76
45

Designing a Statically Typed Actor-Based Concurrent Object-Oriented Programming Language

by

Keung Hae Lee

Dissertation Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

APPROVED:

Dennis Kafura, Chairman

Richard Nance

James Arthur

Monte Boisen

Calvin Ribbens

Adrienne Bloss

June, 1990

Blacksburg, Virginia

Designing a Statically Typed Actor-Based Concurrent Object-Oriented Programming Language

by

Keung Hae Lee

Dr. Dennis Kafura, Chairman
Computer Science

(ABSTRACT)

The research reported in this dissertation investigates extending the power and flexibility of an object-oriented language with *inheritance*, *static typing*, and *concurrency*. A language supporting inheritance, static typing, and concurrency offers a significant leverage in software development. However, when these features are provided together, the benefits of the features are significantly reduced due to the interaction among them. The challenge for the designer of an object-oriented language lies in the difficulty of reconciling the conflicts among these features. This thesis discusses two issues: combining static typing with inheritance and combining concurrency with inheritance. A new model of type and inheritance, called HANA, is presented. The HANA model integrates multiple inheritance, multiple representation, method exclusion, and method name overloading with static typing. The contribution of HANA is that it extends other existing models of type and inheritance with enhanced expressive power, increased reusability, and improved program efficiency. Combining concurrency with inheritance is investigated in the framework of the actor computation model. A language design based on the actor model of concurrent computation faces a serious problem arising from the interference between concurrency and inheritance. A similar problem also occurs in other concurrent object-oriented languages. The problem of concurrency-inheritance conflict is described and a solution based on a concept called *behavior abstraction* is presented.

Acknowledgments

With the deepest gratitude, I acknowledge the support and encouragement received from my major advisor, Dr. Dennis Kafura, throughout this investigation. I am truly grateful for his excellent guidance and the confidence he has shown in me. Without his unfailing patience and enthusiasm, this dissertation would not have been possible.

I am also grateful to Drs. Richard Nance, James Arthur, Monte Boisen, Calvin Ribbens, and Adrienne Bloss for agreeing to serve on my committee. I am particularly grateful for having had the opportunity to give a presentation before the committee once every semester. Their questions, expert advice, and helpful criticism contributed to the quality of this dissertation and significantly improved my technical presentation skills. I also thank Dr. Allison, Dr. Schuetz, and Ms. Patty Foutz for providing financial support during my long stay in the Department of Computer Science.

I would also like to acknowledge the assistance of Greg Lavender. Since I wrote my first technical report, he has been my personal editor. I enjoyed the numerous discussions I had with him. Greg always found time to convert the mess I created into readable form. Without his assistance and friendship, this task would have been much more difficult. In addition, I would like to thank Mat Davis for his reading my papers and providing me with constructive criticism. I also thank Kevin Mayo and Steve Wake for having become friends with me.

Finally, I owe much to my family. I thank my parents, Chongchul Lee and Jaesoon Park, and my parents-in-law, Woohyun Kim and Jaesoon Um for their support, encouragement, and prayer. I also thank my grand mother. I am indebted to my sisters, Kyunghae, Haeja, Myungja, and Hyunhae, who missed their only brother's love for such a long time. I thank

Sukjoo Noh, my brother-in-law, for his encouragement. A special thank goes to Wonsook and Ihnsup Kim, my sister-in-law and brother-in-law, for their support and encouragement.

My daughter Somee always tolerated my absence from home for many nights although she liked my playing with her more. She has always been the source of joy. My wife Kyunghee deserves very special thanks. Without her love, support, encouragement, and understanding, I would not have been able to complete this dissertation. I dedicate this dissertation to my wife Kyunghee.

Table of Contents

List of Figures

Chapter 1 Introduction

1.1 Problem Statement

This thesis investigates an important issue in object-oriented programming: can static typing and concurrency be combined with inheritance in a single language? This combination of language features illustrated in Figure 1 is important for two reasons.

First, inheritance is the characterizing feature of object-oriented programming [Wegner 87]. Inheritance allows new software components to be built on other existing components. Inheritance provides many significant advantages such as software reusability, modifiability, code sharing, and software classification.

Second, both concurrency and static typing have long been central issues of programming language design research and their benefits and importance are well understood. Static typing provides type safety by allowing many programming errors to be detected at compile time. Another advantage is the support for program efficiency which is enabled by static binding of names. Concurrency offers both notational convenience for expressing concurrent events and the support for parallelism and distribution. The support of concurrency is fundamental to solving the problems which involve concurrent activities such as those found in real-time systems and multiprocessing. As the problems to be solved by computers become increasingly more complex, and the number of distributed and parallel processors grows, designing an object-oriented language which offers the advantages of inheritance, static typing, and concurrency is an important task.

The goal of integrating several independently developed language features into the same language framework is to increase the power of the language. The hope is that the benefits offered by each of the features being combined will also be retained by the new language design. However, a common but difficult problem which a language designer faces is the interaction among language features. While combining several language features which are known to be powerful into a single framework is desired, feature interaction typically entails a reduction in the benefits which would be offered by each individual feature. Commonly, a significant reduction in the leverage results when powerful features are put together in a language. A challenge for a language designer lies in resolving the difficulty arising from the conflict among the language features.

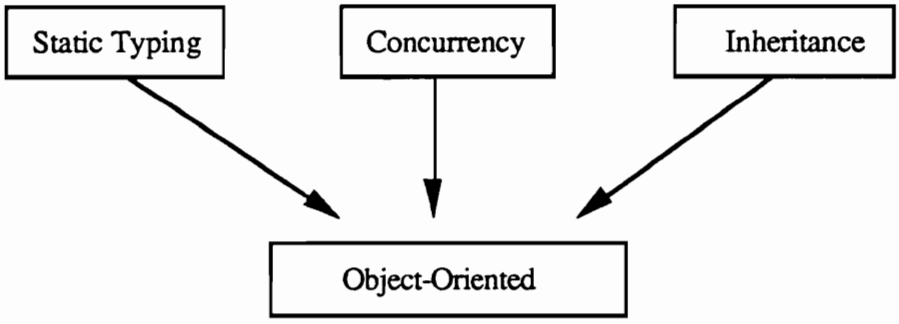


Figure 1 - The Thesis Issue

Combining class inheritance, static typing, and concurrency into the same framework poses difficult problems which arise from the feature interaction among them. The use of class inheritance often conflicts with static typing. In most current object-oriented languages, a subclass cannot selectively inherit the operations of the superclass without violating typing constraints [Snyder 86A, Lee and Kafura 89]. While data abstraction is one of the basic principles underlying object-oriented programming, an abstract data type typically cannot have more than one representation within the same language framework [Lee and Kafura 90]. In addition, the use of inheritance interferes with concurrency [Kafura and Lee 89A,

B]. The interference between inheritance and concurrency results in object-oriented concurrent languages which either do not support inheritance or which do so only by severely compromising some other property. The problem of feature interaction between concurrency and static typing does not seem to be serious compared with the other two. Many existing programming languages support both static typing and concurrency, thus providing existence proofs. Examples of such languages are ADA [DoD 83] and Emerald [Black 87]. No negative interaction between static typing and concurrency occurs in these languages.

1.2 Thesis Plan

Since evidence exists that interaction between concurrency and static typing is not a problem, this thesis focuses on the remaining two issues: combining static typing with inheritance and combining concurrency with inheritance. The actor model developed by Agha [Agha 86] is used as the model of concurrency underlying this research.

Combining static typing with class inheritance raises several related issues such as subtyping, method exclusion, multiple representation, and overloading. A new model of type and inheritance is developed which integrates these language features into a single framework without compromising the benefits of static typing and inheritance. Within the framework of this new model, the interference among these features is reconciled and multiple inheritance is examined in order to extend the leverage of inheritance. Two important issues of multiple inheritance are investigated: name collision and implementation efficiency. The new type model resolves the name collision problem and lends itself to an efficient implementation. An efficient late binding mechanism for a multiple inheritance object-oriented language is investigated.

In order to analyze the concurrency-inheritance conflict in existing concurrent object-oriented languages, a prototype object-oriented language is developed which supports class inheritance and concurrency based on the actor model. The language is used as the tool for exploring a solution to the concurrency-inheritance conflict.

1.3 Contributions

This research explores a unique combination of static typing, class inheritance, and actor concurrency. Some important contributions of this work are the following:

- Analysis and resolution of the feature interaction between static typing and class inheritance,
- Analysis and resolution of the concurrency-inheritance conflicts,
- Development of a new type model which integrates multiple inheritance, subtyping, method exclusion, multiple representation, and method name overloading with static typing,
- Integration of class inheritance and actor concurrency,
- Analysis and resolution of the interaction between subtyping and overloading,
- Resolution of name collision in multiple inheritance,
- Development of an efficient late binding mechanism for multiple inheritance, and
- Development of a prototype actor-based object-oriented language based on a class hierarchical extension.

1.4 Thesis Overview

This thesis is divided into two parts. Part 1 addresses the issue of combining static typing with inheritance while Part 2 is concerned with combining actor concurrency with inheritance. Part 1 consists of Chapters 2 through 5 and part 2 is presented in Chapters 6 and 7. A summary of each chapter follows.

Chapter 2 presents a new model of type and inheritance, called HANA [Lee and Kafura 90]. HANA integrates multiple inheritance, multiple representation, method exclusion, and method name overloading with static typing. The model is built on the recognition of the difference between inheritance and subtyping. In HANA, subtyping is based on both inheritance and interface conformance. The notion of *abstract type* is introduced. Three kinds of inheritance are differentiated: *interface inheritance*, *class inheritance*, and *specification inheritance*. The significance of HANA is that the model extends other existing models of inheritance and subtyping with enhanced expressive power, increased reusability, and improved program efficiency.

