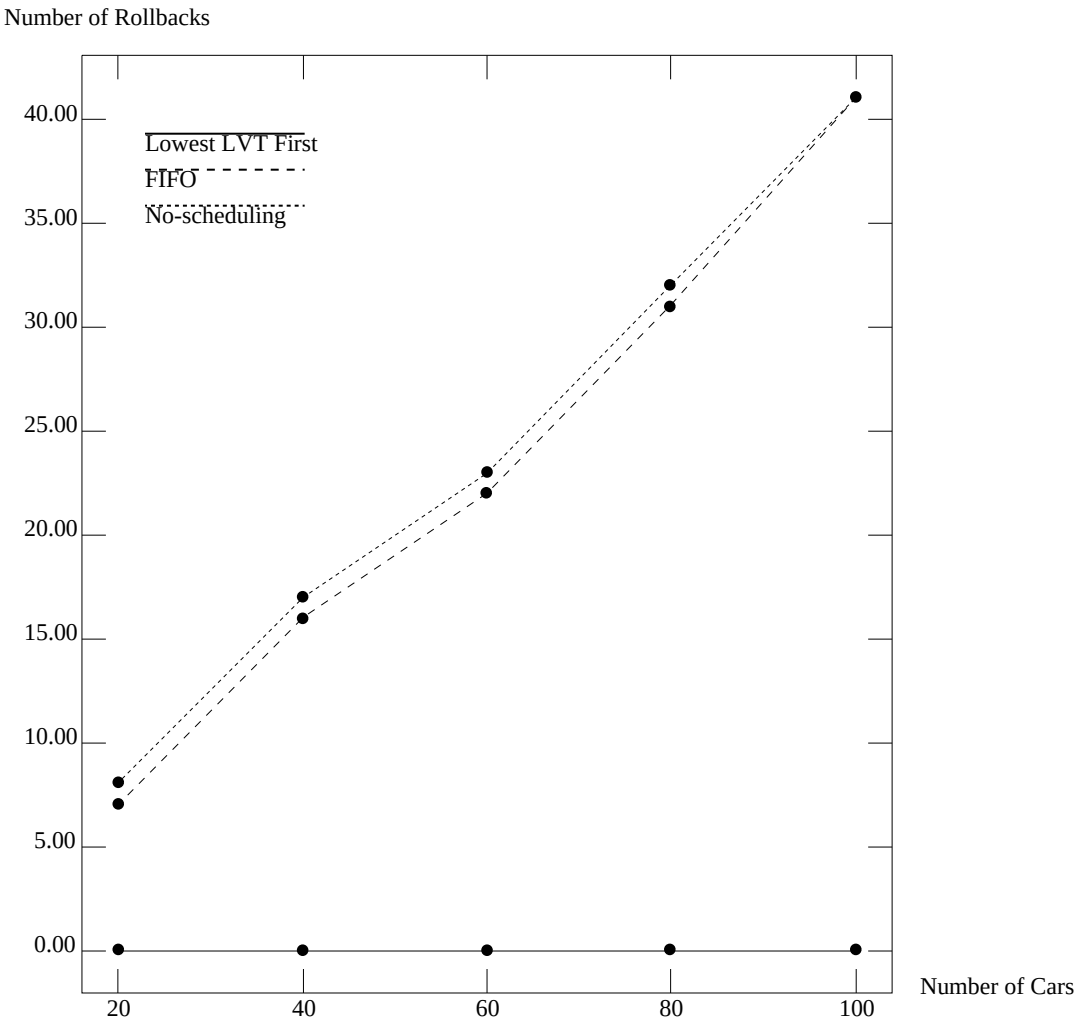


Figure 14.6: The Result of Meta-meta-level Scheduling of the Time Warp Algorithm applied to the Car Wash Problem

Chapter 15

Formal Semantics of the Hybrid Group Architecture—ABCL/Rμ

.

The principle of the hybrid architecture is that it embodies two reflective towers of individual and group. Here, in order to be more strict on the behavior of ABCL/R2 and the hybrid architecture in general, this chapter covers the formal semantic definition. It might be preferable to give complete semantic accounts for ABCL/R2, but it would merely be enormous because of various intricacies of the original ABCL/1. Instead, we define a much smaller language, which retains much of the essential properties of the hybrid architecture of ABCL/R2, and we give complete semantic accounts to that.

ABCL/Rμ is the mini-language designed for this purpose. Its syntax resembles that of ABCL/1. The semantics of the ABCL/Rμ is defined in a similar style as Agha's semantics of Actors given in [1]: the behavior of individual objects are given denotationally, without any non-determinism. Concurrency of in a group of objects is described operationally with a (non-deterministic) transition system of *configurations*. A similar approach was taken for the semantic definition of a OOCR language ACT/R[118] as briefly mentioned in Chapter 11.

The essential differences between ABCL/R2 and ABCL/Rμ are as follows:

- The only primitive data types available are integers and lists. All other data entities are objects.
- The side effects are handled Actor style, i.e., the new state of objects are atomically specified by the replacement behavior command `become`.
- We define simple classes-like specification of behavior with *behavior templates*, instead of providing the `object` expression. This is to facilitate specification of recursive replacement behaviors (see below).

There are numerous technical differences between the Agha actor semantics and ours, primarily for achieving (implementable) reflective system in our language:

- Each transition system of configuration is considered to correspond to a group. For inter-group communication, we define a transition involving two groups simultaneously, instead of creating explicit receptionists.

- The definition an object is (mail address × behavior name × local environment), whereas with actors it was (mail address × behavior). The behavior name serve as a reference to class templates.

- The behavior defined via `(defbehavior ...)` is not actually a behavior, but a *behavior template*. The necessity for this is obvious, because when the actor specifies the replacement behavior, it must have all the information of the 'behaviors' available in the subsequent computation, in particular, its own behavior. However, this is not possible at the time of its definition. One approach is to use a fixpoint operator, but we avoid this in a syntactic way instead.

- There are no longer explicit tags to distinguish the message tasks. As a result, two identical messages sent to an actor are indistinguishable. The messages tasks in the configuration are stored as multisets; weak fairness assumption is obviously still valid due to non-determinism of the task selection.

- The mail addresses are no longer flat, but rather are *structured mail addressees*: the first component is the mail address of the metalevel group manager, the second component is the level of interpretation in the individual reflective tower, and the third component is the index number of the object in the group.

- For each meta-level transition system, the mail addresses of the objects created at that group are ordinal, although their generations are done in an distributed way. Together with the abolition of task tags, a metacircular interpreter faithful to the semantics is now easily implementable.

The reflective part of ABCL/Rμ is defined in an constructive way.

Abstract Syntax

First, we present the abstract syntax of ABCL/Rμ. It resembles ABCL/R2, but much of its elaborate features have been removed. Furthermore, side-effect statements via assignments have been uniformly replaced by the `become` expression.

$$
\begin{array}{rcll}
D & \in & BDef & ::= \texttt{(defBehavior } Name \texttt{ (}Var...\texttt{) } Method...\texttt{)} \\
B & \in & BIde & ::= Name \\
M & \in & Method & ::= \texttt{(=> (}Pttn.\ Var...\texttt{) } Cmd...\texttt{)} \\
 & & & \\
C & \in & Cmd & ::= \texttt{[}Exp\ \texttt{<= [}Exp\texttt{, } Exp\texttt{]]} \mid \texttt{(become } BIde\ Exp...\texttt{)} \\
 & & & \quad \mid \texttt{(para } Cmd...\texttt{)} \mid \texttt{(if } Exp Cmd\{Cmd\}\texttt{)} \\
 & & & \\
E & \in & Exp & ::= Pexp \mid \texttt{(new } BIde\ Exp...\texttt{)} \mid \texttt{(}Pexp\ Exp...\texttt{)} \\
 & & & \\
 & & Pexp & ::= Const \mid Var \mid \texttt{(}Pexp\ Pexp...\texttt{)} \\
 & & & \\
V & \in & Var & ::= Name \\
P & \in & Pttn & ::= Const \mid Name \\
L & \in & Lst & ::= \{Pexp\texttt{, } Pexp...\}
\end{array}
$$

Actor Denotations

Next are the semantic domains. The notations $F_M(A)$, $F_S(B)$, and $S_S(C)$ denote finite multiset of A, finite set of B, and singleton set of C respectively.

Semantic Domains (Non-reified)

The values in the domain below are not reified.

$$
\begin{array}{rcll}
\theta & \in & \mathcal{H} & \textit{(Base Actor Indices)} \\
l & \in & \mathcal{L} & \textit{(Level in Individual Tower)} \\
\mu, m & \in & \mathcal{M} = \mathcal{M} \times \mathcal{L} \times \mathcal{H} + \mathcal{H} & \textit{(Structured Mail Addresses)} \\
v & \in & \mathcal{V} = \mathcal{M} + Num + Fun + List + \cdots & \textit{(Primitive Values)}
\end{array}
$$

For simplicity, we assume that the base actor indices and individual tower level are isomorphic to natural numbers.

