Pure Library and Reflection Based Language Extension Techniques for Object Oriented Systems

UNIVERSITY OF CALIFORNIA
Santa Barbara

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor in Philosophy in Computer Science

by Murat Karaorman

Committee in charge:

Professor John Bruno, chair
Professor Anurag Acharya
Professor Amr El Abbadi
Professor Urs Hölzle

June 2001

Acknowledgements

This has been a long long ride, indeed. Quite a few among my friends and family had long stopped asking me “when?”, and then when I mentioned voluntarily that I was defending my thesis there was initial disbelief, some hesitancy, yet somehow no element of surprise, no one really must have thought I would not take my sweet time through this -- that I did! If you are reading this and know what I am talking about, I salute you!

It has been over twelve years since this journey began. A lifetime. Long enough to meet my love, Lumina, get married and managed to stay married for 10+ years now. Long enough to see my two lovely daughters enter and exit their teenage years, and graduate before me. See my younger brother beat me to “Doctor” title by a few years -- that, too, had been a much awaited happening. The journey has seen me through two top-notch full-time R & D positions, 6 years at Panasonic’s Speech Technology Labs and the last two years at Texas Instruments. Teaching at UCSB’s College of Creative Studies has taken another fun-filled 5-6 years of my free-time there as well:) I feel blessed to have actually met so many people and journeyed through so many seas and countries that this life carried me to. I must now announce my sincerest gratitude to my wife Lumina, for actually staying with me through this juggler’s life.

I dedicate this dissertation to the wonderful and most beautiful women who graced my life. Thank you Mom & Lumina. Thank you Trinity and Lara, and of course thank you Sibel. Trin is the daughter I have always wanted: she has constantly provided great love and joy. She is also in all the vocals and dances. Lara brought great animals and great friendship. Thanks for Dueder and Tigger. Mom & Lumina are the source of my spiritual grounding and love.

Ron’s friendship and support carried me through this entire journey, I just cannot picture it without you, Ron. His wife Pepa, however, is genuinely the greatest, and I owe Pepa more than I can possibly ever give her. Thanks.

I am deeply grateful to the CS Department faculty and staff. It feels like one big family. John Bruno has given me just the right amount of guidance and has been a great match for me as an advisor. Divy and Amr have offered great friendship and mentoring to not only me, but several other close friends here. You make this school very special. Alan Konheim has told my parents many years ago “not to worry, because I am the type who will finish”, and my parents still talk about it. Thanks Alan. Thanks Urs, Anurag and Klaus for your support, knowledge and wisdom. Mary Jane and Leslie Campbell you are my heroes. Thank you Erdal. Thanks also to Ozan, & Nedim. Finally thanks Gokhan and my father Ozer.

Those whose names I did not mention, you know who you are (or more like you know me:) drop me a note and let’s get together sometime. And celebrate.

May we all live in peace and harmony.

VITA

Murat Karaorman

Place of Birth: Bolu, Turkey
Date of Birth: March 26, 1964

Email: murat@cs.ucsb.edu
URL: http://www.cs.ucsb.edu/~murat

EDUCATION

Doctor of Philosophy in Computer Science, University of California, Santa Barbara, California, June 2001

Master of Science in Computer Science,. Bilkent University, Ankara, Turkey, 1988

Bachelor of Science in Computer Science. Middle East Technical Univ., Ankara, Turkey, 1986

PROFESSIONAL EMPLOYMENT

1999 - Present *Texas Instruments*, Santa Barbara, California. Member of Technical Staff. Senior Software Applications Engineer.

1993 - 1999 *Panasonic Speech Technology Laboratory*, Santa Barbara, CA. Research Engineer & Project Leader in the Speech Recognition Group.

1995 - 2001 *College of Creative Studies, University of California, Santa Barbara* CA. Instructor & Program Coordinator of the Computer Science Program

1989 - 1995 *Computer Science Department, University of California, Santa Barbara*, CA. Research Assistant (Distributed Systems Laboratory), Teaching Associate & Assistant

PUBLICATIONS

1. Karaorman, M., Hoelzle, U., Bruno, J. "*jContractor: A Reflective Java Library to Support Design-by-Contract.*" In Second International Conference, Meta-Level Architectures and Reflection. Lecture Notes in Computer Science, Springer, July 1999, Vol: 1616, pp. 175-196.

2.Karaorman, M., Bruno, J., "*Active-RMI: Active Remote Method Invocation System for Distributed Computing Using Active Java Objects.* " In Proceedings of Int'l Conference on Technology of Object Oriented Languages and Systems (TOOLS USA '98) August 3-7,1998. Copyright 1998 IEEE. pp.414-427

3.Karaorman, M., et al. "*An Experimental Japanese/English Interpreting Video Phone System*". Proceedings of Fourth International Conference on Spoken Language Processing - ICSLP '96 (October 3-6, 1996, Philadelphia PA, USA.) pp.1676-80.

4.Murat Karaorman, John Bruno. "*Introducing Concurrency to a Sequential Language.* " Communications of the ACM. September 1993, Vol.36, No.9, pp.103-116.

5.Murat Karaorman. "*Concurrency Libraries for Object-Oriented Systems.*" OOPSLA '94 Doctoral Symposium. (October 23-27, 1994, Portland Oregon.) Workshop presentation.

6.Murat Karaorman. "*A Technique for Implementing Highly Concurrent Wait-free Objects for Shared Memory Multi-processors.*" Unpublished manuscript. Web-document -- http://www.cs.ucsb.edu/~murat/WAIT-FREE.ps

7.Murat Karaorman, John Bruno. "*Design and Implementation Issues for Object-Oriented Concurrency.*" In Proceedings of OOPSLA '93 Workshop on Efficient Implementation of Concurrent Object-Oriented Languages.(September 27, 1993, Washington D.C.) Proc. Ed. L.V.Kale, pp.M-1-8.

8.Murat Karaorman, John Bruno. "*A Concurrency Mechanism for Sequential Eiffel.*" In Proceedings of Int'l Conference on Technology of Object Oriented Languages and Systems (TOOLS USA '92) Conference.(Aug. 3-6, Santa Barbara,Calif.). Prentice Hall 1992, pp.63-77.

9.D.Probert, J.Bruno, M.Karaorman. "*SPACE: A New Approach to Operating System Abstraction.*" In Proceedings of IEEE International Workshop on Object-Orientation in Operating Systems.(October 17-18 1991, Palo Alto, Ca.) pp. 133-137.

10.Ozelci, S., Karaorman, M., Kesim, N., Arkun, E. "*An Experimental Object-Oriented Database Management System Prototype.*" In Computer and Information Sciences-3.} (also in Proceedings of ISCIS III. The Third International Symposium on Computer and Information Sciences, Cesme, Turkey, 29 Oct.-2 Nov. 1988). Edited by: Gelenbe, E.; Orhun, E.; Basar, E. Commack, NY, USA: Nova Sci. Publishers, 1989. p. 457-64.

11.*Design and Implementation of a Secondary Storage Manager for an Object Oriented Database Management System.* M.S. Thesis. Bilkent University, Ankara, September 1988.

PATENTS

1."*Method for goal-oriented speech translation in hand-held devices using meaning extraction and dialogue*." United States Patent# 6,233,561 (May 15, 2001)

2."*Multi-pass Natural Language Parsing with N-Best Tag-Generation for Robust Topic and Information Discovery".* United States & European Patents Pending.

TUTORIAL, CONFERENCE & WORKSHOP PRESENTATIONS

1."*Nuts & Bolts of Object Based Distributed Computing*." Half-day Tutorial Presentation. International Conference on Technology of Object Oriented Languages and Systems (TOOLS USA '98) August 5,1998. Tutorial Notes: 170 pages. Copyright 1998 IEEE.

2."*From C++ to Advanced Java in One Day.*" Full-day Tutorial Presentation as a substitute presenter. (TOOLS USA '98) August 2,1996.

3."*Object-Oriented Distributed Computing.*" Half-day Tutorial Presentation as a substitute presenter. (TOOLS USA '98) August 3,1996.

4.Presented papers (listed in the publications) at TOOLS '92, OOPSLA '93, OOPSLA '94, ICSLP '96, TOOLS '98 , Reflection '99 international conferences and workshops.

RESEARCH & DEVELOPMENT

1. Designed and implemented "*Active-RMI: Remote Object Invocation System for Active Java Objects"* with the related Java class libraries and tools for distributed computing over internet using: asynchronous remote calls with future type results; remote-object creation with transparent class loading; synchronization programming and scheduling. (with John Bruno)

2. Designed and implemented *jContractor,* a set of Java class libraries and an associated contract design pattern to facilitate specification and enforcement of design contracts in Java by allowing programmer to specify and enforce transparently at run-time pre/post conditions for class methods and class invariant conditions. (with Urs Hoelzle)

3. Designed and implemented *Class CONCURRENCY,* library extension to Eiffel for concurrency and distributed computing. Developed a method for translating sequential Eiffel programs to concurrent programs. (with John Bruno)

4. Designed operating system micro-kernel abstractions for concurrent and object-oriented languages for the SPACE Operating System kernel designed at UCSB. (with Dave Probert & John Bruno)

5. Designed a technique for non-blocking and wait-free objects in shared memory multi-processors.

6. *TMS320 DSP Algorithm Standard* Technical Lead at Texas Instruments (with Dr. David Russo & Robert Frankel)

7. *Speech Recognition, Speech Translation, Speech Understanding and Audio Interfaces* at Panasonic Speech Recognition Laboratory (with Dr. Jean-Claude Junqua, Dr. Roland Kuhn). Key project leads and involvements:

i. Speaker Independent Large Vocabulary Continuous Speech Recognition (Olympia Project)

ii. English-Japanese Video Phone/ Speech-to-Speech Translation System (IVP Project).

iii. Computer Telephony Products and Applications [interfacing with a Norstar DM-4 PBX].

iv. Flexible Vocabulary Speaker Independent Recognition Project (FlexVoice)

v. Speech Understanding for Meeting Objectives: SUMO Travel Agent Reservation System.

vi. Car Navigation Demonstration System.

vii. Toolkits for Visualization, maintenance and evaluation.

ABSTRACT

Pure Library and Reflection Based Language Extension Techniques for Object Oriented Systems

by

Murat Karaorman

In order to cope with the increasing complexity and evolving requirements of software, we need better languages and modeling tools which support higher and more suitable levels of abstraction. The original research work conducted as part of this dissertation introduced new language extensions for concurrency, distributed computing and design by contract for the object oriented languages, Eiffel and Java. We demonstrated that powerful new abstractions can be introduced using a purely library based approach and a set of naming and programming conventions without violating the object-oriented principles or compromising language or safety features.

We developed techniques to build extensible, open object-oriented libraries that use and take advantage of the existing low-level system and vendor libraries, platforms, frameworks to support several new high-level abstractions. Programmers use these abstractions by following our naming and programming conventions. We illustrate the applicability of our approach by building and presenting three original research application systems: *Class CONCURRENCY, Active-RMI* and *jContractor. Class CONCURRENCY* introduces concurrency, active objects, asynchronous calls, data-driven synchronization and scheduling abstractions to Eiffel. *Active-RMI* introduces asynchronous remote method invocation with future-type results; asynchronous result delivery; transparent remote-object creation; active-object semantics; user programmable scheduling and synchronization to Java. *jContractor* introduces *Design by Contract* constructs: preconditions, postconditions, class invariants, recovery and exception handling to Java through an intuitive naming convention, and standard Java syntax. *jContractor* uses reflection and dynamic class loading terchniques to discover contract specifications by examing Java class bytecodes and synthesizes and loads an instrumented version of the Java class which incorporates code that enforces the runtime checking of contracts.

Pure Library and Reflection Based Language Extension Techniques for Object Oriented Systems

by

Murat Karaorman
2001

TABLE OF CONTENTS

Chapter 1

Introduction

Designing a software system involves meeting the requirements of the problem at hand with a given set of software and hardware capabilities. The type of software design tools and abstractions available to the designer greatly influences the feasibility, the amount of effort, and the quality of the solution. Operating systems and high-level programming languages and libraries provide the basic software abstractions available to the programmer. The fundamental software and hardware abstractions do not change nearly as fast as the pace at which hardware, memory and communication becomes faster and cheaper. As hardware keeps getting cheaper and more powerful, computer applications continue to reach into larger, more complex and sophisticated domains.

In order to cope with the increasing complexity and evolving requirements of software, there is always a need for better modeling tools and languages supporting higher and more suitable levels of abstraction. This dissertation describes techniques for introducing new fundamental abstractions to object-oriented languages without requiring changes to the language or its development environment. Our approach includes designing class libraries and associated programming and naming conventions to provide new abstraction capabilities to object-oriented programming languages. We illustrate the applicability of this approach by building and presenting three original research application systems. We discuss in detail the design and implementation of class libraries, programming and naming conventions and reflection techniques to introduce high-level language abstractions to Eiffel and Java for concurrency; asynchronous calls and data-driven synchronization; active objects; scheduling; distributed computing; remote object creation; and design by contract.

1.1 Statement of the Problem

Each high-level programming language provides a fixed set of abstractions that programmers use to design software systems and applications. These core programming abstractions typically differ little from one general purpose language to another, and generally offer a low-level, abstract view of a virtual machine. They include support for representing and naming data objects and structures; typing; procedural abstractions: functions, function calls, name spaces, argument marshalling; syntactic abstractions for writing arithmetic and logical expressions and programming flow control. Object oriented languages offer additional abstractions to support data hiding, data encapsulation, inheritance, polymorphism, classes, interfaces. Increasing complexity of software systems demand enhanced productivity, efficiency, and ease of maintenance and continually require programming languages and environments to evolve and offer better suited and higher level modeling capabilities.

The use of libraries is a well-known technique to extend a language with new features and functionality. In fact, most of the statically typed mainstream languages keep basic user-level operating system (O/S) services and abstractions such as streams, files, file-systems, dynamic memory allocation, processes, threads, interprocess communication, networking, etc., outside the language definition, and instead rely on the presence of externally linked system and run-time support libraries and standard application programming interfaces (APIs) to access them. For example, standard I/O libraries extend C language in such a way that most C programmers use and think of these calls (printf, scanf, etc.,) as if they were part of the C language definition. Java uses a rich set of libraries (called packages) and APIs to support windowing abstractions, event-modeling, imaging, etc.,

The library based extension approach is also quite common in interpreted or scripting languages. Examples of some of the popular and successful cases include Common Lisp, Tcl, and Perl. Common Lisp extension CLOS provides object orientation support. Tcl has been extended to provide graphical and windowing support, Tcl/Tk, object-orientation support, Incr-Tcl. Perl has been extended to provide seamless access to just about any operating system function.

There are several advantages to the approach of employing libraries to support new abstractions and functionality: the core language remains more compact and simple and therefore easier to specify, implement and port to different architectures. The libraries can be developed and maintained separately and efficiently and in most cases the libraries can simply be implemented as an indirection layer above the native O/S services.

The availability of extended functionality in the form of generic, system-level libraries is very useful, however, the challenges for building large and complex systems still remain as long as the abstractions offered in the form of library APIs are low level. General systems and libraries typically offer a rich set of low level APIs to offer greater control and flexibility. This assigns more responsibility to a programmer to keep track of state, consistency, and correct usage, and is therefore error prone. The following is a list of some of the techniques that have emerged in practice to cope with evolving complexity of software when using low-level abstractions:

- Layering or hierarchical decomposition of libraries and services, —networking and communication services and protocols, event-handling, windowing systems
- Extending the O/S services to offer more and higher-level user-level APIs — thread, synchronization, scheduling, communication, event-handling, IPC, graphics, …
- Supporting platform specific frameworks —MFC, COM, OLE/ActiveX, VxWorks/Tornado…
- Supporting platform independent frameworks —X, AWT/JFC/Swing, CORBA, RMI,
- Providing automation toolkits, wizardry, and software engineering tools —IDEs, CASE Tools, Interface Builders, Beans, …
- Using Design Patterns [30].

While these techniques offer some help in dealing with complexity they typically require steep learning curves and a fair amount of commitment to a platform even to perform fairly simple or common tasks. Additionally these techniques and frameworks typically do not provide an open or extensible architecture, often requiring going out of the box and reverting to a custom, more complex solution when there is a paradigm mismatch.

Finding the right (high) level of programming abstractions is the greatest challenge with these techniques. The problem is in part due to the fact that the general techniques are too broad or horizontal, as they cater to as large domains of applications as possible. Problems start cropping up in vertical, or specialized domains, such as high-performance computing, real-time systems, scalable and collaborative systems with custom communication needs, etc. Another problem area is in the application domains of emerging technologies and research environments -- some current examples are wireless communications, speech recognition, telephony, internet devices, mobile and ubiquitous computing.

The ideal way to support high-level abstractions to address a particular problem

domain is to have direct language support. This approach can provide additional safety and efficiency. Historically, the following approaches have been used:

1. Design a brand new language to address specific problems in targeted application domains and platforms.
2. Extend an existing language with new and non-standard extensions using customized compilers, preprocessors, or run-time environments.

Both of these approaches offer custom and dedicated solutions but this in itself may cause problems. It may be impossible or impractical for programmers to migrate to a new language and/or development or runtime environment. Maintenance, reliability, security and future support issues might also hinder acceptance.

In this dissertation we describe techniques for providing an alternative approach:

3. Extend an existing object-oriented language using a purely library based approach to provide a new set of high-level abstractions.

1.2 Dissertation Research Overview

The key research work conducted as part of this dissertation consists of introducing new language extensions by designing class libraries for concurrency, distributed computing and design by contract for the object oriented languages, Eiffel and Java. We demonstrate that powerful new abstraction capabilities can be introduced without violating the object-oriented principles or compromising other language or safety features, while using a purely library based approach and a set of naming and programming conventions.

We developed techniques to build extensible, open object-oriented libraries that use and take advantage of the existing lower-level system and vendor libraries, platforms, frameworks to support several new high-level abstractions. We present the design and implementation details of the research prototype systems we have developed to introduce the following high-level programming abstractions:

- Concurrency.
- Active objects.
- Asynchronous calls.
- Data driven synchronization.

- Scheduling.
- Active remote objects.
- Remote object creation.
- Design by Contract.

Eiffel is an object oriented language which has built in support for *design by contract*, but does not have concurrency features. We have designed and implemented an Eiffel class library, *Class CONCURRENCY,* which provides concurrency, active objects, asynchronous calls, data-driven synchronization and scheduling abstractions to Eiffel objects as encapsulated and inheritable properties. Objects which inherit from *Class CONCURRENCY* can acquire a separate thread and state and become active with a programmable scheduler. Active objects' methods can be called asynchronously with a deep-copy pass-by-value semantics for normal object arguments and reference passing semantics for active object arguments. We discuss the concurrency abstractions and design issues in Chapter 2, and present the implementation details of the *Class CONCURRENCY* in Chapter 3.

Java has a basic low-level thread abstraction to support concurrency and has remote method invocation (RMI) to support distributed remote objects. We have designed and implemented the *Active-RMI* system which introduces active objects, asynchronous calls, data driven synchronization, scheduling and remote object creation to Java. We have also designed and implemented a reflective Java library, *jContractor* to introduce *Design by Contract*. We discuss the remote active object and design by contract abstractions and design issues in Chapter 2, and present the implementation work for *Active-RMI* libraries in Chapter 4, *jContractor* libraries in Chapter 5.

Table 1.1 gives a brief summary of the abstractions we have introduced to Eiffel and Java. We discuss the key abstractions and design issues in Chapter 2. We present the design and implementation details of the *Class CONCURRENCY, Active-RMI* and *jContractor* in chapters 3,4 and 5 respectively. *jActivator* system has not been implemented and is briefly discussed in the future work section of the Conclusion.

ABSTRACTIONS	Class Concurrency (Eiffel)	Active-RMI (Java)	jContractor (Java)	jActivator (Java)
Concurrency		N	N	N
Active Local Objects				X
Asynchronous Calls	X	X		X
Data Driven Synchronization	X	X		X
Scheduling	X	X		X
Distributed Computing		N	N	N
Active Remote Objects	X	X		X
Remote Object Creation	X	X		X
Design by Contract	N			
Pre Conditions, Post Conditions, Class Invariants			X	
Special syntax (old, result)			X	
Exception Handling			X	

Legend: X = Newly added feature; N = Native language feature

Table 1.1: Overview of Dissertation Research.

1.3 Summary of Research Contributions

Our primary contribution is to have shown by way of designing and implementing how class libraries and associated programming and naming conventions can be used to introduce new abstractions to object-oriented languages.

Significant contributions to modeling active objects, concurrency and distributed object computing were made through the design and implementation of Class CONCURRENCY and Active-RMI systems.

We have also introduced design by contract abstractions to Java by designing and implementing the *jContractor* system. This work also allowed us to introduce new techniques using reflection and dynamic class loading to perform runtime instrumentation.

Some more specific contributions are listed in the Conclusion, in the section “Summary of Key Contributions” .

Chapter 2

Introducing New Features to Languages using Library Extensions

In this chapter we discuss design issues and techniques for introducing new fundamental abstractions to object-oriented languages without requiring changes to the language or its development environment, and without violating the object-oriented principles or compromising other language or safety features. Our approach is based on designing extensible, open, object-oriented libraries that work with our naming and programming conventions and utilize existing lower-level system abstractions, libraries, and frameworks. This chapter is organized into three sections discussing our approach and design issues while introducing three fundamental sets of abstractions: *concurrency, distributed active objects*, and *Design by Contract*. As part of this dissertation work we have built three systems: *Class CONCURRENCY* to introduce concurrency, active objects, asynchronous calls, data-driven synchronization and scheduling abstractions to Eiffel; *Active-RMI* to introduce asynchronous remote method invocation with future-type results; asynchronous result delivery; transparent remote-object creation; active-object semantics; user programmable scheduling and synchronization to Java; *jContractor* to introduce Design by Contract to Java. The design and implementation details of the *Class CONCURRENCY, Active-RMI* and *jContractor* are individually presented respectively in Chapters 3,4 and 5.

2.1 Introducing Concurrency to Object Oriented Systems

Object-oriented paradigm appears to be well-suited for concurrent programming. Objects can be used to implement not only data-structures but also processes that are

objects with a protected internal state and a prescribed behavior. The extended view of objects as processes having a protected private state and a prescribed behavior, provides the bridge to parallelism, since most approaches in parallel programming are based on the notion of process. Also the communication/synchronization aspects of concurrent programming blends well with the basic message passing (or method invocation) model of computation in object oriented programming. Our approach to introducing concurrency is through supporting the *active object* and *asynchronous remote method invocation* with *data-driven synchronization* abstractions.

There are numerous design considerations that need to be addressed in determining which concurrency abstractions to support and how to integrate them with the object model. In Section 2.1.1 we present these issues. Our main concern is that the integration of concurrency and object-oriented programming should not result in the sacrifice of the advantages of either or both worlds. Concurrency abstractions should be supported as natural extensions to the object model that does not violate the principles of object-oriented programming — such as reusability, data encapsulation, data abstraction, inheritance, polymorphism. Designing a concurrency class library along with an object-oriented concurrent program *design method* and tools to extend an existing object-oriented language has several advantages, and this alternative should be considered before any other scheme which involves making modifications to the syntax, semantics or runtime aspects of the language.

In the Section 2.1.1 we present some of the design issues and approaches taken in designing concurrent object oriented languages. In Section 2.1.6 we present the advantages of choosing a library based approach for introducing concurrency to Eiffel. A full description of our concurrency mechanism, and an associated concurrent programming design method describing how a concurrent application can be designed from sequential object specifications, and how this process can be automated, along with some examples is discussed in Chapter 3, “Introducing Concurrency to a Sequential Object-Oriented Language” on page 41 and in [41].

2.1.1 Design Issues for Concurrency Abstractions

The central abstraction for concurrency in most parallel programming systems is the notion of *process* or *thread* of control, which represents a virtual processor executing instructions within a context. Concurrency implies the possibility of multiple threads executing in parallel as parts of the same computation. The notion of a single thread of control already exists (implicitly) in the sequential object-oriented programming. A single thread starts executing instructions within the context of a ’root’ object, and each method call (return) of an object’s method transfer the thread of control to (from)

the called object. At any given point in time a single thread of control appears to be executing the instructions within the context of some object one of whose methods has most recently been invoked.

Following is a list of design issues pertaining to the *thread* abstraction:

- Is the notion of *active objects* supported? Otherwise how is an independent (parallel) activity (*thread*) represented by the object model?
- If the active object notion is supported, how can an object become active?
- Are *multiple threads* of execution allowed inside an object's methods?
- What is the *granularity* of concurrent access to an object?
- What mechanism is used to ensure *mutual exclusion* in the presence of multiple threads?

Following is a list of issues that are related to the *coordination* of parallel activities:

- How are concurrent threads or activities *synchronized*?
- How do concurrent threads or activities *communicate*?
- If communication is based on message passing:
  Is *message acceptance* or *message sending* explicit?
  Is message delivery synchronous or asynchronous?
- What type of *scheduling policies* are supported?

Following is a list of issues related to the *language model* and *distribution*:

- Do all objects reside in the same *address space*?
  This issue greatly affects other parts of the design. In particular, message passing and synchronization mechanisms greatly depend on the presence (or absence) of a shared memory assumption. Thread abstractions will also be closely related to the address space assumption.
- If objects can reside at distributed sites, how are object references passed as *parameters*? Do they need to deep-copied? Migrated?
- Can all communication be statically type-checked?
- Do special *libraries* exist for extending or refining the concurrency abstractions?
- Which *language* is used to express sequential constructs?
- What type of abstractions or concurrency mechanisms are assumed (or expected)

to be provided by the underlying operating system.

The answers to these questions have a profound affect on the type and amount of concurrency that is attained by the model. Nearly any combination of these key design choices exist in the literature.

We have introduced concurrency to the sequential object-oriented language, Eiffel, with no language extensions or modifications to the compiler by providing special concurrency classes. Table 2.1, “Overview of Introduced Concurrency Abstractions,” on page 13 lists the main abstractions and features of our Eiffel concurrency extensions.

A summary of our design decisions for introducing a concurrency model for Eiffel following the guidelines we have discussed earlier in this section is as follows:

- The concurrency approach is based on active objects.
- Only those objects which inherit from *Class CONCURRENCY* can become active.
- Objects can become active by acquiring an independent *thread* of control when the `split` method is invoked. The `split` method is inherited from the *Class CONCURRENCY* (Library).
- Active objects communicate by *asynchronous remote invocation* of their methods.
- Synchronization is *data-driven* based on *future* type result objects.
- Message acceptance is explicit, achieved by providing a special *scheduler* routine for each active object.
- Each active object supports a single thread of activity.
- All active objects reside at disjoint address spaces, and therefore
- All object references must be deep-copied.

CONCURRENCY ABSTRACTIONS	
Active Object Instantiation & References	
Feature	*Description*
`Class CONCURRENCY`	Must be inherited for an object to become active.
`Create`	Instantiates the proxy of an active object.
`split`	Creates active object in separate address space & sets up connection with the proxy.
`attach`	Connects the proxy with an existing active object.
Asynchronous Call	
Feature	*Description*
`REQUEST Class`	Encapsulates data and communication parameters of an asynchronous call, associated with a unique call-id.
`remoteInvoke`	The proxy takes method name and call parameters, creates a REQUEST object and asynchronously delivers it to the attached active object's request queue. Returns unique call-id associated with future result of the call.
Synchronization & Reply Scheduling (Proxy side)	
Feature	*Description*
`claimResult`	Blocks until result is received for the call associated with the given call-id. Returns the result of the computation.
`resultAvailable`	Polls the proxy's result queue. Non-blocking. Returns true if result is ready for given call-id.
`data`	Applies to a FUTURE object. Blocks until result is received for the call associated with the given FUTURE object. Returns the result of the computation.
`isReady`	Applies to a FUTURE object. Polls the proxy's result queue. Non-blocking. Returns true if result is ready.

Table 2.1: Overview of Introduced Concurrency Abstractions

CONCURRENCY ABSTRACTIONS	
Request Scheduling (Server side)	
Feature	*Description*
`requestQueue`	Server's queue of incoming REQUEST objects asynchronously sent by proxies.
`scheduler`	User defined routine defines the behavior of the active object typically by scheduling incoming requests. Scheduling logic can utilize the name and signature of the operation & the runtime values of the arguments.
`getRequest`	Blocks scheduler code until arrival of new request(s) or a specified time-out.
`pendingRequestExists`	Polls request queue about new request(s) without blocking.
`sendResult`	Asynchronously sends the result of the current request to the responsible proxy.

Table 2.1: Overview of Introduced Concurrency Abstractions

Our choices in addressing the design issues are largely guided by our desire to introduce concurrency to Eiffel using class libraries, and coming up with a concurrency mechanism that respects object-oriented design principles. Subsequent sections provide more discussions about the key elements of our design. A detailed description of the *Class CONCURRENCY,* examples, and an automated design method for converting sequential classes to concurrent classes is presented in Section Chapter 3.