Chapter 3 defines a new type rule accepting many subtype relations which cannot be accepted by other type systems without overloading. The rule is described and its power is demonstrated. While the support of overloading extends the power of object-oriented languages, inheritance interferes with overloading. The feature interaction between inheritance and overloading and a solution for the problem are presented in the chapter.

Chapters 4 and 5 discuss inheritance. Although multiple inheritance is more powerful than single inheritance, supporting multiple inheritance poses two critical problems: name collision and the implementation of late binding. The issue of name collision is discussed in Chapter 4. An efficient late binding mechanism for multiple inheritance is presented in Chapter 5.

Chapter 4 describes how the HANA model resolves the problem of name collision. Name collision arises when a class inherits a method with the same name from more than one parent. The capability of excluding inherited methods without violating static typing offers a sound solution for resolving name collision in multiple inheritance when used together

with the mechanisms of method redefinition and renaming. It is shown that these three mechanisms are orthogonal with respect to the power of resolving name collision in multiple inheritance. Finally, it is shown that the support of method name overloading in HANA eliminates many name collision problems.

Chapter 5 describes a fast and efficient late binding mechanism for statically typed object-oriented programming languages with multiple inheritance. While the benefits of inheritance are well recognized, these benefits typically come at the expense of run-time overhead in both time and space in the case of multiple inheritance. A mechanism for multiple inheritance with an efficiency comparable to that of single inheritance has been sought. Our technique, called *hierarchy partitioning*, is a significant improvement in both space and time over other techniques [Lee and Kafura 89]. A late binding mechanism based on the partitioning of a multiple inheritance hierarchy is presented. A detailed comparison with other related work is also provided.

Chapter 6 describes a prototype design and implementation of an exploratory language called ACT++ (Actors in C++). ACT++ is a concurrent object-oriented language designed for the purpose of exploring the combination of concurrency with inheritance [Kafura and Lee 89C]. ACT++ is planned to evolve into a language for distributed real-time applications in the future [Kafura 88, 89]. The language is a hybrid of the actor model and the object-oriented language C++. The concurrency abstraction of ACT++ is derived from the actor model as defined by Agha [Agha 86]. The chapter describes our experience in building a concurrent extension of C++ with the concurrency abstraction of the actor model. Some problems found in Agha's actor model are discussed in the context of distributed real-time applications. The use of ACT++ discloses the difficulty of combining the actor model of concurrency with class inheritance, which is the subject of Chapter 7.

Chapter 7 focuses on the problem of combining concurrency with inheritance. The interference between inheritance and object-based concurrency has been noted by others [Briot 87, Wegner 87]. While their observations center on the difficulty of locating or copying methods at run-time in systems without shared memory, the fundamental interference between inheritance and concurrency is more deeply rooted. This interference results in concurrent object-based languages which either do not support inheritance or which do so only by severely compromising some other property. An analysis of the problems found in existing concurrent object-oriented languages and a solution which is based on a concept called *behavior abstraction* [Kafura and Lee 89A, B] is presented.

The remaining sections in the current chapter review the background materials on object-oriented programming. Readers familiar with the concepts of object-oriented programming may skip to Chapter 2 with no loss in continuity. Section 1.5 reviews object-oriented programming. Section 1.6 presents a brief review of the literature related to this dissertation research. A more detailed literature review is given in each chapter as needed. A review of the background on the actor model is postponed until Chapter 6 where the discussion on combining concurrency with inheritance starts.

1.5 Background

1.5.1 Object-Oriented Programming

Object-oriented programming is a programming paradigm in which the software system being developed is viewed as a collection of software components called objects. Object-oriented programming integrates the results of software engineering and programming language research efforts over the last two decades within a single framework. Object-oriented programming is rooted in well-known software engineering principles such as information hiding, data abstraction, modularity, and software reusability. Mechanisms implementing these principles provide an object-oriented programming language with powerful features which enable a programmer to cope with increasingly complex problems.

Object-oriented programming was first introduced by SIMULA [Dahl 66], a language designed for implementing computer simulations. Due to SIMULA's origins in computer simulation, object-oriented programming stresses the view of the programming process as a form of modeling. A program is viewed as a set of domain entities, called objects, which interact with each other usually via message passing. In addition to the notion of object, an object-oriented programming language supports the modeling viewpoint of programming with two other language mechanisms: abstract data type (class) and inheritance.

Objects

An *object* is a software component which has an encapsulated internal *state* and a set of *operations*. The internal state is a private part of the object and can be accessed only by its own operations. An operation of an object is usually called a *method* of the object. An object can send a *message* to another object in order to ask the *receiver* object to perform a certain operation on behalf of the *sender*. The action of sending a message for a method is called a *method invocation*. In existing languages two different syntactic styles are commonly used for method invocation; namely, message passing and function call.

Smalltalk originated the message passing style [Goldberg 83] while SIMULA [Dahl 66], C++ [Stroustrup 86], Trellis/Owl [Schaffert 86], and Eiffel [Meyer 88] use a function call syntax. However, it is not uncommon to interpret the function call style using the message passing metaphor. The sender object is called a *client* while the receiver object performing the requested operation is called a *server*.

The privacy of the internal state of an object is ensured by a mechanism called *encapsulation*. Client objects can affect the state of a server object only via method invocation. The internal mechanisms by which a server object responds to a method invocation is not the concern of a client. Hence, a natural view of an object is a collection of private data items protected by the methods which have access to those data items. For this reason, the set of methods of an object is called the *interface*. The set of methods of an object is also often called the *protocol* or *behavior* of the object.

Figure 2 shows the conceptual view of an object called BankAccount. The interface of the object consists of three methods: Deposit, GetBalance, and Withdraw. Other objects cannot access the variable inside the object directly, but must use the methods in the interface of the object. For example, in order to find the current balance of the account, a client sends a GetBalance message, to which the account object responds with the value of the current balance. If the client wants to make a deposit, it can send a Deposit message specifying the amount of the deposit as a part of the message. Finally, if a client wants to withdraw some money, the client can use a Withdraw message, specifying the amount of withdrawal. Clients need to know only the names of the methods and the message formats. An object is responsible for correctly responding to well-formed messages.

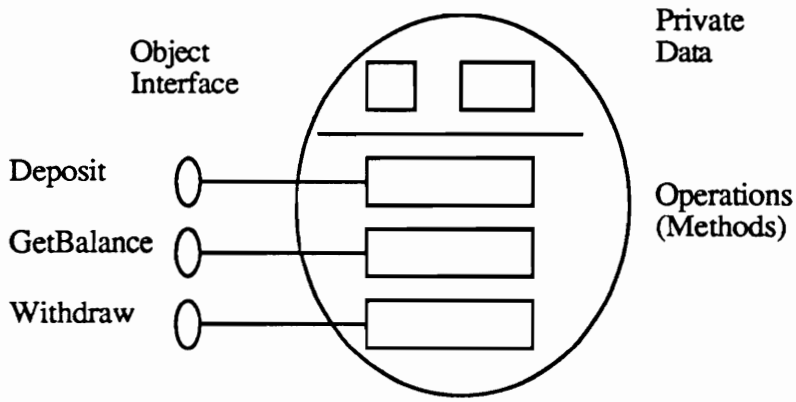


Figure 2 - Illustration of BankAccount Object

Class

While encapsulation protects the internal state of an object from unauthorized access, another mechanism called *data abstraction* separates the behavior of an object from the implementation. Abstraction is an important mechanism in programming. Abstraction mechanisms allow a programmer to concentrate on the desired level of interest without being confused by unnecessary detail. For example, procedural abstraction allows a program to be built using procedure names, thereby liberating the programmer from being concerned about the implementation details of the procedures. Procedural abstraction enables a programmer to concentrate on the higher-level structure of a program. The study of implementation details can be conveniently postponed until the programmer is confident about the higher-level structure of the program.

Data abstraction provides similar benefits at a higher level. The representation of the internal state of an object may be hidden by the behavior of the object. In order to use an object, a programmer only needs to know about its external behavior, i.e., the names and arguments of available methods but not how they are implemented. This separation of

behavior from implementation detail has a significant impact on programming. The interface of an object can be viewed as a contract or specification between the object and its clients. As long as the object retains the same interface, the object can freely change its internal representation with no visible effect on its clients. This principle was introduced by Parnas as *information hiding* [Parnas 72].

The primary importance of data abstraction is the reduced complexity of reasoning about a given problem. In object-oriented design, a problem is often naturally decomposed into a collection of objects. The locality supported by encapsulation and data abstraction allows a program to be studied through local reasoning on these smaller objects. The result is a tremendous saving in programming effort [Liskov 87].

In object-oriented languages, data abstraction is supported by a mechanism called a *class* [Dahl 66, Goldberg 83, Stroustrup 86, Cox 86, Meyer 88]. A class is an abstract data type from which objects having the same property may be instantiated. An object is created (instantiated) from a class by applying the operation for creation, typically called *new* or *create*. An object created from a class is said to be *an instance* of the class. The properties of an object are determined by the class of which it is an instance. All instances of a class have the same properties: the same internal representation and the same operations. A class definition consists of the declaration of methods available to clients and declarations of data items. Figure 3 shows a definition of a BankAccount class expressed in a pseudo code.

```
Class BankAccount
        real currentBalance;
    Method GetBalance
        return (currentBalance);

    Method Deposit (amountToDeposit)
        currentBalance := currentBalance + amountToDeposit
        return (currentBalance);

    Method Withdraw(amountToWithdraw)
        If currentBalance >= amountToWithdraw then
            currentBalance := currentBalance - amountToWithdraw;
            return (amountToWithdraw);
        else
            print "insufficient balance";
```

Figure 3 - Representaion of Class BankAccount

Inheritance

While objects, encapsulation, data abstraction, and instantiation alone make object-oriented programming powerful, the most notable contribution of an object-oriented programming language is the support for software reusability [Meyer 88]. *Reusability* [Biggerstaff 87] allows new software to be built around existing software components which are already implemented and tested. Until recently, only primitive forms of software reusability have been realized. Earlier proposed techniques were either too theoretical for practical use or too ad hoc with little leverage provided to a programmer. Object-oriented programming squarely addresses the issue through a language mechanism called ***inheritance.***

The inheritance mechanism of an object-oriented programming language allows new software components to be built as extensions and modifications of existing components. The reusability inherent in the inheritance mechanism provides critical leverage to productivity, prototyping, reliability, and modifiability of software [Diederich 87]. The inheritance mechanism in object-oriented programming is expected plays a vital role in object-oriented software construction.