Semantic Domains (Reifiable)

$$
\begin{array}{rcll}
\alpha & \in & \mathcal{A} = \mathcal{M} \times BIde \times \mathcal{L} & \textit{(Actors)} \\
\tau & \in & \mathcal{T} = \mathcal{M} \times Pttn \times \mathcal{V} & \textit{(Message Tasks)} \\
\sigma & \in & \mathcal{L} = Var \to \mathcal{V} & \textit{(Local Environment of Actors)} \\
\delta & \in & \mathcal{S} = BIde \to \mathcal{B} \times Var & \textit{(Behavior Environment)} \\
\beta & \in & \mathcal{B} = \mathcal{L} \to \mathcal{V} \to \mathcal{M} \to & \\
 & & \quad F_M(\mathcal{T}) \times F_S(\mathcal{A}) \times S_S(BIde \times \mathcal{L}) \times \mathcal{M} & \textit{(Behavior Templates)}
\end{array}
$$

Semantic Functions

The semantic functions have been divided into several special-purpose ones for convenience. $\mathcal{C}$ is the primary semantic function to handle commands such as message sends and `become`. It returns the new objects and message tasks that were generated as a result of execution of the given command. $\mathcal{E}$ is used to evaluate expressions, while $\mathcal{G}$ returns the objects created within the expressions using `new`. $\mathcal{N}$ governs the generation of object addresses under the current scheme. Finally, $\mathcal{D}$ is more of a convenient shorthand for handling behavior definitions. $\mathcal{C}$, $\mathcal{E}$, $\mathcal{G}$ take the following arguments: the local environment of the object, the local behavior environment of the object, the mail address of the object, and the bookkeeping mail address for new mail address generation.

$$
\begin{array}{lcll}
\mathcal{C} & : & (Cmd + Exp) \rightarrow \mathcal{L} \rightarrow \mathcal{S} \rightarrow \mathcal{M} \rightarrow \rightarrow \mathcal{M} & \\
 & & F_M(\mathcal{T}) \times F_S(\mathcal{A}) \times S_S(BIde \times \mathcal{L}) \times \mathcal{M} & \textit{(Commands)} \\
\mathcal{E} & : & Exp \rightarrow \mathcal{L} \rightarrow \mathcal{S} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{V} & \textit{(Expressions)} \\
\mathcal{G} & : & Exp \rightarrow \mathcal{L} \rightarrow \mathcal{S} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \rightarrow F_S(\mathcal{A}) & \textit{(Actor Generation)} \\
\mathcal{D} & : & BDef \rightarrow \mathcal{S} \rightarrow \mathcal{S} & \textit{(Actor Behavior Definition)} \\
\mathcal{N} & : & Exp \rightarrow \mathcal{M} \rightarrow \mathcal{M} & \textit{(Actor Index Update)}
\end{array}
$$

Auxiliary Functions

$$
\begin{array}{lcl}
succ & : & \mathcal{M} \rightarrow \mathcal{M} \\
succ\,\langle m, l, \theta \rangle & = & \langle m, l, \theta + 1 \rangle \\
\oplus & : & Set \rightarrow Set \\
s_1 \oplus s_2 & = & \textit{if } s_1 = \phi \textit{ then } s_2 \textit{ else } s_1 \\
\uplus & : & Set \rightarrow Set \\
s_1 \uplus s_2 & = & s_1 \cup s_2 \\
\uplus & : & Nat \rightarrow Nat \\
s_1 \uplus s_2 & = & max(n_1, n_2) \\
\uplus & : & Tuple \rightarrow Tuple \\
\langle s_1, \ldots, s_n \rangle \uplus \langle t_1, \ldots, t_n \rangle & = & \langle (s_1 \uplus t_1), \ldots, (s_n \uplus t_n) \rangle
\end{array}
$$

Functions to primitive data values (integers, lists) are defined in an obvious way: they are standard arithmetic functions and list operations such as *car* and *cons*. Also, $\uplus$ denotes a special operator that takes piecewise union or maximum of corresponding tuple elements, depending on their types.

Semantic Definitions

Individual Object(Actor) Semantics

We first present the *ground semantics* of individual ABCL/Rμ objects, i.e., the semantics of non-reified objects.

Commands

$$
\begin{array}{lcl}
\mathcal{C}[\![\texttt{[}E_1\ \texttt{<=}\ \texttt{[}E_2\ .\ E_3\texttt{]]}]\!]\sigma\delta m\mu & = & \mathit{let}\ \ \eta_1 = \mathcal{N}[\![E_1]\!] \\
 & & \qquad\ \ \eta_2 = \mathcal{N}[\![E_2]\!] \\
 & & \qquad\ \ \eta_3 = \mathcal{N}[\![E_3]\!] \\
 & & \mathit{in}\ \ \langle\{\langle\mathcal{E}[\![E_1]\!]\sigma\delta m\mu, \langle\mathcal{E}[\![E_2]\!]\sigma\delta m(\eta_1\mu), \mathcal{E}[\![E_3]\!]\sigma\delta m(\eta_2\eta_1\mu)\rangle\rangle\}, \\
 & & \qquad \mathcal{G}[\![E_1]\!]\sigma\delta m\mu \cup \mathcal{G}[\![E_2]\!]\sigma\delta m(\eta_1\mu) \cup \mathcal{G}[\![E_3]\!]\sigma\delta m(\eta_2\eta_1\mu), \\
 & & \qquad \phi, \eta_3\eta_2\eta_1\mu\rangle \\
\mathcal{C}[\![\texttt{(become}\ B\ E\texttt{)}]\!]\sigma\delta m\mu & = & \langle\phi, \mathcal{G}[\![E]\!]\sigma\delta m succ(\mu), \{\langle m, B, \\
 & & \qquad (\sigma_\perp[(\delta(B)).2 \rightarrow \mathcal{E}[\![E]\!]\sigma\delta m succ(\mu))]\rangle\rangle, \mathcal{N}[\![E]\!]succ(\mu)\} \\
\mathcal{C}[\![\texttt{(para}\ C_1\ C_2\texttt{)}]\!]\sigma\delta m\mu & = & \mathit{let}\ \ \langle T_1, A_1, \gamma_1, \mu_1\rangle = \mathcal{C}[\![C_1]\!]\sigma\delta m\mu \\
 & & \qquad\ \ \langle T_2, A_2, \gamma_2, \mu_2\rangle = \mathcal{C}[\![C_2]\!]\sigma\delta m(\mathcal{N}[\![C_1]\!]\mu) \\
 & & \mathit{in}\ \ \langle T_1 \cup T_2, A_1 \cup A_2, \gamma_1 \oplus \gamma_2, \mu_2\rangle \\
\mathcal{C}[\![\texttt{(if}\ E\ C_1\ C_2\texttt{)}]\!]\sigma\delta m\mu & = & \mathit{if}\ \ \ \mathcal{E}[\![E]\!]\sigma\delta m\mu \\
 & & \mathit{then}\ \ \mathit{let}\ \ \langle T_1, A_1, \gamma_1, \mu_1\rangle = \mathcal{C}[\![E]\!]\sigma\delta m\mu \\
 & & \qquad\qquad\ \ \langle T_2, A_2, \gamma_2, \mu_2\rangle = \mathcal{C}[\![C_1]\!]\sigma\delta m(\mathcal{N}[\![C_1]\!]\mu) \\
 & & \qquad\ \ \mathit{in}\ \ \langle T_1 \cup T_2, A_1 \cup A_2, \gamma_1 \oplus \gamma_2, \mu_2\rangle \\
 & & \mathit{else}\ \ \ \mathit{let}\ \ \langle T_1, A_1, \gamma_1, \mu_1\rangle = \mathcal{C}[\![E]\!]\sigma\delta m\mu \\
 & & \qquad\qquad\ \ \langle T_2, A_2, \gamma_2, \mu_2\rangle = \mathcal{C}[\![C_2]\!]\sigma\delta m(\mathcal{N}[\![E]\!]\mu) \\
 & & \qquad\ \ \mathit{in}\ \ \langle T_1 \cup T_2, A_1 \cup A_2, \gamma_1 \oplus \gamma_2, \mu_2\rangle \\
\mathcal{C}[\![N]\!]\sigma\delta m\mu & = & \langle\phi, \phi, \phi, \mu\rangle \\
\mathcal{C}[\![V]\!]\sigma\delta m\mu & = & \langle\phi, \phi, \phi, \mu\rangle \\
\mathcal{C}[\![\texttt{Me}]\!]\sigma\delta m\mu & = & \langle\phi, \phi, \phi, \mu\rangle \\
\mathcal{C}[\![\texttt{(new}\ B\ E\texttt{)}]\!]\sigma\delta m\mu & = & \langle\phi, \mathcal{G}[\![\texttt{(new}\ B\ E\texttt{)}]\!]\sigma\delta m\mu, \phi, \mathcal{N}[\![E]\!]succ(\mu)\rangle \\
\mathcal{C}[\![\texttt{(}E_1\ E_2\texttt{)}]\!]\sigma\delta m\mu & = & \langle\phi, \mathcal{G}[\![\texttt{(}E_1\ E_2\texttt{)}]\!]\sigma\delta m\mu, \phi, \mathcal{N}[\![\texttt{(}E_1\ E_2\texttt{)}]\!]\mu\rangle \\
\mathcal{C}[\![\{E_1, E_2, \ldots\}]\!]\sigma\delta m\mu & = & cons(\mathcal{E}[\![E_1]\!]\sigma\delta m\mu, cons(\mathcal{E}[\![E_2]\!]\sigma\delta m(\mathcal{N}[\![E_1]\!]\mu), \ldots))
\end{array}
$$

Expressions

$$
\begin{array}{lcl}
\mathcal{E}[\![N]\!]\sigma\delta m\mu & = & \mathrm{N}\ \ \textit{(self-denoting value)} \\
\mathcal{E}[\![V]\!]\sigma\delta m\mu & = & \sigma(V) \\
\mathcal{E}[\![\texttt{Me}]\!]\sigma\delta m\mu & = & m \\
\mathcal{E}[\![\texttt{(new}\ B\ E\texttt{)}]\!]\sigma\delta m\mu & = & \mu \\
\mathcal{E}[\![\texttt{(}E_1\ E_2\texttt{)}]\!]\sigma\delta m\mu & = & \mathcal{E}[\![E_1]\!]\sigma\delta m\mu(\mathcal{E}[\![E_2]\!]\sigma\delta m(\mathcal{N}[\![E_1]\!]\mu)) \\
\mathcal{E}[\![\{E_1, E_2, \ldots\}]\!]\sigma\delta m\mu & = & cons(\mathcal{E}[\![E_1]\!]\sigma\delta m\mu, cons(\mathcal{E}[\![E_2]\!]\sigma\delta m(\mathcal{N}[\![E_1]\!]\mu), \ldots))
\end{array}
$$

Actor Generation

$$
\begin{array}{lcl}
\mathcal{G}[\![E]\!]\sigma\delta m\mu & = & \phi \quad (E \in Pexp) \\
\mathcal{G}[\![\texttt{Me}]\!]\sigma\delta m\mu & = & \phi \\
\mathcal{G}[\![\texttt{(new } B \ E\texttt{)}]\!]\sigma\delta m\mu & = & \{\langle \mu, B, \sigma_\perp[(\delta(B)).2 \rightarrow \mathcal{E}[\![E]\!]\sigma\delta m\mu]\rangle\} \cup \mathcal{G}[\![E]\!]\sigma\delta m succ(\mu) \\
\mathcal{G}[\![\texttt{(}E_1 \ E_2\texttt{)}]\!]\sigma\delta m\mu & = & \mathcal{G}[\![E_1]\!]\sigma\delta m\mu \cup \mathcal{G}[\![E_2]\!]\sigma\delta m\mathcal{N}[\![E_1]\!]\mu \\
\mathcal{G}[\![\{E_1, E_2, \ldots\}]\!]\sigma\delta m\mu & = & \mathcal{G}[\![E_1]\!]\sigma\delta m\mu \cup \mathcal{G}[\![E_2]\!]\sigma\delta m\mathcal{N}[\![E_1]\!]\mu \cup \ldots
\end{array}
$$