2.1.2 Active Objects

Our view of concurrency is based on the notion of *process* and its integration with the notion of *object*. This unification of the notion of a process and object results in the concept of an *active object*. Objects can become active only if they inherit from the *Class CONCURRENCY.* Concurrency can then be viewed as the parallel execution resulting from the creation of these active objects and their interactions with each other.

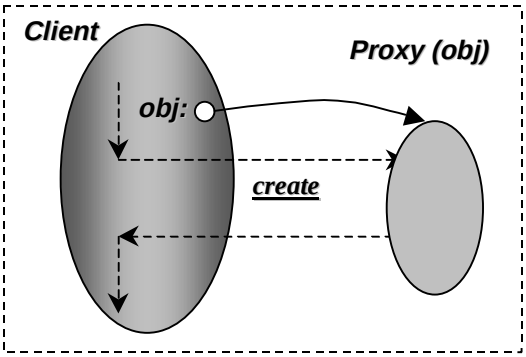


Figure 2.1: . Process State after Create

The *Class CONCURRENCY* preserves the instance creation semantics of sequential Eiffel objects. The `Create` method instantiates a sequential *proxy* object. The actual creation of an active object with its own *process* (or *thread*) is achieved by invoking this *proxy* object's `split` method. This method starts a new process with an independent thread of control and returns back to the client after the *active object* is created and communication ports are established and initialized. Then the active object begins executing a special start-up method, *`scheduler`*. The methodology requires that the *`scheduler`* method is defined for each class that inherits from *CONCURRENCY.* This method specifies how to serve the requests generated by *clients* using the `remoteInvoke` method. All requests are delivered as actual messages to the server using a transparent inter-process communication (IPC) mechanism. Since multiple clients can simultaneously request services, the communication is *buffered.*

For figures 2.1 and 2.2, assume that `obj` inherits from *CONCURRENCY.* Figure 2.1 depicts the situation after the *creation* of the `obj's` *proxy* by executing the following statement in one of *client*'s methods:

```
obj.Create;
```

Figure 2.2 depicts the situation after the following statement in one of *client*'s methods at a later point in the execution:

```
obj.split;
```

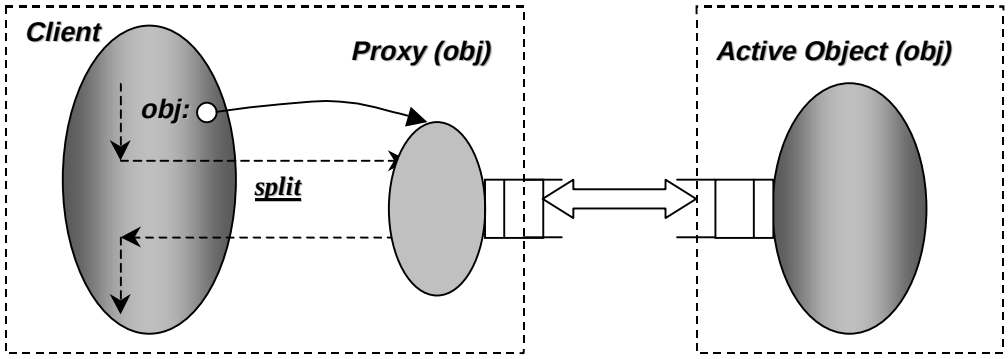


Figure 2.2: Process State after `split`

In both figures, the dotted boxes represent processes residing in distinct address spaces. Solid arrows are the object references pointing to the circles which represent the actual objects. The dotted line arrows inside an object in each process represent a thread executing a method of the object. The dotted line arrows between objects — represented as circles — correspond to the method call and value return between these objects. Finally the striped bi-directional arrow between two objects in separate processes represent the link between the dedicated communication ports of the *proxy* server object, and the corresponding *active* server object. After a `split`, the server object in the client's process act as a *proxy* to the actual active server object. This proxy object transparently and asynchronously relays the requests to the active server's *request queue*, and returns the results of requests which are asynchronously delivered to its *result queue* by the server.

Once a server object has been made *active*, via a `split` operation, new clients that want to invoke the server's methods must create their own proxy copies, and use the `attach` method to set up the association between the proxy and the active server object. A reference to the communication port of the active object must be first obtained from an 'informed' client, as an argument. Calling `attach` is significantly cheaper than `split`, since it doesn't involve creation of a new address space. Using `attach` is the only mechanism to share active objects.

2.1.3 Asynchronous Calls and Synchronization

In the sequential object oriented paradigm, *method invocation* is a *synchronous* procedure call, and objects are *passive* entities, doing work only when their methods are invoked. We call the object invoking the method a *client*, and the invoked object a *server.*

The *Class CONCURRENCY* implements a *non-blocking*, *asynchronous* method invocation mechanism called `remoteInvoke`. Consider the following code where `obj` is an *active* object,

```
callID: INTEGER;
returnVal:T;


callID:=obj.remoteInvoke("method", arg_list);
   .
   .     Concurrent execution with obj
   .
-- use callID to obtain the associated result
returnVal ?= obj.claimResult(callID);
```

The client thread in this example does not wait for the completion of `obj's` execution of `method`, but rather concurrently continues executing its own code until the point in its own execution where it actually needs the result of the execution of `method`. The part of the client's code shown with dots above is executed in *parallel* with the `obj's` code. The result returned by `claimResult` conforms to all types, so it is reverse-assigned to the correct type of `returnVal`, using the **`?=`** reverse-assignment attempt operator. The type, *T*, of `returnVal` is the actual type returned by `obj's` *method* feature. This is a data-driven synchronization scheme, based on asynchronous message passing, and is referred to as the *wait-by-necessity* by Caromel [18] .

The *Class CONCURRENCY* provides two methods to access the result of a `remoteInvoke`. Both of the methods take a single argument, a `callID`, which is returned by the corresponding remote invocation. The method, `claimResult`, as used in the example of the preceding paragraph, returns the result delivered by the server. If the result is not available yet, then `claimResult` *blocks* until it becomes available. The other method, `resultReady`, is a *non-blocking* test for the availability of result of the remote invocation associated with the `callID`. All the underlying communication is encapsulated and hidden from the application. The `callIDs` returned by remoteInvoke are similar to the ConcurrentSmalltalk's CBoxes [72] and ABCL/1's *future type messages* [73].

Since active objects resides in separate and address spaces, objects that appear as parameters of remote methods of active objects can not be passed by reference but must be (deep) copied.

2.1.4 Future Type Results and Higher Level Abstractions

The `remoteInvoke` method is a low-level abstraction and has the following disadvantages:

- The *method-name* argument needs be to passed as a string making standard static type-checking not possible;
- The method cannot take variable number of arguments, requiring the actual arguments to be stored in a variable-sized array.

The lack of support for reflection, runtime instrumentation or dynamic loading of class binaries in Eiffel limited our abilities to provide a higher level abstraction that can be supported directly by the standard language environment. However, in the section *“Generation of Concurrent Class Prototypes from a Sequential Class” on page 51* we describe a design method which provides static-type checking, and variable-size argument passing capabilities for remote invocations which results in a simpler, safer and more transparent utilization of the `remoteInvoke` method.

This design method outlines a method to transparently call `remoteInvoke` using the same typesafe syntax of a sequential call as shown in the following code example, where `active_obj`, is an active object designed by applying our design method. Using future type results provides simpler and more way to transparent access remote invocation results. Explicit invocation of `claimResult` and `resultAvailable` is also eliminated by the utilization of the *`FUTURE`* type. The *`FUTURE`* type objects returned by the proxy object are similar to the Concurrent Smalltalk’s C Boxes[72] and ABCL/1’s future type messages [73].

```
future: FUTURE;
returnVal : T;
future:=active_obj.method(arg1, arg2...);
.
.        concurrent execution with obj
.
returnVal?=future.data;
--implicit claimResult from active_obj
```

2.1.5 Scheduling

Two types of scheduling are supported by our concurrency mechanism: *reply scheduling* and *request scheduling*. *Reply scheduling* is the control the client has over the delivery of the reply and *request scheduling* is the control the active object server has over the acceptance and service of the requests [58]. *Reply scheduling* is addressed by the `remoteInvoke`, `claimResult` and `resultReady` methods as presented in the previous section. This section describes the *request scheduling* methods: `getRequest`, `pendingRequest`, `sendResult`, and the *`scheduler`*.

One of the requirements of the methodology for designing *active* objects is to define a *start-up* method, called *`scheduler`*. The `split` call transfers control to the *`scheduler`* method of the target active object. The *`scheduler`* has exclusive access to the concurrency related internal state of the active object, and specifies the behavior of the *active* object.

Message acceptance is *asynchronous* and *explicit*. *Accepting* a message is separated from actually *serving* the request. Thus the model provides a powerful mechanism for dealing with *local delays*, which is deemed essential in Liskov et al.'s [46] formulation of concurrency requirements for developing client/server type distributed programs.

A queue of request messages, called `request_queue` is created for each active object to store all accepted requests from client objects. Messages delivered to the communication buffer, but not yet accepted into the `request_queue` are called *pending* requests. Each entry in the queue contains all of the parameters of the client's remote invocation request, including its `callID` and reply address. The server object explicitly needs to show intent to accept messages: `getRequest` is a potentially *blocking* call that commits all pending requests into the `request_queue`, and blocks if no pending request exists until the first one arrives. The *non-blocking* variant of this method is the `pendingRequest` method.

The *`scheduler`* has unrestricted access to the `request_queue`, and it can inspect the parameters and names of the requests in the queue in order to select and serve one of them. It can also choose to wait for a certain type of request to arrive, or a certain condition before it selects a request for service. Sending a result back to a client after the service of a request is also done *asynchronously* and *explicitly*, using the `sendResult` primitive.

Since there is a single execution thread inside each *`scheduler`*, the execution of all incoming requests are serialized. This also applies to the passive objects within each process, since no sharing of passive objects is possible due to the deep copy semantics

of arguments of `remoteInvoke` method. Therefore all requests to the methods of a passive object are also serialized. This eliminates the need for synchronization inside the methods of objects and allows us to have all synchronization points correspond to communication events.

2.1.6 On Designing Libraries to Introduce Concurrency

Some of the advantages of designing class libraries to introduce concurrency to a sequential object-oriented language are listed below

- Libraries provide a more *flexible* and *extensible* solution for concurrency since they can be tailored to the specific needs and characteristics of the target operating system and hardware by modifications or refinement of the libraries. Concurrency abstractions that are hard-wired into the language may be impractical or impossible to change. A similar analogy exists in the operating system research. It is more desirable to implement smaller (micro) kernels that move a lot of the traditional kernel abstractions and services out of the kernel (and implement as application-level processes) in favor of reduced complexity and size and enhanced flexibility — even though it might have been more efficient to provide these services within a larger, monolithic kernel.
- It is important that *reuse of sequential libraries* can be supported after concurrency is introduced. Radical changes to the language and the object model may render existing sequential code obsolete. Whereas, extensions through libraries will support the design of sequential objects in much the same way it was before concurrency is introduced. Hence *reusability* is improved since existing sequential code can be incorporated into a concurrent application easily.
- It is more *practical* and *easier* to design and maintain a concurrency library than inventing a new concurrent object-oriented language, or modifying an existing language *and* its compiler (even when there are compilers available for modifications).
- By using a strictly object-oriented technique — designing reusable class libraries — to introduce concurrency and keeping the original language in tact, the *principles of object-oriented programming and design* are respected. Adding concurrency through modifying the language may add a great deal of *complexity* and restrictions to its future evolution. It might even be impossible or difficult to port the language to new hardware or operating system platforms with the added concurrency specifications. Libraries offer a modular and robust mechanism for supporting constantly evolving hardware and O/S platforms.

- Object-oriented libraries support user level extensions.
- Object-oriented libraries can support systematic layered views allowing different levels of abstractions to be delivered to different types of users. While the casual user can use the high-level abstractions of concurrency, more sophisticated users can use (and *reuse*) the entire library, with all of its lower-level abstractions (such as IPC, O/S interaction, etc.,.) and extend or design new higher-level abstractions.
- Designing libraries for concurrency can help designer from fully committing to a specific style of concurrency. September 1993 issue of the Communications of the ACM contains 4 different proposals for introducing concurrency to Eiffel language [19], [41], [48], [55], all with quite distinct approaches.
- *Users* are less likely to switch to a non-standard language extension that is customized for concurrent programming especially if switching would require also switching (or abandoning) tools and existing libraries, whereas library based extensions can be easily incorporated to the user's programming environment.
- Library based approaches can be quite *efficient* since all concurrency and synchronization abstractions must eventually be derived from or mapped onto underlying platform dependent operating system resources and services.

Some issues are harder to deal with when designing libraries. Static type-checking of objects is a problem for communication. Details of having to explicitly initialize or set up communication and provide stub generation, etc. may seem as the disadvantages of the library approach. Along with our library based extension, we provide the programmer with a design method that prescribes how to automatically extend sequential classes to concurrent ones, generate proxies and type-safe stubs for communication, and synchronization (for details see Chapter 3: "Introducing Concurrency to a Sequential Object-Oriented Language" on page 41) Buhr et.al.[15] mention some problems with library design for introducing concurrency, but their arguments are mostly pertaining to the context of the concurrency mechanism they adopted for μC++ [14], a C++ extension.

2.2 Distributed Object Abstractions

In section 2.1 we have presented the *Class CONCURRENCY,* a library based approach to introduce concurrency to a sequential language, Eiffel. In this section, we discuss a similar technique to introduce new distributed computing abstractions to Java: the *Active Remote Method Invocation* system (*Active-RMI).*

2.2.1 Background: Distributed Object Computing

Development and acceptance of open communication standards and protocols have made it feasible and a practically possible to interconnect the tremendously diverse heterogeneous computers and operating systems under a unified framework, the Internet. Open standards and widely accepted protocols such as low-level TCP/IP, UDP/IP [68] or higher level protocols such as HTTP, FTP, RPC[9], etc., make it relatively easy to develop networking and communication oriented software in which the emphasis is either on exchange or streaming of well-structured data/messages, or requesting remote services with well-known interfaces. Such programs are typically tedious and require a programmer to parse and map the message stream to the application semantics explicitly, usually in a platform and language specific fashion. This approach works well for such tasks as basic information retrieval, web-browsing, chatting.

Distributed computing is a more encompassing form of programming which uses networking and communication to enhance a local computation by potentially distributing portions of the computation among different hosts; or to collaborate with other executing programs towards accomplishing a global task. What makes distributed programming challenging is the semantic gap between programming language semantics involving local and remote computations and the communication abstractions. Distributed programs require higher levels of abstractions then TCP/IP or other basic networking protocols. Certain standards and tools have been emerging recently that help programmers write language independent “abstract” interface specifications that can be implemented separately, potentially in different languages through different language bindings. The separation of interface from implementation reduces greatly the complexity the programming of the communication and inter-operability requirements especially in a heterogeneous environment, but the semantic gap between local and remote computation still needs to be explicitly solved by the programmer. Some examples of well-adopted tools and standards for distributing programming are: lower level mechanisms such as RPC[9] DCE[59] or object-based distribution protocols such as CORBA[60], RMI[70], ILU[35], HORB[67] or DCOM[10]. All of these systems add a distribution layer on top of a general purpose programming language system: an intermediate *interface definition language* (IDL) and an IDL-to-application language tool is provided to support creation of remote stubs/proxies, and for registering or locating remote interface implementation objects/servers. Other less popular or experimental solutions exist as research prototypes of distributed operating systems such as Sprite[27], Inferno[49] that support a distributed scope and access primitives as operating system functions; or languages specifically designed for distributed applications such as Emerald[36], Telescript[69], Agent-Tcl[32].

2.2.2 Design Issues and Active-RMI Approach

The *Active Remote Method Invocation* system *(Active-RMI)* is a set of class libraries, tools, and a design method which provides high-level abstractions for building distributed Java applications based on the notion of *active remote objects*. *Active-RMI* is a natural extension of our work involving introducing concurrency extensions to Eiffel language using class libraries[41]. *Active-RMI* is implemented as an extension to the *Java RMI* system which was integrated by Sun Microsystems into the Java Language[31] with the release of JDK 1.1. The key contributions of *Active-RMI* system to distributed computing is the extension of *Java-RMI* with the following features, which are explained in greater detail in Chapter 4: “Active-RMI: Asynchronous Remote Method Invocation System for Distributed Computing and Active Objects” .

1. Asynchronous remote method invocation with future-type results
2. Asynchronous result delivery from the server to the caller.
3. Remote-object creation by the client via transparent class exporting/uploading.
4. Active-object semantics for remote objects created using *Active-RMI* protocol.
5. User programmed scheduling and synchronization of remote method invocations.

Table 2.2, “Overview of Introduced Distributed Computing Abstractions.,” on page 24 provides a general outline of our approach. Using *Active-RMI* one can write complete class specifications in Java describing remote active-object interfaces and implementations, designate remote hosts (or brokers), and remotely create or discover new active-objects on the participating remote hosts. Upon creation or discovery of an active remote-object, an *Active-RMI* program can reference and synchronously or asynchronously remote invoke its public methods using the same syntax as if it were a local Java object. Remote hosts need to be active and willing participants in the *Active-RMI* protocol, but they do not need to have a-priori knowledge of the types of active objects that they will host, as *Active-RMI* protocol transparently exports the needed class definitions in a demand-based fashion. Remotely created objects are started as autonomous agent-like objects, with a thread executing a designated method of the object, which can also access a private request queue of incoming *Active-RMI* requests from clients.

2.2.3 Active Remote Objects

We use the term *active object* to describe an object which has an independent thread; has access to a dedicated request queue of incoming method invocation requests; and

has the ability to inspect, select and serve any of the requests in its request queue. *Active-RMI* system extends the Java object model by giving active objects autonomy over how and when to respond to their clients' requests. Each active objects is created with its dedicated server `scheduler` thread, and an *Active-RMI request queue.* An *Active-RMI* server utility creates each active object with its request queue and a dedicated thread and then returns a remote-stub reference to the client which requested the active remote object's creation. Clients access the remote *Active-RMI* object using a *Java RMI* compatible remote object reference. The active object's thread begins execution in its `scheduler` method.

DISTRIBUTED COMPUTING ABSTRACTIONS	
Remote Server location and remote object creation	
Feature	*Description*
`aRmiServer` *interface*	Allows a client to lookup or transparently create active objects on the remote server.
`createRemoteObject` *method*	Allows a client to lookup or transparently create active objects on the remote server.
Active Remote Objects & Dynamic Class Exporting	
Feature	*Description*
`UnicastActiveRemoteObject` *class*	Used to create active remote objects. Uses RMI remote reference layer, communication subsystem, provides a request queue & default (FIFO) scheduler .
`ClassExporter` *interface* & `implementation` *class*	Used transparently by the createRemoteObject method for exporting referenced classes to the remote server.

Table 2.2: Overview of Introduced Distributed Computing Abstractions.

DISTRIBUTED COMPUTING ABSTRACTIONS

Asynchronous method invocation and synchronization

Feature	*Description*
`_Stub` *classes*	Used transparently by clients to make asychronous remote method calls. The stub object creates and forwards a request object for each call and returns a call id without waiting for result.
`aRmi_Request`	Encapsulates an active-RMI call. Contains unique call-id, method identifiers and serialized call arguments.
`CallHandle` *class*	A unique call-handle is transparently created and placed in call registry for each asynchronous call.
`getResult` *method*	Blocking method of the call-handle object to return the call result.
`resultReady` *method*	Non-blocking method of the call-handle object to check call result availability.

Active-RMI Server, naming and registry services

Feature	*Description*
`aRmiServer` *interface* `&` `implementation` *class*	Standalone server application object providing remote active object creation, naming and registry services.
`aRmiSecurityManager` *class*	Default security manager allowing dynamic class loading & thread groups.
`aRmiClassLoader` *class*	Required class loader for importing active remote object classes.

Table 2.2: Overview of Introduced Distributed Computing Abstractions.

DISTRIBUTED COMPUTING ABSTRACTIONS	
Active Remote Object instantiation.	
Feature	*Description*
`_Skeleton` *classes*	The implementation class corresponding to each active object that gets created on the server with a private request queue and scheduler.
`aRmi_RequestQueue`	Private, dynamically updated queue of incoming Active-RMI Requests.
`scheduler` *method*	The designated method of each active remote object where execution starts upon creation. User programmable method defines the behavior of the active object.
Request scheduling, dispatching and synchronization.	
Feature	*Description*
`waitForRequest` *method*	Request Queue method which blocks scheduler thread until arrival of new request or time-out.
`getFirstRequest`, `getLastRequest` *methods*	Request Queue method returning first (last) request object in request queue with matching criteria. Request gets dequeued.
`peekFirstRequest`, `peekLastRequest` *methods*	Request Queue method returning first (last) request object in request queue with matching criteria. Request queue is not altered.
`getMethodSignature` *method*	Request object method returning String containing requested method's signature.
`args`	Array component of Request object containing array of serialized objects corresponding to actual call arguments.
`serve` *method*	Request object method that dispatches the request. Dispatch implicitly sends result back to caller.

Table 2.2: Overview of Introduced Distributed Computing Abstractions.

2.2.4 Asynchronous Remote Calls and Call Handles

All calls to *Active-RMI* objects are asynchronous. A client object initiates a remote call by transparently invoking the corresponding stub method. The stub method creates a request object for the remote call, marshalling the method descriptor, serialized arguments and a unique call identifier. The stub registers the request object with a user level client-side call *registry* and asynchronously sends the request object to the remote server. Stub method returns the call identifier without waiting for the result back from the remote server.

The client obtains a *call handle* associated with the *Active-RMI* request by looking up the call registry entry using the call identifer returned by the stub method. Client can access the result of the remote method's execution only using the call-handle and its interface functions:

- `getResult`: for blocking receive
- `resultReady`: for non-blocking querying

2.2.5 User Programmable Scheduling and Synchronization

In *Active-RMI* active object model, each request to the active remote object is delivered as an `aRmi_Request` message object into the target object's private request pool, `RequestQueue`. There are no service guarantees, or an implicit order in which the arriving messages will be served. Each active object is created with its private `RequestQueue`, and uses the thread executing its *`scheduler`* to decide how to process requests placed into the queue. The *`scheduler`* method, however, is not restricted to performing only scheduling activities. It can perform general purpose computations and engage in an *agent-like* behavior. The active object's scheduling thread accesses the `RequestQueue` directly and peek into the queue of `aRmi_Requests`, and implement a selection policy to choose and serve one of the requests in the queue. The scheduling policy can involve inspecting each request's type, or its method signature, or the values of its actual arguments. The scheduler thread may also decide not to serve any of the current requests in the queue; or may wait until a certain event takes place, or a certain request arrives. A simple LIFO *`scheduler`* code, enforcing a last-in-first-out style servicing of its request queue is shown below in Figure 2.3.

```
    private void Scheduler()
            throws java.rmi.RemoteException,Exception
    {
       while(true) {
            while (!RequestQueue.empty()) {
                 aRmi_Request currentRequest =
                                   RequestQueue.getLastRequest();
                 currentRequest.serve();
            }
            RequestQueue.waitForRequest(); // blocking call
      }
   }
```

Figure 2.3: A LIFO Scheduler Implementation

2.2.6 Remote Active Object Creation

A special *Active-RMI* server process, implementing the `aRmiServer` interface needs to be running on a server host in order for a client application to be able to remote-create active-RMI objects on that host. A new active remote object can be created on the remote host via `createRemoteObject` method which returns a remote stub reference to the newly created active object. Without the *Active-RMI* server process running, a local object can still create active objects on its own host and export them to clients possibly residing on different hosts, but clients on remote machines will not be able to directly connect to this host and create active remote objects on it. The *Active-RMI* server process is registered using standard *RMI registry* mechanism using the name "*aRmiServer*". Client applications can locate and obtain remote object references to the server using the standard `java.rmi.Naming` utility:

```
aRmiServer rHost = (aRmiServer)java.rmi.Naming.lookup(
                                   "//hostName/aRmiServer");
```

Active-RMI class libraries provide an implementation class, `ARrmiServer_Impl`, which implements the `aRrmiServer` interface. This class can be used to instantiate an *Active-RMI* server object as part of an application. Alternatively, a command-line utility, `aRrmiServer`, is provided and can be started independently at each host offering the Active-RMI remote object creation service.

The primary function of the *Active-RMI* server object implementing the `aRmiServer` interface is to facilitate remote object creation. It provides the `createRemoteObject` method, used by clients to spawn new active remote objects on the host server, and returns an `ActiveRemote` object reference, a subtype of `java.rmi.Remote`. The returned reference acts as a proxy to the actual remote object: client's invocation of the proxy's methods are handled as asynchronous *Active-RMI* requests, executed on the remote host.

Two variants of `createRemoteObject` are defined. Both can be invoked with a single string argument, which contains the implementation class path of the active remote object implementation:

- `createRemoteObject( String` *className*)
- `createRemoteObject( String` *className*, `ClassExporter` *ce*)

The latter form is typically used transparently by the stub implementation of the *Active-RMI Server* object. If the server cannot find some of the classes locally, it throws an exception forcing the client's stub to pass a `ClassExporter` object. When the client responds by passing a class exporter object, a new class loader, `aRmi-ClassLoader,` is created and installed transparently by the server utilizing the remote class exporter object received from the client to download required client-side classes and create the active object on the server. In order to facilitate the need for installing a new class loader as required by *Active-RMI* remote object creation protocol, the `aRmiServer` process uses *Active-RMI*'s security manager, `aRmiSecurity-Manager`, which allows installing new class loaders. Finally, the `aRmiServer` process handling the `createRemoteObject` request creates the active object requested by the client, exports it using the RMI remote reference layer and returns a stub reference to the client. For more details see section "Remote Object Creation And Distribution" on page 66.

2.2.7 Request Scheduling and Server Synchronization

Each request to an active objects gets queued in its `RequestQueue` as an `aRmi_Request` object which represents the client's remote method invocation in the form of a message containing the type and signature of the method and the serialized arguments list, `args`, in the form of an `Object` array. While executing inside the *`scheduler`* method, the active object can examine its `RequestQueue` by peeking into the request objects. The scheduler thread can make selection decisions based on the signature of the requested method, or the contents of its actual parameters. The request gets served by invoking the its `serve` method:

```
thisRequest.serve();
```

which executes the target method of the current active object with the actual arguments, and then asynchronously sends the request back to the client which had invoked the remote method.

The `RequestQueue` supports the standard `java.util.Enumeration` interface to allow iterations over its elements as well as providing several convenience functions:

```
getFirst();
getFirst(String signature);
getLast();
peekFirst() & peekLast()
```

Further synchronization methods provide waiting behavior which block until a particular request type arrives into the `RequestQueue`:

```
waitForRequest();              // any request
waitForRequest(String signature);// request w/matching sig.
```

Each *Active-RMI* request object is associated with a thread which has been suspended just before it began to execute the target method with the passed arguments. Once a decision is made to serve the request, the thread can be resumed by:

More design and implementation details of the Active-RMI system are discussed in Chapter 4, “Active-RMI: Asynchronous Remote Method Invocation System for Distributed Computing and Active Objects” on page 61.

2.3 Design by Contract Abstractions

In sections 2.1 and 2.2 we have presented respectively two library based extension techniques: the *Class CONCURRENCY,* to introduce concurrency to a sequential language, Eiffel, and *Active-RMI* to introduce new distributed computing abstractions to Java. In this section we introduce our third research system, *jContractor*, a purely library-based system which introduces Design by Contract abstractions to Java. The *jContractor* system also relies on a set of naming conventions and utilizes the dynamic class loading and reflection capabilities of Java.

2.3.1 Background: Design by Contract

One of the shortcomings of mainstream object-oriented languages such as C++ and Java is that class or interface definitions provide only a signature-based application interface, much like the APIs specified for libraries in procedural languages. Method signatures provide limited information about the method: the types of formal parameters, the type of returned value, and the types of exceptions that may be thrown. While type information is useful, signatures by themselves do not capture the essential semantic information about what the method does and promises to deliver, or what conditions must be met in order to use the method successfully. To acquire this information, the programmer must either analyze the source code (if available) or rely on some externally communicated specification or documentation, none of which is automatically checked at compile or runtime.