In a language supporting inheritance, a new class can be defined from an old one by specifying the difference between the two. The new class being defined is termed a *subclass* while the existing class is referred to as a *superclass*. A subclass automatically inherits all methods and instance variables of its superclass, as if the subclass defined within itself all methods and instance variables belonging to the superclass. A subclass can add new methods and instance variables to those of its superclass. A subclass is also allowed to reimplement an inherited method.

Suppose that one needs to create a new type of bank account (ProtectedCheckingAccount) which provides overdraft protection. ProtectedCheckingAccount behaves the same way as the BankAccount defined in Figure 3, except the Withdraw method is different. With inheritance, this new class can be defined by inheriting from the existing class, BankAccount. The implementation of the new bank account with overdraft protection is given in Figure 4. The definition of ProtectedCheckingAccount immediately reuses all methods and data items of the BankAccount except the Withdraw method, which is reimplemented. While this is an example of redefining a superclass method, object-oriented languages also allow a subclass to add new methods. A subclass may also exclude or rename inherited methods. However, in current statically typed object-oriented languages, these mechanisms are not safe [Lee and Kafura 90].

```
Class ProtectedCheckingAccount
        Inherits BankAccount
                SavingsAccount mySaving;
        Method Withdraw(amountToWithdraw)
                If currentBalance > amountToWithdraw
                        currentBalance := currentBalance -
                                          amountToWithdraw;
                        return (amountToWithdraw);
                else
                        If mySaving.Withdraw(amountToWithdraw -
                                             currentBalance) then
                                currentBalance := 0;
                                return (amountToWithdraw);
                        else
                                print "insufficient balance";
```

Figure 4 Implementation of Class ProtectedCheckingAccount

1.5.2 Static Typing in Object-Oriented Languages

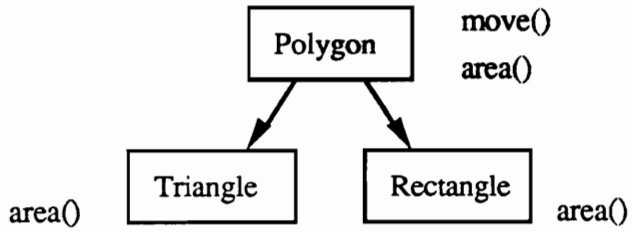
Static typing refers to the type-checking of expressions at compile time. A type correct program written in a language with static typing does not cause an object to perform an illegal operation. When static typing is supported, the type of an expression is determined from its syntax at compile time. Hence, it can be decided at compile-time whether or not a given method invocation results in a type error at run time. Static typing offers significant advantages, such as type safety and program efficiency. In contrast, a language with dynamic typing postpones checking the type of an expression until run-time. A serious problem associated with dynamically typed languages and untyped languages is that type errors are not detected until run time. An example of this problem is the "unknown message" error of Smalltalk. Considerable effort planned on adding static typing to

dynamically typed object-oriented languages, for example, Smalltalk [Suzuki 81, Borning 82, Johnson 87, Grover 90].

In object-oriented languages, static typing is based on the notion of *subtyping*. Subtyping offers a more generalized type compatibility model compared with the notion of type equivalence in other languages. Subtyping determines when one type of object may be used where another type of object is expected. As in conventional languages, an object of type *X* can be assigned to a variable of another type *Y* if *X* and *Y* are the same type. In addition, an assignment is still valid when *X* and *Y* are not the same type, only if *X* is a subtype of *Y*. Most current languages view a class *X* as a subtype of a class *Y* if *X* is a subclass of *Y* [Snyder 86A, America 87B].

Figure 5 illustrates a class hierarchy and a program using subtyping. Both *Triangle* and *Rectangle* are subclasses of *Polygon*. Since *Triangle* needs a different formula for computing its area, the method *area()* of *Triangle* is semantically different from that of *Rectangle*. Since *Triangle* is a subtype of *Polygon*, the variable *aTriangle* can be assigned to variable *p*. Similarly, the variable *aRectangle* can be legally assigned to *p*. When the last line of the program is executed the object named by variable *p* is either an object of *Triangle* or an object of *Rectangle*. Hence, the binding of *p.area()* cannot be done until run time when the actual object becomes known. Delaying binding until run-time is called *late binding*[1]. The mechanisms of subtyping and late binding allow objects with different method implementations to exibit the same external behavior. The actual method to be performed is determined by the actual type of the object currently named by a variable as demonstrated by the last line.

[1]Late binding is also commonly called dynamic binding, run-time binding, and method dispatching.




```
p:Polygon                   /* variable        */
aTriangle:Triangle;         /* object          */
aRectangle:Rectangle;       /* object          */
...
if ... then
        p:=aTriangle;       /*ok. subtype      */
else
        p:=aRectangle;      /*ok. subtype      */
x:=p.area();            /* late binding     */
```

Figure 5 - Subtyping Example

1.6. Literature Review

1.6.1 Object-Oriented Programming Languages

Object-oriented programming originated with SIMULA [Dahl 66] in the area of computer simulation. Smalltalk-80 [Goldberg 83] brought this programming paradigm to the attention of mainstream computer scientists. Smalltalk-80 demonstrated the power of the message passing metaphor and the significance of class hierarchies in rapid prototyping [Goldberg 83].

Since the introduction of Smalltalk, object-oriented programming has gained popularity within the artificial intelligence community. Many object-oriented language designs are based on AI languages. Flavors [Moon 86] and Loops [Bobrow 86] are examples of Lisp-like object-oriented languages while Vulcan [Kahn 86] is a prolog-based object-oriented language.

As the object-oriented programming paradigm became accepted as a powerful technique, several object-oriented languages aimed at commercial software production were developed. These languages include derivatives of C, such as C++ [Stroustrup 86], which is statically typed and adopts the function call syntax for method invocation, Objective-C [Cox 86], which is dynamically typed and adopts the message passing style of Smalltalk-80, and a new language, Eiffel [Meyer 88], which is statically typed like C++.

SIMULA was the first language which supported both class and inheritance. The language CLU [Liskov 77] supports class but no inheritance. Ada [DoD 83] supports data abstraction using packages. However, an Ada package is not a type. Ada does not support inheritance. Early actor languages support the object notion but without classes [Lieberman 87, Theriault 83].

Design decisions in an object-oriented languages have several dimensions. Some frequently debated issues are static typing versus dynamic typing, flexibility versus efficiency/safety, class versus prototype, and inheritance versus delegation [Lieberman 88]. Trellis/Owl [Schaffert 86], C++, and Eiffel support static typing and class inheritance. Smalltalk, Objective-C, and most Lisp based languages supports dynamic typing and class inheritance. Some researchers prefer prototypes to classes. Prototype based languages include Act1 [Lieberman 87], ThingLab [Borning 86], and Self [Ungar 87]. [Tomlinson 88B] argues that a class is a more useful construct unless few of the objects in a system have common behaviors. [Lieberman 86] proposes delegation as an alternative to inheritance. He argues that delegation is more powerful than inheritance. [Stein 87] showed that delegation and inheritance are equally powerful. [Tomlinson 88B] demonstrates that inheritance require less message traffic than delegation. [Kafura and Lee 89A] argue that inheritance provides a higher degree of reusability than does delegation.

Inheritance is closely tied to the notion of a class. A class captures static properties of objects such as attributes and methods in an explicit form. A new class can be defined as an extension of existing classes with the support of inheritance. Class hierarchies provide a natural classification of components and enhance modularized modification. These expectations have been evidenced in [Goldberg 83, Campbell 87, Johnson 88]. Classes and inheritance are valuable where reusability and maintainability are emphasized more than flexibility. Prototype and delegation based languages are more suitable when the power of dynamic and flexible sharing can be maximally exploited, as in rapid prototyping. The support for classes and inheritance is more natural when strong type-checking and efficient code are favored over flexibility and dynamics [Lieberman 88].

The feature interaction between static typing and inheritance is discussed in [Snyder 86A, Lalonde 86, America 87B, Cook 89B, Canning 89, Lee and Kafura 90].

1.6.2 Actor model

The actor model is a concurrent computation model originally developed by Hewitt with the inspiration of Smalltalk [Hewitt 77]. Hewitt proposed a philosophy which views computation as pure message passing. The original actor model was elaborated on by many other researchers [Hewitt and Baker 77, Hewitt and Atkinson 79, Clinger 81, Hewitt and deJong 83]. Clinger provided a mathematical basis for the actor model with a denotational semantic description [Clinger 81]. The actor model described in this paper is derived from Agha's work [Agha 86].

Many language designs are based on the actor model of computation. Most current message passing based, concurrent object-oriented languages are influenced by the actor paradigm. The languages based on the actor model are Act1 [Lieberman 81], Act2 [Theriault 83], Act3 [Agha 86], and Omega [Attardi 87]. These languages are Lisp derivatives developed for AI problems. They do not support class inheritance.

Other languages which are based on the actor model but include significant modifications are ABCL/1 [Yonezawa 87], Cantor [Athas 88], Lamina [Delagi 88], and Actra [Barry 87]. ABCL/1 provides neither class nor inheritance. The Cantor design is targeted at VLSI based architectures that support fine-grained concurrency. The use of Lamina and the Actra for real-time problems was demonstrated in the signal processing domain. None of these language support the notion of the replacement behavior of Agha's actor model.