Actor Index

$$
\begin{array}{lcl}
\mathcal{N}[\![E]\!]\mu & = & \mu \quad (E \in Pexp) \\
\mathcal{N}[\![\texttt{Me}]\!]\mu & = & \mu \\
\mathcal{N}[\![\texttt{(new } B \ E\texttt{)}]\!]\mu & = & succ(\mu) \\
\mathcal{N}[\![\texttt{(}E_1 \ E_2\texttt{)}]\!]\mu & = & \mathcal{N}[\![E_2]\!](\mathcal{N}[\![E_1]\!]\mu)
\end{array}
$$

Behavior Definition

$$
\begin{array}{lcl}
\mathcal{D}[\![D]\!]\delta & = & \delta[B \rightarrow \langle \lambda\sigma\delta m\mu.\lambda\langle p, v\rangle. \\
 & & \quad if\ p = \mathcal{E}[\![P_1]\!]\sigma\delta m\mu\ then\ \mathcal{C}[\![C_1]\!](\sigma[V_1 \rightarrow v])\delta m\mu \\
 & & \quad \vdots \\
 & & \quad if\ p = \mathcal{E}[\![P_n]\!]\sigma\delta m\mu\ then\ \mathcal{C}[\![C_n]\!](\sigma[V_n \rightarrow v])\delta m\mu, \\
 & & V_{acq}\rangle]
\end{array}
$$

where $D \equiv$ `(defbehavior` B `(`V_{acq}`)`
`(=>` `(`P_1 `.` V_1`)` C_1`)`
$\vdots$
`(=>` `(`P_n `.` V_n`)` C_n`))`

Transition System for Groups

The behavior of groups are defined operationally in terms of possible transitions between two (valid) configurations. The message passing command outputs a task into the task multiset of the given configuration; transition via processing of a task corresponds to consumption of a message (message reception + method invocation).

Initial Group Configuration

Given a group with definition `(`D_1 $\ldots$ D_n C`)` and a mail address of the group manager m, the initial group configuration c_{init} is defined as follows:

$$
\begin{array}{lcl}
c_{init} & = & let\ \ \langle\langle T, A, \gamma, \mu\rangle, \delta\rangle = \\
 & & \quad let\ \ \delta_1 = \mathcal{D}[\![D_1]\!]\delta_\perp,\ \delta_2 = \mathcal{D}[\![D_2]\!]\delta_1,\ \ldots\ ,\ \delta_n = \mathcal{D}[\![D_n]\!]\delta_{n-1} \\
 & & \quad in\ \ \langle \mathcal{C}[\![C]\!]\sigma_\perp\delta_n\langle m, 1, 1\rangle\langle m, 1, succ(1)\rangle, \delta_n\rangle \\
 & & in\ \ \langle A, T, \delta, m, \mu\rangle
\end{array}
$$

Note that we do not need to obtain the fixed point of δ as seen in the semantics of inheritance by Reddy[93]. This is because actual references to behavior templates do not occur until the evaluation of C, and δ is not modified at that time nor thereafter.

Group Transition System

First of all, we define in-group transition. Given two group configurations c_1 and c_2, where:

$$c = \langle A, T, \delta, m, \mu \rangle$$
$$c' = \langle A', T', \delta, m, \mu' \rangle$$

then, c is said to have a *possible transition* to c' by processing a message task $\tau = \langle m, \langle P, v \rangle \rangle \in T$, written:

$$c \xrightarrow{\tau} c'$$

if $\exists \alpha = \langle m, B, \sigma \rangle \in A$ and for $\langle T, A, \alpha', \mu'' \rangle = (\delta(B).1)\sigma\delta m\mu_1\langle P, v \rangle$, the followings hold:

$$\begin{aligned} A' &= (A_1 - \{\alpha\}) \cup A \cup \{\alpha'\} \\ T' &= (T_1 - \{\tau\}) \cup T \\ \mu' &= \mu'' \end{aligned}$$

Next, we model inter-group communication. Group communication can be *strict*, where the reception of the message at the receiving group needs to be guaranteed, and *non-strict*, where reception need not be guaranteed.

Strict Inter-Group Communication

Given two groups with configurations c_1 and c_2, where:

$$\begin{aligned} c_1 &= \langle A_1, T_1, \delta_1, m_1, \mu_1 \rangle \\ c_2 &= \langle A_2, T_2, \delta_2, m_2, \mu_2 \rangle \end{aligned}$$

then, c_1 and c_2 are said to have a *strict possible communication* with a message task $\tau = \langle \langle m_2, l, \mu \rangle, \langle P, v \rangle \rangle \in T_1$, and with resulting configurations into c_1' and c_2', s.t.

$$\begin{aligned} c_1' &= \langle A_1', T_1', \delta_1, m_1, \mu_1' \rangle \\ c_2' &= \langle A_2', T_2', \delta_2, m_2, \mu_2' \rangle \end{aligned}$$

if $\exists \alpha = \langle \langle m_2, l, \mu \rangle, B, \sigma \rangle \in A_2$, and for $\langle T, A, \alpha', \mu' \rangle = (\delta(B).1)\sigma\delta_2 m_2\mu_2\langle P, v \rangle$, the followings hold:

$$\begin{aligned} A_1' &= A_1 \\ T_1' &= (T_1 - \{\tau\}) \\ \mu_1' &= \mu_1 \\ A_2' &= (A_2 - \{\alpha\}) \cup A \cup \{\alpha'\} \\ T_2' &= T_2 \cup T \\ \mu_2' &= \mu'. \end{aligned}$$

This is written as:

$$\langle c_1, c_2 \rangle \xrightarrow{\tau} \langle c_1', c_2' \rangle$$

Non-strict Inter-Group Communication

Given two groups with configurations c_1 and c_2, where:

$$\begin{aligned} c_1 &= \langle A_1, T_1, \delta_1, m_1, \mu_1 \rangle \\ c_2 &= \langle A_2, T_2, \delta_2, m_2, \mu_2 \rangle \end{aligned}$$

then, c_1 and c_2 are said to have a *possible communication* with a message task $\tau = \langle \langle m_2, l, \mu \rangle, \langle P, v \rangle \rangle \in T_1$, and with resulting configurations into c_1' and c_2', s.t.

$$\begin{aligned} c_1' &= \langle A_1', T_1', \delta_1, m_1, \mu_1' \rangle \\ c_2' &= \langle A_2', T_2', \delta_2, m_2, \mu_2' \rangle \end{aligned}$$

if the followings hold:

$$\begin{aligned} A_1' &= A_1 \\ T_1' &= (T_1 - \{\tau\}) \\ \mu_1' &= \mu_1 \\ A_2' &= A_2 \\ T_2' &= T_2 \cup T \\ \mu_2' &= \mu_2. \end{aligned}$$

This is written as:

$$\langle c_1, c_2 \rangle \xrightarrow{\tau} \langle c_1', c_2' \rangle$$

Reflection in ABCL/Rμ

The approach to modeling the infinite towers of ABCL/R2 can be done in two ways. One is to employ a semi-denotational model of the tower by in the manner of Friedman and Wand[116] and by Danvy and Malmkjær ([36], etc.), which has been recently adopted to the context of OO-languages by Nakajima[82]. However, we avoid this approach, because: (1) it would be difficult to model the inherent concurrency in the metaobjects, and (2) it would be difficult to model the existence of the evaluator object in the group tower. Instead, we model the behavior of the current ABCL/R2 system implementation itself. To make the semantics tractable and understandable, we throw away compilation and various other optimizations, and stick to the modified version of the original metacircular definitions.

The only way in the usual actor model to invoke an operation is via message sends. Thus, any reflective operation must also be invoked via message-passing between the actors belonging to different levels. We describe how such inter-level communications are achieved in ABCL/Rμ.

Reification and Reflection of Mail Addresses

The basic policy of ABCL/Rμ is to encode the structure of the reflective towers into the mail addresses. In other words, the distinguishment of the object as to its role is possible for the transition system just by observing the mail address contained in the task. The

address maintains within itself the level information, with value 1 being the base level and the subsequent values denoting the appropriate meta levels.

As a result of this encoding, the expressions that access the meta-levels, namely, $\uparrow E$, $\downarrow E$, is easily defined. Also, the access to the meta-group $\Uparrow E$, can be given a concise definition. The job of delivering the message becomes the responsibility of the transition system. Below are their simple definitions:

$$
\begin{array}{lcl}
\mathcal{E}[\![\Uparrow E]\!]\sigma\delta m\mu & = & \mathit{if}\ \mathcal{E}[\![E]\!]\sigma\delta m\mu \in \mathcal{M} \\
 & & \mathit{then\ let}\ \ \langle m', l', \mu'\rangle = \mathcal{E}[\![E]\!]\sigma\delta m\mu \quad \textit{(address of group manager)} \\
 & & \qquad \mathit{in}\ \ m' \\
 & & \mathit{else\ error} \\
\mathcal{E}[\![\uparrow E]\!]\sigma\delta m\mu & = & \mathit{if}\ \mathcal{E}[\![E]\!]\sigma\delta m\mu \in \mathcal{M} \\
 & & \mathit{then\ let}\ \ \langle m', l', \mu'\rangle = \mathcal{E}[\![E]\!]\sigma\delta m\mu \quad \textit{(address of metaobject)} \\
 & & \qquad \mathit{in}\ \ \langle m', l', \mu' + 1\rangle \\
 & & \mathit{else\ error} \\
\mathcal{E}[\![\uparrow E]\!]\sigma\delta m\mu & = & \mathit{if}\ \mathcal{E}[\![E]\!]\sigma\delta m\mu \in \mathcal{M} \\
 & & \mathit{then\ let}\ \ \langle m', l', \mu'\rangle = \mathcal{E}[\![E]\!]\sigma\delta m\mu \quad \textit{(address of denotation)} \\
 & & \qquad \mathit{in}\ \ \mathit{if}\ \mu' - 1 > 0 \\
 & & \qquad\qquad \mathit{then}\ \langle m', l', \mu' - 1\rangle \\
 & & \qquad\qquad \mathit{else\ error} \\
 & & \mathit{else\ error}
\end{array}
$$

Metacircular Definitions of Metaobjects and Meta-Groups in ABCL/Rμ

Here, we present the metacircular definition of ABCL/Rμ, which is faithful to the base ABCL/Rμ semantics. The definitions use several syntactic shorthand for brevity: for example, multiple-argument message and now-type message passings can be defined in terms of insensitive actors as described in [1]. Also, the `match` macro has been defined on lists much the same way as has been defined for ABCL/R2.