A programmer needs semantic information to correctly design or use a class. Meyer introduced Design By Contract as a way to specify the essential semantic information and constraints that govern the design and correct use of a class [51]. This information includes assertions about the state of the object that hold before and after each method call; these assertions are called *class invariants*, and apply to the public interface of the class. The information also includes the set of constraints that must be satisfied by a client in order to invoke a particular method. These constraints are specific to each method, and are called *preconditions* of the method. Each precondition specifies conditions on the state of the object and the argument values that must hold prior to invoking the method. Finally, the programmer needs assertions regarding the state of the object after the execution of a method and the relationship of this state to the state of the object just prior to the method invocation. These assertions are called the *postconditions* of a method. The assertions governing the implementation and the use of a class are collectively called a *contract*. Contracts are specification constructs which are not necessarily part of the implementation code of a class, however, a runtime monitor could check whether contracts are being honored. An example of a contract is shown in a language independent form in Table 2.3, “Contract Specification for Inserting Element to Dictionary,” on page 32. This example shows the informal contract specifications for inserting an element into the *dictionary*, a table of bounded capacity where each element is identified by a certain character string used as key.

2.3.2 Design by Contract Library for Java

We have introduced *Design By Contract* to Java by *jContractor*, a purely library-based system and a set of naming conventions. The *jContractor* system does not require any

special tools such as modified compilers, runtime systems, modified JVMs, or pre-processors, and works with any pure Java implementation. Therefore, a Java programmer can use *jContractor* library and follow a simple and intuitive set of conventions to practice *Design By Contract* .

Design By Contract Example: Dictionary		
	Obligations	***Benefits***
Client	(Must ensure precondition) Make sure table is *not full* & key is a *non-empty* string	(May benefit from postcondition) Get updated table where the given element now appears, associated with the given key.
Supplier	(Must ensure postcondition) Record given element in table, associated with given key.	(*May assume precondition)* No need to do anything if table given is *full*, or key is *empty* string.

Table 2.3: Contract Specification for Inserting Element to Dictionary

Each class and interface in a Java program corresponds to a translation unit with a machine and platform independent representation as specified by the Java Virtual Machine (JVM) `class` file format [45]. Each class file contains JVM instructions (bytecodes) and a rich set of meta-level information. *jContractor* utilizes the meta-level information encoded in the standard Java class files to instrument the bytecodes on-the-fly during class loading. During the instrumentation *jContractor* parses each Java class file and discovers the *jContractor* contract information by analyzing the class meta-data.

2.3.3 Design Issues and jContractor Approach

The *jContractor* design resolves three key design issues when adding contracts to Java:

- how to express preconditions, postconditions and class invariants and incorporate

them into a standard Java class definition;

- how to refer to (old) entry values of members or arguments, and to check method results when writing postconditions using standard Java syntax; and
- how to check and enforce contracts at runtime.

An overview of *jContractor*'s approach to solving these problems is given below:

- Programmers add contract code to a class in the form of methods following *jContractor*'s naming conventions: *contract patterns*. The *jContractor* class loader recognizes these patterns and rewrites the code to reflect the presence of contracts.
- Contract patterns can be inserted either directly into the class or they can be written separately as a *contract class* where the contract class' name is derived from the target class using *jContractor* naming conventions. The separate contract class approach can also be used to specify contracts for interfaces.
- *jContractor* finds the *contract patterns* during class loading or object instantiation by utilizing the meta-level information found in Java class files and by taking advantage of dynamic class loading in order to perform "reflective", on-the-fly bytecode modification.
- Programmers enable the run-time enforcement of contracts either by engaging the *jContractor* class loader or by explicitly instantiating objects from the *jContractor* object factory. Programmers can use exactly the same syntax for invoking methods and passing object references regardless of whether contracts are present or not.
- *jContractor* uses an intuitive naming convention for adding *preconditions*, *postconditions, class invariants, recovery* and *exception handling* in the form of `protected` methods. Contract code is hence distinguished from the functional code. The name and signature of each contract method determines the actual method with which the contract is associated.
- Postconditions and exception handlers can access the *old* value of any attribute by using a special object reference, *`OLD`*. For example *`OLD.count`* returns the value of the attribute *`count`* just prior to the execution of the method. *jContractor* emulates this behavior by transparently rewriting class methods during class loading so that the entry values of *`OLD`* references are saved and then made available to the postcondition and exception handling code.
- *jContractor* provides a class, *`RESULT`*, with a `static boolean` method, *`Compare`*. In a postcondition it is possible to check the *result* associated with the method's execution by calling *`RESULT.Compare(<expression>)`*. A true or false is returned based oncomparing the value of *`<expression>`* with the result.

2.3.4 *jContractor* Library and Contract Patterns

Table 2.4, "Overview of jContractor Design By Contract Abstractions," on page 34 contains a summary of key Design By Contract abstractions and the corresponding *jContractor* patterns. One of the key contributions of *jContractor* is that it supports all Design By Contract principles using a *pure-Java*, *library-based* approach. Therefore, any Java developer can immediately start using Design By Contract without making any changes to the test, development, and deployment environment after obtaining a copy of *jContractor* classes.

DESIGN BY CONTRACT ABSTRACTIONS	
Pre-Conditions	
Pattern	*Description*
`methodName_PreCondition`	Method evaluating a boolean result based on the same arguments & object state at the time of *methodName* invocation. Instrumented code for *methodName* with matching signature executes the pre-condition method before executing body. Reports error and aborts method if pre-condition method returns false.
Post-Conditions	
Pattern	*Description*
`methodName_PostCondition`	Method evaluating a boolean result based on the values of arguments & object state at the time when *methodName* returns without an exception condition. Special OLD & RESULT state can be accessed in evaluating post-condition. Instrumented code for *methodName* with matching signature executes the pre-condition after method body is executed. Reports error and aborts method if post-condition method returns false.

Table 2.4: Overview of jContractor Design By Contract Abstractions

DESIGN BY CONTRACT ABSTRACTIONS	
Class Invariants	
Pattern	*Description*
`className_ClassInvariant`	Method evaluating a boolean result based on the object state at the time of invocation of each public method and immediately upon successful return from the method. Instrumented code for each public method executes the class invariant method before and after executing the method body. Reports error and aborts method if class invariant method returns false.
Exception Handling	
Pattern	*Description*
`methodName_OnException`	Method that gets called when *`methodName's`* execution ends *abnormally*, throwing an Exception. The exception handler provides an opportunity for doing recovery by restoring invariants, resetting state.
Accessing OLD state & RESULT	
Pattern	*Description*
`OLD.attr &` `OLD.m(...)`	Expression evaluates to *value* of *`attr`* or result of method m() on method entry. *`OLD`* references can only be used inside postcondition and exception handler methods. Instrumented code for the target method records OLD values on method entry.
`RESULT.compare(<expr>)`	Expression evaluates to *true* if the result evaluated by the method body is equal to the expression. RESULT references can only be used inside postcondition methods. Instrumented code for the target method records the value of the result just before method body returns.

Table 2.4: Overview of jContractor Design By Contract Abstractions

DESIGN BY CONTRACT ABSTRACTIONS	
Reflection Based Runtime Instrumentation	
`jContractorClassLoader`	Instrumentation of classes is performed on compiled class byte codes on-the-fly by the jContractor class loader. Instrumentation logic is based on using reflection to search contract patters in the meta-level data found in standard Java format class bytecodes.
`jContractorFactory`	Instrumentation of designated classes can be performed on compiled class byte codes on-the-fly by explicitly instantiating objects from jContractor Factory. Instrumentation logic is the same one used by the intrumenting class loader.

Table 2.4: Overview of jContractor Design By Contract Abstractions

2.3.5 Adding Contracts to Java Programs

A programmer writes a contract by taking a class or method name, say *put*, then appending a suffix depending on the type of constraint, say *_PreCondition*, to write the *put_PreCondition.* Then the programmer writes the method body describing the precondition. The method can access both the arguments of the *put* method with the identical signature, and the attributes of the class. When *jContractor* instrumentation is engaged at runtime, the precondition gets checked each time the *put* method is called, and the call throws an exception if the precondition fails.

The code fragment in Figure 5.1 on page 88 shows a *jContractor* based implementation of the `put` method for the *Dictionary* class. An alternative approach is to provide a separate *contract class*, *Dictionary_CONTRACT*, as shown in Figure 5.2, which contains the contract code using the same naming conventions. The contract class can (optionally) extend the target class for which the contracts are being written, which is the case in our example. For every class or interface *X* that the *jContractor ClassLoader* loads, it also looks for a separate contract class, *X_CONTRACT*, and uses contract specifications from both *X* and *X_CONTRACT* (if present) when performing its instrumentation. The details of the class loading and instrumentation will be presented in subsequent sections.

2.3.6 Interaction Of Contracts With Inheritance And Polymorphism

Contracts are essentially specifications checked at run-time. They are not part of the functional implementation code, and a "correct" program's execution should not depend on the presence or enabling of the contract methods. Additionally, the exceptions that may be thrown due to runtime contract violations are not checked exceptions, therefore, they are not required to be part of a method's signature and do not require clients' code to handle these specification as exceptions. In the rest of this section we discuss the contravariance and covariance issues arising from the way contracts are inherited.

The inheritance of preconditions from a parent class follows *contravariance*: as a subclass provides a more specialized implementation, it should weaken, not strengthen, the preconditions of its methods. Any method that is redefined in the subclass should be able to at least handle the cases that were being handled by the parent, and in addition handle some other cases due to its specialization. Otherwise, polymorphic substitution would no longer be possible. A client of *X* is bound by the contractual obligations of meeting the precondition specifications of *X*. If during runtime an object of a more specialized instance, say of class *Y* (a subclass of *X*) is passed, the client's code should not be expected to satisfy any stricter preconditions than it already satisfies for *X*, irrespective of the runtime type of the object.

jContractor supports contravariance by evaluating the a *logical-OR* of the precondition expression specified in the subclass with the preconditions inherited from its parents. For example, consider the following client code snippet:

```
// assume that class Y extends  class X
X x;
Y y = new Y();    // Y object instantiated
x = y;            // x is polymorphically attached to a Y object
int i = 5; …
x.foo(i);         // only PreCondition(X,[foo,int i])  should be met
```

When executing *`x.foo(),`* due to dynamic binding in Java, the *foo()* method that is found in class Y gets called, since the dynamic type of the instance is Y. If *jContractor* is enabled this results in the evaluation of the following precondition expression:

PreCondition(X,[foo,int i]) $\vee$ *PreCondition(Y,[foo,int i])*

This ensures that no matter how strict *PreCondition(Y,foo)* might be, as long as the *PreCondition(X,foo)* holds true, `x.foo()` will not raise a precondition exception.

While we are satisfied with this behavior from a theoretical standpoint, in practice a programmer could violate contravariance. For example, consider the following precondition specifications for the `foo()` method defined both in X and Y, still using the example code snippet above:

PreCondition(X, [foo,int a]) : $a > 0$ *(I)*

PreCondition(Y, [foo,int a]) : $a > 10$ *(II)*

From a specification point of view *(II)* is stricter than *(I)*, since for values of a*:* $0<a<=10$*, (II)* will fail, while *(I)* will succeed, and for all other values of a, *(I)* and *(II)* will return identical results. Following these specifications, the call of previous example:

```
x.foo(i);   // where i is 5
```

does not raise an exception since it meets *PreCondition(X,foo,int a)*. However, there is a problem from an implementation view, that Y's method `foo(int a)` effectively gets called even though its own precondition specification, *(II)*, is violated. The problem here is one of a design error in the contract specification. Theoretically, this error can be diagnosed from the specification code using formal verification and by validating whether following *logical-implication* holds for each redefined method `m()`:

$$PreCondition(ParentClass,m) \Rightarrow PreCondition(SubClass,m)$$

For the previous example, it is easy to prove that *(I)* does not logically-imply *(II)*. It is beyond the scope of *jContractor* to do formal verification for logical inference of specification anomalies. *jContractor* does, however, diagnose and report these types of design anomalies, where any one of the *logical-OR*'ed precondition expressions evaluates to *false*. In the above example, *jContractor* would throw an exception to report that the precondition has been illegally strengthened in the subclass, thus forcing the programmer to correct the precondition.

A similar specification anomaly could also occur when a subclass strengthens the parent class's invariants, since *jContractor* checks the class invariants when preconditions are evaluated. The subclass' invariant's runtime violation is caught by *jContractor* instrumented code as an exception, with the correct diagnostic explanation.

The inheritance of postconditions is similar: as a subclass provides a more specialized implementation, it should strengthen, not weaken the postconditions of its interface

methods. Any method that is redefined in the subclass should be able to guarantee at least as much as its parent's implementation, and then perhaps some more, due to its specialization. *jContractor* evaluates the *logical-AND* of the postcondition expression found in the subclass with the ones inherited from its parents. Similar anomalies as discussed above for preconditions can also appear in postcondition specifications due to programming errors. *jContractor* will detect these anomalies should they manifest during runtime execution of their respective methods.

2.3.7 Factory Style Instrumentation Issues

When factory style instrumentation is used, *jContractor* constructs a contractor subclass as a direct descendant of the original base class. Therefore, it is possible to pass objects instantiated using the instrumented subclass to any client expecting an instance of the base class. Other than enforcing the contract specifics, an instrumented subclass, say `Foo_Contractor,` has the same interface as the base class, `Foo`, and type-wise conforms to `Foo`. This design allows the contractor subclasses to be used with any polymorphic substitution involving the base class. Consider the class hierarchy shown in Figure 2.4:

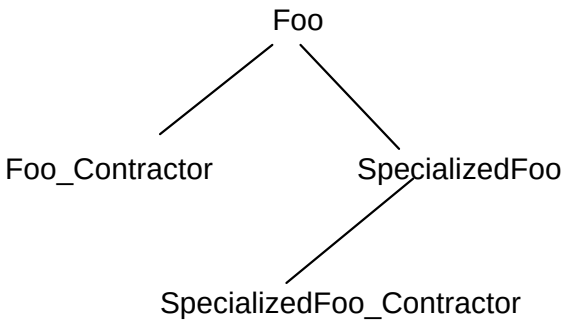


Figure 2.4: Example jContractor Factory Class Hierarchy

jContractor allows for the polymorphic substitution of either `SpecialFoo` objects or the instrumented `SpecialFoo_Contractor` objects with `Foo` objects.the instrumented `SpecialFoo_Contractor` objects with `Foo` objects.

2.4 Summary

In this chapter we introduced three bodies of original research work we have conducted to introduce new language extensions for concurrency, distributed computing and design by contract for the object oriented languages, Eiffel and Java. We discussed design considerations and techniques to build extensible, open, object-oriented libraries that use and take advantage of the existing low-level system and vendor libraries, platforms, frameworks to support these new language extensions. These high-level programming abstractions are provided to a programmer by following our naming and programming conventions. The design and implementation details of the *Class CONCURRENCY, Active-RMI* and *jContractor* are individually presented respectively in Chapters 3,4 and 5.

Chapter 3

Introducing Concurrency to a Sequential Object-Oriented Language

3.1 Introduction

The work described in this chapter introduces concurrency to the object-oriented language Eiffel by providing a set of *class libraries* and an associated design methodology. The concurrency mechanism we provide is well-suited for client/server style distributed applications. Since no changes are made to the Eiffel Language [53], or its runtime system, the essential principles of sequential object-oriented programming offered by Eiffel are not sacrificed. We present our concurrency abstractions as encapsulated behavior of Eiffel objects, that can be inherited from the *Class CONCURRENCY.*

The main concurrency abstractions provided by our mechanism are objects as processes — *active objects* — and asynchronous *remote method invocation* with *data-driven synchronization*. The *Class CONCURRENCY* encapsulates the high-level concurrency abstractions and provides them to objects through inheritance. In addition to the class libraries we also developed a *design method* which promotes: (1) incremental development, (2) stepwise refinement of active objects from ordinary sequential Eiffel objects, (3) utilization of existing Eiffel software Libraries. This design method views objects as the unit of design, and facilitates key object-oriented principles such as reusability, data encapsulation, and extensibility.

We have discussed the fundamental concurrency abstractions in Chapter 2, therefore we will summarize the key points in the next section, and in the subsequent sections present the design and implementation details of the *Class CONCURRENCY.* In Sec-

tion 3.5.1, “Generation of Concurrent Class Prototypes from a Sequential Class” on page 51 we propose a methodology for writing concurrent applications. The *methodology* describes how a concurrent application can be designed from sequential object specifications, and how this process can be automated. In Section 3.6.1 we give an example of a bounded buffer implementation and a discussion of various aspects of our concurrency model.

3.2 Concurrency Model Overview

Our concurrency model unifies the notion of a *process* and the notion of an *object*, arriving at the concept of an *active object*. Objects become active via inheritance from the *Class CONCURRENCY*. Concurrency is viewed as the parallel execution resulting from the creation of active objects and their interactions with each other.

In order for an object to become active, it must *inherit* from the *Class CONCURRENCY*. The `Create` function is called first to create a sequential *proxy* object. The actual creation of an active object with its own process is achieved by invoking this *proxy* object’s `split` method, which is an inherited behavior from the *Class CONCURRENCY*. The `split` method starts a new process with an independent thread of control and returns back to the client after the *active object* is created and communication ports are established and initialized. Then the active object begins executing a special start-up method, *`scheduler`*. The design method requires that a method called *`scheduler`* is defined for each class that inherits from *CONCURRENCY*. The *`scheduler`* method specifies how to serve the requests generated by *clients* using the `remoteInvoke` method. All requests are delivered as actual messages to the server using a transparent inter-process communication (IPC) mechanism. Since multiple clients can simultaneously request services, the communication is *buffered*.

After a `split`, the server object in the client’s process act as a *proxy* to the actual active server object. This proxy object transparently and asynchronously relays the requests to the active server’s *request queue*, and returns the results of requests which are asynchronously delivered to its *result queue* by the server.

Once a server object has been made *active*, via a `split` operation, new clients that want to invoke the server’s methods must create their own proxy copies, and use the `attach` method to set up the association between the proxy and the active server object. `Attach` is significantly cheaper than `split`, since it doesn’t involve creation of a new address space, `Attach` is also the only mechanism to share active objects. Algorithms for `split` and `attach` are given in Section 3.3.1.

In the sequential object oriented paradigm, *method invocation* is a *synchronous* procedure call, and objects are *passive* entities, doing work only when their methods are invoked. The *Class CONCURRENCY* implements a *non-blocking, asynchronous* method invocation mechanism called `remoteInvoke`. Consider the following code where `obj` is an *active* object,

```
call_id:INTEGER;
return_value: T;
call_id := obj.remoteInvoke("method",arg_list );
.
.  }concurrent execution with obj
.
return_value ?= obj.claimResult(call_id);
--Usecall_id to obtain the associated result.
```

The client does not wait for the completion of `obj's` execution of *method*, but concurrently continues executing its own code until the point in its own execution where it actually uses the result of the execution of *method*. The part of the client's code shown with dots above is executed in *parallel* with the `obj's` code. The result returned by `claimResult` conforms to all types, so it is assigned to the *return_value*, using the *?=* reverse-assignment attempt operator. The type, *T*, of *return_value* is the actual type returned by `obj's` *method* feature. This is a data-driven synchronization scheme, based on asynchronous message passing.

The *Class CONCURRENCY* provides two methods to access the result of a remote method invocation. Both of the methods take a single argument, a *call_id*, which is returned by the corresponding remote invocation as a handle to obtain the actual result in the future. First method, `claimResult`, as used in the example of the preceding paragraph, returns the result delivered by the server. If result is not available yet, then `claimResult` *blocks* until it becomes available. The other method, `resultReady`, is a *non-blocking* test for the availability of result of the remote invocation associated with the *call_id*. All the underlying communication is encapsulated and hidden from the application.

Since active objects reside in separate address spaces, objects that appear as parameters of remote methods of active objects can not be passed by reference but must be (deep) copied.

We have introduced a new design method which provides static-type checking, and variable-size argument passing capabilities for remote invocations which results in a simpler, safer, and more transparent utilization of the `remoteInvoke` method. Using *FUTURE* type results also provides simpler and more transparent accessing of the result of the remote invocation. Explicit invocation of `claimResult`, and `resultAvailable` is also eliminated by the utilization of the *FUTURE* type. Following code fragment is an example that shows how the design method improves the shortcomings of the previous example which used the library abstractions for concurrency directly.

```
future: FUTURE;
return_value  : T;


future := act_obj.method(arg1, arg2 ... );
.
.  }  concurrent execution with obj
.
return_value ?= future.data;
--implicit claimResult from act_obj
```

Message acceptance in our model is *asynchronous* and *explicit*. Accepting a message is separated from actually serving the request. Messages delivered to the communication buffer but not yet accepted into the *`request_queue`* are called *pending* requests. A queue of request messages, *`request_queue`*, contains all the accepted requests of the client objects. Each entry in the queue contains all of the parameters of the client's remote invocation request, including the *call_id* and reply address. The server object explicitly needs to show intent to accept messages: `getRequest` is a potentially *blocking* call that commits all pending requests into the *`request_queue`*, and blocks if no pending request exists until the first one arrives. A non-blocking version of this method is the `pendingRequestExists` method.

The *`scheduler`* has unrestricted access to the *`request_queue`*, and it can inspect the parameters and names of the requests in the queue in order to select and serve one of them. It can also choose to wait for a certain type of request or a certain condition to arrive before it selects a request for service. Sending a result back to a client after the service of a request is also done *asynchronously* and *explicitly*, using the `sendResult` primitive.

Since there is a single thread inside the *scheduler*, the scheduler serializes the execution of its methods. This also applies to the passive objects within each process, since no sharing of passive objects is possible due to the deep copy semantics of arguments of `remote_invocation`. Therefore, all requests to the methods of a passive object are also serialized. This eliminates the need for synchronization inside the methods of objects and allows us to have all synchronization points correspond to communication events.

Objects that are inherently sequential may be designed the same way they would be designed in sequential Eiffel. This enhances *reusability* in two ways: objects already designed for sequential applications can be readily *reused* in concurrent applications; non-concurrent applications can directly use such objects designed in concurrent applications.

Reuse of active objects is supported in a different way. Since most of the synchronization and request-service policy is implemented inside the *scheduler* method, a concurrent class can be extended via inheritance and redefinition of the *scheduler* method.

3.3 Design of the Class CONCURRENCY

The *Class CONCURRENCY* encapsulates a state as well as a behavior that collectively describes the notion of *active* objects. The state contains the data structures involved in the communication, scheduling and synchronization events that take place during the life-time of each individual object. The behavior provides the interface to create active objects, and request asynchronous execution of their features and communicate with them.

The *Class CONCURRENCY* is part of the *Parallel Library* which also includes other classes used for the implementation of low-level IPC and UNIX interface.

A short description of the *Class CONCURRENCY* is given in Appendix A; in the subsequent sections we provide more detailed explanations of the contents of this class. In Section 3.6 we give an example application using the concurrency mechanism described here.

3.3.1 Design and Implementation Details

Pseudo code describing the `split` method is given in Figure 3.1. A `split` call implicitly starts a handshake protocol between the newly created active server process and the calling client, the *proxy* object. The *proxy*'s port is created first, and its parameters are passed to the server as part of its initialization. Then the server creates its own communication port, and acknowledges the *proxy* by providing info about its own communication parameters. At this point the handshake is accomplished and the *proxy* and *server* are ready to start sending request and reply messages to each other. Each proxy object has its own communication port. All proxies send their request messages to the same port at the server process.

```
split()
   begin
    initialize_caller_side_IPC_parameters();

    -- create and initializes a dedicated communication port
    -- pass caller side IPC info to the new process created
    -- by fork() or other system call in Unix.
    spawn_new_process();

    in parent process --  i.e. proxy side
         create_Result_queue();
         wait_acknowledge(); --  from the child
         -- "ack" contains info about childs' IPC parameters
         record_server_info(); --  using "ack" message
         return(); --  from proxy to caller
    in child process --  i.e. newly created active object
         initialize_server_side_IPC_parameters();
         acknowledge_parent();
         --  sends "ack" to caller, piggybacking the new IPC info;
         obj := create_server_object(); --  first and only object
                                            in this process.
         obj.scheduler(); --  invoke obj's scheduler method.
         -- transfers control to scheduler of the newly created active
         -- object scheduler describes obj's concurrent activity.
   end --  split
```

Figure 3.1: Pseudo code: *split* method

Figure 3.2 gives the pseudo code for the `attach` method. `Attach` is similar to `split`, except that it sets up an association with an already existing active server object.

```
attach(server_info)
   begin
         --  create and initialize a dedicated communication port
         --  Note: no new process is spawned.
         initialize_caller_side_IPC_parameters();

         create_result_queue();
         record_server_info(); --  using the parameter passed.

         return(); --  without any handshake
   end --  attach
```

Figure 3.2: Pseudo code: *attach* method

Figure 3.3 describes the implementation of the `remoteInvoke` method. The arguments of the `remoteInvoke` method are the name of the method to be invoked and the parameters supplied to that method. The `remoteInvoke` method can only be issued to a proxy object. This proxy will create a request package and asynchronously deliver it to the active server. The `remoteInvoke` call returns a unique *call_id*, which can be used to access the result of the invoked method. In order to access the result of a previously invoked method, another abstraction is provided via the *`claimResult`* method which takes the call_id number as the argument and returns the result if it had been delivered by the server. If result is not available yet, then *`claimResult`* blocks until it is ready. The type of the result object returned by `claimResult` is *ANY*, which statically conforms to all types. This enables `claimResult` to be a general result delivery method. The result is assigned to an object of the correct result type expected by the client by using Eiffel's *reverse-assignment attemp* operator, *?=*.

```
remoteInvoke(method, method_parameters)
    begin
         Request_package := create_request_package();
         claim_no := unique_claim_no();
         set_method_fields(Request_package, method,
                           method_parameters);
         set_IPC_fields(Request_package, claim_no, IPC_info);
         -- IPC_info about both server and client packed in server
         -- will use this stored IPC info to send the result back

         -- asynchronous send to the server using info on package
         send_package( Request_package );

         return ( claim_no ); --  to the client code
         --  without waiting for the result from the server
    end --  remoteInvoke
```

Figure 3.3: Pseudo code: *remoteInvoke* method

The method, `resultReady` (which also takes a *`claim_number`* argument) is non-blocking and can be used to test if result of the associated request is available.

Since the Eiffel language at the time the concurrency mechanism was developed did not support the notion of meta-classes, we supplied the *method-name* to `remoteInvoke` as a string. Also, since Eiffel does not support variable-number of arguments for methods, `remoteInvoke` expects a single variable sized generic array that contains the actual arguments of the call. The reverse-assignment operator, *?=*, is used to extract the actual arguments from the arguments array. Passing the name of the method as a string, and packaging/unpacking the arguments manually is error prone, and not checkable by the compiler. To alleviate these problems and to provide type-safety, the methodology described in next section provides an alternative way of doing the *remote method call* without explicitly using the `remoteInvoke` method.

Each *active* object maintains a queue of requests, called *Request Queue*. Each entry in the queue contains all of the parameters of the clients invocation, including the `claim_number` and communication parameters to send back the result of the computation. This server object explicitly needs to show intent to accept messages. `getRequest` is a blocking call that commits all pending requests into the *Request Queue*, and blocks if no pending request exists until the first one arrives. A non-blocking version of this method is the `pendingRequestExists` method. The *Request Queue* is avail-

able as a local object, and is implemented as a queue that supports all the general queue methods of Eiffel Data Structure Library.

The server object chooses one of the pending requests from the queue, and then applies the *method* to itself, using the parameters that are also packed into the request, and then returns the results back to the client upon completion of service also in an asynchronous way, using the `sendResult` method. The `claim_number` is also returned along with the result to the object who invoked the method, so that this (client) object can claim the result, using the `claimResult` method. This is a blocking method which forces synchronization with the server if the result associated with `claim_number` has not been returned yet. All the results of remote-invocations get queued into *result_queue* in an asynchronous manner and stay there until they are *claim*ed. On the other hand, the `resultReady` method does a simple non-blocking search of *result_queue*, to test whether a particular result is delivered yet.

3.4 Implementation Status

The described system is implemented using version 2.2 Eiffel by ISE [53], running on Sun's UNIX based Sun OS 3.0. It is written largely in Eiffel, with the low level operating system and inter-process-communication (IPC) routines written in C language as external Eiffel routines. The asynchronous communication is implemented using Internet domain sockets, and (UDP) datagram messages.

3.5 A Method for Designing Active Objects

The first step of the methodology is to identify the active objects. Our approach is to start the design using a sequential prototype of the active object. To illustrate the idea, consider the following sequential Eiffel class, *Class A*, shown in Figure 3.4. The ellipsis in the code denote implementation details that are omitted for a clearer presentation.