1.6.3 Object-Oriented Concurrent Programming Languages

Surveys on concurrent object-oriented programming languages are found in [Tomlinson 88A, Papathomas 89, Yonezawa and Tokoro 87]. The interaction between concurrency and inheritance was noticed by [Briot 87, Wegner 87, Kafura and Lee 89A]. The languages which support both concurrency and inheritance are Concurrent Smalltalk [Yokote 87], and Extended Eiffel [Caromel 88]. Concurrent Smalltalk adopts a low level wait-signal synchronization. The language does not provide tight encapsulation for objects. Extended Eiffel separates synchronization code into a body from the methods, which contain only sequential actions. The body must be completely redefined each time a subclass adds a new method. Other languages with the object notion and concurrency support include ABCL/1 [Yonezawa 87], POOL-T [America 87A], Hybrid [Nierstrasz 87], and Emerald [Black 86].

Chapter 2 A Model of Type and Inheritance

2.1 Introduction

Most statically typed object-oriented languages use subtyping as the basis of type checking. Subtyping rules determine when objects of one type may be used where another type is expected. Many current object-oriented languages consider inheritance and subtyping to be identical. For example, in SIMULA, C++, Trellis/Owl, and Eiffel, a class X is a subtype of another class Y if and only if X inherits from Y. This view of subtyping makes it difficult to support exclusion of an inherited method and multiple implementations of a type. Furthermore a loophole in type safety results. These problems and the difference between inheritance and subtyping are well recognized [LaLonde 86, Snyder 86A, America 87B, Cook 89B, Canning 90].

Method exclusion is motivated by the reuse of existing type definitions. Method exclusion is also implicitly used by method redefinition (See Chapter 3). When a subclass excludes an inherited method from its interface, a type error may occur at run time. Consider the following code fragment:

```
class A { f() {...}, ... };
class B { inherit A; exclude A.f(); ...}
a1:A;
a1 := B.create(...);
a1.f();                    /* dynamic type error */
```

While the invocation of a1.f() is correct from the typing standpoint, the above sequence of statements causes f() to be applied to an instance of B, which does not understand the message because B excluded f() from its definition. Hence, an error occurs at run-time.

Two approaches to dealing with method exclusion is that of CommonObjects [Snyder 86B] and exception mechanisms [Borgida 86, Wegner and Zdonic 88]. CommonObjects supports method exclusion by allowing a class to be a subtype of a class which is not its ancestor in the class hierarchy. Exception mechanisms also allow a class to delete an inherited method. An invocation of a deleted method is viewed as an exception, which triggers a certain recovery action rather than causing an error. While Sina/st [Aksit and Tripathy 88] also supports method exclusion, the language has no notion of subtyping.

Many examples may be found which argue for the flexibility of having several representations of a type, e.g., a regular matrix and a sparse matrix, locations represented in rectangular coordinates and polar coordinates, files stored in disk and tape. A variety of techniques have been used to support multiple representations. In Exemplar Based Smalltalk [LaLonde 86], a class may have different exemplars which are prototype instances having different implementations. However, Exemplar Based Smalltalk provides only run-time type checking like Smalltalk. [Decouchant 89] describes a language which separates types from classes. A type may be implemented by several different classes. Duo-Talk [Lunau 89] also supports multiple representations by separating types from classes. However, Duo-Talk has no class inheritance. [Canning 89] separates the type hierarchy from the class hierarchy where class is not a type. A type may be implemented by multiple classes.

Two extant languages support multiple representations without inheritance. Emerald [Black 87], a statically typed language equating subtyping with interface conformance, provides multiple representations. No inheritance is supported by Emerald. A delegation based language, Act1 [Lieberman 87], relies on message passing and run-time binding to support

multiple representations. Since a variable is not typed, objects having different implementations may be assigned to the same variable.

From these examples, two observations can be made about existing object-oriented languages which support inheritance and static typing. One observation is that method exclusion is a problem when subtyping and subclassing are equated. The other observation is that multiple representation is a problem when a class is both a type and an implementation at the same time. It is impossible to associate a type with multiple implementations if every type is defined by a class.

In this thesis, a new model of type and inheritance, called HANA, intended for a statically typed object-oriented language is proposed. HANA supports multiple inheritance, method exclusion, multiple representation, and method name overloading. In HANA, subtyping is based on both inheritance and interface conformance. Hence, HANA distinguishes subtyping from inheritance, but does not separate them. Rather, a new kind of type, called *abstract type,* is introduced for the support of multiple representations.

Our decision to base subtyping on inheritance (but not to equate them) is motivated by a semantic reason and an implementation concern. Emerald, whose subtyping is based on signature compatibility, can support subtype polymorphism and multiple representations. Signature compatibility is weak in capturing semantic relationships among types, as observed by [America 87B, Liskov 87]. Consider a stack which supports two methods get() and put(Integer), and a queue whose interface also consists of these two methods. A stack certainly is not a queue even if they are signature compatible. Hence, the possibility exists that two signature compatible types are actually incompatible types. As [Snyder 86A, America 87B] explain, a complete solution would be possible only if formal semantic specification techniques are included in a programming language. [Wegner and Zdonic 87]

similarly observe that some levels of behavior compatibility go beyond signature compatibility. However, since the current techniques of formal specification cannot capture all aspects of object behaviors, other alternatives are needed. The expressive power of Emerald subtyping is limited since all types having the same interface are effectively collapsed into a single type. Although not a perfect solution, inheritance offers a middle ground since with inheritance, the difference between stacks and queues can readily be expressed.

The second concern is the cost of supporting multiple representation. While multiple representation provides programming flexibility, its disadvantage is the extra cost associated with method binding. Method binding may be implemented efficiently when subtyping is based on inheritance [Snyder 86A, Stroustrup 87, Lee and Kafura 89]. For example, a language like C++ uses static binding, and even when late binding is needed, it can be done efficiently. In the presence of multiple representations, this implementation advantage is difficult to exploit. Implementation schemes for a type system supporting multiple representations are described in [Black 87, Connor 89]. In both schemes, method binding involves extra mapping between the type of a variable and the type of the object which might be assigned to that variable.

While HANA supports multiple representations, it pays the cost of expensive binding only where the multiple representation feature is used. At an extreme, if all types used in a program have only single implementations, the type system reduces to one which supports no multiple representations, and so no overhead exists. The additional overhead incurred by the use of multiple representation is relatively small if a program is mostly based on types having single implementations.

HANA supports method exclusion as a type definition mechanism without compromising the integrity of the type system. An immediate advantage offered by method exclusion is increased reusability through selective inheritance. The capability of excluding an inherited method is also useful as a mechanism for resolving name collision in multiple inheritance. Chapter 4 shows that the distinction between inheritance and subtyping as used by HANA loosens the difficulty of the name collision problem. Method exclusion together with method overriding and renaming offer a viable solution for the name collision problem in multiple inheritance.

2.2 Abstract Type and Concrete Type

There are two kinds of types in HANA, namely, *abstract type* and *concrete type*. An abstract type is a specification of an *interface*, that is, the set of operations applicable to its objects. Each operation in the set is denoted only by a method name and its signature. We use the term *method specification* to refer to the name and signature of a method. An abstract type definition is only an interface and has no attached implementation. In contrast, a concrete type is a combination of an interface and a representation. The notion of concrete type is equivalent to that of class in other models. Each class definition yields a new concrete type.

2.3 Inheritance

An abstract type may inherit from other abstract types. Similarly, a class may inherit from other classes. The inheritance between abstract types is called *interface inheritance* while the inheritance between concrete types is called *class inheritance*. Hence, two distinct inheritance hierarchies exist in HANA: an *interface hierarchy* and a *class hierarchy*. These two different hierarchies are related by the third kind of inheritance, called *specification inheritance*. HANA assumes multiple inheritance in both interface inheritance and class inheritance. In order to concentrate on the idea of separating interface hierarchy from class

hierarchy, the discussion in this section ignores the problem of name collision. The issue of name collision is discussed in Chapter 4.

2.3.1 Interface Inheritance

An abstract type may inherit methods from other abstract types, add new methods, or delete inherited methods. The definition of an abstract type T has the form:

```
Interface T {
        inherit     S1,S2,...,Sk;
        exclude     f1,f2,...,fm;
        add         g1,g2,...,gn;
}
```

The set of methods in T is computed by the following set algebra:

$$T = S1+S2+...+Sk - \{f1,f2,...,fm\} + \{g1,g2,...,gn\}$$

where "+" and "-" denote set union and set subtraction, respectively.

2.3.2 Class Inheritance

A class definition may inherit from other classes, delete some inherited methods, adding new methods, or rename inherited methods. Furthermore, a new method overrides all inherited methods that have the same name and signature.

Rule 1 - Overriding of an inherited method:
A new method f overrides an inherited method g if and only if f has the same specification as g.

This method overriding rule is different from that of other languages which define it in terms of conformance [Schaffert 86, Meyer 88]. This new overriding rule is motivated by

HANA's support of method name overloading. The significance of this new overriding rule is discussed in Chapter 3. In HANA, the overriding rule of other languages can be expressed using a combination of exclusion and addition of methods. An example class definition is given below:

```
Class T1 {
        inherit C1, C2;             /* inherit from classes C1 and C2 */
        exclude C1.f1, C2.f2;       /* exclude f1 of C1 and f2 of C2  */
        add
                f3() { ...}         /* new method                     */
}
```

The set of methods in the interface of T1 is computed by

$$T1 = (C1 - \{f1\}) + (C2 - \{f2\}) + \{f3\}$$

where + and - denote set operations.