Translation of ABCL/Rμ programs into First-Class Data

Although it is possible to manipulate the ABCL/Rμ programs directly, it would be far easier to deal with parsed abstract syntax trees. We first define the translation rule between standard ABCL/Rμ program and its representation as a list data. Each syntactic constituent is given an explicit tag to distinguish its type. For the behavior definition, the function `find-script` finds the matching method given the behavior identifier and the message pattern.

Translation Rules

$$\mathcal{TR} : (Cmd + Exp + BDef) \rightarrow List$$

$$\begin{array}{lcl}
\mathcal{TR}[\![\texttt{[}E_1\ \texttt{<= [}E_2\ \texttt{.}\ E_3\texttt{]]}]\!] & = & \{\texttt{:send}\ \mathcal{TR}[\![E_1]\!]\ \mathcal{TR}[\![E_2]\!]\ \mathcal{TR}[\![E_3]\!]\} \\
\mathcal{TR}[\![\texttt{(become}\ B\ E\texttt{)}]\!] & = & \{\texttt{:become}\ \mathcal{TR}[\![B]\!]\ \mathcal{TR}[\![E]\!]\} \\
\mathcal{TR}[\![\texttt{(para}\ C_1\ C_2\texttt{)}]\!] & = & \{\texttt{:para}\ \mathcal{TR}[\![C_1]\!]\ \mathcal{TR}[\![C_2]\!]\} \\
\mathcal{TR}[\![\texttt{(if}\ E\ C_1\ C_2\texttt{)}]\!] & = & \{\texttt{:if}\ \mathcal{TR}[\![E]\!]\ \mathcal{TR}[\![C_1]\!]\ \mathcal{TR}[\![C_2]\!]\} \\
\mathcal{TR}[\![N]\!] & = & \{\texttt{:self-denoting}\ N\} \\
\mathcal{TR}[\![V]\!] & = & \{\texttt{:variable}\ V\} \\
\mathcal{TR}[\![\texttt{Me}]\!] & = & \{\texttt{:me}\} \\
\mathcal{TR}[\![\texttt{(new}\ B\ E\texttt{)}]\!] & = & \{\texttt{:new}\ \mathcal{TR}[\![B]\!]\ \mathcal{TR}[\![E]\!]\} \\
\mathcal{TR}[\![\texttt{(}E_1\ E_2\texttt{)}]\!] & = & \{\texttt{:apply}\ \mathcal{TR}[\![E_1]\!]\ \mathcal{TR}[\![E_2]\!]\} \\
\mathcal{TR}[\![\{E_1\texttt{,}E_2\texttt{,...}\}]\!] & = & \{\texttt{:list}\ \mathcal{TR}[\![E_1]\!]\ \mathcal{TR}[\![E_2]\!]\} \\
\mathcal{TR}[\![D]\!] & = & \{\texttt{:behavior}\ \{ \\
 & & \quad \{\mathcal{TR}[\![P_1]\!]\ \{\texttt{:script}\ \mathcal{TR}[\![V_1]\!]\ \mathcal{TR}[\![C_1]\!]\}\} \\
 & & \quad\quad \vdots \\
 & & \quad \{\mathcal{TR}[\![P_n]\!]\ \{\texttt{:script}\ \mathcal{TR}[\![V_n]\!]\ \mathcal{TR}[\![C_n]\!]\}\} \\
 & & \quad \mathcal{TR}[\![V_{acq}]\!]\}\}
\end{array}$$

Metaobject definition in ABCL/Rμ

The metacircular definition of metaobjects is given in Figure 15.1. The definition is much simpler compared to that of ABCL/R2 metaobject we have described in Chapter 12.

The metaobjects are usually created directly via the evaluator object. This is different from ABCL/R2, which has a separate metaobject generator. This design was chosen due to the availability of class-like behavior templates, and also to simplify the base semantics.

One may notice that there are no `become` command in the definition of the metaobject. Indeed, the metaobject (and the object it represents) disappears once its script has been evaluated. This is the consequence of the adoption of actor semantics; if one follows the definition of the evaluator closely, it is immediately obvious that a replacement metaobject is created upon execution of the `become` command within its script. An alternative design would be to have the metaobject be persistent and handle the state change associated with `become` by itself, but it would complicate the replacement behavior protocol, and the resulting metacircular definition would be less faithful to the base semantics of ABCL/Rμ.

Definition of Primary Evaluator in ABCL/Rμ

The primary evaluator is also defined analogously in Figures 15.2 and 15.3. It closely follows the denotational definition of evaluation of individual objects. It has the following three notable characteristics:

```
(defbehavior MetaObject (beh-id local-env beh-env eval group)
   (=> (:message Message Reply Sender)
      (match Message
         (is {Pattern Args}
            [eval <=
               [:eval beh-env beh-id local-env Pattern Args Me group]])
         (otherwise (error)))))
```

Figure 15.1: Metaobject Definition in ABCL/Rμ

1. It serves the role of all the different types of evaluations as defined by the respective semantic functions in the base semantic definition of ABCL/Rμ (It also adds the script invocation for convenience).

2. Each evaluation request is sent as a message, which contains all the information necessary to evaluate a given command, expression, etc.; thus, any evaluation does not 'loop back' to the group manager or the metaobject, but rather, is interpreted as a successive recursive evaluation request to the evaluator, as defined in the base semantics. Here, This behavior is similar in principle to the evaluator in ABCL/R2.

3. The evaluator is a stateless object; thus, multiple parallel invocation is possible in theory. This is evident the scripts, in that all the evaluation messages perform an immediate `become` after it receives a evaluation message. In fact, evaluation of totally distinct subexpressions may be arbitrarily interleaved; again, this is similar in principle to ABCL/R2.

The resulting tasks are sent to the group manager, which manages the transition system at the meta-level.

Definition of Group Manager in ABCL/Rμ

The group manager is relatively simple; it basically manages the behavior of the transition system described earlier (Figure 15.4).

The group manager starts evaluation when it receives the `:start` message along with the initial command to evaluate. It forwards the evaluation of the command to the evaluator of the group as if it were an evaluation request from an object. It passes appropriate initialization arguments to the evaluator in the process, such as the behavior environment of the group as well as an reference to itself. In the normal course of operation, it receives the evaluation result from objects in the group in parallel. At the same time, it selects a single message task from its pool of tasks, and forwards it to the target metaobject. Reception of results and task selection/forwarding are done totally in parallel, taking advantage of the arbitrary interleavings possible for configurations in the base semantics of ABCL/Rμ.

```
(defbehavior Eval ()  ;; The Primary Evaluator
                      ;; — computational resource for the group.
                      ;; (has no state variables)
   ;; evaluation of a script
   (=> [:eval beh-env beh-id env pat args sender group]
      (para
         (become Eval())       ;; immediately invoke next evaluation (stateless)
         (match (find-script   ;; find the first-class value of behavior
                [Me <== [:eval-expr pat env beh-env]] ;; evaluate pattern
                beh-id)        ;; under given behavior id
            (is {script: formal-args script-body}
               [Me <= [:eval-command
                  script-body
                  [env <== [:extend-env env formal-args args]] ;; bind args
                  beh-env sender group]])
            (otherwise (error)))))

   ;; evaluation of a command
   (=> [:eval-command command env beh-env sender group]
      (para
         (become Eval())      ;; immediately invoke next evaluation (stateless)
         (match command       ;; Assume parse to convenient form
            (is {:send Target Pattern Args} ;; message send
               ;; create the new triplet and send to the group
               [group <= [:eval-result {
                  {:task    ;; create message send task
                     [Me <== [:eval-evlis
                                 {Target Pattern Args}
                                 env beh-env]]
                     sender},
                  (union [Me <== [:eval-gen Target env beh-env group]]
                         [Me <== [:eval-gen Pattern env beh-env group]]
                         [Me <== [:eval-gen Args env beh-env group]])
                  {} }]])
            (is {:become Beh-Id Args} ;; handle become
               ;; create the new triplet and send to the group
               [group <= [:eval-result {
                  {}
                  [Me <== [:eval-gen Args env beh-env group]]
                  {Beh-Id (new Environ
                     (cdr (lookup-beh beh-env Beh-Id))
                     [Me <= [:eval-expr Args env beh-env]])}}]])
            (is {:para Com1 Com2} ;; parallel execution
                     :            ;; (ommitted)
                    )
                     :            ;; (other commands)
      (otherwise (error))))
;; (next page)
```

Figure 15.2: Primary Evaluator in ABCL/Rμ

```
    ;; evaluation of expressions
    (=> [:eval-expr args env beh-env]
       (para
          (become Eval())
          (match args
             (is {:self-denoting val}    ;; self-denoting value
                !val)
             (is {:variable name}        ;; variable
                !(lookup-var env name))
             (is {:new Beh-Id expr}      ;; object creation (mail address only)
                !(create Beh-Id expr)
                      :    ;; (rest omitted)
                                  )))

    ;; evaluation of object generation
    (=> [:eval-gen args env beh-env group]
       (para
          (become Eval())
          (match args
             (is {:self-denoting val}    ;; self-denoting value
                !{})                     ;;   return empty
             (is {:variable name}        ;; variable
                !{})                     ;;   return empty
             (is {:new Beh-Id expr}      ;; object creation
                !(union
                   (new Metaobject        ;; create new (meta) object
                      Beh-Id
                      (new Environ (cdr (lookup-beh beh-env Beh-Id))
                         [Me <== [:eval-expr Args env beh-env group]])
                      beh-env Me group)
                   ;; add created objects in subexpressions
                   [Me <== [:eval-gen expr env beh-env]])))
                      :  ;; (rest omitted)
                                     )

    ;; composite evaluation messages
    (=> [:eval-evlis arg-list env beh-env]
             :
                       )
)
```

Figure 15.3: Primary Evaluator in ABCL/Rμ (cont'd)

```
(defbehavior Group (Actors Tasks BehEnv Eval)  ;; The Group Manager
                   ;; Actually models the transition system itself

   ;; initial script evaluation
   (=> [:start init-command]
      [Eval <= [:eval-command init-command (new Environ nil nil)
               BehEnv Me Me]])

   ;; receive evaluation result from a metaobject
   (=> [:eval-result result] ;; get evaluation result of a single actor
      (match result
         (is {Tasks Actors Replacement} ;; should be this
            (para
               (become (union Actors Replacement) Tasks BehEnv Eval)
               [Me <= [:process-next-task]]))
         (otherwise (error))))

   ;; process next task in the group
   (=> [:process-next-task] ;; self loop
      (para
         [Me <= [:process-one-task (select-one-task Tasks)]]
         (become (Actors Tasks BehEnv Eval))))

   ;; process one task
   (=> [:process-one-task task] ;; process a specified task
      (para
         (become (Actors (remove-task task Tasks) BehEnv Eval))
         (match task
            (is {:task target {pattern args} sender}
               [target <= [:message pattern args sender]]))
         (otherwise (error))))
)
```

Figure 15.4: Group Manager in ABCL/Rμ

Inter-group communication is handled simply by sending a message `:process-one-task` to the destination group manager, with the appropriate reified task data structure. This is more elegant compared to the receptionist approach taken in the metacircular definition of ACT/R, in which the receptionist actor in the meta-level must perform extra work to translate the inter-group messages to internal (meta) task of the destination group.