We propose a way to extend the sequential class, *Class A* to a concurrent one. The concurrent version, *Class Conc_A* is given in Figure 3.5. *Class Conc_A* exports the same methods, *foo* and *bar*. But both *foo* and *bar* are *redefined* as remote methods in *Conc_A*.

```
class A
   export
        foo, bar
   inherit X,Y, ...

   feature

. . . --  implementation details

   foo(arg1: T1 ): T_foo is
        do
             . . .
        end
end; --  Class A
```

Figure 3.4: Sequential Class: A

```
class Conc_A
   export foo, bar
   inherit  CONCURRENCY
          redefine scheduler;
      A
          rename
      foo as A_foo,
   feature
      args : ARRAY[ANY];
      scheduler is
          local foo_arg1 : T1;
          do
              from
                  getRequest;
              until false
              loop -- forever
                  current_request := request_queue.remove;
                  if current_request.feat_name.equal("foo")
                  then
                      foo_arg1?= current_request.
                                          parameters.item(1);
                      send_result(A_foo(foo_arg1));
                  else
                      -- error handling needed
                  end
                  if  request_queue.empty or
```

```
                    pendingRequestExists
                then
                    getRequest;
                end; - if
            end; --  loop
    end; --  scheduler

    foo(arg1: T1 ): FUTURE is
        require  is_proxy
        do
            args.put( arg1, 1);
            Result.Create( Current,
                        remoteInvoke("foo", args);
        end;
end; -- class
```

Figure 3.5: Concurrent Version of Class A.

3.5.1 Generation of Concurrent Class Prototypes from a Sequential Class

A concurrent class can be generated from a sequential one in an automated way. The example shown above extends the sequential *Class A* to a concurrent class, *Conc_A*. The `scheduler` generated for *Conc_A* uses a FIFO policy. This scheduler can then be modified to express more complex synchronization constraints.

The generation of a concurrent class, such as *Conc_A* as given above, is very much like a stub generation process for an RPC mechanism. The key steps in extending *Class A* to *Class Conc_A* are:

1. Export the same features in *Conc_A* as in *A* by copying A's export clause;
2. Inherit from the classes *CONCURRENCY* and *A*.
3. Redefine *scheduler* inherited from *CONCURRENCY.*
4. Rename the exported features so that their sequential implementation in *Class A* can be utilized inside *Class Conc_A*. The new name is derived from the old name in *Class A* by prefixing it with the class name, *A*. For example, the name *A_foo* in the *Class Conc_A* can be used to execute the method *foo* of *Class A*. Since *A_foo* is not exported, the sequential implementation of *foo* found in *Class A* is hidden from clients of *Class Conc_A*.
5. Construct the exported features of *Class A.* Using the same names for the exported features in *Conc_A* results in the sequential implementation of these features to be overridden by the new ones defined in *Conc_A*. For example, *foo*

method in *Conc_A* can be automatically generated by (i) copying the definition in *Class A*, (ii) changing its return type, say *T*, to *FUTURE*, and then (iii) rewriting the body part. The new body consists of two parts: (1) constructing of an arguments list, by copying the formal arguments of *foo* into an array, and then (2) creating and returning a *FUTURE* type result object. The result, which is a *FUTURE* type object, is initialized by providing the following two arguments to its *Create* method. First is the value of *Current*., which is the proxy object, and second is the claim number returned by the *remoteInvoke ("foo", arguments-array)* call. A 'require *is_proxy*' pre-condition is added to the method, to ensure that invocations of *foo* will be handled only by proxy objects that have executed a `split` or `attach`.

Redefinition solves a problem which arises due to *polymorphism*. An active object of class *Conc_A*, say *conc_obj*, can be polymorphically assigned to an object reference of *Class A*, say *a_obj*. Without the redefinition of *foo*, the call *a_obj.foo()* would result in the invocation of *foo* defined in *Class A*. This violates the remote invocation protocol of the dynamic and active object *conc_obj*. The redefinition ensures correct behavior by invoking the *foo* method of Conc_A.

6. Construct the *`scheduler`* method. A FIFO scheduler can be constructed creating a loop. At each iteration of the loop: (1) pending requests are committed to the *request_queue*, (2) one request is selected from the queue, (3) the requested operation is performed and finally (4) the results are sent back asynchronously, using `sendResult`.
7. Generate the necessary data structures used by *remoteInvoke* (the *args* array), and other temporaries needed when a request is selected by the scheduler for servicing. Depending on which method is being served, temporary objects are needed to supply the correct-typed arguments to the method. For example, the local object *foo_arg1: T1* can be used to during the servicing of a *foo* request. Before the actual call, the temporaries must be initialized, using the reverse-assignment attempt (*?=* operator) with the argument array part of the request package.

Note that the steps described above propose a way of *automating* the process of extending a sequential class to a concurrent class prototype. This methodology helps overcome two problems related with type-safety, and static type-checking, both of which arise due to directly calling the `remoteInvoke` method by clients. These two problems are: (1) having to pass the *method* argument as a string; (2) handling of variable number of arguments. The methodology provides a way to indirectly call `remoteInvoke` using the same (type-safe) syntax of a sequential call.

Using *FUTURE* objects as results of remote method invocations eliminates the need to keep track of the *call_id* and *server_object proxy* associated with the remote invoca-

tion. Also eliminated is the need to explicitly call `claimResult` to obtain the result. Using *FUTURE* type objects makes it simpler and more transparent to use the `claimResult` and `resultAvailable` methods of the *Class CONCURRENCY.*

3.5.2 Reusability Issues

An important aspect of our design method is its support for software composition and reuse. This support is due to the fact that our concurrency mechanism is compatible with the key object oriented principles for providing *reusability.* These principles are: data encapsulation, data abstraction, polymorphism, inheritance and genericity [53].

Data encapsulation property of objects in sequential Eiffel is respected by our methodology. Active objects have a protected internal state which can only be accessed or modified by invoking a method specified in its interface. Proxy objects do not have any direct means to modify the state of the active objects. They are only responsible from delegating requests and collecting replies. Therefore proxy objects do not violate the data encapsulation property of the active objects. The active object has complete autonomy over managing its internal state.

Data abstraction is the mechanism which frees the client of a class from having to know its internal representation. In our methodology the client of an active server does not need to know how its request will be delivered and serviced. The client simply contracts a job (by remote method invocation) to the server and then check on the contractor's status only when the need arises. The only point the client needs to synchronize with the server is when the result is needed, and this does not require any extra concurrency/synchronization related information about the server's implementation to be known by the client.

Inheritance is the key technique used in the methodology to create concurrent objects. The way inheritance is used in our methodology as a software development technique allowing extension, specialization and stepwise refinement is essentially the same as in sequential Eiffel programming.

Objects that are inherently sequential may be designed the same way they would be designed in sequential Eiffel. This enhances *reusability* in two ways: objects already designed for sequential applications can be readily *reused* in concurrent applications; non-concurrent applications can directly use such objects designed in concurrent applications.

Reuse of active objects is supported in a different way. Since most of the synchroniza-

tion and request-service policy is implemented inside the *scheduler* method, a concurrent class can be extended through inheritance and redefinition of the *scheduler*.

A discussion of how our methodology supports *polymorphism* is given in the section titled: “A Method for Designing Active Objects” on page 49 in step 3.

3.6 An example: Bounded Buffer

Figure 3.6 presents the *BUFFER* Class, which defines a sequential buffer class with a bounded storage. We design *ACTIVE_BUFFER* Class by applying the methodology presented in the previous section. We modify the FIFO scheduler generated by the methodology in order to express different synchronization needs of the active bounded buffer example. The Class *ACTIVE_BUFFER* is presented in Figure 3.7.

Figure 3.6: Sequential Buffer Class

```
class BUFFER [T] -- A Sequential Bounded Buffer Class
export
        put {PRODUCER}, get {CONSUMER}
feature
    buffer : FIXED_QUEUE [T] ; -- actual storage
    put ( Item: T ): BOOLEAN is --
        do
            if not buffer.full then
                  buffer.put (Item); -- append to buffer
                  Result := true;
            else -- return code for failure
                  Result := false;
            end; -- if
        end; -- put
    get : T is --
        do
            if not buffer.empty then
                 -- remove the oldest item in buffer
                 buffer.remove;

                 -- return the removed item
                 Result := buffer.item;

            else -- return a Void object for failure
                  Result.Forget; -- Result made Void
            end; -- if
        end; -- get end -- class BUFFER
```

Figure 3.7: ACTIVE BUFFER Class

```
class ACTIVE_BUFFER -- A Concurrent extension to BUFFER
export put {PRODUCER}, get {CONSUMER}
inherit
   CONCURRENCY
         redefine scheduler;
   BUFFER
         rename
               put as BUFFER_put,
               get as BUFFER_get
feature
    args : ARRAY[ANY]; -- argument list for remoteInvoke
    scheduler is -- execution begins here
          local
               put_Item: DATA;
          do
               from
                    -- buffer allocated only by the active object
                     buffer.Create;

                     -- initially block until request(s)arrives
                     getRequest;
               until false
               loop -- i.e. loop forever
                    -- pick a request
                    current_request := request_queue.remove;
                    if current_request.feat_name.equal("put")
                          and not buffer.full then
                          put_Item?=current_request.
                                           parameters.item(1);

                          -- ack producer asynchronously
                          sendResult(BUFFER_put(put_Item));

                     elsif current_request.feat_name.equal("get")
                                and not buffer.empty then
                          sendResult( BUFFER_get ); -- to consumer
                     else -- cannot serve the current_request
                          request_queue.put_left(current_request);
                          -- skip over
                     end;
                     if request_queue.empty
                          or pendingRequestExists then
                          getRequest ;  -- blocks only if empty
                     elsif request_queue.offright then
                          -- go back to oldest in queue
                          request_queue.start;
                     end;
```

```
            end; -- loop
        end; -- scheduler

    put ( Item: DATA ): FUTURE is -- executed only by the proxy
        require is_proxy
        do
            args.put( Item, 1); -- initialize arguments array
            Result.Create(Current, remoteInvoke( "put", args) );
        end; -- put

    get : FUTURE is -- executed only by the proxy
        require is_proxy
        do
            Result.Create(Current, remoteInvoke( "get", args) );
        end; -- get
end -- class ACTIVE_BUFFER
```

In order to prevent the producers from producing unboundedly, put returns an acknowledgment back to the producer in an asynchronous manner. The implementation of *PRODUCER*, Figure 3.8, allows a producer to have at most one outstanding put request. The data-driven synchronization of the producer allows it to produce the next item without having to wait for an immediate acknowledgment.

Figure 3.8: Producer Class of Bounded Buffer Example

```
class PRODUCER
inherit
    CONCURRENCY
          redefine scheduler;
feature
    b_buffer : ACTIVE_BUFFER ; -- shared buffer object
    Data_item : DATA; -- unit of production
    buffer_params : SOCKET; -- IPC info to attach buffer
    synch : FUTURE;

    scheduler is -- execution begins here
          do
               from
                   b_buffer. Create ; -- create proxy
                   getRequest ; -- wait until buffer info arrives
                   buffer_params?=request_queue.first.parameters(1);
                   b_buffer.attach(buffer_params);

                     〈  Produce First Data Item  〉
               until false
               loop -- forever
                   -- asynchronous remote method invocation.
                   synch:= b_buffer.put( Data_item );

                   -- at this point producer continues to execute

                   〈  Produce Next Data Item  〉

                   -- wait result here
                   synch.data;

                   〈  More processing 〉
               end; -- loop
          end; -- scheduler
end -- class PRODUCER
```

Figure 3.9: Consumer Class of Bounded Buffer Example

```
class CONSUMER
inherit
    CONCURRENCY  redefine scheduler;
feature
    b_buffer : ACTIVE_BUFFER;
    data_item : DATA;
    buffer_params : SOCKET;
    box: FUTURE; -- FUTURE DATA result returned by remote invocation
    scheduler is -- execution begins here
         do
              from
                 b_buffer.Create ; -- create proxy
                 getRequest ; -- wait until buffer info arrives
                 buffer_params?=request_queue.first.parameters(1);

                 b_buffer.attach(buffer_params);
              until false
              loop -- forever
                   -- generate a remote request for “get”
                   box := b_buffer.get;

                   -- wait here until data becomes available
                   data_item ?= box.data;
                   〈  Consume Data Item  〉
              end; -- loop
         end; -- scheduler
end -- class CONSUMER
```

3.6.1 Notes on the Bounded Buffer Example

An interesting feature of the given implementation is that the *ACTIVE_BUFFER* Class has complete autonomy over how it services the requests of its clients, the *scheduler* routine being the only place where these decisions are made. This is important with respect to reusability since one may reuse the *ACTIVE_BUFFER* Class with ease, only redefining the *scheduler* routine, in order to introduce a new protocol or specification. Since all the synchronization code is confined to the scheduler. One can easily extend the given class by inheriting and/or modifying the synchronization code and functional code separately. Another advantage of this approach is that the functional code remains "pure", and can be written without worrying about the synchronization issues which could otherwise create dependencies between relatively independent

parts of code in different modules, and compromise the encapsulation and procedural abstraction properties.

To show how flexible and expressive the approach is, we discuss alternative ways of implementing different synchronization constraints using our buffer example. Let's say we would like to allow consumers to be served even if there are earlier requests from producers. We can simply change the policy executed at each iteration of the scheduler loop to choose the oldest "*get*" type request from the *request queue* if buffer is not empty. This can be done by scanning the request queue, and serving a "*put*" request only when (1) there is no "*get*" request and buffer is not full; or (2) when buffer is empty. Another application might be to respect real-time scheduling constraints of the service requests, where this information is made available as a parameter of the service request. Since we have the ability to "look into" the request messages without selecting them, we can program many complex scheduling algorithms.

3.7 Summary

We have presented a method of writing concurrent applications in Eiffel Language. The method introduces concurrency as an inheritable property of objects specified in the *Class CONCURRENCY,* and provides a methodology using inheritance to write concurrent, distributed applications. Indeed what we have done in this work can be seen as a way of placing Eiffel Language on top of the concurrency that exists at the underlying processor or operating system level, rather than placing concurrency into the Eiffel Language.

In our approach, the active server object can choose the type of requests to respond to and in what order. This is a very powerful mechanism addressing the problem of dealing with *local delays* [46].

An important issue addressed by our methodology is *reusability. Reusability* is enhanced by separating the sequential and concurrent aspects of a class. Also the `scheduler` method can be used to express most of the synchronization code and the concurrent behavior of the object. Reusability and modifiability are enhanced since one can easily extend a given class by inheriting and/or modifying the synchronization code and functional code separately. Reusability is also supported by preserving the *data abstraction, data encapsulation, genericity* and *polymorphism* principles of object-oriented programming. Section 4.2 addresses the reusability issues in greater detail.

Another issue addressed by the methodology is the static type checking of method names and the arguments that are to be *remote_invoked*.

The ability to express powerful synchronization constraints as reusable software components emerges as a strong point of the method. The concurrency mechanism presented here is designed for writing distributed applications. It does not assume a shared-memory model and supports distribution of the active objects over a network.

An issue that requires further investigation is about identifying the synchronization and correctness needs in the presence of multi threaded schedulers in shared-address spaces. Without having explicit control over the scheduling and preemption of multiple threads in a shared-address space many new technical difficulties are introduced. Some of these difficulties are related with mutual exclusion of the execution of an objects's methods, and non-reentrant system calls. Some of these issues have been mentioned in [15]. A concurrency mechanism with multi threaded active objects must satisfactorily address the interference problem with respect to data encapsulation, procedural abstraction and reusability issues, that emerge due to the potential arbitrary interleavings of an object's methods.

Chapter 4

Active-RMI: Asynchronous Remote Method Invocation System for Distributed Computing and Active Objects

4.1 Introduction

In this chapter we discuss the design and implementation of the *Active-RMI (Active Remote Method Invocation)* system. *Active-RMI* is a set of class libraries, tools, and a design method which provides high-level abstractions for building distributed Java applications based on active remote-objects. *Active-RMI* is a natural extension of our earlier work[41] involving introducing concurrency extensions to Eiffel language using class libraries. *Active-RMI* is implemented as an extension to the *Java-RMI* system which was integrated by Sun into the Java Language[31] with the release of JDK 1.1. The key contributions of *Active-RMI* system to distributed computing is the extension of *Java-RMI* with the following features:

1. Asynchronous remote method invocation with future-type results and asynchronous result delivery mechanism from the server to the caller.
2. On-demand remote-object creation by a client, with transparent class export/uploading.
3. Active-object semantics for remote objects created using *Active-RMI* protocol.
4. User programmed scheduling and synchronization of remote method invocations.

Active-RMI libraries allows programmers to design and implement remote active-object interfaces. Using standard Java syntax, *Active-RMI* applications can create new

active remote objects or lookup existing active remote objects on participating hosts. Standard Java object references and method invocation syntax is used to perform asynchronous method calls on remote objects. Remote hosts where new active objects get created need to be active and willing participants in the *Active-RMI* protocol, but they do not need to have a-priory knowledge of the types of active objects that they will host. *Active-RMI* protocol transparently exports the needed class definitions in a demand-based fashion. Remote objects are created as autonomous agent-like objects, with a thread starting execution at a designated scheduler method. Active objects can access their private request queue of incoming *Active-RMI* requests from clients. We use the term "*active*" to describe the remote *Active-RMI* object's ability to execute arbitrary code as well as its ability to make autonomous scheduling decisions about which requests (if any) to serve after inspecting its request queue.

4.2 Overview of Active Remote Method Invocation System

Writing a distributed application using *Active-RMI* requires providing the following definitions for each active remote-object type:

- *Interface* specifications.
- *Implementation classes* for the interfaces.
- *Scheduler method* implementing active object behavior and synchronization.

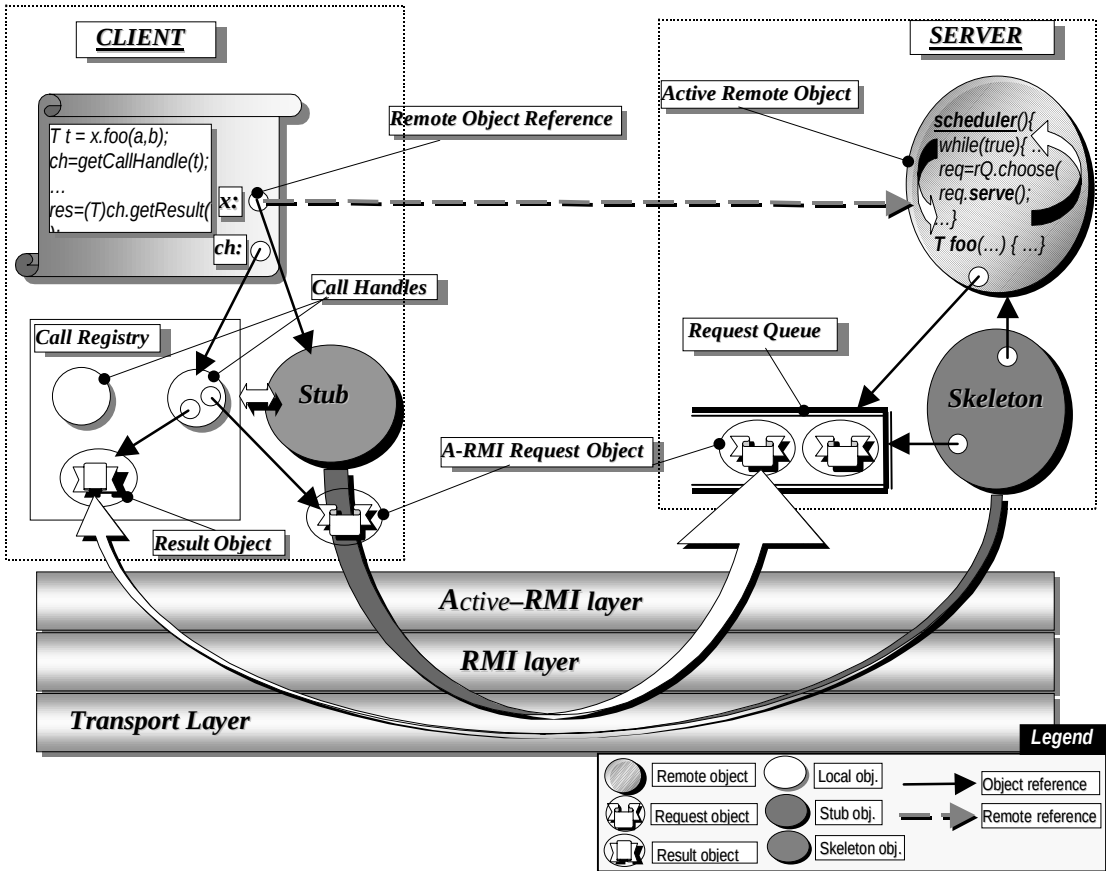


Figure 4.1: Asynchronous Remote Method Invocation

We call any object that creates or obtains a reference to an active remote-object a ***client*** of the remote object. Using its interface and implementation definitions, an active remote object can be created and referenced by client programs using an RMI based transport layer connecting different hosts participating in the distributed computation. Active remote-objects can be created on any host connected to the Internet that has an *Active-RMI Server daemon* running. We refer to the host on which an *Active-RMI* object has been created as a ***server***. It is possible for a client object to be involved in some distributed computation where it acts as a server to some other client. The *Active-RMI* system is built as an extension to Sun's Remote Method Invocation (RMI) system, and uses the same Java virtual machine, remote reference layer and a similar stub/skeleton creation and deployment mechanism.

4.2.1 Remote Object Referencing And Type-safety

A client object declares and references an *Active-RMI* object using its interface and implementation specifications using the same syntax as a standard Java object. The client can *obtain* a remote object reference either by using *Active-RMI* naming/ lookup service or by receiving the remote object reference as a parameter or result from another client. Alternatively, a client can *create* a new active remote-object on a remote *Active-RMI* server host using *Active-RMI* remote-object creation service. The client can invoke public methods of an active remote object as specified in its remote interface specification using *standard Java syntax* for object method calls. Since both the interface and implementation specifications of *Active-RMI* objects, and all *Active-RMI* libraries are written in Java, any standard Java compiler can statically enforce Java's strong typing for the client's code

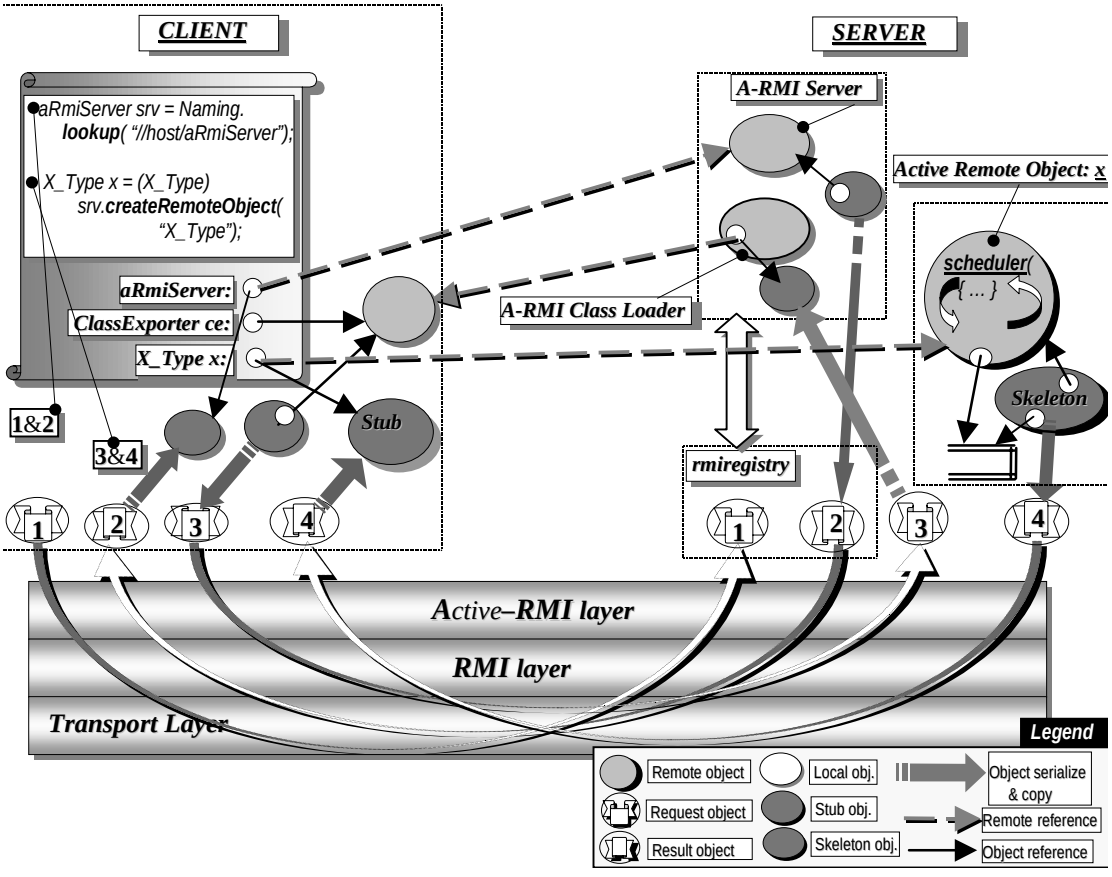


Figure 4.2: Active Remote Object Creation

4.2.2 Semantics Of *Active-RMI* Remote Method Call

Despite the similarity in syntax, the semantics of an *Active-RMI* remote-method call is different from that of a standard Java method call. There are three main differences: *asynchronous call* semantics; *by-value parameters-passing* instead of passing local object references; and the remote computation's being carried out in a different Java runtime environment, in a separate thread and address space. Figure 1. illustrates the object layout and references and the communication aspects of an asynchronous *Active-RMI* call. The call, on the *client* side returns immediately after registering the call locally with its *Active-RMI* call registry, without waiting for the remote computation to be completed. The value returned is a unique pseudo-result that the client needs to convert into a *call-handle*. The call-handle object can be used to do a blocking wait or non-blocking query about the delivery of actual result from the server. Remote method calls involve delivering the call as a request-packages into remote object's request queue, and receiving the result also as a network package delivered through a call-handle. When marshalling the local-object reference parameters of the call into a request package, or the result back to the client, *Active-RMI* uses object-serialization to flatten and pass deep-copies of local objects.

4.2.3 Active Objects And Scheduling

On the server_side, upon receiving a create remote-object request from a client, an *Active-RMI* server object creates the active remote object with the requested type and starts a new thread which begins executing in the object's `scheduler` method. A remote-stub reference is returned to the client, which can be used to invoke remote methods of the active object, or can be passed to other clients. The `scheduler` method, if provided as part of the object's implementation overrides a default (FIFO) scheduler and is written as a standard Java language method. By allowing *Active-RMI* objects to describe a behavior of their own using its `scheduler` method, and have autonomy over how and when to serve incoming requests *Active-RMI* introduces *active objects* as en extension of the passive object model of Java. *Active-RMI* system also provides each active object with a dynamic queue of requests, where each entry in the queue is an *aRmi_Request* object which has been asynchronously delivered by the underlying remote reference layer as a result of a client's invocation of one of the remote *Active-RMI* object's public methods. The scheduler method can wait for arrival of requests into the queue, inspect the contents of each request message, including the method's signature and the values of its actual parameters. When the scheduler decides to "*`serve`*" one of the requests, it is taken off the queue, executed and the result is asynchronously delivered back to the client.

4.2.4 Remote Object Creation And Distribution

Figure 2. illustrates the object layout, references, communication aspects and data-flow during remote creation of an active object. On each host where clients are allowed to create active remote-objects, a daemon process running our *Active-RMI* library's `ARrmiServer_Impl` must be started. This process registers itself with a standard *Java RMI rmiregistry* service under the name "*aRmiServer*". Clients can locate the *aRmiServer* using *rmiregistry* lookup with that name on the server host.

A new active remote object is created on the server using the `createRemoteObject` method of `aRmiServer`, by passing the remote-object's class name. If a local class by the requested class name is not found at the server host, an exception is thrown forcing the client's stub to transparently send a `ClassExporter` object, `ce`. The server uses the class exporter object to transparently install a new `aRmiClass-Loader`, downloads required client-side classes from the client and finally creates the requested active object on the server. A stub reference to the active object is returned to the client. In order to facilitate the potential need for installing a new class loader as required by *Active-RMI*'s remote object creation protocol, the `aRmiServer` process is started with a custom security manager, `aRmiSecurityManager`, which allows installing new class loaders.

4.3 Architecture of Active Remote Invocation System

The *Active-RMI* system is designed as an extension to Sun's *Java RMI* system and uses RMI libraries at implementation level whenever suitable. The classes and interfaces comprising the *Active-RMI* class library, and its inheritance hierarchy is shown in Figure 3. While this approach creates a tight coupling between our implementation and Java RMI, it provides some valuable benefits: since *Java RMI* is part of the standard distribution of Java Developer Kit as of JDK 1.1, *Active-RMI* libraries can be quite compact, reducing the amount and size of network traffic and initialization overhead for certain applications that need to dynamically download the *Active-RMI* class libraries; *Active-RMI* system uses the same standard Java Virtual Machine as *RMI* for runtime support, allowing our design to introduce new features and extensions without making modifications to the Java language or its virtual machine; applications that already use *RMI* can be easily extended to take advantage of *Active-RMI* extensions; future enhancements to *RMI* with regards to performance, reliability, security, etc., can directly benefit *Active-RMI* applications as well.