2.3.3 Specification Inheritance

A class definition may specify an abstract type in its inheritance list. In this case, the class inherits a specification from the abstract type. Further, this kind of inheritance is called *specification inheritance*. In contrast to interface inheritance and class inheritance, specification inheritance only associates a class with an abstract type of which the class is an implementation. By definition, the interface of a class inheriting a specification of an abstract type must satisfy the specification of the abstract type.

Rule 2 - Spectification Inheritance:
If a class inherits a specification from an abstract type, the interface of the class must conform to the abstract type.

2.4 Typing

HANA is intended for a statically typed language. In such a language, the type of each expression is determined by the analysis of the program text such that if a program is accepted by the compiler, no object is ever requested to perform an unknown operation. As with most other statically typed languages, the static type-checking is based on the explicit type declaration of each identifier. In HANA, every object is an instance of some class, which determines the object's type.

Rule 3 - Type:
The type of an object is the class of which it is an instance.

The compatibility of assignment is defined by subtyping:

Rule 4 - Assignment Compatibility:
An object or variable of type X can be assigned to a variable of type Y if and only if X is a subtype of Y.

Argument passing is defined similarly.

Rule 5 - Argument Passing Compatibility:
An object of type X can be passed as an argument of type Y if and only if X is a subtype of Y.

2.5 Subtyping

In HANA, subtyping is based on two conditions: *inheritance* and *conformance*. The subtype relation between two types is defined as follows:

Rule 6 - Subtype:
A type X is a *subtype* of another type Y if and only if one of the following holds:
(1) X and Y are identical
(2) X *inherits from* Y and X *conforms to* Y.

Further, if X is a subtype of Y, we call Y a supertype of X. The definition of conformance follows.

Definition:

> Type X *conforms to* type Y if and only if for any method y in Y, there exists a method x in X such that x conforms to y.

The conformance between two methods is defined by the well-known rule of contravariance [Cardelli 84]:

Definition:

> Method f(S1,S2,...,Sn):T *conforms to* method f(U1,...,Un):V if and only if type Si is a supertype of Ui for all i in {1,...n} and type T is a subtype of V.

According to the HANA definition of subtype, a type X conforming to another type Y is not always a subtype of Y. Furthermore, a subtype relation is not automatically established by inheritance. Since the subtype relation is a more constrained form of the inheritance relation, the subtype relation is a subset of the inheritance relation and also a subset of the conformance relation. In general, the type hierarchy is embedded within the inheritance hierarchy, which contains types related not only by subtype but also by inheritance with no conformance.

2.6 Multiple Representation

Multiple representation means that if two different classes implement the same abstraction, their instances may be assigned to the same variable or passed as arguments to the same method, even if no inheritance relation exists between them. In HANA, multiple representation is supported by abstract type and specification inheritance. An abstract type may have several different implementations. Each implementation of an abstract type is a

class. Specification inheritance indicates which classes implement a given abstract type. Since a class inheriting from an abstract type always conforms to the abstract type by definition, the class is a subtype of the abstract type. When a class inherits from more than one abstract type, the class is a subtype of each of these abstract types.

A variable or parameter declaration may use either an abstract type or a concrete type (class) as a type designator. There is a significant difference between declaring a variable as a class type and as an abstract type. For example, consider the inheritance hierarchy and the program shown in Figure 6. X is an abstract type while Y and Z are two concrete types implementing X. W is a subtype of Y. Variable y may be assigned only the instances of Y or W. In contrast, the set of objects which can be assigned to x includes not only those objects, but also instances of class Z.

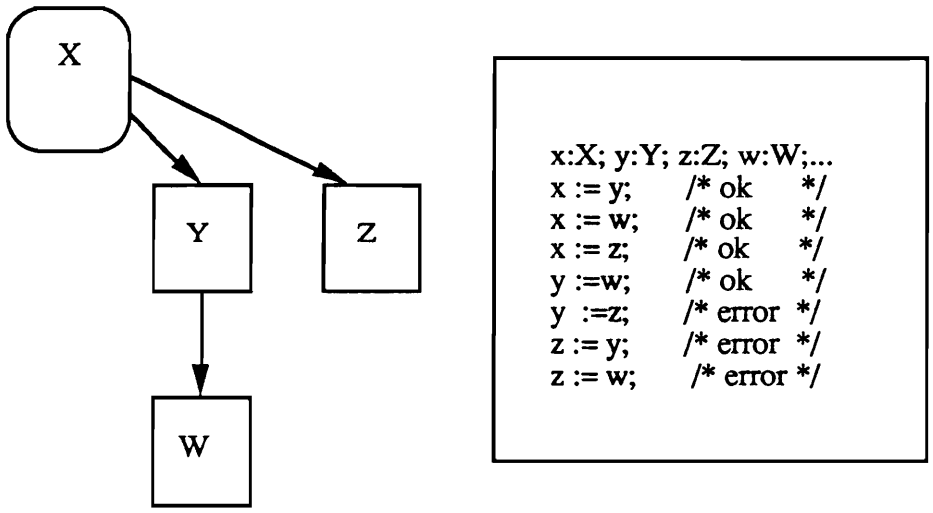


Figure 6 - An Example of Multiple Representation

Two interesting cases exist: an abstract type with no implementations and a class inheriting from no abstract types. An abstract type without an implementation may be useful for classifying behaviors. Such abstract types factor the common aspects of several similar

behaviors. A class definition with no abstract type in its inheritance list is equivalent to a class found in other object-oriented languages. Some programs need only a class hierarchy, in which case method invocations can be efficiently implemented [Lee and Kafura 89]. Even when a program needs the support of multiple representation, more expensive binding overhead is limited to invocations on abstract type variables. A program can be efficient when class types are dominant.

2.7 Method Exclusion

In HANA, a type may exclude or rename inherited methods. The restriction on a class definition is that if the class inherits from an abstract type, the net effect of exclusion and renaming must result in a class whose interface conforms to the abstract type. Consider the following definition of an abstract type Stack.

```
Interface Stack {
        empty():Boolean;
        top():Integer;
        push(Integer);
        pop():Integer;
}
```

Figure 7 illustrates two representations of Stack: ArrayStack and ListStack. Class ArrayStack is an implementation using an array. Class ArrayStack has no superclass. We assume that only methods are exported by a class. Hence, ArrayStack exports empty():Boolean, top():Integer, push(Integer), and pop():Integer, which exactly match the set of methods defined by the abstract type Stack.

Class ListStack implements Stack using a list data structure. Let us assume that ListQueue is a class which has an interface of Queue. Since ListStack inherits from ListQueue, ListStack inherits all methods exported by ListQueue. Among these, methods front():Integer and get():Integer are renamed to top():Integer and pop():Integer, respectively. ListStack excludes the inherited method enqueue(Integer) and adds a new method push(Integer):Stack. The resulting interface of ListStack is {empty():Boolean, top():Integer, push(Integer), pop():Integer}, which satisfies the requirement of a Stack interface.

```
Class ArrayStack is Stack {
        current: Integer;
        store: Array [1..MAX] of Integer;
        empty(): Boolean {...}
    top(): Integer {...}
    push(item: Integer) {...}
    pop(): Integer {...}

Class ListStack is Stack {
    inherit     ListQueue;
    exclude     ListQueue.enqueue;
    rename      ListQueue.front()/top, ListQueue.get()/pop;
    add
        push(item: Integer) {...}
}
```

Figure 7 - Definitions of ArrayStack and ListStack

The above example shows the power of differentiating subtyping and inheritance. A class may rename or exclude an inherited method without breaking subtyping constraints. In the above example, while ListStack uses both renaming and exclusion of inherited methods,

their use does not result in a typing error since ListStack is not a subtype of ListQueue. Reusability is enhanced because a class can reuse any other classes by selectively inheriting their methods without regard to possible type implications.

Chapter 3 Method Name Overloading

3.1 Introduction

Overloading of names is traditionally called ad hoc polymorphism [Wegner and Cardelli 85]. While most programming languages provide overloading to a certain extent, the importance of overloading as an explicit language feature has been demonstrated by several recent language designs. Two notable ones are ADA and C++. Overloading enhances the expressiveness as well as the extensibility of a language. These benefits are the main motivations behind the support of overloading in most current languages.

This chapter examines the role of overloading in a statically typed object-oriented language. Overloading plays two significant roles. First, overloading extends the expressive power of an object-oriented language. As presented in this work, overloading accommodates a new overriding rule which is different from that of Trellis/Owl and Eiffel. The rule accepts many subtype relations which other type systems without overloading cannot. This rule is described and its power is demonstrated. Second, overloading offers a natural solution for many name collision problems which occur in object-oriented languages. The issue of name collision is discussed in the next chapter.

The observation presented in this thesis argues for the support of overloading in object-oriented languages; however, inheritance interferes with overloading. This problem of the feature interaction between inheritance and overloading and a solution for the problem are presented.

3.2 Overloading Is Not Ad Hoc

Consider the following problem, which is adopted from [Canning 89]. Suppose a class called Point is already defined. Class Point is a definition of point objects in rectangular coordinates. One of the methods defined by the Point class is equal() which takes an argument of type Point and determines if the receiver object and the argument have the same coordinates. The definition of Point might look like:

```
Class Point {
        ...
        x():Real { return x-value; }
        y():Real { return y-value; }
        equal(p1:Point):Boolean { return (x()=p1.x() and y()=p1.y()); }
}
```

Suppose that we want to create a new class, ColorPoint, which inherits from Point. Class ColorPoint adds an instance variable for color and overrides the equal(Point) method with its own version of the equal method, equal(ColorPoint). The new equal method takes an argument of type ColorPoint and checks if the receiver object is equal to the argument object in both location and color. Class ColorPoint is defined as follows:

```
Class ColorPoint {
        inherit Point;
            color : ColorType;
        add equal(c1:ColorPoint):Boolean;
        { return (x()=c1.x() and y()=c1.y() and color()=c1.color());}
}
```

An interesting question arises. Is ColorPoint a subtype of Point?