In order to handle reflective behavior, the group manager almost must perform the appropriate reification process, which is implicit in the design of the metacircular definition.

Chapter 16

Discussions and Future Work

16.1 Summary of Part II

We have proposed a new reflective architecture called the *hybrid group architecture*, whose primary purpose is to realize the tasks requiring group-wide object coordination such as distributed resource management in OOCR languages, while maintaining linguistic lucidity in reflection. We presented the language ABCL/R2 which is based on this architecture, and described its key features:

- Heterogeneous Object Groups and Group Shared Resources
- Non-reifying Objects
- Meta-groups and Individual/Group Reflective Towers

By the use of reflective architecture, we have shown that we can cleanly achieve orthogonal encapsulation of meta-level features relevant to concurrency, such as scheduling.

Although ABCL/R2 facilitates object groups, usefulness of the concept of object group has been been widely recognized. But unfortunately, most work on object groups from the language aspect of OOCP has been for homogeneous groups[30, 90]; as for heterogeneous groups, its definition or construction has been mostly vague (for example, [59]). In this work, we have shown that heterogeneous object groups are not ad-hoc concepts, but can be defined constructively and lucidly in an OOCR language, and how cooperative actions of objects in a group with respect to resource management at the base-level can be described in the meta-level architecture. In a sense, our proposal would serve as one reference model for (heterogeneous) object groups, and construction of reusable frameworks based on groups.

We have also presented schemes for efficiently implementing ABCL/R2 and other OOCR languages. The extended lazy creation scheme of meta-groups and metaobjects reduces the meta-level interpretation overhead. Efficient script execution is achieved by the partial compilation of non-reflective operations, and allowing for the mixture of reflective and non-reflective operations. The light-weight objects provide efficient execution of continuation objects and light-weight metaobjects. The system reduces 'unnecessary reification' via dynamic progression scheme using the light-weight metaobjects and the

forwarding mechanism. Self-reification of group kernel objects and compilation scheme of non-reifying objects allow full compilation of object scripts. The basic run-time performance of ABCL/R2, as a result, compares favorably to or sometimes even exceeds that of the ABCL/1 compiler and also conventional threads programming using C + Sun LWP in non-reflective programming, and the overhead of reflective computation is reduced by orders of magnitude compared to our previous OOCR language prototypes. This allows us to achieve practical execution efficiency for typical programs that are mixtures of non-reflective and reflective code.

The implementation of ABCL/R2 described in this paper is running on the TOP-1 Common Lisp[105] on OMRON LUNA-88K, a shared-memory computer with four 88000 CPUs running Mach. A pseudo-parallel version that runs on Kyoto Common Lisp and CMU Common Lisp was also created. The latter version is available via anonymous ftp on the Internet from camille.is.s.u-tokyo.ac.jp (133.11.12.1) so that researchers all over the world can experience the joys and intricacies of OOCR programming. We hope that it will also serve as a platform for experiments in concurrent programming, e.g., compare several concurrency control algorithms, since such algorithms can usually be encapsulated easily in the meta-level and above as we demonstrate in this paper.

16.2 Directions for the Future

Our work is by no means complete or our proposal ultimate. There are still some limitations with ABCL/R2 which we must strive to solve. For example, there is a difficulty in the management of two distinct resources exhibiting collaborative behaviors; this is necessary for realization of features proposed in advanced operating systems, where the virtual memory management coordinates with the thread scheduler. We could extend our architecture further, and/or make an approach from the group-wide reflection to the individual-based.

Recently, there have been several interesting proliferation of work on OOCR architectures other than the ones we have mentioned so far. Most of them are focused heavily on practice: they include OO||[10], Actra[74, 75], C++ extension in Choices-OS[66], RbCl[51], C++ Meta-Information Protocol[22], and Open-C++[29]. They could be classified as belonging to one architecture or the other, but the more important point is that, they *sacrifice* the full ability to do reflection for the sake of achieving practical speed and also incorporation into practical OO-languages that are being widely used, such as C++. For example, in Open C++, reflective computation and the associated metaobject protocol is restricted to reification of message passing mechanism of C++. Although this might seem to be a strong restriction compared to fully reflective languages such as our ABCL/R2, Chiba nevertheless describes in [29] that, despite these restrictions, one is able to program and create metalevel frameworks for many distributed language features such as broadcasting, atomic updates, and global synchronization.

One could compare our approach to the abovementioned restriction approach. In our scheme, the user is presented with a fully-reflective language (in fact, it faithfully runs the metacircular interpreter, albeit slowly), and it is up to the optimizing compiler and efficient runtime scheme to distinguish what are being reified and what are not.

Performance is achieved by 'avoiding' going meta as much as possible, until truly needed. However, this puts heavy burden on the compiler and run-time, and precise determination of when meta- is needed is often difficult. By contrast, in the restriction approach, it is the user's responsibility to determine if the language provides enough reflective capabilities to suit his needs. Also, such system usually do not come with optimizing compiler for reflection (they usually employ a simple compiler on top of existing compilers such as C++), and thus currently it is up to the user to reflect precisely what parts of the system are reified and what are not. This is somewhat bothersome, because the user has to have a clear notion what to reify beforehand in designing his system.

Eventually, the two strategies could be combined by investigating what parts of the language/system is essential in being reflected upon, and what are not. Then, one could devise an optimizing compilation scheme for it. In fact, such an approach is actually favorable, because by restricting the degree of reflection, the compiler can have more assumptions about the state of the system, and thus can perform better optimizations.

In any case, one could say that the big challenge is to reduce the cost of reflective operations further by the use of more elaborate compilation schemes. Since we have achieved comparable speed to non-reflective version of the language for non-reflective operations, if we could 'collapse' and compile away much of the reflective code, we would be constantly able to achieve comparable (or greater) speed for reflective programs. In particular, the most difficult problem would be how to 'reflect' the dynamically user-customized meta-level code into base-level compilation. For this purpose, we need to concentrate on three research areas:

1. The current version of ABCL/R2 allows too much freedom in the reflective programming of both the individual tower and the group tower. We should instead (1) divide the meta-level into smaller sub-functional parts (e.g., divide the evaluator into sub-evaluators) and (2) devise appropriate Metaobject Protocols[58] that allow only valid customization, thereby giving the compiler more a-priori information.

2. Develop a practical partial evaluation scheme so that the amount of compilable code could be maximized. For example, one could collapse some of the evaluator code into the metaobject so that evaluation request to the evaluator could be eliminated. (The use of partial evaluation in reflection has been suggested in [35, 118, 26], but to our knowledge, no actual reflective languages exist that have actually implemented it.)

3. Integrate an on-line compiler into the system, which is used for dynamic compilation of objects that had been dynamically modified by the user via reflection. For this purpose, various compiler technologies developed for SELF (e.g., [27], among many others) could be applicable, but many other technologies specific to OOCR architectures would have to be developed.

The ultimate goal is to have reflection be used generally in constructing malleable architectures. Figure 16.1 is a road map describing the relationships between our previous and current work, plus our future directions in search for a better OOCR architecture. Other current research topics include:

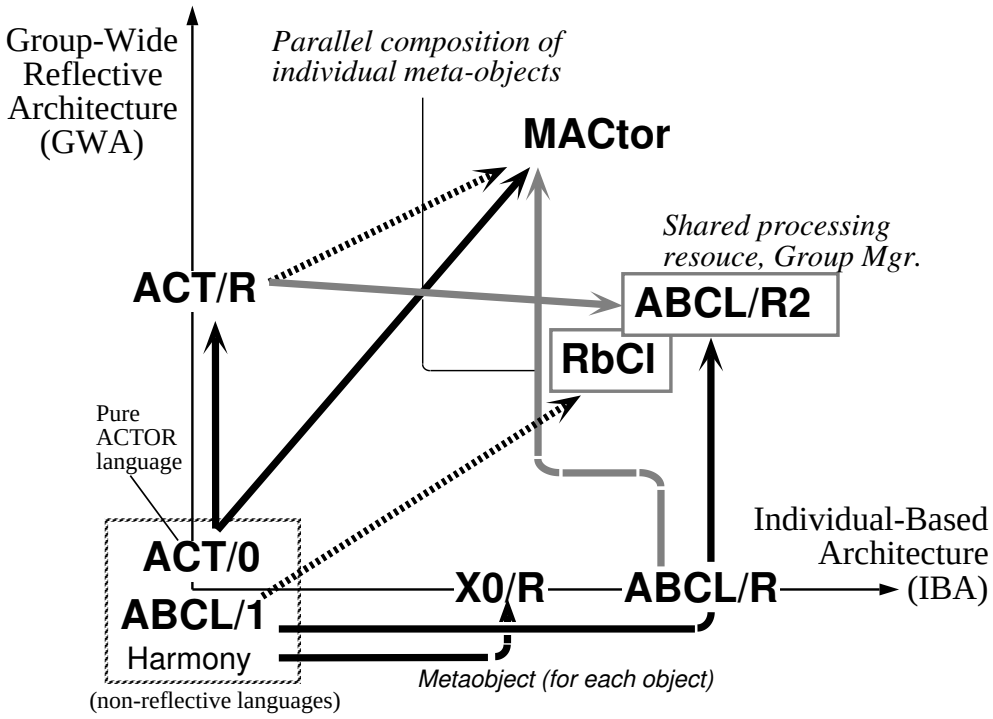


Figure 16.1: The Road Map of Our Previous and Current Works, Plus Future Directions

- To lay a more theoretical foundation for the hybrid group architecture as an abstract model, rather than giving direct formalization of ABCL/Rμ. We are currently taking an alternative approach by again using Actor transition system as the base computational model. The work is currently called *MACTOR*, and has a different formulation from ABCL/Rμ.