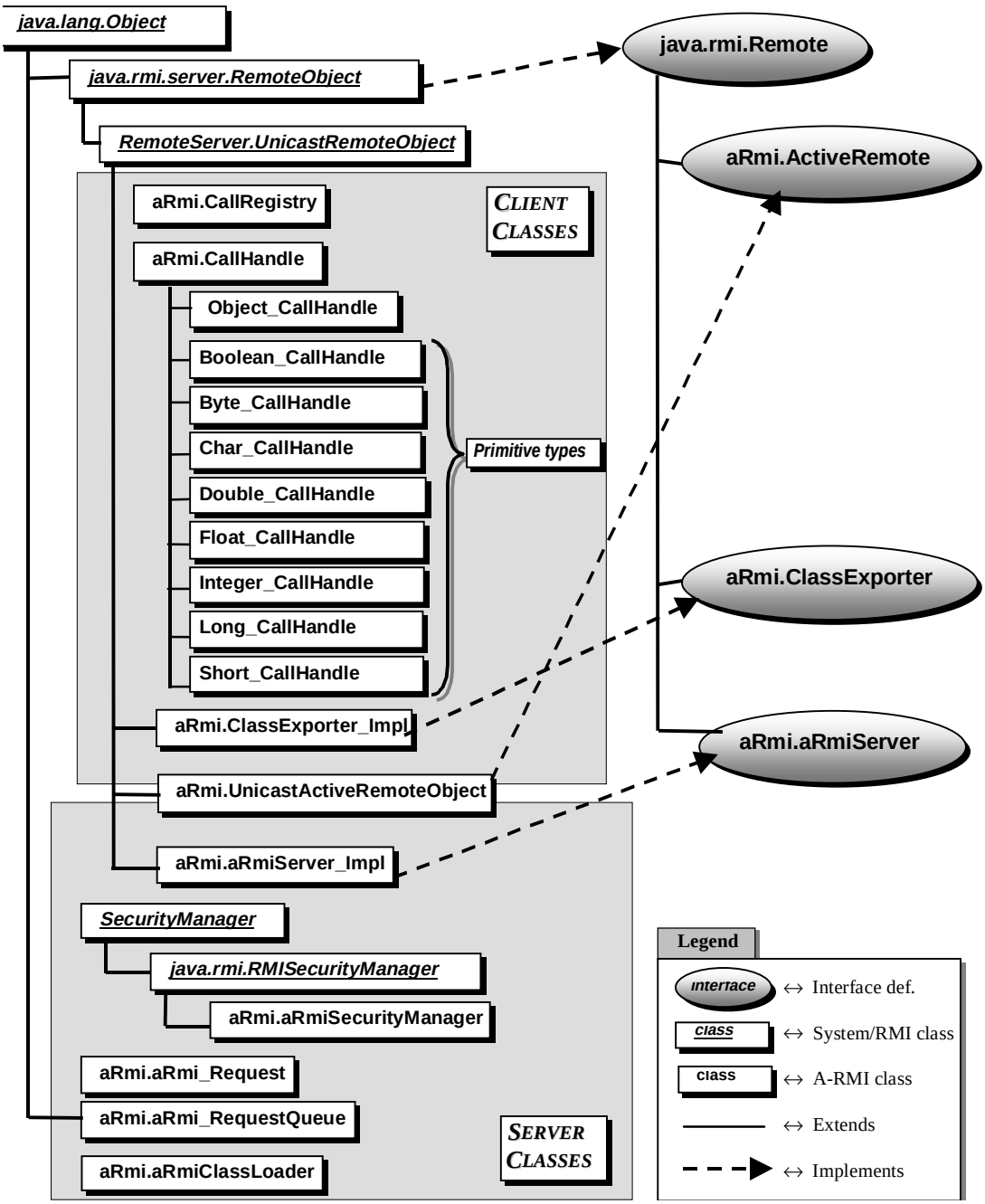


Figure 4.3: Active-RMI Class Library Organization

Active-RMI Class libraries are organized with two primary components: classes and tools to support *client* applications and those to support *server* implementations. This delineation is highlighted in Figure 3 where the grouping of client and server classes is shown as separate blocks. A high-level description of the functions supported by the client and server classes are described here, and the rest of this section elaborates on the functional description of the classes.

4.4 Client Abstractions and Functions

4.4.1 Active Remote Object Interfaces And Implementations

A client program can refer to an active remote object using its interface definition. Each active remote interface must extend the interface `ActiveRemote` and contain a list of the public methods that can be remotely invoked. The `ActiveRemote` interface of the *Active-RMI* library extends the Java RMI's `Remote` interface. Both of these interfaces have empty bodies and simply serve as *marker* types used by the automatic stub and skeleton generator utilities and by the Java Virtual Machine (JVM) to transparently use the remote reference layer. The JVM implicitly invokes the corresponding stub method during the method invocation of an `ActiveRemote` object instead of the method from its implementation class. The example below describes an active remote object interface, `ActObj,` which has a single method `foo` taking an `int` and an `Object` argument and returning a result of object type, `Bar:`

```
interface ActObj extends ActiveRemote
{
   public Bar foo(int a, T tObj);
}
```

A concrete class must implement the active remote interface in order for a client or server process to be able to create the actual active object. Active object implementation classes must extend *Active-RMI*'s `UnicastActiveRemoteObject` class. A client or server program, providing such an implementation class can create new active objects on authorizing hosts. Once the active object is created a remote stub reference can be exported to any client using its active remote interface definition. The *Active-RMI* library implementation of the `UnicastActiveRemoteObject` class extends the Java RMI library class `UnicastRemoteObject,` and provides the active remote object functions of the *Active-RMI* protocol to both clients and servers. An example implementation outline of the `ActObj` interface is presented below:

```
class ActObj_Impl  extends UnicastActiveRemoteObject
                   implements ActObj ...
{
        public Bar foo(int a, T tObj) { ... } ...
}
```

4.4.2 Active Remote Object Creation

Active remote objects can be created on a remote host using the *Active-RMI* server interface, `aRmiServer`. An *Active-RMI* server process implementing the `aRmiServer` interface is provided with the *Active-RMI* system and can be started by a user on that host machine as a user level service. This process needs to be registered with the Java RMI *rmiregistry* service under the name: *"aRmiServer"*. Once the registry name binding is established clients can lookup and locate the *Active-RMI* server object:

```
aRmiServer rServer=(aRmiServer)Naming.lookup(
                                        "//hostname/aRmiServer");
```

After the remote *Active-RMI* server reference, `rServer`, is obtained, it can be used to create active objects on the remote server by using the `aRmiServer` object's exported interface's `createRemoteObject` method which takes the name of a class implementing the remote active interface as a parameter used to create the remote active objects on the server:

```
ActObj obj = (ActObj)rServer.createRemoteObject(
                    "ActObj_Impl");
```

or

```
ActObj obj=(ActObj)rServer.createRemoteObject(
                    "ActObj_Impl",new ClassExporter_Impl());
```

The latter form is typically used transparently by the stub implementation of the *aRmiServer* object, when the former form typically used by the client programmer raises a remote exception due to classes not being found on the remote server host.

The `createRemoteObject` returns a stub handle as a a remote-reference to the newly created object on the remote server. Calling the public methods of this object as specified in interface `ActObj` is transparently carried out as an asynchronous remote method call.

4.4.3 Asynchronous Remote Method Calls And Call Handles

Methods of active remote objects are always called *asynchronously*, i.e. the call returns to the client immediately with a pseudo result, which can be converted to a *call handle.* A unique call-handle is associated with each asynchronous remote method invocation. The client can instantiate the call handle through the constructor of the proper `CallHandle` *subclass*, passing as argument the temporary result returned by the remote call -- the exception is a `Void_CallHandle`, whose constructor takes no arguments. The call-handle can be used to either do a blocking wait on the result of the remote computation, via `getResult`, or do non-blocking query of result's availability, via `resultReady`. In the example code snipet below, the client's invocation of the method, `foo`() of the active remote object, `obj`, returns a result whose type is the same type that is declared in `obj` 's remote interface. The type of the result that a remote method returns can be any of the primitive Java language types or some `Object` type which implements the `serializable` interface. In the following example:

```
Bar t  = obj.foo(arg1,arg2);
Object_CallHandle th = new Object_CallHandle(t);
```

obj's `foo()` method is invoked with two actual arguments, `arg1` and `arg2`, and returns immediately with a temporary unique value, `t`, of type *`Bar`*. This value is subsequently used to obtain a call handle, *`th`*, of type `Object_CallHandle`. The `waitFinished`() method of `CallHandle` class can be used for synchronizing with the completion of the remote computation without actually collecting the result, this is the only way for synchronizing with a void return type remote method.

Since Java primitive types are not derived from `Object` class, the `getResult` method cannot be written as a generic method returning a generic object or primitive type. Therefore, actual call-handles need to be created as one of the specialized subclasses of `CallHandle`, such as an `Integer_CallHandle` for `int` result type calls, or `Object_CallHandle` for remote calls returning an `Object` or array type, etc. *Active-RMI* also defines and uses a *call registry* abstraction to transparently generate and manage the call-handles.

4.4.4 Call Registry And Synchronization With Remote Host

Active-RMI system transparently maintains a *call-registry* object associated with each client. The call registry is used by the client stubs to create unique temporary results

and to register the call details and the associated remote reference layer state information. This registry information is used when the constructor of a `CallHandle` class is called to associate the remote asynchronous call with a call-handle for future result delivery. Using a call handle obtained from the call-registry the client can either do a blocking wait, using the `getResult` method of the `CallHandle` class, and wait for the result's arrival from the remote server; or do a non-blocking query, using the `resultReady` method which immediately returns true or false depending on whether the result is ready. In the following example:

```
int result = ih.getResult();
```

the client waits for the remote computation to finish and ship back the result of the computation associated with the `Integer_CallHandle`, `ih`, and the received value is assigned to result. For primitive type results no casting is necessary since the call handle already holds the type information. For remote methods returning arbitrary object-types, however, the value returned needs to be cast to the right type. In the code segment below, the `Object_CallHandle, oh,` is being used to synchronize with a remote call returning a `T` type object:

```
T resObj =  (T) oh.getResult();
```

4.5 Stub and Skeleton Classes

Active-RMI uses stub classes on the client side for marshalling and registry of the remote calls and skeleton classes on the remote server host for request dispatching and scheduling. The interface approach and stub/skeleton naming conventions are identical to the standard Java RMI calls. The construction and use of Active-RMI stub and skeleton classes is also similar to that of standard Java RMI stubs.

4.5.1 Client side Stub Classes

When a client calls a method of an active remote object the actual method that is invoked is the corresponding *stub class* method. The stub object acts as a proxy implementing the remote object's interface by marshalling each call into a request package and delivering it to the remote object on the server side. This process is very similar to the standard Java RMI system and the current implementation is built as an extension to the RMI remote reference layer. Unlike RMI, however, *Active-RMI* stubs return back immediately to the caller a unique temporary result after implicitly registering

the call-details with a local client-side registry, without waiting for the actual result of the computation from the remote host. It is the client's responsibility to keep track of the remote call status and result using a call handle for each remote invocation. Client obtains the call handle from the registry using the unique temporary returned by the stub method. The actual result for a remote call gets delivered asynchronously by the remote active object and is ultimately returned to the client application when the call-handle's `getResult` method is invoked.

4.5.2 Server Side Skeleton Classes

Skeleton Classes for active remote objects also work similar to their Java RMI counterparts. Skeleton classes are uploaded to the server by the `aRmiClassLoader` possibly through the use of a `ClassExporter` object at the time the first remote object is created. When a client invokes a method of a remote object, the corresponding stub method is called by the RMI Java runtime, which in turn marshals the call parameters and *dispatches* the call to the corresponding skeleton object on the remote server. *Active-RMI* skeleton classes' *`dispatch`* implementation, unlike its RMI counterpart, does not immediately serve the request but instead converts the call parameters into an `aRmi_Request` object and delivers it to the active object's `RequestQueue`. Active object's *`scheduler`* is responsible for the final call completion and the asynchronous delivery of the result back to the client by invoking the `serve` method of the request object.

4.5.3 Stub and Skeleton Class Examples

In this section we present stub, skeleton and implementation classes implementing the simple `Hello` interface:

```
public interface Hello
           extends java.rmi.Remote
{
    public String sayHello(int id)
           throws java.rmi.RemoteException;
}
```

```
import  activeRmi.*;

public final class HelloImpl_Stub
    extends java.rmi.server.RemoteStub
    implements Hello, java.rmi.Remote
{
    private static java.rmi.server.Operation[] operations = {
        new java.rmi.server.Operation(
                    "java.lang.String sayHello(int)")
    };
private static final long interfaceHash =
                                  -5352162705594599214L;

public HelloImpl_Stub() {  super();   }
public HelloImpl_Stub(java.rmi.server.RemoteRef rep)
         {super(rep);}
         // Methods from remote interfaces
         // Implementation of sayHello
public java.lang.String sayHello(int $_int_1) throws
                                   java.rmi.RemoteException {
        int opnum = 0;
        java.rmi.server.RemoteRef sub = ref;
        java.rmi.server.RemoteCall call =
          sub.newCall((java.rmi.server.RemoteObject)this,
                      operations, opnum, interfaceHash);
        try {
            java.io.ObjectOutput out = call.getOutputStream();
            out.writeInt($_int_1);
        } catch (java.io.IOException ex) {
            throw new java.rmi.MarshalException(
                     "Error marshaling args", ex);
        };
        try { sub.invoke(call);
        } catch (java.rmi.RemoteException ex) {
            throw ex;
        } catch (java.lang.Exception ex) {
          throw new java.rmi.UnexpectedException(
                     "Unexpected exception", ex);
        };
         java.lang.String callID = new java.lang.String();
         FutureRegistry.registerCall(callID, call, ref);
         return callID;
    }
}
```

Figure 4.4: Active-RMI Hello Implementation Stub Class

```
public final class HelloImpl_Skel
    extends java.lang.Object
    implements java.rmi.server.Skeleton,
    Dispatcher, Runnable
{
    private static java.rmi.server.Operation[] operations = {
        new java.rmi.server.Operation(
                    "java.lang.String sayHello(int)")
    };

    private static final long interfaceHash = -
5352162705594599214L;

    public java.rmi.server.Operation[] getOperations() {
        return operations;
    }
    public aRmi_RequestQueue  RequestQueue;
    public HelloImpl server;

    public void dispatch(java.rmi.Remote obj,
         java.rmi.server.RemoteCall call, int opnum, long hash)
         throws java.rmi.RemoteException, Exception {

         if (hash != interfaceHash)
             throw new java.rmi.server.SkeletonMismatchException(
                                       "Hash mismatch");
        server = (HelloImpl)obj;

        java.io.ObjectOutput out;
        RequestQueue = server.RequestQueue;
        if (activeThread == null) {
            activeThread = new Thread ( (Runnable)this );
            activeThread.start();
        }

        switch (opnum) {
        case 0: { // sayHello
            int $_int_1;
            try {
                java.io.ObjectInput in = call.getInputStream();
                $_int_1 = in.readInt();
            } catch (java.io.IOException ex) {
                throw new java.rmi.UnmarshalException(
                     "Error unmarshaling arguments", ex);
            } finally {
                call.releaseInputStream();
            };
```

```
        try {
               out = call.getResultStream(true);
        } catch (java.io.IOException ex) {
              throw new java.rmi.MarshalException(
                    "Error marshaling return", ex);
        };

        Object args[] = new Object[1];
        args[0] = new java.lang.Integer($_int_1);
        Thread thisThread = Thread.currentThread();
        server.RequestQueue.push( new aRmi_Request(this, server,
                               call, thisThread , opnum, args));

        activeThread.resume();
        thisThread.suspend();

        /*
         * suspended thread resumes here to finish computation
         */
        java.lang.String $result = server.sayHello($_int_1);

        try {
         out.writeObject($result);
        } catch (java.io.IOException ex) {
         throw new java.rmi.MarshalException(
                    "Error marshaling return", ex);
        };

        break; // thread will break free from here.
        }
       default:
           throw new java.rmi.RemoteException(
                    "Method number out of range");
        }
}

public void run() {
      try {
          defaultScheduler();
      } catch (java.rmi.RemoteException ex) {

      } catch (Exception ex) {}
}

public  void defaultScheduler()
                    throws java.rmi.RemoteException, Exception
{
    while(true) {
```

```
          while (!RequestQueue.empty()) {
                aRmi_Request currentRequest =
                 RequestQueue.getFirstRequest();
                currentRequest.serve();
                RequestQueue.removeRequest(currentRequest);
          }

          Thread.yield();
          waitForRequest(1000L); // block till next
requestdispatched
          Thread.yield();
         }
    }

    public void waitForRequest(long ms) {

       Thread.currentThread().suspend();
    }

    public boolean debug = false;

    private boolean started = false; // set to true when
                                     // activeThread starts
    private Thread activeThread = null;
}
```

Figure 4.5: Active-RMI Hello Implementation Skeleton Class

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import activeRmi.*;

public class HelloImpl
         extends UnicastActiveRemoteObject
         implements Hello
{
    private String name;

    public activeRmi.aRmi_RequestQueue  RequestQueue;
    public HelloImpl()
         throws java.rmi.RemoteException, Exception
    {
         RequestQueue = new activeRmi.aRmi_RequestQueue();
    }

    public String sayHello(int id) throws RemoteException
    {
         return  "Hello aRmi World! <id=" + id + ">";
    }
}
```

Figure 4.6: Active-RMI Hello Implementation Class

4.6 Exceptions

Exceptions that may be thrown as a result of a remote method invocation are passed through the RMI layer and can be caught in the client with the usual Java syntax and semantics. Remote exceptions as specified by Java RMI are handled in the same fashion in *Active-RMI*.

4.7 Server Abstractions and Functions

4.7.1 Starting *Active-RMI* server

A special *Active-RMI* server process, implementing the `aRmiServer` interface needs to be running on a server host in order for a client application to be able to remote-cre-

ate active-RMI objects on that host. A new active remote object can be created on the remote host via `createRemoteObject` method which returns a remote stub reference to the newly created active object. Without the *Active-RMI* server process running, a local object can still create active objects on its own host and export them to clients possibly residing on different hosts, but clients on remote machines will not be able to directly connect to this host and create active remote objects on it. The *Active-RMI* server process is registered using standard RMI registry mechanism using the name "*aRmiServer*". Client applications can locate and obtain remote object references to the server by `java.rmi.Naming` utility:

```
aRmiServer rHost = (aRmiServer)java.rmi.Naming.lookup(
                                          //hostName/aRmiServer");
```

Active-RMI class libraries contains a class called `ARrmiServer_Impl` which implements the `aRrmiServer` interface. This class can be used to instantiate an *Active-RMI* server object as part of an application. Alternatively, a command-level utility, *aRrmiServer* is provided and can be started independently as a standalone daemon process or user level application.

4.7.2 *Active-RMI* Server And Remote Object Creation

The primary function of the *Active-RMI* server object implementing the `aRmiServer` interface is to facilitate remote object creation. It provides the method, `createRemoteObject` to be used by clients to spawn new active remote objects. Invoking the `createRemoteObject` method of an `aRmiServer` object results in the creation of a new object on that server host. An `ActiveRemote` object reference, a subtype of `java.rmi.Remote`, is returned to the client. The returned reference acts as a proxy to the actual remote object: client's invocation of the proxy's methods are handled as asynchronous *Active-RMI* requests, executed on the remote host. The `createRemoteObject` method can be invoked with a single string argument, which contains the implementation class path of the active remote object implementation:

```
Hello obj= (Hello) rHost.createRemoteObject(
                                  "aRmiExamples.HelloImpl");
```

or with an additional `ClassExporter` argument,

```
Hello obj= (Hello) rHost.createRemoteObject(
               "aRmiExamples.HelloImpl",new ClassExporter_Impl());
```

The latter form is typically used transparently by the stub implementation of the *Active-RMI Server* object. If the server cannot find some of the classes locally, it throws an exception forcing the client's stub to pass a `ClassExporter` object. When the client responds by passing a class exporter object, a new class loader, `aRmi-ClassLoader,` is created and installed transparently by the server utilizing the remote class exporter object received from the client to download required client-side classes and create the active object on the server. In order to facilitate the need for installing a new class loader as required by *Active-RMI* remote object creation protocol, the *aRmiServer* process uses *Active-RMI*'s security manager, `aRmiSecurity-Manager`, which allows installing new class loaders. Finally, the *aRmiServer* process handling the `createRemoteObject` request creates the active object requested by the client, exports it using the RMI remote reference layer and returns a stub reference to the client. Figure 4.2 depicts a typical object creation scenario.

4.7.3 Active Object Semantics

We use the term *active object* to describe an object which has an independent thread; has access to a request queue of incoming method invocation requests; and has the ability to inspect, select and serve any of the requests in its request queue. The *Active-RMI* system extends the Java object model by giving active objects autonomy over how and when to respond to their clients' requests. Active objects are created using *Active-RMI* `UnicastActiveRemoteObject` class which uses the RMI remote reference layer, the communication subsystem, and provides the *Active-RMI request queue* and a default (FIFO) `scheduler`. The *Active-RMI* server, creates each active object with a request queue and a dedicated thread and then returns to the client a remote-stub reference. The active object's thread begins execution in its `scheduler` method.

In *Active-RMI* active object model, each request to the active remote object is delivered as an `aRmi_Request` message object into the target object's private request pool. There are no service guarantees, or an implicit order in which the arrived messages will be served. The active object inherits the `RequestQueue` constructed by the *Active-RMI* protocol as a private instance variable, and uses the thread executing its `scheduler` to decide how to process requests placed into the queue. The `scheduler` method, however, is not restricted to only implementing scheduling activities and it can perform general purpose, *agent-like* computation. The active object's scheduling thread can access the `RequestQueue` directly and peek into the queue of `aRmi_Requests`, and implement a selection policy to choose and serve one of the requests in the queue. The scheduling policy can involve inspecting each request's type, or its method signature, or the values of its actual arguments. The scheduler

thread may also decide not to serve any of the current requests in the queue; or may wait until a certain event takes place, or a certain request arrives. A simple `scheduler` code, enforcing a last-in-first-out (LIFO) style servicing of its request queue is shown below:

```
private void  scheduler()
              throws java.rmi.RemoteException,Exception
{
   while(true) {
         while (!RequestQueue.empty()) {
               aRmi_Request currentRequest =
                     RequestQueue.getLastRequest();
               currentRequest.serve();
         }
         RequestQueue.waitForRequest(); // blocking call
   }
 }
```

4.7.4 *Active-RMI* Scheduler and Request Queue

Each request to an active objects gets queued in its `RequestQueue` as an `aRmi_Request` object which represents the client's remote method invocation in the form of a message containing the type and signature of the method and the serialized arguments list, `args`, in the form of an `Object` array. While executing inside the *`scheduler`* method, the active object can examine its `RequestQueue` by peeking into the request objects and possibly select one based on the signature of the requested method, or the contents of its actual parameters and finally serve the request by invoking the request object's `serve` method. To access the signature and then the *k-th* parameter of the earliest call in the request queue following code segment can be used:

```
aRmi_Request aRequest = RequestQueue.getFirst();
String sig = aRequest.getMethodSignature();
Object argK = aRequest.args[k-1];
```

The actual parameters of the call are available in the public `Object[]` component, `args`. All primitive-type arguments are stored as objects using their `java.lang.`*primitive* wrapper classes, and can obtained by unwrapping from its object format. The signature stored in the request can be used to enforce type checking when unbundling the parameters. The `RequestQueue` supports the standard `java.util.Enumeration` interface to iterate over its elements as well as providing several convenience functions, for example:

```
aRequest = RequestQueue.getFirst();
aRequest = RequestQueue.getFirst("foo");
aRequest = RequestQueue.getFirst("foo(int)");
```

the no-arg version of `getFirst()` method dequeues and returns the first request in `RequestQueue`. The `getFirst("foo")` call scans and returns the first request in `RequestQueue` whose target method's name is "`foo`". The third variant returns the first request in the queue whose signature matches "foo(int)". If there are no pending requests or no match is found a `null` value is returned. The `getLast()` method is provided with a similar semantics except that the `RequestQueue` is scanned in reverse order. Other variants for queue access, `peekFirst` and `peekLast` methods, return a request object without dequeueing. Further synchronization methods provide waiting behavior which block until a particular request type arrives into the `Request-Queue`:

```
waitForRequest();        // any request
waitForRequest("foo");// any foo request
waitForRequest("foo(int)");// any foo(int) request.
```

4.7.5 Request Handling And Synchronization Support

Each *Active-RMI* request object is associated with a thread which has been suspended just before it began to execute the target method with the passed arguments. Once a decision is made to serve the request, the thread can be resumed by:

```
thisRequest.serve();
```

which executes the target method of the current active object with the actual arguments, and then asynchronously sends the request back to the client which had invoked the remote method.

4.8 Implementation and Performance Issues

The latest *Active-RMI* system uses the standard Java RMI remote reference layer as a transport layer. The prototype implementation is written entirely in pure Java and has been tested using Sun Microsystem's Java Developers Kit, JDK 1.1 and 1.2 releases on machines running various Windows and Solaris O/S flavors. The dependence on standard RMI support has the following implications: synchronous Active-RMI remote calls are at best only as efficient as standard RMI calls; performance and portability of Active-RMI protocol will always benefit from enhancements and deployment of Sun's Java *RMI* infrastructure. Our key performance objective is to incur negligible run-time overhead for *Active-RMI* calls in excess of the standard *RMI* runtime overhead. We discuss how we attain our objectives by briefly outlining the *Active-RMI* activities on the client and the server.

On the client side the the only additional overhead is due to creation of a unique call identifier (standard object creation) and its registration with the call registry (hashtable insertion and lookup). Communication layer and marshalling costs within the stub methods are identical to standard *RMI* calls. Client side thread creation overhead is minimized by a demand based scheme: a new client side thread is created only when a *getResult* operation blocks because the remote result is not ready. Further optimization is possible (though not implemented in current implementation) by using shared thread pools and management techniques.

Server side *Active-RMI* overhead is primarily due to the creation of *request* objects and the related queue management overhead inside the *scheduler* method. The only thread created by the *Active-RMI* system is at setup time, during the creation of the active object *scheduler* thread. Standard *RMI* runtime already creates a dedicated I/O thread for dispatching remote request messages to their target object's skeleton methods, so no new additional thread creation is performed by *Active-RMI*. The skeleton method simply suspends the current dispatch thread after enqueueing the request to the request queue. The scheduler *serves* a request simply by resuming the suspended *RMI* dispatch thread.

4.9 Summary

We have introduced Active Remote Method Invocation system, *Active-RMI*, as a set of class libraries, tools, and a design method for building distributed applications using an active remote object abstraction in Java. *Active-RMI* provides a very high level of programming for writing complex object distribution and synchronization applications

entirely in Java. Three key abstractions provided by the *Active-RMI* model are: the active remote objects with user level scheduling; asynchronous method invocation with data-driven, non-blocking synchronization using call-handles; and transparent remote object creation. The first two abstractions allow us to support both reply and request scheduling. *Reply scheduling* is the control the client has over the delivery of reply/result, which we address by the asynchronous remote method invocation mechanism and by providing both blocking and non-blocking result handling capability using call-handles. *Request scheduling* is the control the active object server has over the acceptance and serving of incoming requests. We support request scheduling by delivering the requests to an active server object's private request queue, and by providing a `scheduler` method abstraction for coding active object behavior with run-time support to access its own request queue and to make autonomous scheduling decisions. Our approach provides a mechanism for dealing with local delays, which is deemed essential in Liskov et al.'s[46] formulation of concurrency requirements for developing client/server type distributed programs.

Having a designated `scheduler` method with run-time request queue access helps coding complex synchronization specifications externally to the object's methods and thus helps enhance maintainability and reusability. The separation of synchronization and functional specifications is not only useful for reducing complexity but also allows our model to easily program *agent-like* active objects.

An important area of further study for us is the security modeling. We have currently a rather strict security mechanism based on the RMI security model, however, it is desirable to have less rigid authorization and security abstractions.

Performance is a critical mission of the *Active-RMI* system development, and our implementation and design approach incurs minimal cost on top of underlying the Java RMI system overhead.

An issue that needs further investigation is identifying the synchronization and correctness needs in the presence of multi-threaded schedulers. We are attempting to gain more insight by building systems based on *Active-RMI* and comparing with alternative approaches.

Chapter 5

jContractor: A Reflective Java Library for Design by Contract

5.1 Introduction

In this Chapter we discuss the design and implementation of *jContractor*, a purely library-based system and a set of naming conventions to support *Design By Contract* in Java. The *jContractor* system does not require any special tools such as modified compilers, runtime systems, modified JVMs, or pre-processors, and works with any pure Java implementation. Therefore, a programmer can practice *Design By Contract* by using the *jContractor* library and by following a simple and intuitive set of conventions.

Each class and interface in a Java program corresponds to a translation unit with a machine and platform independent representation as specified by the Java Virtual Machine (JVM) `class` file format [45]. Each class file contains JVM instructions (bytecodes) and a rich set of meta-level information. *jContractor* utilizes the meta-level information encoded in the standard Java class files to instrument the bytecodes on-the-fly during class loading. During the instrumentation process *jContractor* parses each Java class file and discovers the *jContractor* contract information by analyzing the class meta-data.

The *jContractor* design addresses three key issues which arise when adding contracts to Java: how to express preconditions, postconditions and class invariants and incorporate them into a standard Java class definition; how to reference entry values of attributes, to check method results inside postconditions using standard Java syntax; and how to check and enforce contracts at runtime.

A discussion of key Design By Contract abstractions is introduced in Chapter 2 section titled “Design by Contract Abstractions” on page 30. In this chapter we first give a brief overview of *jContractor*'s approach and in the subsequent sections discuss design and implementation details.

5.2 jContactor Overview

jContractor provides an intuitive set of high-level programming abstractions to define and perform runtime checking of Design by Contract specifications.

- Programmers add contract code to a class in the form of methods following *jContractor*'s naming conventions: *contract patterns*. The *jContractor* class loader recognizes these patterns and rewrites the code to reflect the presence of contracts.
- Contract patterns can be inserted either directly into the class or they can be written separately as a *contract class* where the contract class' name is derived from the target class using *jContractor* naming conventions. The separate contract class approach can also be used to specify contracts for interfaces.
- The *jContractor* library instruments the classes that contain *contract patterns* on the fly during class loading or object instantiation. Programmers enable the runtime enforcement of contracts either by engaging the *jContractor* class loader or by explicitly instantiating objects from the *jContractor* object factory. Programmers can use exactly the same syntax for invoking methods and passing object references regardless of whether contracts are present or not.
- *jContractor* uses an intuitive naming convention for adding *preconditions, postconditions, class invariants, recovery* and *exception handling* in the form of `protected` methods. Contract code is hence distinguished from the functional code. The name and signature of each contract method determines the actual method with which the contract is associated.
- *Postconditions* and *exception handlers* can access the *old* value of any attribute by using a special object reference, *`OLD`*. For example *`OLD.count`* returns the value of the attribute *`count`* just prior to the execution of the method. *jContractor* emulates this behavior by transparently rewriting class methods during class loading so that the entry values of *`OLD`* references are saved and then made available to the postcondition and exception handling code.
- *jContractor* provides a class, *`RESULT`*, and a `static` method, *`Compare`*. Inside a method's postcondition *`RESULT.Compare(`*<*expression*>*`)`* returns true or false by comparing the value of the <*expression*> to the current result.

5.3 *jContractor* Library and Contract Patterns

jContractor is a purely library-based approach to support Design By Contract constructs using standard Java. Table 2.4, “Overview of jContractor Design By Contract Abstractions,” on page 34 contains a brief summary of key *jContractor* constructs and patterns.

A programmer writes a contract by taking a class or method name, say `put`, then appending a suffix depending on the type of constraint, say `_PreCondition`, to write the `put_PreCondition`. Then the programmer writes the method body describing the precondition. The method can access both the arguments of the `put` method with the identical signature, and the attributes of the class. When *jContractor* instrumentation is engaged at runtime, the precondition gets checked each time the `put` method is called, and the call throws an exception if the precondition fails.

The code fragment in Figure 5.1 shows a *jContractor* based implementation of the `put` method for the `Dictionary` class. An alternative approach is to provide a separate *contract class*, `Dictionary_CONTRACT`, as shown in Figure 1-b, which contains the contract code using the same naming conventions. The contract class can (optionally) extend the target class for which the contracts are being written, which is the case in our example. For every class or interface `X` that the *jContractor ClassLoader* loads, it also looks for a separate contract class, `X_CONTRACT`, and uses contract specifications from both `X` and `X_CONTRACT` (if present) when performing its instrumentation. The details of the class loading and instrumentation will be presented in subsequent sections.

Figure 5.1: Dictionary Class Implementing Contract for `put` Method

```
class Dictionary... {
   protected Dictionary OLD;
   Object put(object x, String key)
   {
         putBody();
   }
   protected boolean put_PreCondition(object x, String key)
   {
         return( (count <= capacity)
               && !(key.length() == 0));
   }
   protected boolean put_PostCondition (object x, String key)
   {
         return( (has(x))
               && (item(key)== x)
               && (count == OLD.count +1 ))
   }
   protected boolean Object put_OnException(Exception e)
                           throws Exception
   {
         count = OLD.count;
         throw e;          //rethrow exception.
   }
   protected boolean Dictionary_ClassInvariant()
   {
         return(count >= 0);
   }
}
```

Figure 5.2: Separate Contract Class for `Dictionary`

```
Class Dictionary_CONTRACT extends Dictionary...
{
   protected boolean put_PostCondition(Object x, String key)
   {
         return ((has(x)) && (item(key)==x) && (count==OLD.count+1)
   }
   protected boolean Dictionary_ClassInvariant(){
               return(count >= 0);
   }
}
```

5.3.1 Runtime Contract Monitoring

In order to enforce contract specifications at run-time, the contractor object must be instantiated from an instrumented class. This can be accomplished in two possible ways: (1) by using the *jContractor class loader* which instruments all classes containing contracts during class loading; (2) by using a factory style instantiation using the *jContractor* library.

The simplest and the preferred method is to use the *jContractor class loader*, since this requires no changes to a client's code. The following code segment shows how a client declares, instantiates, and then uses a `Dictionary` object, *`dict`*. The client's code remains unchanged whether *jContractor* runtime instrumentation is used or not:

```
Dictionary  dict;            // Dictionary (Figure-5.1) defines contracts.

dict = new Dictionary();     // instantiates dict from instrumented or
                             // non- instrumented  class depending on
                             // jContractor class-loader being engaged.

dict.put(obj1,"name1");      // If jContractor is enabled, put-contracts
                             // are enforced,  i.e. contract  violations
                             // result in an exception being thrown.
```

The second approach uses the *jContractor* object factory, by invoking its *`New`* method. The factory instantiation can be used when the client's application must use a custom (or third party) class loader and cannot use *jContractor class loader*. This approach also

gives more explicit control to the client over *when* and *which* objects to instrument. Following code segment shows the client's code using the *jContractor* factory to instantiate an instrumented `Dictionary` object, *`dict`*:

```
dict = (Dictionary) jContractor.New("Dictionary");
                                        // instruments  Dictionary

dict.put(obj1,"name1");                 // put-contracts are enforced
```

Syntactically, any class containing *jContractor* design-pattern constructs is still a pure Java class. From a client's perspective, both instrumented and non-instrumented instantiations are still Dictionary objects and they can be used interchangeably, since they both provide the same interface and functionality. The only semantic difference in their behavior is that the execution of instrumented methods results in evaluating the contract assertions, (e.g., *`put_PreCondition`*) and throwing a Java runtime exception if the assertion fails.

Java allows method overloading. *jContractor* supports this feature by associating each method variant with the pre- and postcondition functions with the matching argument signatures.

For any method, say *`foo`*, of class *`X`*, if there is no *`boolean`* method by the name, *`foo_PreCondition`* with the same argument signature, in either *`X`*, *`X_CONTRACT`* or one of their descendants then the default precondition for the *`foo`* method is "true". The same "default" rule applies to the postconditions and class invariants.

5.3.2 Naming Conventions for Preconditions, Postconditions and Class Invariants

The following naming conventions constitute the *jContractor* patterns for pre- and postconditions and class invariants:

Precondition: *protected boolean methodName + _PreCondition + (<arg-list>)*
Postcondition: *protected boolean methodName + _PostCondition + (< arg-list >)*
ClassInvariant: *protected boolean className + _ClassInvariant ( )*

Each construct's method body evaluates a *`boolean`* result and may contain references to the object's internal state with the same scope and access rules as the original method. Pre- and postcondition methods can also use the original method's formal

arguments in expressions. Additionally, postcondition expressions can refer to the old values of object's attributes by declaring a pseudo object, `OLD`, with the same class type and using the `OLD` object to access the values.

5.3.3 Exception Handling

The postcondition for a method describes the contractual obligations of the contractor object only when the method terminates successfully. When a method terminates abnormally due to some exception, it is not required for the contractor to ensure that the postcondition holds. It is very desirable, however, for the contracting (supplier) objects to be able to specify what conditions must still hold true in these situations, and to get a chance to restore the state to reflect this.

jContractor supports the specification of general or specialized exception handling code for methods. The instrumented method contains wrapper code to catch exceptions thrown inside the original method body. If the contracts include an exception-handler method for the type of exception caught by the wrapper, the exception handler code gets executed.

If exception handlers are defined for a particular method, each exception handler must either re-throw the handled exception or compute and return a valid result. If the exception is re-thrown no further evaluation of the postconditions or class-invariants is carried out. If the handler is able to recover by generating a new result, the postcondition and class-invariant checks are performed before the result is returned, as if the method had terminated successfully.

The exception handler method's name is obtained by appending the suffix, *"_OnException"*, to the method's name. The method takes a single argument whose type belongs to either one of the exceptions that may be thrown by the original method, or to a more general exception class. The body of the exception handler can include arbitrary Java statements and refer to the object's internal state using the same scope and access rules as the original method itself. The *jContractor* approach is more flexible than the Eiffel's "*rescue*" mechanism because separate handlers can be written for different types of exceptions and more information can be made available to the handler code using the exception object which is passed to the handler method.

5.3.4 Supporting Old Values and Recovery

jContractor uses a clean and safe instrumentation “trick” to mimic the Eiffel keyword, *old*, and support Design By Contract style postcondition expressions in which one can refer to the “*old*” state of the object just prior to the method’s invocation. The trick involves using the "syntax notation/convention", *OLD.x* to mean the value that x had when method body was entered. Same notation is also used for method references as well, e.g., *OLD.foo()* is used to refer to the result of calling the member *foo()* when entering the method. We will later explain how the *jContractor* instrumentation process rewrites expressions involving *OLD* to achieve the desired effect. First we illustrate its usage from the example in Figure 1. The class Dictionary first declares *OLD*:

```
private Dictionary OLD;
```

Then, in the postcondition of the put method taking *<Object x, String key>* arguments, the following subexpression is used

```
(count == OLD.count + 1)
```

to specify that the execution of the corresponding put method increases the value of the object's *count* by 1. Here *OLD.count* refers to the value of *count* at the point just before the *put*-method began to execute.

jContractor implements this behavior using the following instrumentation logic. When loading the Dictionary class, *jContractor* scans the postconditions and exception handlers for *OLD* usage. So, when it sees the *OLD.count* in *put_PostCondition* it inserts code to the beginning of the put method to allocate a unique temporary and to save count to this temporary. Then it rewrites the expression in the postcondition replacing the *OLD.value* subexpression with an access to the temporary. In summary, the value of the expression *OLD.expr* (where *expr* is an arbitrary sequence of field dereferences or method calls) is simply the value of *expr* on entry to the method.

It is also possible for an exception handler or postcondition method to revert the state of *attr* to its old value by using the *OLD* construct. This may be used as a basic recovery mechanism to restore the state of the object when an invariant or postcondition is found to be violated within an exception-handler. For example,

```
attr = OLD.attr;
```

or

```
attr = OLD.attr.Clone();
```

The first example restores the object reference for *attr* to be restored, and the second example restores the object state for *attr* (by cloning the object when entering the method, and then attaching the object reference to the cloned copy.)

5.3.5 Separate Contract Classes

jContractor allows contract specifications for a particular class to be externally provided as a separate class, adhering to certain naming conventions. For example, consider a class, *X*, which may or may not contain *jContractor* contract specifications. *jContractor* will associate the class name, *X_CONTRACT*, with the class *X*, as a potential place to find contract specifications for *X*. *X_CONTRACT* must extend class *X* and use the same naming conventions and notations developed earlier in this paper to specify pre- and postconditions, exception handlers or class invariants for the methods in class *X*.

If the implementation class *X* also specifies a precondition for the same method, that precondition is *logical-AND*'ed with the one in *X_CONTRACT* during instrumentation. Similarly, postconditions, and class invariants are also combined using *logical-AND*. Exception handlers in the contract class override the ones inherited from *X*.

The ability to write separate contract classes is useful when specifying contracts for legacy or third party classes, and when modifying existing source code is not possible or viable. It can be used as a technique for debugging and testing system or third-party libraries.

5.3.6 Contract Specifications for Interfaces

Separate contract classes also allow contracts to be added to interfaces. For example, consider the *interface IX* and the class *C* which implements this interface. The class *IX_CONTRACT* contains the pre- and postconditions for the methods in *IX*. Methods defined in the contract class are used to instrument the class "implementing" the interface.

```
interface IX
{
   int foo(<args>);
```

```
}

class IX_CONTRACT
{
   protected boolean foo_PreCondition(<args>) { ... }
   protected boolean foo_PostCondition(<args>){ ... }
}
```

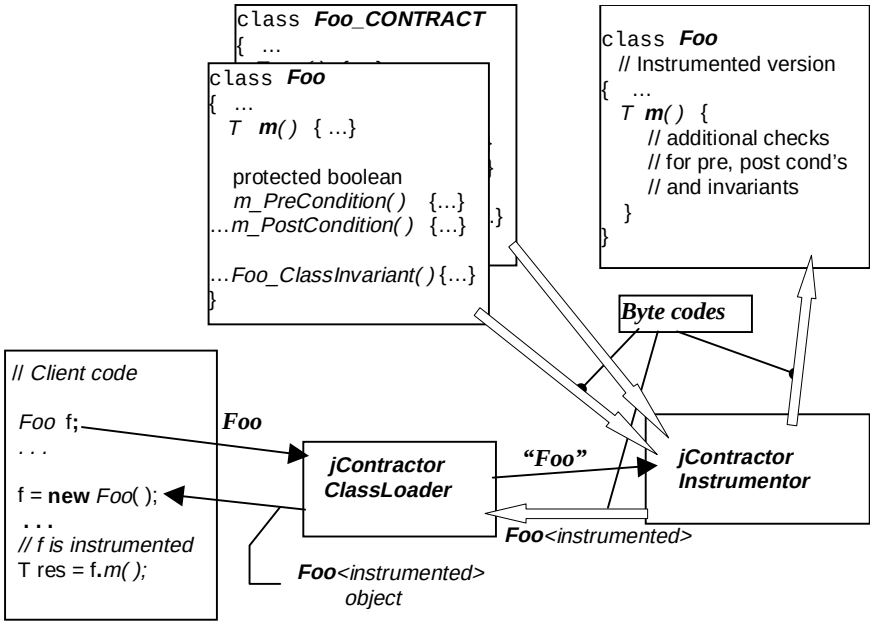
Contracts for interface classes can only include pre- and postconditions, and can only express constraints using expressions involving the method's arguments or interface method calls, without any references to a particular object state. If the implementation class also specifies a precondition for the same method, the conditions are *logical-AND*'ed during instrumentation. Similarly, postconditions are also combined using *logical-AND*.

5.4 Design and Implementation of *jContractor*

The *jContractor* package uses *Java Reflection* to detect Design By Contract patterns during object instantiation or class loading. Classes containing contract patterns are instrumented on the fly using the *jContractor* library. We begin by explaining how instrumentation of a class is done using the two different mechanisms explained in section 2.1. The rest of this section explains the details of the instrumentation algorithm.

The primary instrumentation technique uses the *jContractor class loader* to transparently instrument classes during class loading. The scenario depicted in Figure 5.3 illustrates how the *jContractor Class Loader* obtains instrumented class bytecodes from the *jContractor instrumentor* while loading class `Foo`. The *jContractor class loader* is engaged when launching the Java application. The instrumentor is passed the name of the class by the class loader and in return it searches the compiled class, `Foo`, for *jContractor* contract patterns. If the class contains contract methods, the instrumentor makes a copy of the class bytecodes, modifying the public methods with wrapper code to check contract violations, and returns the modified bytecodes to the class loader. Otherwise, it returns the original class without any modification. The object instantiated from the instrumented class is shown as the `Foo<Instrumented>` object in the diagram, to highlight the fact that it is instrumented, but syntactically it is a `Foo` object.

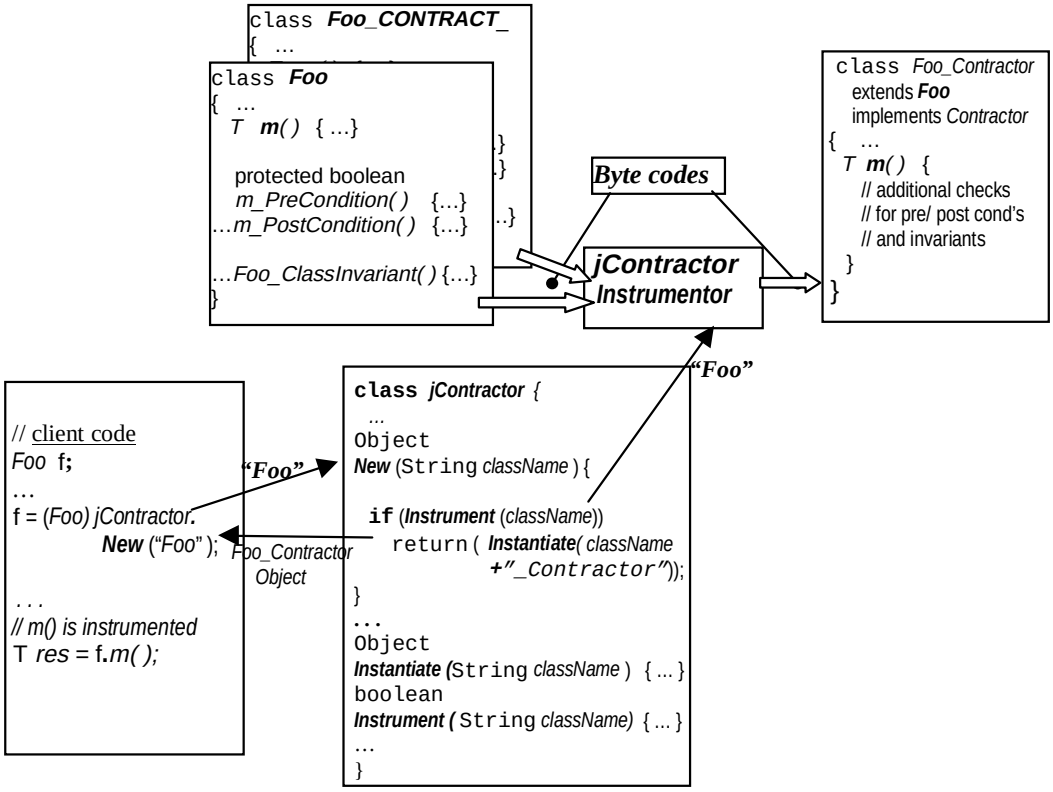
Figure 5.3: *jContractor* Class Loader based Instrumentation



If the command line argument for *jContractor* is not present when starting up the application, the user's own (or the default) class loader is used, which effectively turns off the *jContractor* instrumentation. Since contract methods are separate from the public methods, the program's behavior remains exactly the same except for the runtime checking of contract violations. This is the preferred technique since the client's code is essentially unchanged and all that the supplier has to do is to add the *jContractor* contract methods to the class.

The alternative technique is a factory style object instantiation using the *jContractor* library's New method. New takes a class name as argument and returns an instrumented object conforming to the type of requested class. Using this approach the client explicitly instructs *jContractor* to instrument a class and return an instrumented instance. The factory approach does not require engaging the *jContractor* class loader and is safe to use with any pure-Java class loader. The example in illustrates the factory style instrumentation and instantiation using the class `Foo`. The client invokes *jContractor*.`New`() with the name of the class, "`Foo`". The New method uses the *jContractor* Instrumentor to create a subclass of `Foo`, with the name, `Foo_Contractor` which now contains the instrumented version of `Foo`. New instantiates and returns a `Foo_Contractor` object to the client. When the client invokes methods of the returned object as a `Foo` object, it calls the instrumented methods in `Foo_Contractor` due to the polymorphic assignment and dynamic binding.

Figure 5.4: *jContractor* Factory Style Instrumentation and Instantiation

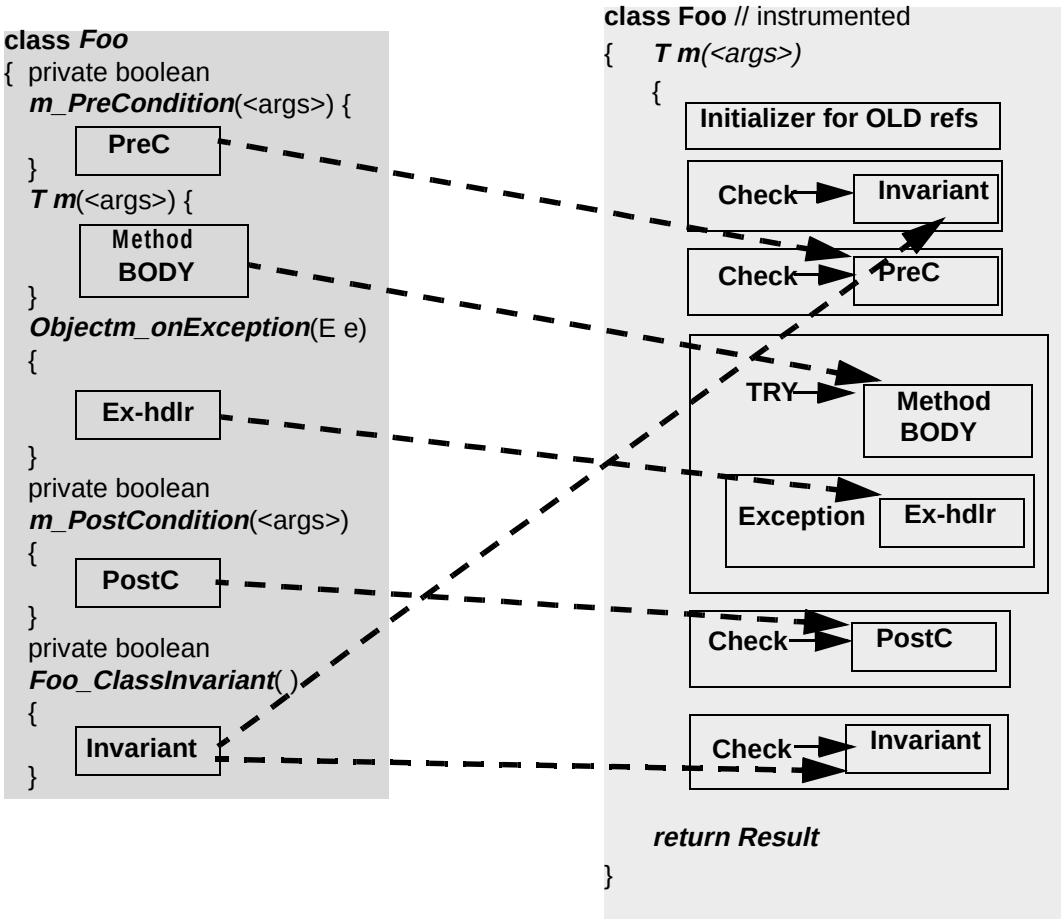


The remainder of this section contains details of the instrumentation algorithm for individual *jContractor* constructs.

5.4.1 Method Instrumentation

jContractor instruments contractor objects using a simple code rewriting technique. Figure 5.5 illustrates the high-level view of how to map code segments from original class methods into the target instrumented version. *jContractor*'s key instrumentation policy is to inline the contract code for each method within the target method's body, to avoid any extra function call. Two basic transformations are applied to the original method's body. First, `return` statements are replaced by an assignment statement – storing the result in a method-scoped temporary – followed by a labeled `break`, to exit out of the method body. Second, references to "old" values, using the `OLD` object reference are replaced by a single variable – this is explained in more detail in Section 5.4.3.

Figure 5.5: jContractor Instrumentation Overview



A check wrapper checks the boolean result computed by the wrapped block and throws an exception if the result is `false`. A *TRY wrapper* executes the wrapped code inside a try-catch block, and associates each exception handler that the contract specifies with a catch phrase inside an exception wrapper. *Exception wrappers* are simple code blocks that are inserted inside the catch clause of a try-catch block with the matching `Exception` type. Typically, exception handlers re-throw the exception, which causes the instrumented method to terminate with the thrown exception. It is possible, however, for the exception handler to recover from the exception condition and generate a result. Figure 5.5 illustrates the overview of these code wrapping transformations.

5.4.2 Instrumentation Example

In this section we show examples of a conceptual source-code level instrumentation and then present a concrete byte-code level instrumentation based on current jContractor implementation. Figure 5.6 illustrates a Java *source code-equivalent* of the Dictionary Class (Figure 5.1 on page 88) *after* it is instrumented. Note that jContractor instrumentation uses only the byte-codes in the Java class files, and generates instrumented byte-codes in Java class file format -- no source code (such as the one shown in Figure 5.6) is ever generated, it is simply used here to illustrate the transformation logic.

Figure 5.6: Factory Instrumented Dictionary Class

```
class Dictionary_Contractor  extends Dictionary …{
  …
  Object put(Object x, String key)
  {
      Object   $put_$Result;
      boolean  $put_PreCondition,
               $put_PostCondition,
               $ClassInvariant;
      int      $OLD_$count = this.count;

      $put_PreCondition = (  (count <= capacity)
                          && (! key.length()==0) );

      if (!$put_PreCondition) {
            throw new PreConditionException();
      }
      $ClassInvariant = (count >= 0);
      if (!$ClassInvariant) {
            throw new ClassInvariantException();
      }
      try {
            $put_$Result = putBody();
      }
      catch (Exception e) {      // put_OnException
            count = $OLD_$count; // restore(count)
            throw e;
      }
      $put_PostCondition =((has(x)) && (item (key) == x) &&
                           (count == $OLD_$count + 1));
      if (!$put_PostCondition) {
            throw new PostConditionException();
      }
      $ClassInvariant = (count >= 0);
      if (!$ClassInvariant) {
            throw new ClassInvariantException();
      }
      return $put_$Result;
    }
  }
```

For a complete bytecode instrumention example see the simplified Dictionary implementation, shown in Figure 5.7. *jContractor* instruments the class put method to perform class invariant and post-condition checks. A *JavaClass* bytecode listing for the instrumented *put* method is shown in Figure 5.8.

Figure 5.7: Simple Dictionary Class

```
public class dict
{
    protected dict     OLD;
    protected int      count = 0,
                       capacity = loopCount;
    Hashtable ht = new Hashtable(capacity);

    public Object put(Object x, String key)
    {
        int i=0;
        try {
            count++;
            // insert <x,key> pair into the hashtable
            return    ht.put(x,key);
        } catch (Exception e) {
            System.out.println(e);
        }
        return "Error in put("+x+", "+key+")";
    }

    protected boolean put_PostCondition(Object x, String key)
    {
        if (!RESULT.compare(null)) {
            System.out.println("Result not NULL: <"+x+">");
            return false;
        }
        return (  (has (x))    && (item (x) == key)
                               && (count == OLD.count+1) );
    }

    protected boolean _ClassInvariant()
    {
        return (count >= 0 && capacity >= count);
    }

    private boolean has (Object x) {
        if (ht.get(x) != null)
            return true;
        else
            return false;
    }

    private Object item (Object  x) {
        return ht.get(x);
    }
}
```

5.4.3 Instrumentation of OLD References

jContractor takes the following actions for each unique *OLD-expression* inside a method's postcondition or exception handler code. Say the method's name is *m()* and the expression is *OLD.attr*, and *attr* has type T, then *jContractor* incorporates the equivalent of the following code while rewriting *m():*

```
T  $OLD_$attr = this.attr;
```

The effect of this code is to allocate a temporary, *$OLD_$attr*, and record the value of the expression, *attr*, when the method code is entered. The code rewriting logic then replaces all occurrences of *OLD.attr* inside the contract code with the temporary variable *$OLD_$attr* whose value has been initialized once at the beginning of the method's execution.

5.4.4 Instrumentation of RESULT References

jContractor allows the following syntax expression inside a method's postcondition method to refer to the result of the current computation that led to its evaluation:

```
RESULT.Compare(expression)
```

RESULT is provided as part of the *jContractor* library package, to facilitate this syntax expression. It exports a single *static boolean* method, *Compare()*, taking a single argument with one variant for each built-in Java primitive type and one variant for the *Object* type. These methods never get invoked in reality, and the sole purpose of having them (like the *OLD* declarations discussed earlier) is to allow the Java compiler to legally accept the syntax, and then rely on the instrumentation logic to supply the right execution semantics.

During instrumentation, for each method declaration, *T m()*, a temporary variable *$m_$Result* is internally declared with the same type, *T*, and used to store the result of the current computation. Then the postcondition expression shown above is rewritten as:

```
($m_$Result == (T)(expression))
```

Figure 5.8: Instrumented `put` Method

public Object put(Object, String)
Code(max_stack = 100, max_locals = 100, code_length = 318)

```
0:   aload_0
1:   getfield        dict.count I (42)                    Class Invariant Wrapping
4:   istore          %5
6:   aload_0
7:   invokevirtual   dict._ClassInvariant ()Z (35)
10:  ifne            #59
13:  getstatic       java.lang.System.out Ljava/io/PrintStream; (50)
16:  new             <java.lang.StringBuffer> (25)
19:  dup
20:  ldc             "\njContractor Exception: Class Invariant VIOLATION: \nwhen: put(" (200)
22:  invokespecial   java.lang.StringBuffer.<init> (Ljava/lang/String;)V (33)
25:  aload_1
26:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/Object;)Ljava/lang/StringBuffer; (38)
29:  ldc             ", " (9)
31:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
34:  aload_2
35:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/Object;)Ljava/lang/StringBuffer; (38)
38:  ldc             ", " (9)
40:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
43:  invokevirtual   java.lang.StringBuffer.toString ()Ljava/lang/String; (57)
46:  invokevirtual   java.io.PrintStream.println (Ljava/lang/String;)V (53)
49:  new             <java.lang.RuntimeException> (182)
52:  dup
53:  ldc
     "Class Invariant VIOLATION: put(Ljava/lang/Object;Ljava/lang/String;)Ljava/lang/Object;"
55:  invokespecial   java.lang.RuntimeException.<init> (Ljava/lang/String;)V (183)
58:  athrow
59:  nop
60:  aload_0
61:  dup
62:  getfield        dict.count I (42)
65:  iconst_1
66:  iadd
67:  putfield        dict.count I (42)
70:  aload_0
71:  getfield        dict.ht Ljava/util/Hashtable; (46)
74:  aload_1
75:  aload_2
76:  invokevirtual   java.util.Hashtable.put (Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
(54)
79:  goto            #123
82:  astore_3
83:  getstatic       java.lang.System.out Ljava/io/PrintStream; (50)
86:  aload_3
87:  invokevirtual   java.io.PrintStream.println (Ljava/lang/Object;)V (52)
90:  new             <java.lang.StringBuffer> (25)
93:  dup
94:  ldc             "Error in put(" (12)
```

```
96:  invokespecial  java.lang.StringBuffer.<init> (Ljava/lang/String;)V (33)
99:  aload_1
100: invokevirtual  java.lang.StringBuffer.append (Ljava/lang/Object;)Ljava/lang/StringBuffer; (38)
103: ldc            ", " (9)
105: invokevirtual  java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
108: aload_2
109: invokevirtual  java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
112: ldc            ")" (6)
114: invokevirtual  java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
117: invokevirtual  java.lang.StringBuffer.toString ()Ljava/lang/String; (57)
120: goto           #123
123: nop
124: dup
125: astore         %4
127: aconst_null
128: aload          %4
130: if_acmpeq      #137
133: iconst_0
134: goto           #138
137: iconst_1
138: nop
139: nop
140: ifne           #174
143: getstatic      java.lang.System.out Ljava/io/PrintStream; (50)
146: new            <java.lang.StringBuffer> (25)
149: dup
150: ldc            "Result not NULL: <" (14)
152: invokespecial  java.lang.StringBuffer.<init> (Ljava/lang/String;)V (33)
155: aload_1
156: invokevirtual  java.lang.StringBuffer.append (Ljava/lang/Object;)Ljava/lang/StringBuffer; (38)
159: ldc            ">" (10)
161: invokevirtual  java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
164: invokevirtual  java.lang.StringBuffer.toString ()Ljava/lang/String; (57)
167: invokevirtual  java.io.PrintStream.println (Ljava/lang/String;)V (53)
170: iconst_0
171: goto           #210
174: aload_0
175: aload_1
176: invokespecial  dict.has (Ljava/lang/Object;)Z (45)
179: ifeq           #202
182: aload_0
183: aload_1
184: invokespecial  dict.item (Ljava/lang/Object;)Ljava/lang/Object; (47)
187: aload_2
188: if_acmpne      #202
191: aload_0
192: getfield       dict.count I (42)
195: iload          %5
197: iconst_1
198: iadd
199: if_icmpeq      #206
202: iconst_0
203: goto           #210
206: iconst_1
```

```
207:  goto            #210
210:  nop
211:  ifne            #260
214:  getstatic       java.lang.System.out Ljava/io/PrintStream; (50)
217:  new             <java.lang.StringBuffer> (25)
220:  dup
221:  ldc             "\njContractor Exception: POSTCONDITION EXCEPTION: \nwhen: put(" (204)
223:  invokespecial   java.lang.StringBuffer.<init> (Ljava/lang/String;)V (33)
226:  aload_1
227:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/Object;)Ljava/lang/StringBuffer; (38)
230:  ldc             ", " (9)
232:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
235:  aload_2
236:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/Object;)Ljava/lang/StringBuffer; (38)
239:  ldc             ", " (9)
241:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
244:  invokevirtual   java.lang.StringBuffer.toString ()Ljava/lang/String; (57)
247:  invokevirtual   java.io.PrintStream.println (Ljava/lang/String;)V (53)
250:  new             <java.lang.RuntimeException> (182)
253:  dup
254:  ldc
      "POSTCONDITION EXCEPTION: put(Ljava/lang/Object;Ljava/lang/String;)Ljava/lang/Object;"
256:  invokespecial   java.lang.RuntimeException.<init> (Ljava/lang/String;)V (183)
259:  athrow
260:  nop
261:  aload_0
262:  invokevirtual   dict._ClassInvariant ()Z (35)
265:  ifne            #314
268:  getstatic       java.lang.System.out Ljava/io/PrintStream; (50)
271:  new             <java.lang.StringBuffer> (25)
274:  dup
275:  ldc             "\njContractor Exception: ClassInvariant VIOLATION:\nwhen: put(" (208)
277:  invokespecial   java.lang.StringBuffer.<init> (Ljava/lang/String;)V (33)
280:  aload_1
281:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/Object;)Ljava/lang/StringBuffer; (38)
284:  ldc             ", " (9)
286:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
289:  aload_2
290:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/Object;)Ljava/lang/StringBuffer; (38)
293:  ldc             ", " (9)
295:  invokevirtual   java.lang.StringBuffer.append (Ljava/lang/String;)Ljava/lang/StringBuffer; (39)
298:  invokevirtual   java.lang.StringBuffer.toString ()Ljava/lang/String; (57)
301:  invokevirtual   java.io.PrintStream.println (Ljava/lang/String;)V (53)
304:  new             <java.lang.RuntimeException> (182)
307:  dup
308:  ldc
      "ClassInvariant VIOLATION:put(Ljava/lang/Object;Ljava/lang/String;)Ljava/lang/Object;"
310:  invokespecial   java.lang.RuntimeException.<init> (Ljava/lang/String;)V (183)
313:  athrow
314:  nop
315:  aload           %4
317:  areturn
```

5.4.5 Use of Reflection

Each class and interface in a Java program corresponds to a translation unit with a machine and platform independent representation as specified by the Java Virtual Machine `class` file format. Each class file contains JVM instructions (bytecodes) and a rich set of meta-level information. During the instrumentation process *jContractor* parses and analyzes the meta-information encoded in the class byte-codes in order to discover the *jContractor* contract patterns. When the class contains or inherits contracts, *jContractor* instrumentor modifies the class bytecodes on the fly and then passes it to the class loader. The class name and its inheritance hierarchy; the method names, signatures and code for each class method; the attribute names found and referenced in class methods constitute the necessary and available meta information found in the standard Java class files. The presence of this meta information in standard Java class byte codes and the capability to do dynamic class loading are essential to our way building a pure-library based *jContractor* implementation.

Core Java classes include the `java.lang.reflect` package which provides reflection capabilities that could be used for parsing the class information, but using this package would require prior loading of the class files into the JVM. Since *jContractor* completes its instrumentation *before* loading the class files, it cannot use core reflection classes directly and instead uses its own class file parser.

5.5 Implementation and Performance Issues

jContractor is implemented entirely in Java and can be used with any JDK 1.1 or later. It has been tested with the JDK 1.1 and 1.2 on Solaris, and Windows9x/NT and jBuilder on Windows9x/NT. Current implementation uses BCEL: The Byte Code Engineering Library (formerly known as JavaClass) libraries during instrumentation, which must be installed separately. BCEL is a freely available open source project under the dual GNU Lesser General Public License (LGPL) and the Mozilla Public License (MPL).

Our performance studies show that jContractor instrumented class methods run more efficiently than explicitly hand instrumented version of the methods performing functionally equivalent run-time contract monitoring. We compared the performance of the uninstrumented and instrumented versions of the Dictionary “put” operation shown in

Figure 5.7 and Figure 5.8, respectively as well as the manual Design By Contract variant, *put_HandCodedDBC* shown in Figure 5.9

#operations	No DBC	Manual DBC	jContractor DBC
10,000	3,265 +/- 25	4,010 +/- 50	3,920 +/- 20
50,000	9,100 +/- 30	22,650 +/- 20	22,200 +/- 20

Table 5.1: Execution Times for Dictionary "put" Operations (in ms)

The cost of checking for contract viloations using hand coded checks (Manual DBC) is 18.6% - 22.8% of actual "put" processing time. Using *jContractor* reduces the contract checking overhead by about 14% to 16.2%-20%. This example shows one of the performance benefits of jContractor in addition to the convenience of on-the-fly instrumentation. jContractor instrumentor optimizes contact enforcement code by inlining and auto-allocating needed temporaries. Cost of runtime DBC checks appears high using the Dictionary example, but the cost depends entirely on the actual pre-, postcondition expressions. Evaluating Dictionary put post condition, for example, checks whether the item is "in" the dictionary (i.e. 1 lookup), and also checks whether the key-object association is correct which is another lookup. While these checks are costly operations with respect to the basic put operation, the ability to effortlessly turn contract checking on and off at runtime highlights another benefit of the jContractor system during system debugging and testing.

Figure 5.9: Hand Coded Design by Contract Version of Dictionary Class

```
public class dict
{ ...
    protected boolean put_PostCondition2(Object x, String key,
                  Object RESULT, int oldCount)
    {
        if (RESULT != null) {
            System.out.println("Result not NULL: <"+x+">");
            return false;
        }
        return ( has(x) && (item (x) == key)
                 && (count == oldCount+1) );
    }

    public Object put_HandCodedDBC(Object x, String key)
    {
        Object result = null;
        int oldCount = count;

        if (_ClassInvariant()) {
```

```
            result = put(x, key);

            if (_ClassInvariant()) {
             if (!put_PostCondition2(x, key, result, oldCount))
             {
                 System.out.println("PostCondition violation");

             }
            } else {
             System.out.println("\nClass Invariant violation");
            }
        } else {
             System.out.println("\nClass Invariant violation");
        }
        return result;
    } ...
```

5.6 Summary

We have introduced *jContractor,* a purely library-based solution to write Design By Contract specifications and to enforce them at runtime using Java. The *jContractor* library and naming conventions can be used to specify the following Design By Contract constructs: pre- and postconditions, class invariants, exception handlers, and *old* references. Programmers can write contracts using standard Java syntax and an intuitive naming convention. Contracts are specified in the form of protected methods in a class definition where the method names and signatures constitute the *jContractor* naming conventions. *jContractor* checks for these patterns in class definitions and rewrites those classes on the fly by instrumenting their methods to check contract violations at runtime.

The greatest advantage of *jContractor* over existing approaches is the ease of deployment. Since *jContractor* is purely library-based, it does not require any special tools such as modified compilers, runtime systems, pre-processors or JVMs, and works with any pure Java implementation.

The *jContractor* library instruments the classes that contain *contract patterns* during class loading or object instantiation. Programmers enable the run-time enforcement of contracts by using a command line switch at start-up, which installs the *jContractor* instrumenting class loader. *jContractor* object factory provides an alternative mechanism that does not require engaging the *jContractor* ClassLoader to instantiate instru-

mented objects. Clients can instantiate objects directly from the *jContractor* factory, which can use any standard class loader and does not require a command line switch. Either way, clients can use exactly the same syntax for invoking methods or passing object references regardless of whether contracts are present or not. Contract violations result in the method throwing proper runtime exceptions when instrumented object instances are used.

We also describe a novel instrumentation technique that allows accessing the *old* values of variables when writing postconditions and exception handling methods. For example, `OLD.count` returns the value of the attribute `count` at method entry. The instrumentation arranges for the attribute values or expressions accessed through the `OLD` reference to be recorded at method entry and replaces the `OLD` expressions with automatically allocated unique identifiers to access the recorded values.

Chapter 6

Comparisons with Other Approaches and Related Work

6.1 Library Based Language Extensions

Many statically typed general purpose programming languages use libraries to extend the language with new features and functionality. For example, C and C++ rely on the presence of externally linked system and run-time support libraries and standard application programming interfaces (APIs) to support user-level operating system (O/S) services and abstractions such as streams, files, file-systems, dynamic memory allocation, processes, threads, interprocess communication, networking, etc., For example, standard I/O libraries extend C language in such a way that most C programmers use and think of these calls (printf, scanf, etc.,) as if they were part of the C language definition. Platform specific frameworks such as MFC, COM, OLE/ActiveX provide Windows C/C++ programmers support for designing graphical user interfaces(GUI) and a standardized component and communication model. AWT/JFC/Swing provide a platform independent Java framework for designing GUIs. OMG's CORBA and Microsoft's DCOM provide language and platform independent frameworks for developing distributed applications. Java has direct language support for threads and implicit dynamic memory allocation, yet, relies on a rich set of libraries organized as core language packages to support windowing abstractions, networking, imaging, etc.,

Many interpreted or scripting languages, such as Common Lisp, Tcl, and Perl, also use a library based extension approach. Common Lisp extension CLOS provides object orientation. Tcl extension, Tk, provides graphical and windowing support; Incr-Tcl extension provides object-orientation. Perl uses packages to provide seamless access to just about any operating system level function.

Standardized libraries and frameworks typically offer a rich set of low level APIs to offer greater control and flexibility, but tend to be error-prone as they assign significant responsibility to a programmer to keep track of state, consistency, and correct usage and require steep learning curves and a fair amount of commitment to a platform even to perform fairly simple or common tasks.

Complex systems require higher levels of programming abstractions than those typically provided by standardized system level libraries and frameworks. The ideal way to support high-level abstractions is to have direct language support. Language level support can provide simplicity, safety and performance efficiency. Traditionally one of the following approaches have been used to introduce new high-level language abstractions:

1. Design a brand new language with new features and abstractions.
2. Extend an existing language with new and non-standard extensions using customized compilers, preprocessors, or run-time environments.

Both of these approaches offer custom and dedicated solutions but this in itself may cause problems. It may be impossible or impractical for programmers to migrate to a new language and/or development or runtime environment. Maintenance, reliability, security and future support issues might also hinder acceptance. In this dissertation we have described techniques using an alternative approach:

3. Extend an existing object-oriented language using a purely library based approach to provide new high-level abstractions.

We introduced new language extensions by designing class libraries and a set of naming and programming conventions for *concurrency, distributed computing* and *design by contract* for the object oriented languages, Eiffel and Java. In subsequent sections of this chapter we present a comparison of each system with other related work in literature.

6.2 Introducing Concurrency to Sequential Object-Oriented Languages

6.2.1 Eiffel Concurrency Extensions

There are several proposals for concurrent programming with Eiffel: Eiffel// [18], Meyer's proposal [54], CEiffel [47], and Colin and Geib's Concurrency Classes [25]. Our mechanism is most similar to Eiffel//. Primary differences are: our concurrency

mechanism does not modify the Eiffel compiler, provides full support for coping with local delays, and allows post-actions after returning results in active objects.

Meyer's proposed extension to Eiffel also takes a process based approach that uses the assertion mechanism of Eiffel for synchronization. Colin and Geib's Concurrency Classes do not modify the Eiffel language, similar to our approach, however they use a single address space with light-weight threads, and have a very different approach in the way they view an activity (or thread) independent from objects.

CEiffel is based on expressing concurrency properties attached to classes, or methods in the form of annotations. CEiffel also provides strong support for reusability, including a novel mechanism which allows concurrent objects to be reused as sequential objects by a compiler switch which ignores the concurrency annotations. Both our mechanism and Eiffel// differ from CEiffel and Colin and Geib's model in not supporting multiple-threads in active objects.

6.2.2 Related Work Introducing Concurrency to Object-Oriented Languages

Three distinct approaches exist for introducing concurrency to object-oriented systems:

- *Design a new* concurrent object-oriented language.
- *Extend* an existing object-oriented language.
- *Design a Concurrency Library.* Use an existing object-oriented language and provide concurrency abstractions through external libraries.

Many references and comparative discussions about general concurrent object-oriented languages can be found in [3], and [61]. Most of the earlier systems fall into the first approach: design a new object-oriented language with built-in concurrency. Some examples are: Hybrid [57], POOL [5] (and its variants), SR [6], ABCL/1 [73] and Java [31]. These *new* languages provide powerful concurrency abstractions and general-purpose programming capabilities. Most of the *extensions* introduce concurrency to their respective languages using some combination of the following techniques:

- *inheritance* from special concurrency classes that the modified compiler recognizes — e.g. Eiffel// [19] , PRESTO [8] ;
- special keywords, modifiers or preprocessing techniques to modify or extend the language syntax and semantics — e.g. μC++ [14] , CEiffel [48];

- extension to the syntax and semantics of the language to support a general concurrency paradigm such as the Actor model [2] — e.g. ACT++ [37], Actalk [11].

Our concurrency mechanism, and Colin and Geib's [25] Eiffel Classes, fall in the *library approach*. The approach of introducing concurrency via a class definition of Process is also used in the *Choices* operating system [16] where they use the C++ language. The library based solutions are attractive since they do not replace the existing software development platform. However, the sequential execution semantics of the host language may impose restrictions on providing type-safety and intra-object concurrency.

The order in which we have presented the three approaches follows a chronological order. Most of the earlier concurrent object-oriented languages were *new* languages. As object-oriented thinking matured and sequential object-oriented languages started to become popular, numerous proposals were made to *extend* these sequential languages for concurrent programming. The *library approach* is most recent, and has been influenced by most of the earlier work on concurrency. The latest trends for object-based concurrency emphasize and address issues such as *reusability* and compatibility with object-oriented software engineering techniques as fundamental requirements [48], [58]. A major focus of our work has been to address the issues of reusability and compatibility with the object-oriented paradigm.

Our concurrency mechanism fits into the non-uniform, and non-orthogonal category of Papathomas' classification of object-oriented concurrency approaches [61], since we allow both active and non-active objects to co-exist and have single threaded active objects. Some other languages that belong to this classification are POOL-T [5], ABCL/1 [73], and Eiffel// [18]. The basic mechanism in our methodology to access results of asynchronous calls, i.e. using the *call_id*s returned by `remoteInvoke`, are similar to the ConcurrentSmalltalk's CBoxes [72] and ABCL/1's *future type messages*[73].

6.3 Distributed Computing and Active Object Extensions

Distributed computing uses networking and communication to enhance a local computation by potentially distributing portions of the computation among different hosts; or to collaborate with other executing programs towards accomplishing a global task. What makes distributed programming challenging is the semantic gap between programming language semantics involving local and remote computations and the communication abstractions. Distributed programs require higher levels of abstractions then TCP/IP or other basic networking protocols.

Most distributed computing systems use widely available and standardized communication and distribution protocols: lower level mechanisms such as RPC[9] DCE[59] or object-based distribution protocols such as CORBA[60], RMI[70], ILU[35], HORB[67] or DCOM[10]. All of these systems add a distribution layer on top of a general purpose programming language system: an intermediate *interface definition language* (IDL) and an IDL-to-application language tool is provided to support creation of remote stubs/proxies, and for registering or locating remote interface implementation objects/servers. Numerous experimental and research prototypes exist: distributed operating systems such as Sprite[27], Inferno[49] support distributed scope and access primitives as operating system functions; languages such as Emerald[36], Telescript[69], Agent-Tcl[32] are specifically designed for writing distributed applications.

Active Remote Method Invocation system, *Active-RMI*, is a set of class libraries, tools, and a design method which collectively provides high-level abstractions for building distributed Java applications based on the notion of *active remote objects*. *Active-RMI* is a natural extension of our work involving introducing concurrency extensions to Eiffel language using class libraries[41]. *Active-RMI* is implemented as an extension to the *Java RMI* system without extending the Java Language[31] or its run-time environment. *Active-RMI* provides a very high level of programming for writing complex object distribution and synchronization applications entirely in Java. Three key abstractions provided by the *Active-RMI* model are: the *active remote objects* with user level *scheduling*; *asynchronous method invocation* with data-driven, non-blocking synchronization; and transparent *remote object creation*.

In this section we discuss languages or language extensions which provide programming abstractions and services for building distributed, parallel and mobile applications.

6.3.1 Java Based Related Work

Java appears to be an ideal language for distributed computing with standard built-in features: platform independence, multi-threading and synchronization constructs, remote method invocation (RMI) system, dynamic and networked class loading. However, the abundance of distributed computing extensions for Java discussed in this section indicate the need for additional and higher-level abstractions. Also, Brose et.al. show that [12,13] that Java’s method-calling semantics, pass-by-value, lead to unacceptable latencies when accessing arrays, and even class instances. Java, even with RMI, does not exhibit access transparency, or identical method calling syntax and semantics for both local and remote objects.

ProActive PDC [20] is the most similar approach to Active-RMI. ProActive PDC (formerly known as Java//) is a library for Parallel, Distributed, and Concurrent programming in Java. It is entirely API-based, needing no special compiler or JVM. The idea is to run the same program as a sequential application, a multithreaded single-node application, and a distributed application. The programmer provides hints to the system through its API, and also uses the API to get implicit futures (using wait-by-necessity), continuations (a transparent delegation mechanism), and active objects. Running a program on the different kinds of platforms supported by ProActive PDC is achieved through object composition. The user defines a sequential object, which the system then composes with a proxy and a so-called body. The proxy turns local calls into messages, which are decoded by the body. Futures and continuations are provided by creating specialized subclasses of user objects which contain the appropriate code.

Sumatra is an extension of Java that supports resource-aware mobile programs. Sumatra has not modified Java language syntax and can run all legal Java programs without modification. All added functionality is provided by extending the Java class library and by modifying the Java interpreter, without affecting the virtual machine interface. Sumatra adds four programming abstractions to Java: object-groups, execution-engines, resource-monitoring and asynchronous events. Computation begins at a single site and spreads to other sites in three ways: (1) remote method instantiation, (2) remote thread creation, and (3) thread migration. Active-RMI directly supports the first two mechanisms. Remote method instantiation is essentially an RMI style synchronous remote method invocation. Remote thread creation differs from remote method instantiation in that the new thread is independent of the creating thread; the creating thread continues execution once the creation is complete. This is very similar to Active-RMI aynchronous remote method call, except that remote execution is performed within a previously created active object context. Active-RMI does not directly support thread migration which involves stopping the execution of the calling thread at the current site, transferring its state to another site and resuming execution at that site. Sumatra uses a master daemon which runs on all machines that allow creation of execution-engines and listens on a well-known socket. When a execution-engine creation request is received, it creates a new interpreter process which attempts to bind to specified socket. This daemon is very similar to the Active-RMI's *armiServer* utility.

The *Kan* system [34] extends Java language with asynchronous method calls (used for expressing concurrency), guards (used for expressing dataflow and synchronization constraints), and nested atomic transactions (used for expressing atomicity). Kan hides distribution, replication, migration, and faults from the programmer when writing parallel and distributed applications. Kan extends Java's syntax and relies on a specialized compiler which produces standard Java bytecodes, containing calls into the Kan runtime system. That system itself is written in pure Java, so the system and user

applications run on any standard Java Virtual Machine, using Java sockets for communication.

cJVM[7] distributes the JVM onto homogeneous clusters of computers on a high-speed network, and runs unmodified Java programs on each JVM. It was designed to support servers, by distributing the server load across the cluster. Hence, it performs best on applications with a large number of independently executing threads.

Do! [44] uses a modified Java compiler to automatically generate distributed code from multithreaded program source code. The Java language is not extended, but the user is given an API for providing hints to the compiler about appropriate mappings of threads and objects to distributed system nodes. The generated code uses standard Java RMI for communication. It uses a runtime library that supports the creation and manipulation of remote objects.

JavaParty [62] is an extension to Java. It adds transparency to remote objects, bypassing the complexity of RMI. It also transparently migrates objects for greater availability. JavaParty compiles down to standard Java bytecodes, allowing JavaParty programs to run on any JVM. It also supports easy integration of standard Java class files, compiled externally to the JavaParty system. The JavaParty project has produced improved Serialization [63] and RMI [56] implementations. However, their solutions are not portable across JVMs.

Javelin [24] is a prototype infrastructure for Internet-based parallel computing using Java. In Javelin model there are three kinds of participating entities: brokers, clients and hosts. Javelin allows machines connected to the Internet to make a portion of their idle resources available to remote clients and at other times utilize resources from other machines when more computational power is needed. Javelin system provides a parallel programming language layer which offers support for the SPMD programming model and Linda Tuple Space from within an applet. Javelin describes simple programming models that enable programmers to express many parallel programming constructs in their client applet code. These programming models are realized by executing specialized servlets on the broker. Standard Java language and syntax is used.

Nile [66] project provides a self-managing, fault-tolerant, heterogeneous system composed of hundreds of commodity workstations, with access to a distributed database whose size is on the order of hundreds of terabytes. It is written in Java for heterogeneity. CORBA is used as a data management layer. It is structured to run embarrassingly parallel applications; i.e., those with independent parallel subtasks, such as web indexers. The database itself is widely distributed, with replication providing some degree of fault tolerance. The failure of a job is automatically detected, and the job is restarted if the failure can be repaired or worked around. The basic operation of Nile is to divide

the application into subparts and distribute those subparts to the constituent computing nodes, then collect and collate the results. If a subpart fails, recovery consists of assigning the subpart to a new computing node.

Parallel Java [38] is an extension to the Java language to support parallel constructs. It is based on earlier work on a C++ extension, called Charm++. The parallel extensions provide for the creation of remote objects via proxies, with automatic load balancing. Objects with a port on every node are called object groups, and allow for easy expression of algorithms requiring global coordination, such as barriers. Parallel Java is part of a larger effort, named Converse, which is aimed at providing multilingual parallel support. That is, Parallel Java programs can interact with parallel libraries written in other languages supported by Converse. Parallel Java uses Java Serialization and Reflection to provide portable means of accessing remote objects.

6.3.2 Other Languages and Systems

Compositional C++ (CC++) [22] is a superset of C++, adding the keywords sync, global, par, parfor, atomic,and spawn. It provides support for explicit parallelization of programs by way of declaring blocks of code as parallel, in either a synchronous or asynchronous manner. Loops can be declared as parallel as well. Synchronization is provided in the form of atomic methods. The C++ language is extend to provide global pointers to CC++ objects. Each main process is itself an object, which allows processes to operate on one another. The consistency guarantee is cache consistency (see Section 6.2.5), a fairly weak guarantee. CC++ is a parallel language able to specify blocks of code as atomic, construct global pointers to objects, and spawn new threads at the language level.

Charm++ [39] is a C++ extension. The basic unit of computation is the chare. A chare can be located on a specific node, or it can be a branch office chare, with local components on every node. A number of common modes of information sharing are supported by means of shared variables. Several types are available, including read-only variables, accumulator variables, monotonic variables, write-once variables, and distributed tables. In general, an object can choose its own method ordering protocol through system calls that indicate the messages which the object is willing to receive. It does so by expressing the method protocol dependencies as a directed acyclic graph. However, there is no intra-object concurrency possible and no values can be returned in response to a message (i.e., concurrent methods all have return type void). The consistency guarantee is broadcast consistency.

COOL [21] is a specialized language which supports concurrent method execution in an object that adheres to the multiple reader-single writer protocol. This is done by specifying a method to be either of type mutex (writer) or non-mutex (reader). The COOL runtime system then ensures this consistency model by using read and write locks. Intra-object concurrency is supported. Condition variables and monitors are used to implement inter-object communication. While these provide synchronization, they are unable to pass values back (that is, concurrent methods cannot return values).

Concert [23] supports distributed objects with an aim towards providing fine-grained parallelism. Object-based concurrency control and encapsulation, and a dynamic concurrency model are provided. The thrust of the project is to use aggressive whole program compilation, interprocedural optimization, and an efficient runtime system which works in concert with the compiler optimizations.

Emerald [36] is a strongly-typed pure object-oriented language. Objects can migrate between nodes. Objects can be declared immutable, which simplifies sharing. There is both inter-object and intra-object concurrency. Objects can be declared as monitors, which simplifies handling intra-object concurrency. Objects with a process (executing in parallel with the monitor) are active; those without a process are passive. All method calls are synchronous; new threads of control arise by creating a new active object. A garbage collector reclaims unreferenced objects.

Obliq [17] is a language based on Modula-3 which supports distributed object-oriented programs through migrating threads. A computation can roam over the network, while maintaining connections between its constituent parts. Objects are local to some computing node, but threads migrate. Hence, a distributed computation consists of the migration of a thread over the necessary set of objects. Obliq is a lexically-scoped untyped interpreted language. It contains a notion of hierarchical spaces; an Obliq computation may involve multiple threads of control within an address space, multiple address spaces on a machine, heterogeneous machines over a local network, and multiple networks over the Internet.

6.4 Extensions for Design by Contract

The idea of associating boolean expressions (assertions) with code as a means to argue the code's correctness can be traced back to Hoare [33] and others who worked in the field of program correctness. The idea of extending an object-oriented language using only libraries and naming conventions appeared in [41]. The notion of compiling assertions into runtime checks first appeared in the *Eiffel* language [52].

Eiffel is an elegant language with built-in language and runtime support for Design By Contract. *Eiffel* integrates preconditions (*require-clause*), postconditions (*ensure-clause*), class invariants, *old* and *rescue/retry* constructs into the definition of methods and classes. *jContractor* is able to provide all of the contract support found in *Eiffel*, with the following differences: *jContractor* supports exception-handling with finer exception resolution – as opposed to a single *rescue* clause; *jContractor* does not support the *retry* construct of *Eiffel*. We believe that if such recovery from an exception condition is possible, it is better to incorporate this handler into the implementation of the method itself, which forestalls throwing the exception at all. *jContractor'* support for *old* supports cloning semantics where references are involved, while Eiffel does not.

Duncan & Hölzle introduced Handshake[28], which allows a programmer to write external contract specifications for Java classes and interfaces without changing the classes themselves. Handshake is implemented as a dynamically linked library and works by intercepting the JVM's file accesses and instrumenting the classes on the fly using a mechanism called binary component adaptation (BCA). BCA is developed for on the fly modification of pre-compiled Java components (class bytecodes) using externally provided specification code containing directives to alter the pre-compiled semantics [42]. The flexibility of the approach allows Handshake to add contracts to classes declared `final`; to system classes; and to interfaces as well as classes. Some of the shortcomings of the approach are that contract specifications are written externally using a special syntax; and that Handshake Library is a non-Java system that has to be ported to and supported on different platforms.

Kramer's *iContract* is a tool designed for specifying and enforcing contracts in Java [43]. Using *iContract*, pre-, postconditions and class invariants can be annotated in the Java source code as "comments" with tags such as: @pre, @post. The *iContract* tool acts as a pre-processor, which translates these assertions and generates modified versions of the Java source code. *iContract* uses its own specification language for expressing the boolean conditions.

Mannion and Philips have proposed an extension to the Java language to support Design By Contract [50], employing *Eiffel*-like keyword and expressions, which become part of a method's signature. Mannion's request that Design By Contract be directly supported in the language standard is reportedly the most popular "non-bug" request in the Java Developer Connection Home Page (bug number 4071460).

Porat and Fertig propose an extension to C++ class declarations to permit specification of pre- and postconditions and invariants using an assertion-like semantics to support *Design By Contract* [64].

Chapter 7

Conclusion

The original research work conducted as part of this dissertation introduced new language extensions for concurrency, distributed computing and design by contract for the object oriented languages, Eiffel and Java. We demonstrated that powerful new abstraction capabilities can be introduced using a purely library based approach and a set of naming and programming conventions without violating the object-oriented principles or compromising other language or safety features.

We developed techniques to build extensible, open object-oriented libraries that use and take advantage of the existing low-level system and vendor libraries, platforms, frameworks to support several new high-level abstractions. Programmers use these abstractions by following our naming and programming conventions. We illustrate the applicability of our approach by building and presenting three original research application systems: *Class CONCURRENCY, Active-RMI* and *jContractor. Class CONCURRENCY* introduces concurrency, active objects, asynchronous calls, data-driven synchronization and scheduling abstractions to Eiffel. *Active-RMI* introduces asynchronous remote method invocation with future-type results; asynchronous result delivery; transparent remote-object creation; active-object semantics; user programmable scheduling and synchronization to Java. *jContractor* introduces *Design by Contract* to Java. Table 1.1, “Overview of Dissertation Research.,” on page 6 lists feature outline of these three systems we have built and the new abstractions we introduced for each system.

The *Class CONCURRENCY* provides concurrency, active objects, asynchronous calls, data-driven synchronization and scheduling abstractions to Eiffel objects as encapsulated and inheritable properties. Objects which inherit from the *Class CONCURRENCY* acquire a separate thread and private state and become active with a programmable scheduler. Active objects’ methods can be called asynchronously using

deep-copy pass-by-value semantics for normal object arguments and reference passing semantics for active object arguments.

Active-RMI extends Java with asynchronous remote method invocation & future-type results; asynchronous result delivery; transparent remote-object creation; active-object semantics; user programmable scheduling and synchronization. *Active-RMI* objects are started as autonomous agent-like objects, with complete, user programmable control over scheduling and execution of incoming requests.

jContractor is a pure Java library which requires no special tools such as modified compilers, modified JVMs, or pre-processors to support Design By Contract constructs: preconditions, postconditions, class invariants, recovery and exception handling. *jContractor* uses an intuitive naming convention, and standard Java syntax. The designer of a class specifies contracts by providing contract methods following *jContractor* naming conventions. *jContractor* uses Reflection to synthesize an instrumented version of a Java class by incorporating code that enforces the present *jContractor* contract specifications. Programmers enable the run-time enforcement of contracts by either engaging the *jContractor* class loader or by explicitly instantiating objects using the *jContractor* object factory. Programmers can use exactly the same syntax for invoking methods and passing object references regardless of whether contracts are present or not.

Our work illustrates advantages of employing libraries to support new abstractions and functionality: the core languages are left in tact allowing programmers to use their standard development tools and environments. Developing the libraries independently as a layer above the native O/S or language services leads to performance efficiency, portability and ease of development and maintenance.

We have identified the following advantages for using the pure library based approach when introducing new high-level programming abstractions to an object-oriented language:

- Libraries provide a more *flexible* and *extendible* solution since they can be tailored to the specific needs and characteristics of the target operating system and hardware by modifications or refinement of the libraries. New abstractions that are hard wired into the language may be impractical or impossible to change. A similar analogy exists in the operating system research. It is more desirable to implement smaller (micro) kernels that move a lot of the traditional kernel abstractions and services out of the kernel (and implement as application-level processes) in favor of reduced complexity and size and enhanced flexibility — even though it might have been more efficient to provide these services within a larger, monolithic kernel.

- *Reuse of existing libraries* can be supported after new features are introduced. Radical changes to the language and the object model may render existing code obsolete. Hence *reusability* is improved.
- It is more *practical* and *easier* to design and maintain a library than inventing a new concurrent object-oriented language, or modifying an existing language *and* its compiler (even when there are compilers available for modifications).
- By using a strictly object-oriented technique — designing reusable libraries — to introduce new abstractions and keeping the original language in tact, the *principles of object-oriented programming and design* are not violated.
- Adding new features by modifying the language may add a great deal of *complexity* and restrictions to its future evolution. It might even be impossible or difficult to port the language to new hardware or operating system platforms with the added concurrency specifications. Libraries offer a modular and robust mechanism for supporting constantly evolving hardware and O/S platforms.
- Object-oriented libraries support user level extensions.
- Object-oriented libraries can support systematic layered views allowing different levels of abstractions to be delivered to different types of users. While the casual user can use the high-level abstractions, more sophisticated users can use (and *reuse*) the entire library, with all of its lower-level abstractions (such as IPC, O/S interaction, etc.,.) and extend or design new higher-level abstractions.
- Designing libraries can help the language designer from fully committing to a specific solution.
- *Users* are less likely to switch to a non-standard language extension that is customized for programming especially if switching would require also switching (or abandoning) tools and existing libraries. Whereas library based extensions can be easily incorporated to the user's programming environment.

7.1 Summary of Key Contributions

- We have shown by way of designing and implementing how class libraries and associated programming and naming conventions can be used to introduce new abstractions to object-oriented languages.
- We have introduced high-level concurrency abstractions: active objects, asynchronous calls, data-driven synchronization and scheduling to Eiffel by designing and implementing the *Class CONCURRENCY.*
- We have introduced high-level distributed computing abstractions: active

objects, asynchronous calls, data driven synchronization, scheduling, remote object creation by designing and implementing *Active-RMI* system.

- We have introduced design by contract abstractions to Java by designing and implementing *jContractor* system.
- We have introduced user-level programmable scheduling capability to active objects. Scheduling and synchronization decisions can be made based on the type, signature and contents of the request messages.
- We have shown how reflection capabilities of languages can be used to introduce type-safe and syntactically clean abstractions -- *jContractor* and *Active-RMI*.
- We have shown how to enhance syntax and enforce type safety by introducing an automated design method for using library based abstractions when reflection and meta programming capabilities are limited -- *Class CONCURRENCY.*
- We have introduced techniques using reflection and dynamic class loading to perform runtime instrumentation.
- We have introduced a factory-based runtime instrumentation technique which can be used when overloading class loaders is not possible -- *jContractor.*
- We have introduced new runtime instrumentation techniques to introduce new syntax -- OLD & RESULT in jContractor.
- We have shown new techniques to use dynamic class loading capabilities of Java to provide high level abstractions for remote object creation.

7.2 Open Problems and Future Directions

An important area of further study for us is the security modeling. We have currently a rather strict security mechanism based on the RMI security model, however, it is desirable to have less rigid authorization and security abstractions.

An issue that requires further investigation is about identifying the synchronization and correctness needs in the presence of multi threaded schedulers in shared-address spaces. Without having explicit control over the scheduling and preemption of multiple threads in a shared-address space many new technical difficulties are introduced. Some of these difficulties are related with mutual exclusion of the execution of an objects's methods, and non-reentrant system calls. Some of these issues have been mentioned in [15]. A concurrency mechanism with multi threaded active objects must satisfactorily address the interference problem with respect to data encapsulation, pro-

cedural abstraction and reusability issues, that emerge due to the potential arbitrary interleavings of an object's methods.

We have done preliminary design of a new system, *jActivator*, which can be seen as the next generation of *Active-RMI*. *jActivator* will use *jContractor* style class loading time instrumentation to implement active objects and asynchronous method calls.

Appendix A