Most type systems interpret the addition of equal(ColorPoint) as a redefinition of equal(Point). For example, in Trellis/Owl, C++, and Eiffel, equal(ColorPoint) overrides equal(Point). As a result, equal(Point) is replaced by equal(ColorPoint) in the interface of ColorPoint. Note that equal(ColorPoint) does not conform to equal(Point). Hence, ColorPoint is not a subtype of Point. However, in most type systems where subtyping is equated with inheritance, ColorPoint is considered a subtype of Point. A type correct program in these type systems may generate a type error at run-time. Consider the following example:

```
p1, p2: Point; c: ColorPoint;
...
p1          := c;
equality := p1.equal(p2) then
...
```

Although this program is type correct, if ColorPoint is assumed to be a subtype of Point, the invocation p1.equal(p2) causes equal(ColorPoint) to be applied to a Point object, which is a run time error. This problem was first noticed by Cook [Cook 89A].

For type safety, the definition of ColorPoint should not be viewed as a subtype of Point. However, a language which equates subtyping with inheritance automatically considers ColorPoint as a subtype of Point and thus cannot be type safe. Some type systems disallow the definition of a class like ColorPoint in order to avoid a run-time error. For example, Trellis/Owl and Eiffel require a redefinition of a method to conform to the method being redefined [Schaffert 86, Meyer 88]. A disadvantage associated with this approach is

reduced reusability. ColorPoint cannot inherit from Point but must be defined from scratch even though ColorPoint and Point have almost the same implementation and behavior.

[Canning 89] proposes a solution to this problem by separating inheritance from subtyping. Class is not a type. The separation allows ColorPoint to inherit Point. In this scheme, however, ColorPoint, which is a subclass of Point, is not a subtype of Point since class is not a type. A drawback of this approach is the implementation overhead incurred by the separation of class and type. Binding becomes expensive since a type lacks information about how objects are implemented. Furthermore, the scheme totally excludes the possibility of viewing ColorPoint as a Point.

The above example suggests that it is not possible to define ColorPoint as a subtype of Point without sacrificing type safety in current type systems. This is unfortunate since ColorPoint is almost like Point (even their names suggest that.). There are many similar situations. In general, a similar problem occurs if a class implements a method which takes an argument of its own type. While type safety dictates that ColorPoint should not be a subtype of Point, the cost is flexibility. However, viewing a ColorPoint as a Point is natural and useful. It should be possible to define ColorPoint to be a subtype, if desired. Overloading offers a solution to this problem.

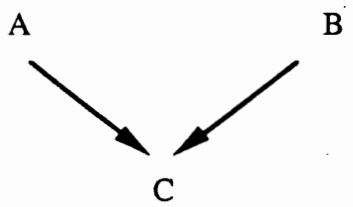
The problem occurs because method overriding effectively excludes an inherited method. In the ColorPoint example, addition of equal(ColorPoint) causes the equal(Point) method to be excluded from the interface of ColorPoint. As discussed in Chapter 2, method exclusion cannot be supported without violating typing constraints where subtyping is equated with inheritance. Hence, the definition of ColorPoint should not be allowed for type safety reasons; otherwise, ColorPoint should not be treated as a subtype of Point as in [Canning 89].

In HANA, the addition of equal(ColorPoint) does not override equal(Point). This is enabled through overloading which distinguishes the two seemingly equal methods. Since their parameter types are different, the interface of ColorPoint contains both equal(Point) and equal(ColorPoint). Hence, ColorPoint defined as above is a subtype of Point in HANA.

3.3 Interaction between Overloading and Parameter Subtyping

While the support of overloading is important, overloading tends to interfere with parameter subtyping. For example, if a type contains two methods that are not identical but have the same operation name and the same number of parameters, there is potential ambiguity in method binding. This section describes the feature interaction between overloading and parameter subtyping.

Consider types A, B, C, and T shown in Figure 8. Type C is a subtype of both A and B. Type T defines two overloaded methods, f(A):G and f(B):G. Let c be a variable of type C and consider a method invocation of the form f(c) applied to an object of type T. Note that f(c) is a legal invocation of both f(A):G and f(B):G since C is a subtype of both A and B. To which one should the invocation f(c) be bound?

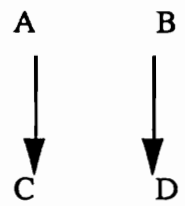


T = { f(A):G, f(B):G,... }

Figure 8 - Interaction between Overloading and Multiple Inheritance

While it may seem that the interaction occurs only with multiple inheritance, a similar problem also exists in a single inheritance model. For example, consider the types shown in Figure 9. Type C is a subtype of type A while type D is a subtype of B. Suppose that another type T contains two methods f(A,D):G and f(C,B):G. Consider variables c and d whose types are declared as follows:

c : C; d : D; t : T;



T = { f(A,D):G, f(C,B):G,... }

Figure 9 - Interaction between Overloading and Single Inheritance

Is t.f(c,d) an invocation of R.f or S.f? The invocation f(c, d) is valid syntax for both alternatives if subtyping of parameters is supported. It is ambiguous which operation should be performed by the invocation.

Hence, feature interaction occurs between subtyping and overloading in both single inheritance and multiple inheritance. In formalizing these problematic situations, the assumption is made that the desired result type of each method invocation can be inferred from the context of the invocation [Baker 82].

3.4 Binding Ambiguity

In order to capture all possible type definitions which cause ambiguity in method binding, the term *binding ambiguity* is formally defined:

Definition:

> If an invocation f(a1, a2, ..., ak) applied to an object of type T has more than one method which can be legally bound to the invocation, type T is said to have *binding ambiguity*.

Lemma

A type T has binding ambiguity if and only if T contains two (overloaded) methods f1(S1,S2,...,Sn):R1 and f2(T1,T2,...,Tn):R2 such that for each i in {1, 2, ..., n }, there exists a type Ui such that Ui is a subtype of both Si and Ti and there exists a type which is a supertype of both R1 and R2.

Proof

Suppose that a type T has binding ambiguity. By the definition of binding ambiguity, there exists an invocation syntax f(u1,u2,...,uk) which can be bound to some methods f1(S1,S2,...,Sk):R1 and f2(T1,T2,...,Tk): R2 belonging to T. Let M denote the desired type of the expression f(u1,u2,...,uk). Then the type of ui is a subtype of Si for each i in {1,2,...,k} and R1 is a subtype of M since f1(u1,u2,...,uk) can legally replace f(u1,u2,...,uk). Let Ui denote the type of ui for each i in {1,2,...,k}. Then Ui is a subtype of Si for each i in {1,2,...,k}. Note that Ui is also a subtype of Ti for each i in {1,2,...,k}. Furthermore, R2 is a subtype of M since f2(u1,u2,...,uk) is a legal binding for f(u1,u2,...,uk). Hence, there exist a type Ui which is a subtype of both Si and Ti for each i in {1,2,...,k}, and type M which is a supertype of R1 and R2.

To prove the reverse direction, suppose T has two methods f1(S1,S2,..., Sn):R1 and f2(T1,T2,...,Tn):R2 and there exist a type Ui which is a subtype of both Si and Ti for each i in {1,2,...,n}, and a type M which conforms to R1 and R2. Let ui denote an object of type Ui for each i in {1,2,...,n}. Invoking f1 with arguments u1,u2,...,un will generate a result of type R2, which is a subtype of M. Hence, f(u1,u2,...,un) is a legal invocation for f1 if the invocation appears where an object of type M is expected. Similarly, f(u1,u2,...,un) can be a legal invocation for f2. Therefore, by definition, T has binding ambiguity.

3.5 Resolution of Binding Ambiguity

One approach is to completely disallow naming conflicts to occur [Snyder 86B]. Such a rule excludes many useful type hierarchies since a type with binding ambiguity cannot be defined. Binding ambiguity is not a problem until the potentially ambiguous method is actually invoked. Hence, excluding certain type definitions simply because they have binding ambiguity is too restrictive. Rather, the compiler should detect an ambiguous <u>invocation</u> while type <u>definitions</u> with binding ambiguity should be allowed. If an ambiguous invocation exists, the programmer may resolve the ambiguity by providing more specific type information. We have found an intuitive rule which resolves many ambiguous invocations. The rule is based on the fact that "conform to" relation is a partial ordering.

Rule 7 - Resolution of Binding Ambiguity:

> **Let F be the set of methods which can be legally bound to an invocation t.f(a1,a2,...,an). The invocation is bound to f in F such that f conforms to any g in F, if such f exists. Otherwise, the invocation is a type error.**

Our binding rule selects the greatest lower bound of F. If this is not the desired binding, the programmer should narrow down the choice by providing more specific type information.

For example, the programmer may cast the type of an argument to a super type to resolve binding ambiguity. In this way, we can have the flexibility of being able to select a specific one among several possible bindings without constraining the set of legal type definitions.

The example program in Figure 10 illustrates how this rule resolves binding ambiguity. We are using the earlier definitions of Point and ColorPoint classes. Since type ColorPoint has two equal methods with different signatures, it has binding ambiguity. While the invocation in line 6 has two possible bindings, equal(Point) and equal(ColorPoint), the resolution rule selects equal(ColorPoint) since the type of the input argument (cc2) is ColorPoint. The invocation in line 7 is bound to equal(Point). Line 9 and 10 are the invocations of equal on a Point variable which currently contains an instance of ColorPoint. Since the interface of Point contains only one equal method (equal(Point)), both invocations are bound to that equal method. Hence, while ColorPoint is a subtype of Point, no type errors were introduced.

```
1    pp1, pp2:Point;  cc1, cc2:ColorPoint;  v1,v2, v3, v4:Boolean;
2    pp1:=Point.create(0,3);
3    pp2:=Point.create(0,3);
4    cc1:=ColorPoint.create(0,3,R);          /* color = red        */
5    cc2:=ColorPoint.create(0,3,B);          /* color = blue       */
6    v1 := cc1.equal(cc2);                   /*equal(ColorPoint)*/
7    v2 := cc1.equal(pp1);                   /* equal(Point)       */
8    pp1 := cc1;
9    v3 := pp1.equal(pp2)                    /* equal(Point)       */
10   v4 := pp1.equal(cc2)                    /* equal(Point)       */
```

Figure 10 - An Example Resolution of Binding Ambiguity

A question which might arise at this point is whether the particular bindings chosen by the resolution rule of HANA are meaningful. The interpretation of the bindings used in line 6 and 9 are obvious. The invocations in line 7 and 10 checks on the equality of the locations of the receiver object and the argument object.