- To have an effective distributed implementation of ABCL/R2 on MPPs. Here, the structured mail address might serve as key in depicting the topology of node distribution. We also need to perform some technology transfers from the efficient compiler/runtime architecture of ABCL/onAP1000.

- To construct and apply reflective system to real-world applications in concurrent/distributed systems. We are experimenting with ABCL/R2 in programming of other examples, such as the dynamic optimization and deadlock detection. In particular, an extensive experiment is currently being conducted to improve the execution speed of parallel programs via transparent monitoring of individual object execution by the monitoring meta-level objects, and performing appropriate scheduling via such meta-level information. Such programming experiences would guide us in creating robust meta-level application frameworks, other efficient implementation schemes, as well as more sophisticated OOCR architectures.

In conclusion, we stress that it is not the language but the language architecture as a whole (that manifests itself with reflection) that provides the *mechanism* for integrating various user *policies* into the system for solving user-specific problems, and, such malleable system architectures are especially valuable for parallel and distributed computing using concurrent objects.

Bibliography

[1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] Gul Agha. Concurrent object-oriented programming. *Communications ACM*, 33(9):125–141, 1990.

[3] Mehmet Aksit and Anand Tripathi. Data abstraction and mechanism in Sina/ST. In *Proceedings of OOPSLA '88*, volume 23, pages 267–275. SIGPLAN Notices, ACM Press, September 1988.

[4] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of ECOOP'87*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer-Verlag, 1987.

[5] Pierre America. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of OOPSLA '90*, volume 25, pages 161–168. SIGPLAN Notices, ACM Press, October 1990.

[6] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands, May,1990*, number 489 in Lecture Notes in Computer Science, pages 60–90. Springer-Verlag, February 1991.

[7] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An overview of SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[8] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[9] J. K. Annot and P. A. M. Haan. POOL and DOOM: The object oriented approach. In P. C. Treleaven, editor, *Parallel Computers: Object-Oriented, Functional, Logic*, chapter 3, pages 47–79. John Wiley & Sons, Ltd., 1990.

[10] Peter C. Bahrs. On Reflection in Object-Oriented Heterogeneous Environments for Concurrent Processing. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[11] J. P. Bahsoun, L. Feraud, and C. Betourne. A "two degrees of freedom" approach for parallel programming. In *Proceedings of the 1990 IEEE International Conference on Programming Languages*, pages 261–270, 1990.

[12] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[13] Brian M. Barry. Prototyping a real-time embedded system in smalltalk. In *Proceedings of OOPSLA'89*, volume 24, pages 255–265. SIGPLAN Notices, ACM Press, October 1989.

[14] John K. Bennett. The design and implementation of Distributed Smalltalk. In *Proceedings of OOPSLA'87*, volume 22, pages 318–330. SIGPLAN Notices, ACM Press, October 1987.

[15] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of OOPSLA'86*, volume 21, pages 78–86. SIGPLAN Notices, ACM Press, September 1986.

[16] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in emerald. *IEEE Transactions on Software Engineering*, 13(1):65–75, January 1987.

[17] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of OOPSLA'90*, volume 25, pages 303–311. SIGPLAN Notices, ACM Press, October 1990.

[18] Jean-Piere Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent OOP. In *Proceedings of ECOOP'87*, volume 276 of *Lecture Notes in Computer Science*, pages 33–40. Springer-Verlag, 1987.

[19] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding concurrency to a statically type-safe object-oriented programming language. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 18–21. SIGPLAN Notices, ACM Press, April 1989.

[20] Christopher Burdorf and Jed Marti. Non-Preemptive Time Warp Scheduling Algorithm. *Operating Systems Review*, 24(2):7–18, April 1990.

[21] Roger M. Burkhart. Reflective functions for the C language. In *Proceedings of ECOOP/OOPSLA '90 Workshop on Reflective and Metalevel Architectures in Object-Oriented Programming*, Ottawa, Canada, October 1990.

[22] Frank Buschmann, Konrad Kiefer, Frances Paulisch, and Michael Stal. Meta-Information-Protocol: Run-time type information for C++. In *Proceedings of the IMSA International Workshop on Reflection and Meta-level Architecture*, pages 82–87. Research Institute of Software Engineering (RISE), Japan, November 1992.

[23] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science*, volume 16, pages 89–102. Springer-Verlag, 1974.

[24] Denis Caromel. A general model for concurrent and distributed object-oriented programming. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 102–104. SIGPLAN Notices, ACM Press, April 1989.

[25] Denis Caromel. Programming abstractions for concurrent programming — a solution to the explicit/implicit control dilemma. In B. Meyer, J. Potter, M. Tokoro, and J. Bezivin, editors, *Proceedings of TOOLS 3, Sydney*, pages 245–253, November 1990.

[26] Craig Chambers. Towards Efficient Implementation of Computational Reflection. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[27] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–15, Phoenix, Arizona, October 1991. Published as SIGPLAN Notices 25(11), November, 1991.

[28] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of OOPSLA'89*, volume 24, pages 49–70. SIGPLAN Notices, ACM Press, October 1989.

[29] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of ECOOP'93*, 1993. (to appear).

[30] Andrew Chien and William J. Dally. Concurrent aggregates. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 187–196. SIGPLAN Notices, March 1990.

[31] William Douglas Clinger. *Foundations of Actor Semantics.* PhD thesis, Massachusetts Institute of Technology, May 1981.

[32] Pierre Cointe. Metaclasses are first class. In *Proceedings of OOPSLA'87*, volume 22, pages 156–167. SIGPLAN Notices, ACM Press, October 1987.

[33] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of OOPSLA'89*, volume 24, pages 433–443. SIGPLAN Notices, ACM Press, October 1989.

[34] Antonio Corradi and Letizia Leonardi. Parallelism in object-oriented programming languages. In *Proceedings of the 1990 IEEE International Conference on Programming Languages*, pages 271–280, 1990.

[35] Olivier Danvy. Across the bridge between reflection and partial evaluation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 83–116. Elsevier Science, North-Holland, 1988.

[36] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in the reflective tower. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 327–341. ACM Press, 1988.

[37] Dennis Decouchant. Design of a distributed object manager for the Smallatalk-80 system. In *Proceedings of OOPSLA'86*, volume 21, pages 444–452. SIGPLAN Notices, ACM Press, September 1986.

[38] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[39] D. Decouchant et. al. A synchronization mechanism for typed objects in a distributed system. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 105–107. SIGPLAN Notices, ACM Press, April 1989.

[40] Jacques Ferber. Conceptual reflection and Actor languages. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 177–193. North-Holland, 1988.

[41] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings of OOPSLA'89*, volume 24, pages 317–326. SIGPLAN Notices, ACM Press, October 1989.

[42] Brian Foote. Object-oriented reflective metalevel architectures: Pyrite or panacea? In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[43] Narain Gehani and William D. Roome. *The Concurrent C Programming Language*. Prentice Hall, 1989.

[44] J. E. Grass and R. H. Campbell. Mediators: A synchronization mechanism. In *Proceedings of the 1986 IEEE International Conference on Distributed Computing Systems*, pages 468–477, 1986.

[45] Carl E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.

[46] John Hogg and Steven Weiser. OTM: applying objects to tasks. In *Proceedings of OOPSLA'87*, volume 22, pages 388–393. SIGPLAN Notices, ACM Press, October 1987.

[47] Kohei Honda. Interaction types. (unpublished manuscript), 1992.

[48] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[49] Waldemar Horwat. Concurrent Smalltalk on the message-driven processor. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[50] Yuuji Ichisugi, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. An object-oriented concurrent reflective architecture for distributed computing environments. In *Proc. 29th Annual Allerton Conference on Communication, Control and Computing*, Allerton, Illinois, October 1991. (To Appear).

[51] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. Rbcl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of the IMSA International Workshop on Reflection and Meta-level Architecture*, pages 24–35. Research Institute of Software Engineering (RISE), Japan, November 1992.

[52] Yutaka Ishikawa. Reflection facilities and realistic programming. *SIGPLAN Notices*, 26(8):101–110, August 1991.

[53] Yutaka Ishikawa. Communication mechanisms on autonomous objects. In *Proceedings of OOPSLA'92*, volume 27, pages 303–314. SIGPLAN Notices, ACM Press, October 1992.

[54] Yutaka Ishikawa and Mario Tokoro. A concurrent object-oriented knowledge representation language Orient-84K: its features and implementations. In *Proceedings of OOPSLA'86*, volume 21, pages 232–241. SIGPLAN Notices, ACM Press, September 1986.

[55] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[56] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor based concurrent object-oriented languages. In *Proceedings of ECOOP'89*, pages 131–145. Cambridge University Press, 1989.

[57] Kenneth M. Kahn and Vijay A. Saraswat. Actors as a special case of concurrent constraint programming. In *Proceedings of OOPSLA'90*, volume 25, pages 57–65. SIGPLAN Notices, ACM Press, October 1990.

[58] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts, 1991.

[59] Luping Liang, Samuel T. Chanson, and Gerald W. Newfeld. Process Groups and Group Communications: Classifications and Requirements. *IEEE Computer*, pages 56–66, February 1990.

[60] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA'86*, volume 21, pages 214–223. SIGPLAN Notices, ACM Press, September 1986.

[61] Henry Lieberman. Concurrent object-oriented programming in Act/1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. The MIT Press, 1987.

[62] Steven E. Lucco. Parallel programming in a virtual object space. In *Proceedings of OOPSLA'87*, volume 22, pages 26–34. SIGPLAN Notices, ACM Press, October 1987.

[63] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of ECOOP'92*, 1992.

[64] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, volume 22, pages 147–155. SIGPLAN Notices, ACM Press, October 1987.

[65] Pattie Maes. Issues in computational reflection. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 21–35. North-Holland, 1988.

[66] Peter Mandany, Panos Kougiouris, Nayeem Islam, and Roy H. Campbell. Practical examples of reification and reflection in C++. In *Proceedings of the IMSA International Workshop on Reflection and Meta-level Architecture*, pages 76–81. Research Institute of Software Engineering (RISE), Japan, November 1992.

[67] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of OOPSLA'92*, volume 27, pages 127–144. SIGPLAN Notices, ACM Press, October 1992.