```
class CONCURRENCY export
    split, attach, remoteInvoke, claimResult, resultReady
          -- Non-exported features:
          -- current_request, has_split, is_proxy,
          -- request_queue, result_queue, sendResult
          -- getRequest, pendingRequestExists
   inherit
   IPC
          -- Interprocess Communication primitives
   feature
   request_queue : LINKED_LIST [REQUEST] is
          -- implemented as a linked list of REQUEST
          -- objects. REQUEST is a class of Parallel
          -- Parallel Library; see appendix B.
          -- LINKED_LIST is a generic class of Eiffel
          -- data structure library [eif]
   result_queue : LINKED_LIST [RESULT] is
          -- queue of all results received;
          -- see appendix A for RESULT
   current_request : REQUEST ;
          -- most recently dequeued request_queue item.
          -- contains the IPC information about the client,
          -- the feature name, and a list of actual
          -- parameters (of type: unbounded_array[ANY])
   has_split : BOOLEAN ;
          -- split or attach sets to true.
   is_proxy : BOOLEAN ;
          -- set to true by attach or split
          -- set to false in the active object
   scheduler() is
          deferred
                -- requires any Class inheriting from
                -- CONCURRENCY to provide a scheduler.
   split() is
          require
                not has_split -- precondition for split
          -- creates a new process to function as a server
          -- creates a socket on the client,
          -- initializes IPC parameters,
          -- starts handshake protocol with server,
          -- sets has_split and is_proxy to true,
          -- returns and unblocks client.
          -- The server process completes handshake,
          -- starts-up a new Eiffel runtime environment
```

```
      -- initializes server IPC params,
      -- starts executing server's scheduler.
attach( old_server_info : SOCKET) is
      require
            not old_server_info.Void
      -- like split except no new process is created,
      -- sets has_split and is_proxy to true,
      -- no handshake done;
      -- returns immediatelly after initializing
      -- client side of IPC parameters.
remoteInvoke( feature_name: STRING,
            parameters: ARRAY[ANY] ): INTEGER is
      require is_proxy;
            -- cannot remoteInvoke unless a is_proxy.
      -- asynchronously delivers the request to the
      -- server, and returns without blocking.
      -- returns a request handle (claim_number)
      -- to the client identifying the request

result_available (claim_number : INTEGER ):
            BOOLEAN  is
      require is_proxy;
      -- scans the result_queue to check if the result
      -- associated with claim_number has arrived.
      -- Returns, without blocking, true
      -- if associated result is in queue.
claimResult (claim_number: INTEGER ): ANY is
      require is_proxy;
      -- if the result associated with claim_number
      -- is available: returns (by dequeueing it
      -- from result_queue) the corresponding result
      -- otherwise, blocks until the result arrives.
      -- The result is of type ANY. Every class in
      -- Eiffel is a descendant of this Kernel Library
      -- class ANY, and therefore all class types
      -- conform to it; thus claimResult is general
      -- purpose, and applicable to any result type.
sendResult ( result_value : ANY) is
      -- result of the current_request is delivered to
      -- the client.
      -- result_value is of type ANY, therefore
      -- the actual result needs to be
      -- reverse-assigned to the original result type
      -- after calling this method.
      -- result is delivered asynchronously.
getRequest() is
      -- If there are pending requests in the system,
      -- place them into the request_queue, and
      -- return (without dequeueing);
```

```
            -- otherwise: block until some request(s) arrives,
            -- place them into request_queue; return.
        pendingRequestExists() is
            -- Return true: if there are pending requests
            -- in the system.
            -- Return false: otherwise.
end -- CONCURRENCY
```

Appendix B

```
class RESULT export
    claim_no, set_claim_no,return_value, set_return_value
feature
    claim_no : INTEGER ;
         -- this is the request handle the client uses
         -- to associate the result with the request.
    set_claim_no ( id_res : INTEGER ) is
         -- exported routine to set claim_no.
    return_value : ANY;
         -- result of the request as set by the server
         -- type must conform to ANY
    set_return_value( val : ANY )  is
         -- exported routine to set return_value
end-- RESULT
```

Appendix C

```
class REQUEST export
    claim_no, set_claim_no, req_type, set_req_type,
    feat_name, set_feat_name,
    parameters, set_parameters
feature
    claim_no: INTEGER;
         -- unique id for each request
    set_claim_no( req_num: INTEGER) is
         -- exported feature to set claim_no
```

```
    req_type: INTEGER;
         -- func, rout, attr, rattr, battr, msg, etc.
         -- (optionally) used to determine what action
         -- to take by the server
    set_req_type( r_type : INTEGER )  is
         -- exported feature to set req_type
    feat_name : STRING ;
         -- name of the method to be remote invoked
    set_feat_name( f_name : STRING)  is
         -- exported feature to set feat_name
    parameters : ARRAY [ANY] ;
         -- the array contains the arguments of
         -- the method to be remote invoked.
         -- the client needs to initialize the elements in
         -- the same order used to execute the routine.
    set_parameters ( par_list : ARRAY [ANY] ) is
         -- exported feature to deep clone parameters
end-- REQUEST
```

Appendix D

```
class FUTURE export
   data, is_ready
feature
   proxy: CONCURRENCY;
         --proxy object used during remote invocation
   call_id: INTEGER;
         --the handle returned by remoteInvoke
   returned_data : ANY;
         --the data object returned by claimResult
   create(proxy_obj: Conc_A, call_no: INTEGER) is
         do
               proxy := proxy_obj;
               calli_id:= call_no;
         end;
   data: ANY is -- blocking acces to result
         do
               remote _access;
               Result := returned_data;
         end;
```

```
remote_access is
      once
            returned _data := proxy.claimResult(call_id);
      end;
is_ready : BOOLEAN is
      do
            Result := proxy.resultReady(call_id);
      end;
end; - FUTURE
```

References

[1] A. Acharya, M. Ranganathan, J. Saltz, "Sumatra: a Language for Resource Aware Mobile Programs, Mobile Object Systems, Lecture Notes in Computer Science, No. 1222, Springer Verlag (D), pp. 111-130, February 1997.

[2] Agha, G.H. ACTORS: A Model of Concurrent Computation in Distributed Systems, The MIT Press, Cambridge, Mass., 1986.

[3] Agha, G. Concurrent Object-Oriented Programming. Communications of the ACM. 33(9) (September 1990) 125.

[4] Agha, G., Wegner, P., Yonezawa, A. ACM SIGPLAN Workshop on Object-Based Concurrent Programming. ACM, April 1989. ACM SIGPLAN Notices, Vol.24, No.4.

[5] America, P., POOL-T: A Parallel Object-Oriented Language., Object-Oriented Concurrent Programming, ed. M. Tokoro, A. Yonezawa, pp. 199-220, MIT Press, Cambridge, Mass. 1987.

[6] Andrews, G.R., et.al. An Overview of the SR Language and implementation. ACM Transactions on Programming Languages and Systems.10 (January 1988) 51.

[7] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In ICPP '99, pp.4-11, Aizu-Wakamatsu, Fukushima, Japan, 21-24 September 1999.

[8] Bershad, B.N., et.al. PRESTO: A System for Object Oriented Parallel Programming. Software-Practice and Experience. 18 (August 1988) 713.

[9] Birrel, A.D., Nelson, B.J. Implementing Remote Procedure Calls. *In Proc. ACM Symp. on Transactions on Computer Systems*, pp.19-59, February 1984.

[10] Box, Don. *Creating Components with DCOM and C++*. Addison Wesley, 1997.

[11] Briot, J-. P. Actalk: A testbed for classifying and designing actor languages in Smalltalk-80 environment. In Proceedings of the Third ECOOP Conference '89.(July 10-4 1989, Nottingham), Cambridge University Press, 1989, pp.109-129.

[12] Brose, G., Löhr, K. -P., Spiegel, A. Java does not distribute. In Christine Mingins, Roger Duke, and Bertrand Meyer, editors, *TOOLS Pacific '97*, pages 144-52, Melbourne, Australia, 24-27 November 1997.

[13] Brose, G., Löhr, K. -P., Spiegel, A. Java resists transparent distribution. *Object Magazine*, 7(10):50-2, December 1997.

[14] Buhr, P.A., et.al. µC++: Concurrency in the Object-oriented Language C++. Software-Practice and Experience.22(2) (February 1992) 137.

[15] Buhr, P.A., Ditchfield, G. Adding concurrency to a programming language. In Proceedings of USENIX C++ Technical Conference (August 10-13, 1992, Portland Or.) USENIX Association, Berkeley, 1992, pp. 207-223.

[16] Campbell, R., Islam, N., Madany, P. Choices, Frame-works and Refinement. Computing Systems, 5(3) (1992) 217-257.

[17] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Winter 1995.

[18] Caromel, D. Concurrency and Reusability: From Sequential to Parallel. Journal of Object-Oriented Programming,3(3) (September 1990) 34.

[19] Caromel, D Toward a Method of Object-Oriented Concurrent Programming. Communications of the ACM 37(8) (Septemeber 1993), pp. 90-102.

[20] Denis Caromel, Wilfried Klauser, and Julien Vayssire. Towards seamless computing and metacomputing in Java. Concurrency: Practice and Experience, 10(11--13):1043--61, September-November 1998.

[21] Rohit Chandra, Anoop Gupta, and John Hennessy. COOL: An object-based language forparallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[22] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In Utpal Banerjee et al., editor, *Languages and Compilers for Parallel Computing '92*, volume 757 of *Lecture Notes in Computer Science*, pages 124–44, New Haven, CT, USA, 3–5 August 1992. Springer-Verlag.

[23] Andrew A. Chien, et.al., Supporting high level programming with high performance: The Illinois Concert system. In Hermann Hellwagner, editor, *HIPS '97 (IPPS '97 Workshop)*, pages 15–25, Geneva, Switzerland, 1 April 1997.

11th International Parallel Processing Symposioum; IEEE Computer Society Technical Committee on Parallel Processing; ACM SIGARCH.

[24] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using Java. Concurrency: Practice and Experience, 9(11):1139--60, November 1997

[25] Colin, J-.F., Geib, J-.M. Eiffel Classes for concurrent programming.In Proceedings of TOOLS-4 '91 Conference. Prentice Hall 1991, pp.23-34.

[26] Cox, B. Object Oriented Programming -- an evolutionary approach. Addison Wesley, Reading Mass. 1986

[27] F.Douglis and J.Ousterhout. Process migration in the Sprite operating system. In *Proc. of the 7th International Conf. Distributed Computing Systems*, pp.18-25, 1987.

[28] Andrew Duncan and Urs Hölzle. *Adding Contracts to Java with Handshake*. Technical Report TRC98-32, University of California, Santa Barbara, 1998.

[29] Ellis, M., Stroustrup, B. The Annotated C++ Reference Manual. Addison-Wesley, Reading Mass., 1990.

[30] Erich Gamma, Ralph Johnson Richard Helm, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[31] J.Gosling et.al. *The Java Language Specification.* Addison Wesley, SunSoft Java Series, 1996.

[32] Robert Gray. *Agent Tcl: A flexible and secure mobile-agent system.* Ph.D. Thesis. Dartmouth College, Hanover, N.H., June 30, 1997.

[33] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), October 1969.

[34] Jerry James. *Reliable Distributed Objects: Reasoning, Analysis, and Implementation*. Ph.D. Dissertation. Computer Science Department, University of California at Santa Barbara, March 2000.

[35] B.Janssen and Mike Spreitzer. *ILU 2.0 Reference Manual.* Xerox PARC, *ftp://ftp.parc.xerox.com/pub/ilu/ilu.html* 1996.

[36] E.Jul, et.al. Fine grained mobility in the Emerald system. In *Proc. ACM Symp. on Transactions on Computer Systems*, pp.109-33, 1988.

[37] Kafura, D.G., Lee, K.H. Inheritance in ACtor based concurrent object-oriented languages. In Proceedings of ECOOP '89(July 10-14, Nottingham). Cambridge University Press, 1989, pp.131-145.

[38] L. V. Kale, Milind Bhandarkar, and Terry Wilmarth. Design and implementation of parallel Java with global object space. In Hamid R. Arabnia, editor, PDPTA '97, volume 1: Computer Science Research, Education, and Applications, pages 235--44, Las Vegas, NV, USA, 29 June 1997.

[39] L. V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object-oriented system based on C++. In Andreas Paepcke, editor, *OOPSLA '93*, pages 91–108, Wash-ington, DC, USA, 26 September–1 October 1993. ACM SIGPLAN, ACM Press.

[40] Karaorman, M., and Bruno, J. A concurrency mechanism for sequential Eiffel. In Proceedings of TOOLS USA '92 Conference(Aug. 3-6, Santa Barbara, Calif.). Prentice Hall 1992, pp.63-77.

[41] Murat Karaorman and John Bruno. Introducing concurrency to a sequential language. *Communications of the ACM*. September 1993, Vol.36, No.9, pp.103-116.

[42] Ralph Keller and Urs Hölzle. *Binary Component Adaptation*. Proc. of ECOOP '98, Lecture Notes in Computer Science, Springer Verlag, July 1998.

[43] Reto Kramer. *iContract – The Java Design by Contract Tool*. Proc. of TOOLS '98, Santa Barbara, CA August 1998. Copyright IEEE 1998.

[44] Launay, P., Pazat, J. -L., Generation of distributed parallel Java programs. In D.Pritchard anf J. Reeve, editors, Eura-Par '98, V.1470 Lecture Notes in Conputer Science, pp.729-732, Southampton, UK, 1-4 September 1998. Springer.

[45] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996

[46] Liskov, B., Herlihy, M., Gilbert, L. Limitations of synchronous communication with static process structure in languages for distributed computing. Concurrent Programming, ed. Gehani and McGettrick, Addison-Wesley 1988.

[47] Löhr, K.-P. Concurrency Annotations improve reusability. In Proceedings of TOOLS USA '92 Conference(Aug. 3-6, Santa Barbara, Calif.). Prentice Hall 1992, pp.53-62.

[48] Löhr, K.-P Concurrency Annotations for Reusable Software. Communications of the ACM. 37(8) (September 1993), pp. 81-89.

[49] Lucent Technologies. *Inferno 2.0 User's Guide*. http://www.lucent-inferno.com/ , 1998.

[50] Mike Mannion and Roy Phillips. *Prevention is Better than a Cure*. Java Report, Sept.1998.

[51] Bertrand Meyer. *Applying Design by Contract*. In Computer IEEE), vol. 25, no. 10, October 1992, pages 40-51.

[52] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1992.

[53] Meyer, B., Object-Oriented Software Construction. Prentice-Hall, New York, 1988.

[54] Meyer, B., Sequential and Concurrent Object-Oriented Programming. In Proceedings of TOOLS 2 Conference(SOL/Angkor, Paris, June 1990). pp. 17- 28, 1990.

[55] Meyer, B. Systematic Concurrent Object-Oriented Programing. Communications of the ACM. 37(8) (Septemeber 1993), pp. 56-80.

[56] Nester, C., Philippsen, M., Haumacher, B> A more efficient RMI for Java. In Java Grande ‘99 [6], pp.152-159.

[57] Nierstrasz, O.M. Active Objects in Hybrid. ACM SIGPLAN Notices. 22 (December 1987) 243.

[58] Nierstrasz, O.M. Next 700 concurrent object-oriented languages -- Reflections on the future of object-based concurrency. In Object Composition, ed. D.C. Tsichritzis, pp.165-187, Centre Universitaire d'Informatique, University of Geneva, June 1991.

[59] Open Software Foundation. *Introduction to OSF DCE: Rev 1.0*. Prentice Hall, 1992.

[60] Object Management Group. *The Common Object Request Broker Architecture and Specification, 2nd ed.*1995.

[61] Papathomas, M. Concurrency Issues in Object-Oriented Languages.Object Oriented Development Technical Report, Centre Universitaire Informatique, University of Geneva, ed. D. Tsichritzis, pp. 207-245, 1989.

[62] Michael Philippsen and Matthias Zenger. Javaparty---transparent remote objects in Java. Concurrency: Practice and Experience, 9(11):1225--1242, November 1997.

[63] Michael Philippsen and Bernhard Haumacher. More efficient object serialization. In International Workshop on Java for Parallel and Distributed Computing, San Juan, Puerto Rico, 12 April 1999. Held in conjunction with IPPS/SPDP '99.

[64] S.Porat and P.Fertig. *Class Assertions in C++*. Journal of Object Oriented Programming, 8(2):30-37, May 1995.

[65] Probert, D., Bruno, J., Karaorman, M. SPACE: A New Approach to Operating System Abstraction. In Proceedings of International Workshop on Object Orientation in Operating Systems (October 17-18, 1991 Palo Alto, CA). IEEE Computer Society Press, 1991, pp 133-137.

[66] Aleta Ricciardi, Michael Ogg, and Fabio Previato. Experience with distributed replicated objects: The Nile project. Theory and Practice of Object Systems, 4(2):107--15, 1998.

[67] Hirano Satoshi. The Magic Carpet for Network Computing: HORB Flyer's Guide. Electrotechnical Laboratory *http://ring.etl.go.jp/openlab/horb*, 1996.

[68] Stevens. W.R. *TCP/IP Illustrated, Volume 1 The Protocols*. Addison Wesley, 1994.

[69] James White. Telescript technology: mobile agents. *http://genmagic.com/Telescript/WhitePapers*, 1996.

[70] A.Wolrath, R.Riggs, and J.Waldo. A distributed object model for Java. In *2nd Conf. On Object-Oriented Technologies and Systems (COOTS)*.pp.219-231, Toronto, Ontario, 1996.

[71] D. Wu, D. Agrawal, A. El Abbadi, and A. Singh. A Java-based framework for processing distributed objects. In Proc. Intl. Conf. on Conceptual Modeling, pages 333-46, Los Angeles, CA, 1997.

[72] Yokote, Y., and Tokoro, M. Concurrent Programming in ConcurrentSmalltalk. Object-Oriented Concurrent Programming, pp. 129-158, MIT Press, Cambridge, Mass. 1987.

[73] Yonezawa, A., et.al., Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. Object-Oriented Concurrent Programming, pp. 55-89, MIT Press, Cambridge, Mass. 1987.