Chapter 4 Name Collision

4.1 Introduction

Inheritance is the distinguishing feature of object-oriented programming. Inheritance provides such significant benefits as reusability, maintainability, code sharing, and subtyping. While multiple inheritance, i.e., the capability of inheriting from more than a single parent, is a more powerful mechanism than single inheritance, which allows a type to inherit from only one parent, two problems have discouraged the support of multiple inheritance [Cox 86]: name collision and implementation inefficiency. This chapter focuses on name collision while Chapter 5 presents a new technique for efficiently implementing multiple inheritance. The following discussion concentrates on why name collision is a problem in object-oriented programming.

As programming tasks become more and more complex, a programmer typically relies on techniques such as decomposition to help manage complexity. In object-oriented programming, a programming task is naturally decomposed into a set of abstract data types or classes. The design technique based on decomposing a task into abstract data types is a powerful programming methodology for coping with complexity [Liskov 87]. The modular definition encouraged by abstract types allows a program to be built and understood through local reasoning on separate types or modules.

In order for this decomposition technique to be effective, a programmer defining a new type should not be concerned with how other types are defined. The names used in other type definitions should not concern the programmer. In other words, names used in a type should be local to that type. The need for locality of names is more pronounced when

software is developed by several programmers in a team or when program libraries are shared by many programmers.

Modular type definition creates a large number of local name spaces. A method applied to an object is locally evaluated by that object. Where inheritance is not supported, method names remain local to the type which defines them. However, when a type can inherit methods from other types, the names of the inherited methods are moved into the scope of the inheriting type. Hence, inheritance weakens the locality of a method name by broadening its scope. Inheritance causes a method name defined in one name space to be observed in another name space. A type inheriting from multiple parents can see the names of all inherited methods in its own name space. Note that weakening method name locality is different from violating encapsulation. Weakening of locality still exists even when complete encapsulation is supported.

In multiple inheritance, name collision occurs if a type inherits from two independently developed types having a method with a common name. A similar problem exists in single inheritance as well [Knudsen 88]. Hence, we observe that in object-oriented programming, where inheritance is essential and modular type definition is highly encouraged, name collision is unavoidable and common rather than exceptional.

The HANA model resolves the name collision problem without compromising the integrity of the type system. The capability of excluding inherited methods without violating static typing offers a sound solution for resolving name collision in multiple inheritance when used together with other mechanisms. It is shown that the mechanisms of method exclusion, addition, and renaming are orthogonal with respect to the power of resolving name collision in multiple inheritance. It is also described how the support of method name overloading helps resolve many name collision problems.

An example name collision problem is shown in Figure 11.

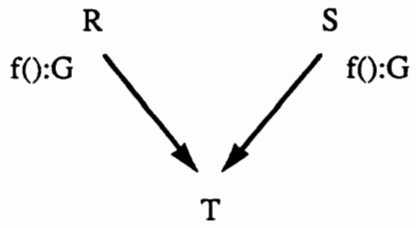


Figure 11 - Name Collision

Suppose that method f() is implemented by two classes R and S which are the parents of class T. Consider the following scenario:

```
var t: T;
        t = T.create();        /* create an instance of T */
        t.f();                 /* which f()? */
```

Ambiguity arises with regard to which f() should be invoked by t.f(). A similar scenario can be constructed for abstract types.

Trellis/Owl [Schaffert 86] and extended Smalltalk [Borning 82] consider a class inheriting methods with the same name from two different parents as an error unless the two methods are actually the same operation, in which case the inherited methods are automatically merged. This approach has two disadvantages. First, many classes which can be defined with inheritance will be rejected. Second, automatic merging is undesirable. There are at least two serious problems which make automatic merging undesirable in class inheritance.

On the one hand, two methods with the same specification may be two different operations which have the same specification by accident. In this case, merging the two methods will cause one of them to be excluded from the subclass. The implication of excluding an inherited method is discussed further in this section. On the other hand, even when they actually denote the same implementation inherited twice via different paths, automatic merging breaks the encapsulation of the superclasses by making the use of inheritance in the superclass visible to the subclass. This problem was first noticed by Snyder [Snyder 86A]. For this reason, CommonObjects [Snyder 86B] explicitly forbids a class from inheriting two methods with the same name, regardless of their implementations.

Clearly, automatic merging is not desirable for class inheritance; therefore, it is necessary to resolve name collision explicitly in order to bind each method invocation to a unique method. There are three possibilities:

1. Method Exclusion,
2. Method Overriding,
3. Method Renaming.

Languages can be found which use each of the three mechanisms. Method exclusion is implicitly used by Flavors, CommonLoops, and Sina/st, in which an ambiguous invocation is always bound to a fixed method among the conflicting methods. Each of these languages uses a precedence rule to determine a unique method. Multiple inheritance in C++, Trellis/Owl, and Smalltalk uses method overriding. Eiffel uses both method renaming and overriding. While a language may use one or more of these three approaches, there are subtle but significant differences in their use. These mechanisms are orthogonal and resolve name conflicts in different ways.

In the above example, the two versions of f() have the same signature. This example is a special case of a more general problem. In general, a type may inherit methods which have a common name but different signatures. The following sections concern conflicting methods having the same signature. The general form of name collision is resolved by the support of overloading and is discussed in Section 4.7. Since the semantics of interface inheritance is different from that of class inheritance, the following discussion considers name collision in interface inheritance and that in class inheritance separately.

4.2 Name Collision in Abstract Types

Resolving name collision in interface inheritance is simple. An abstract type is merely an interface whose methods have no attached implementations. Since two methods with the same specification are not distinguishable, only one of them is contained in the type. The treatment of an interface as a set of methods correctly captures this idea. Hence, the following rule, referred to as *automatic merging*, is used for abstract types.

Rule 8 - Automatic Merging:

> **If an abstract type inherits two methods with an identical specification, the type contains only one of them.**

4.3 Name Collision in Concrete Types

The HANA model assumes a complete encapsulation of a class. An instance variable defined in a class can be accessed only through its methods. The assumption is made that for each instance variable, a pair of operations, one for reading and the other for writing the variable, are automatically provided. In such a system, name collision involves only method names, and hence, it is sufficient to consider only collision between method names.

The following definitions of class X and Y are used for the comparison of the three approaches:

```
Class X {
        ...
        f() { ... };
}

Class Y {
        ...
        f() { ... };
}
```

4.4 Resolution by Exclusion

In this approach, a child class may exclude any inherited methods so that every method name in its interface becomes unique. For example, consider the following definition:

```
Class T {
        inherit X, Y;
        exclude X.f();
        ...
}
```

Observe that T should not be a subtype of X. Otherwise, a serious problem arises. Assuming that the above definition makes T a subtype of X, consider the following program.

```
t:T; x:X;
t = T.create();
x = t;
...             /* invocations of X's methods on x     */
x.f();          /* invocation of Y.f()                 */
```

Since X.f() is excluded from T's interface, the f() applied to an instance of T always invokes Y.f(). It is not possible to invoke X.f() on the instances of T. However, other methods of X are still available in the interface of T. If the method f() in X and Y are implemented by different classes, the use of Y.f() together with other methods of X on the same object may break invariants assumed by the methods of X. Hence, this use of subtyping will cause objects named by x to be inconsistent. The method intended by x.f() is X.f(), not Y.f(). This example shows that T should not be considered a subtype of X.

The mechanism of method exclusion is useful when a subclass inherits methods of a superclass selectively. In this situation, the subclass is not and does not need to be a subtype of the superclass. However, method exclusion does not allow T to be a subtype of both X and Y. Hence, method exclusion alone does not completely solve the problem of name collision.

4.5 Resolution by Overriding

The second approach is to define a new method in the subclass which overrides conflicting methods. Continuing with the above definitions of X and Y, consider the following definition of T.

```
Class T {
        inherit X, Y;
        f() { Y.f(); };    /* override f() defined in X and Y */
        ...
}
```

In this definition, T defines its own f() which overrides X.f() and Y.f(). The new method f() simply invokes Y.f() so that an invocation of f() on the instances of T always has a unique binding. Since the new f() defined by T overrides the f() of both X and Y, T conforms to both X and Y, and should be considered a subtype of both. While the inconsistency problems discussed above might occur, the view is taken that this is what the programmer intends. Otherwise, method exclusion should be used instead of method overriding. While the new f() in this example simply invokes Y.f(), it may have any definition. For example, the new f() might invoke both X.f() and Y.f(), which is not possible with method exclusion.

Although method overriding offers some flexibility which method exclusion does not, method overriding cannot replace method exclusion. Method overriding alone cannot solve the problem of an inconsistent object, which is resolved by method exclusion. Yet, these two mechanisms are not sufficient to completely solve the name collision problem. Neither allows the desired version of f() to be selected.