[68] Satoshi Matsuoka and Satoru Kawai. Using Tuple Space communication in distributed object-oriented languages. In *Proceedings of OOPSLA'88*, volume 23, pages 276–283. SIGPLAN Notices, ACM Press, September 1988.

[69] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Synchronization constraints with inheritance: What is not possible — so what is? Technical Report 10, Department of Information Science, the University of Tokyo, 1990.

[70] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of ECOOP'91*, number 512 in Lecture Notes in Computer Science, pages 231–250. Springer-Verlag, 1991.

[71] Satoshi Matsuoka and Akinori Yonezawa. Metalevel solution to inheritance anomaly in concurrent object-oriented languages. In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[72] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.

[73] Satoshi Matsuoka and Akinori Yonezawa. Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. (Submitted), 1993.

[74] Jeff McAffer. A unified distributed simulation system. In *Proceedings of the 1990 Winter Simulation Conference*, pages 415–422, New Orleans, LA, December 1990.

[75] Jeff McAffer. Variably Asynchronous, Reflective Tasks in Smalltalk. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[76] Paul L. McCullough. Transparent Forwarding: First steps. In *Proceedings of OOPSLA'87*, volume 22, pages 331–341. SIGPLAN Notices, ACM Press, October 1987.

[77] José Meseguer. A logical theory of concurrent objects. In *Proceedings of OOPSLA'90*, volume 25, pages 101–115. SIGPLAN Notices, ACM Press, October 1990.

[78] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *Proceedings of ECOOP'93*, 1993. (to appear).

[79] Robin Milner. *Communication and Concurrency*. Prentice Hall, Engle Cliffs, 1989.

[80] Tatsuo Minohara and Mario Tokoro. Multiple meta-objects support an object. In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1990.

[81] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[82] Shin Nakajima. What makes a language reflective and how? In *Proceedings of the IMSA International Workshop on Reflection and Meta-level Architecture*, pages 125–136. Research Institute of Software Engineering (RISE), Japan, November 1992.

[83] Jürgen Nehmer, Dieter Haban, Friedemann Mattern, Dieter Wybraniez, and H. Dieter Rombach. Key concepts of the INCAS multicomputer project. *IEEE Transactions on Software Engineering*, 13(8):913–923, August 1987.

[84] Christian Neusius. Synchronizing actions. In *Proceedings of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 1991.

[85] Oscar Nierstrasz. Active objects in Hybrid. In *Proceedings of OOPSLA'87*, volume 22, pages 243–253. SIGPLAN Notices, ACM Press, October 1987.

[86] Oscar Nierstrasz and Michael Papathomas. Viewing objects as patterns of communicating agents. In *Proceedings of OOPSLA '90*, volume 25, pages 38–43. SIGPLAN Notices, ACM Press, October 1990.

[87] Oscar Nierstrasz and Michael Papathomas. Towards a type theory of active objects. In *Proceedings of the 1990 ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming, Ottawa, Canada, Oct. 1990*, volume 2 of *OOPS Messenger*, pages 89–93. ACM Press, April 1991.

[88] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL/1-D: A distributed programming system with multi model reflection framework. In *Proceedings of the IMSA International Workshop on Reflection and Meta-level Architecture*, pages 36–47. Research Institute of Software Engineering (RISE), Japan, November 1992.

[89] Joseph Pallas and David Ungar. Multiprocessor Smalltalk: a case study of a multiprocessor-based programming environment. In *Proceedings of the SIGPLAN '88 Conference on Programming Design and Implementation*, pages 268–277, 1988.

[90] Flavio De Paoli and Mehdi Jazayeri. FLAME: a language for distributed programming. In *Proceedings of the 1990 IEEE International Conference on Programming Languages*, pages 69–78, 1990.

[91] M. Papathomas. Concurrency issues in object-oriented programming languages. In D. Tsichritzis, editor, *Object Oriented Development*, chapter 12, pages 207–245. Université de Geneve, 1989.

[92] Ramana Rao. Implementational reflection in Silica. In *Proceedings of ECOOP'91*, number 512 in Lecture Notes in Computer Science, pages 251–267. Springer-Verlag, July 1991.

[93] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented language. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 289–297, 1988.

[94] J. H. Reppy. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN '88 Conference on Programming Design and Implementation*, pages 250–259, 1988.

[95] Joel Richardson, Peter Schwartz, and Luis-Felipe Cabrera. Cacl: Efficient fine-grained protection for objects. In *Proceedings of OOPSLA '92*, volume 27, pages 263–275. SIGPLAN Notices, ACM Press, October 1992.

[96] John R. Rose. A Minimal Metaobject Protocol for Dynamic Dispatch. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[97] Etsuya Shibayama. How to invent distributed implementation schemes of an object-based concurrent language — a transformational approach —. In *Proceedings of*

OOPSLA'88, volume 23, pages 297–305. SIGPLAN Notices, ACM Press, September 1988.

[98] Etsuya Shibayama. Reuse of concurrent object descriptions. In B. Meyer, J. Potter, M. Tokoro, and J. Bezivin, editors, *Proceedings of TOOLS 3, Sydney*, pages 254–266, November 1990.

[99] Etsuya Shibayama. *An Object-Based Approach to Modeling Concurrent Systems*. PhD thesis, the University of Tokyo, Tokyo, Japan, December 1991.

[100] Brian C. Smith. Reflection and semantics in Lisp. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, 1984.

[101] Brian C. Smith. What do you mean, meta? In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[102] Alan Snyder. Encapsulation and inheritance. In *Proceedings of OOPSLA'86*, volume 21, pages 38–45. SIGPLAN Notices, ACM Press, September 1986.

[103] Symbolics Inc. *Genera User's Guide*, 1990.

[104] Tomoyuki Tanaka. Actor-based reflection without meta-objects. Technical Report RT-0047, IBM Research, Tokyo Reserach Laboratory, August 1990.

[105] Tomoyuki Tanaka and Shigeru Uzuhara. Multiprocessor Common Lisp on TOP-1. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1990.

[106] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Efficient implementation of fine-grained object-wise synchronization schemes. (To be submitted), 1993.

[107] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of 4th ACM Symposium on Principles and Practices of Parallel Programming (PPoPP'93), San Diego*, May 1993. (to appear).

[108] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Incorporating locality management and gc in massively-parallel object-oriented languages. In *Proceedings of Joint Symposium on Parallel Processing*. IPSJ, May 1993. (to appear).

[109] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with Enabled-Sets. In *Proceedings of OOPSLA'89*, volume 24, pages 103–112. SIGPLAN Notices, ACM Press, October 1989.

[110] Anand Tripathi, Eric Berge, and Mehmet Aksit. An implementation of object-oriented concurrent programming language SINA. *Software—Practice and Experience*, 19(3):235–256, March 1989.

[111] Jan van den Bos and Chris Laffra. PROCOL: a parallel object language with protocols. In *Proceedings of OOPSLA'89*, volume 24, pages 95–102. SIGPLAN Notices, ACM Press, October 1989.

[112] Frank van Harmlen. A classification of meta-level architectures. In Abramson and Rogers, editors, *Meta-Programming in Logic Programming*, chapter 5, pages 103–122. The MIT Press, 1989.

[113] Prasad Vishnubhotla. Synchronization and scheduling in ALPS objects. In *Proceedings of the 1988 IEEE International Conference on Distributed Computing Systems*, pages 256–264, 1988.

[114] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, volume 20, pages 256–266, Gold Coast, Australia, May 1992.

[115] Ken Wakita and Akinori Yonezawa. Linguistic supports for development of organizational information systems. In *Proceedings of ACM COCS*, November 1991.

[116] Mitchell Wand and Danel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. North-Holland, 1988.

[117] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA'88*, volume 23, pages 306–315. SIGPLAN Notices, ACM Press, September 1988. (Revised version in [127]).

[118] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands*, May 1990. also number 489 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[119] Peter Wegner. Dimensions of object-based language design. In *Proceedings of OOPSLA'87*, volume 22, pages 168–182. SIGPLAN Notices, ACM Press, October 1987.

[120] Rebecca Wirfs-Brock and Ralph Johnson. Surveying current research issues in object-oriented design. *Commun. ACM*, 33(9):105–124, September 1990.

[121] Masahiro Yasugi, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Proceedings of 6th ACM International Conference on Supercomputing*. ACM, 1992.

[122] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of OOPSLA '92*, volume 27, pages 414–434. SIGPLAN Notices, ACM Press, October 1992.

[123] Yasuhiko Yokote, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. Reflective object management in the Muse operating system. In *Proceedings of the IEEE International Workshop on Object-Orientation in Operating Systems*, pages 16–23. IEEE Computer Society Press, 1991.

[124] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In Stephen Cook, editor, *Proceedings of ECOOP'89*, pages 89–106. Cambridge University Press, 1989.

[125] Yasuhiko Yokote and Mario Tokoro. The design and implementation of Concurrent Smalltalk. In *Proceedings of OOPSLA'86*, volume 21, pages 331–340. SIGPLAN Notices, ACM Press, September 1986.

[126] A. Yonezawa, H. Matsuda, and E. Shibayama. Discrete event simulation based on an object oriented parallel computation model. Technical Report C-64, Dept. of Information Science, Tokyo Institute of Technology, 1984.

[127] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* Computer Systems Series. The MIT Press, 1990.

[128] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of OOPSLA'86*, volume 21, pages 258–268. SIGPLAN Notices, ACM Press, September 1986.

[129] Akinori Yonezawa and Takuo Watanabe. An introduction to object-based reflective concurrent computations. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 50–54. SIGPLAN Notices, ACM Press, April 1989.

[130] Kaoru Yoshida. *A'UM: A Stream-based Concurrent Object-Oriented Programming Language.* PhD thesis, Keio University, Japan, March 1990.

Appendix A

Previous Work that Addressed Conflicts of Inheritance vs. Concurrency/Distribution in OOCP languages

Here, we review previous work on OOCP languages that addressed other, less problematic conflicts between inheritance/delegation and concurrency/distribution compared to this thesis. Wegner[119] classifies various dimensions of OOCP language design, such as inheritance, concurrency, persistence, etc, and how they are related with or independent from one another. A recent survey of programming languages for distributed systems by Bal et al.[12] analyzes OOCP languages in terms of comparisons with other class of languages, but does not cover the issues addressed in this paper.

A.1 Conflict between Inheritance and Distribution

One well known difficulty, as Wegner describes in [119], is the consistency maintenance of class definitions among the distributed processors in class-based OOCPs. For example, updating of the definition of class K would pose the following difficulties: the changes in the class definition must be propagated consistently to all its instances, which are distributed throughout the system. Not only is this a non-trivial task, but it is more difficult when the descendent classes of K as well as their instances are also distributed throughout the system, because they must be notified of the changes as well. If the notification of the changes is not properly propagated, the system may become inconsistent, because the instances may or may not be aware of the changes depending on how the classes are distributed and how the method lookup is performed. The difficulty can be somewhat resolved with replication of class definitions at each node, but that too would cause consistency maintenance problems as well as requiring large amount of memory per each node.

Distribution of class definitions was not a problem for languages designed for (or implemented on) shared address-space architectures [89, 125, 54], but naturally arose for

distributed address-space architectures. The problem has been identified in practice, for example, in several works in distributed versions of Smalltalk systems [14, 37, 68, 76]. In fact, some distributed programming languages such as Emerald[15, 16] and the most recent version of SR[7] goes to the extreme, allowing only the inheritance of *method specifications*, but no automatic code sharing of the inherited methods.

Wegner[119] also points out that object modularity and sharing achieved by inheritance are conflicting goals:

> However, there is a potential conflict between the independence required for concurrency and the structured sharing required for inheritance. This is particularly true when concurrency is augmented by the stronger requirement that processes be distributed ... In fact, we can say that distribution is inconsistent with inheritance ... *(because)* the inconsistency between the goals of modularity and sharing are incompatible.

We expect that various consistency maintenance schemes developed for distributed databases, such as *transactions*, could be applied in order to achieve a reasonable solution for OOCP systems of coarse to medium grain granularity. Fine grain systems are in need of extensive future research, however.

A.2 Conflict between Delegation and Synchronization

Briot and Yonezawa[18] discusses the problems of interaction of *message delegation* in the style of Lieberman[60] verses the *synchronization* of delegated messages. Consider the situation where the original receiver object a delegates a message to object b, and the activated method in b contains a reference to the state variable (i.e., instance variable) of the originator a. Then, variable access is done via a message send from b to a. But this is not trivial, because nondeterminism in the message communication could potentially disturb the atomicity of the method execution. In order to avoid the problem, selective message reception scheme as exemplified by the *insensitive actors* in the Actor model and the *waiting-mode* in ABCL/1 is necessary; but this causes another problem of deadlocks in the presence of recursion i.e., a message sent to self.

To resolve the above conflict, Briot and Yonezawa propose two new schemes of delegation in OOCP, and discuss their merits and drawbacks:

- the *recipe-query scheme* — a receiver object queries another object for the *recipe (i.e., method)* object handling the received message, instead of delegating the message to another object. The merit is that the atomicity of message processing is preserved; the drawback is that it could incur heavy overhead of copying the actual body of methods.

- the *copy-everything scheme* — the variable dictionaries and method dictionaries are copied to the inheriting object. Although this approach is effective for relatively static configurations, the storage overhead could potentially be enormous.

Other problems with delegation we point out is that, (1) if the target of the delegation is shared by multiple objects, then the target could become a potential bottleneck, and (2) if the chain of delegation is long, then all the intermediate objects could get suspended until the message is finally processed. These problems can be overcome by assuming actor-style *replacement behavior* in the underlying language semantics, thereby allowing pipelined processing of delegated messages.

Appendix B

Appendix: Definition of the Time Warp Group

Figure B.1 illustrates how one defines a Time Warp Group, and how one could program a scheduler for it. The scheduling algorithm is the Lowest LVT (Local Virtual Time) First scheduler[20], but it can be interchanged dynamically with any valid Time Warp scheduler. Figures B.2, B.3, B.4, and B.5 are the skeletal definition of the Time Warp group.

```
;;; The Time Warp group

[group TW-group
 (meta-gen TW-meta-gen)
 (evaluator [TW-eval-gen <== [:new Lowest-LVT-First-Scheduler]])
 ;; ... additional scripts & initialization expressions here ...
 ]

;;; The Lowest Local Virtual Time First Scheduler

[object Lowest-LVT-First-Scheduler
 (state [queue := [priority-queue-gen <== [:new :test-fun #'<]]])
 (script
  (=> [:schedule Message :with Id LVT]
      (match Message
       (is [:anti-message . ARGS]
           ;; Annihilation of the positive of this anti-message.
           (if [queue <== [:have? [:message . ARGS]]]
               [queue <= [:remove [:message . ARGS]]]
               [queue <= [:enq Message :with LVT]]))
       (otherwise
        [queue <= [:enq Message :with LVT]])))
  (=> :next @ C
      [queue <= :deq @ C])
  (=> :contents @ C
      [queue <= :listify @ C])
  (=> [:copy List]
      [queue <= [:enq-list List]])
  )]
```

Figure B.1: The Usage of Time Warp Group

```
[object TW-Meta-Gen   ;;; The Metaobject Generator of the Time Warp group
 (script
   (=> [:new StateVars LexEnv Script Evaluator GMgr]
       ![object TW-meta
         (state [queue := [queue-gen <== :new]]
                [pqueue := [priority-queue-gen <== :new]]
                [state := [undoable-env-gen <== [:new StateVars LexEnv]]]
                [output-history := [output-history-gen <== :new]]
                [scriptset := Script]
                [evaluator := Evaluator]
                [group := GMgr]
                [mode := ':dormant]
                [LVT := 0])
         (script
           ;; Ordinary messages (omitted)
                :
           ;; TimeWarp messages
           (=> [Type Message Reply Sender VST VRT]
                   where (member Type '(:message :anti-message))
               [pqueue <= [:enq [Message Reply Sender VST VRT] :with VRT]]
               (when (eq mode :dormant)
                 [mode := ':active]
                 [Me <= :begin-tw]))
           (=> :begin-tw
               (if [queue <== :empty?] ; check ordinary message queue first
                   (match [pqueue <== :next]
                     (is [Message Reply Sender VST VRT] where (>= VRT LVT)
                         [state <= [:push LVT]]
                         [LVT := VRT]
                         (match (find-script Message scriptset)
                           (is [Vars Body]
                               [evaluator <=
                                   [:do-progn Body
                                      [env-gen <== [:new Bindings state]]
                                      [den Me] GMgr output-history LVT] @
                                   [cont ignore
                                      [Me <= :end]]])
                           (is NIL
                               (warn "~A cannot handle the message: ~S"
                                     [den Me] Message)
                               [Me <= :end])))
;; (next page)
```

Figure B.2: Group Kernel Objects of the Time Warp Group (Metaobject Generator)

```
                    (is [Message Reply Sender VST VRT] where (< VRT LVT)
                        ;; State rolls itself back to the most recent time
                        ;; before VRT and returns the value of the time.
                        [LVT := [state <== [:rollback-to VRT]]]
                        ;; Input queue for TimeWarp messages also rewound
                        [pqueue <= [:rollback-to VRT]]
                        ;; Send anti-messages.
                        (dolist
                          (h [output-history <== [:history-since VRT]])
                          (match h
                            (is [Target Message Reply VST VRT]
                                [Target <= [:anti-message Message
                                            Reply [den Me] VST VRT]])))))
                    [Me <= :begin]))

            (=> :begin
                ... same as 'vanilla' meta objects ...)

            (=> :end
                (if (not [queue <== :empty?])
                    [Me <= :begin]
                    (if (not [pqueue <== :empty?])
                        [Me <= :begin-tw]
                        [mode := ':dormant])))
            )])
    )]
```

Figure B.3: Group Kernel Objects of the Time Warp Group (Metaobject Generator)(cont'd)

```
[object TW-Eval-gen       ;;; The evaluator of the Time Warp group
 (script
   (=> [:new Scheduler]
       [object TW-eval
        (meta TW-eval-meta-gen)
        (script
         (=> [:do Exp Env Id Gid Outputs LVT] @ C
             (match (parse-exp Exp)
                    :
             (is [:variable Var]  ;; variables and pseudo-variables
                 (match Var
                  (is 'Me ![den Id])
                  (is 'Group !Gid)
                  (is 'LVT !LVT)
                  (otherwise [Env <= [:value-of Var] @ C])))
                    :
             (is [:send-tw-mesg Target Message Reply VRT]
                 [Me <= [:do-evlis
                           [Target Message Reply VRT]
                           Env Id Gid
                           Outputs LVT] @
                  [cont [target* message* reply* vrt*]
                   [C <= ()]
                   (if (not (null target*))
                     (progn
                       [[meta target*] <=
                           [:message message* reply* LVT vrt*]]
                       [Outputs <=
                          [:push [target* message* reply* LVT vrt*]]]
                       ))]])
                    :
             (is [:object-def Name Meta-gen-spec State Script]
                 [Me <= [:do Meta-gen-spec Env Id Gid Outputs LVT] @
                  [cont meta-gen*
                   (if (null meta-gen)
                     [Gid <= [:new State Script] @
                      [cont object
                       (if Name
                         [Env <= [:set-global Name object] @ C]
                         !C)]])]])
             )))]
       [[meta TW-eval] <= [:set-scheduler Scheduler]]
       !TW-eval
  ))]
 )]
```

Figure B.4: Group Kernel Objects of the Time Warp Group (TimeWarp Evaluator)

```
[object TW-Eval-meta-gen    ;;; The special metaobject for TW-Eval
 (script
  (=> [:new StateVars LexEnv Scripts Evaluator GMgr]
      ![object Eval-Meta
        (meta non-reifying-meta)
        (state [scheduler := nil]
               [evaluator := Evaluator]
               [scriptset := Scripts]
               [mode := ':dormant])

        (script
         (=> [:message Message Reply Sender]
             (match Message  ; = [:do Exp Env Id Gid VRT]
              (is [_ _ _  Id _ _]
                  [scheduler <= [:schedule Message :with Id Time]]))
             (if (eql mode ':dormant)
               (progn [mode := ':active]
                      [Me <= :begin])))
         (=> :begin
             ... almost same as 'vanilla' meta objects ...)
         (=> :end
             ... same as 'vanilla' meta objects ...)

         ;; Additional methods
         (=> [:set-scheduler NewScheduler]
             [scheduler := NewScheduler])
         (=> [:change-scheduler NewScheduler]
             [NewScheduler <= [:copy [scheduler <== :contents]]]
             [scheduler := NewScheduler])
         )]))]
```

Figure B.5: Group Kernel Objects of the Time Warp Group (Metaobject of Evaluator)