4.6 Resolution by Renaming

Method renaming solves the last problem. With method renaming, T might be defined as follows:

```
Class T {
        inherit X, Y;
        rename X.f()/myf;  /* rename X.f() by myf */
        ...
}
```

The claim is made that both X.f() and Y.f() can be invoked on the instances of T, and the inconsistency problem does not exist with this approach. Consider the program in Figure 12, which uses the above definition of T. The number at the left of each line is a line number.

```
1    x:X; y:Y; t:T;
2    t := T.create();
3    x := t;          /* T is a subtype of X    */
4    y := t;          /* T is a subtype of Y    */
5    t.f();           /* invoke Y.f()           */
6    t.myf();         /* invoke X.f()           */
7    x.f();           /* invoke.X.f()           */
8    y.f();           /* invoke Y.f()           */
```

Figure 12 - Resolution by Renaming

The call t.f() in line 5 invokes Y.f(). Note, however, that X.f() can also be applied to an instance of type T as shown in line 6. Since X.f() has been renamed to myf within T, one can refer to X.f() through the new name myf.

The beauty of the renaming mechanism is demonstrated in line 7. Note that x.f() is bound to X.f() rather than Y.f(). If an instance of T is treated as an object of type X so that all

operations applied to that object may be those of X, the other two mechanisms do not help. Obviously, method renaming can not replace exclusion and overriding.

In summary, the claim that the mechanisms of method exclusion, overriding, and renaming are orthogonal to each other in resolving name collision in multiple inheritance has been substantiated through the previous examples.

4.7 Resolution by Overloading

The previous focus has been on the name collision problem in which two conflicting methods have the same specification, but this problem is a special case of a more general problem. This section considers the name collision problem in a more general setting. Having already considered the case in which two conflicting methods have the same specification, our discussion in this section addresses only the other case, that is, when two inherited methods have a common name but different signatures.

The support of method name overloading is essential for addressing the general case. While renaming may be used for resolving name collision, renaming is too restrictive since it may be difficult sometimes to find a new name which is intuitive and descriptive. For example, consider inventing a different name for the "+" operator. Further, even if a new name can be found, it requires extra effort by the programmer. Thus, it is desired to keep the number of situations which require renaming as small as possible.

The capability of overloading method names offers a viable solution for name collision when conflicting methods have different signatures. With the support of method name overloading, two methods with the same name but different signatures are considered distinct. Hence, two methods with different signatures do not cause name collision even if

they have the same operation name. Not only does this save programming effort, but it also improves program expressiveness.

While overloading resolves name collision when conflicting methods have different signatures, one issue requires discussion: *binding ambiguity*. Binding ambiguity exists when a class inherits two methods having an identical name and the same number of parameters but different parameter types. Assuming that B is a subtype of A, consider the following example.

```
class X { f(A):R {...} }
class Y { f(B):R {...} }
class T { inherits X, Y; ...}
```

In this example, class T inherits from X and Y. Hence, T has two f's with different argument types. Consider the invocation t.f(a1) where t is a variable of type T. If a1 has type A, f(B):R cannot be used for this invocation, and hence, f(A):R of class X should be selected. However, if the type of a1 is B, then both X.f and Y.f may be used for the invocation. While this is an example of binding ambiguity arising from name collision, which has not been addressed separately, the rule introduced in Section 3.5 is sufficiently powerful to resolve this kind of binding ambiguity. Our binding rule selects Y.f whose parameter type matches the type of the argument used in the invocation. If the desired method is X.f, not Y.f, the programmer must provide more specific information. For example, an invocation like t.f((A)a1) should be used where (A)a1 is meant to cast the type of a1 to A. In this way, the flexibility of being able to select a specific one among many possible bindings is made available.

Chapter 5 Hierarchy Partitioning: An Efficient Late Binding Mechanism

5.1 Introduction

Another difficulty in supporting multiple inheritance is the overhead associated with late binding of methods. Late binding of methods is required when redefinitions of inherited methods and subtyping are supported. While an efficient mechanism is available for the implementation of single inheritance, a comparable technique for multiple inheritance has been sought. Most reported techniques are either too inefficient or provide only partial support.

This chapter presents a fast and efficient late binding mechanism for statically typed multiple inheritance object-oriented languages. The mechanism is a generalization of the virtual function table approach pioneered by single inheritance in C++ [Stroustrup 87A]. This new technique, called *Hierarchy Partitioning*, provides an efficiency close to that of single inheritance C++ for the implementation of multiple inheritance.

This chapter assumes that the reader is familiar with object-oriented programming languages, preferably, a statically typed object-oriented language like C++ [Stroustrup 86] or Eiffel [Meyer 88]. While the HANA model is used in the following discussion, note that the techniques described in this chapter are general and not limited to HANA.

The remainder of this chapter is broken into the following sections. Section 5.2 briefly describes the common index technique, a fast binding mechanism for single inheritance. Section 5.3 examines the possibility of extending the common index technique to multiple inheritance. Several problems associated with the approach is noted. Section 5.4 introduces the hierarchy partitioning technique by an example. Section 5.5 describes how

the new technique is used in program translation. Section 5.6 provides a more rigorous description of Hierarchy Partitioning. An analysis of the expected performance of Hierarchy Partitioning is also given in this section. Section 5.7 is a comparison with other proposed mechanisms.

5.2 Late Binding for Single Inheritance

The fast method dispatching technique for single inheritance described in this section is used in C++ [Stroustrup 87]. In single inheritance, method dispatching is done by indexing into a run-time table, called the *method table*. Figure 13 shows a single inheritance hierarchy and the associated method tables. A method table is created for each class at compile time. The method table of a class is an array of addresses of methods which are available to its instances and may be redefined by its subclasses. The method table of a subclass extends its parent's method table with additional entries for new methods defined by the subclass. A new method which redefines an inherited method uses the entry of the method being redefined. For example, the method table of class C in Figure 13 contains only one entry for f(), which points to the f() defined by class C.

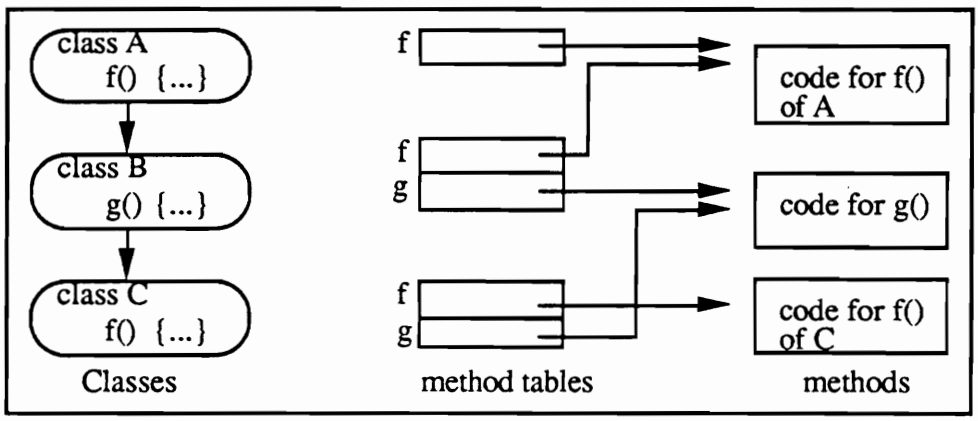


Figure 13 - Method tables in Single Inheritance Hierarchy

The compiler generates an index for each method invocation. At run time, the index is used to locate the address of the method to be executed in the method table of the receiver object. Note that this fast binding via an indexing operation is enabled by the property that two methods having the same name use a common index.

Static Binding

A strategy for detecting a method which is allowed to be redefined by a subclass is to use an explicit keyword for such a method in the definition of a class. This technique has been successfully used in C++. The programmer specifies the keyword "virtual" in the declaration of a method that may be redefined by a subclass. Static binding is used for all non-virtual functions. Static binding may also be used for non-virtual functions with multiple inheritance. Since an implementation of multiple inheritance may also use static binding for non-virtual methods, the following discussion of multiple inheritance is concerned only about late binding of virtual methods.

5.3 Late Binding for Multiple Inheritance

The common index technique which enables fast method dispatching for single inheritance does not work for multiple inheritance. Consider the classes shown in Figure 14. A redefined method replaces the entry occupied by the method being redefined as in single inheritance. Note that method b has index 0 in B's table while its index in C's table is 1. When class B is compiled, it is not known what b's indices will be in its subclasses. Hence, the indexing technique is not directly applicable to multiple inheritance.

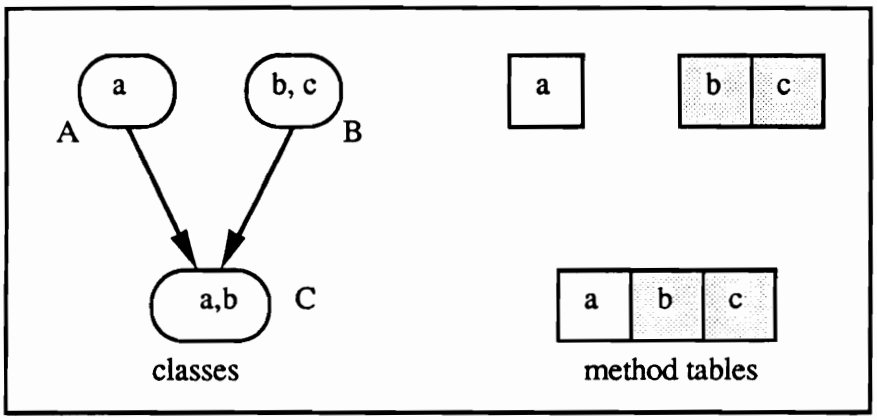


Figure 14 - Method Table with Shared Entries

Figure 15 is an attempt to fix the problem described above. A separate entry is used for each different definition in a method table. For example, the method table of C is a combination of the method tables of its superclasses, A and B, with the additional methods defined by C itself. Unfortunately, this scheme has a problem, too. A situation which is similar to the previous example occurs. Observe that b is the first entry in B's method table, but is the second and fifth entry in C's table.

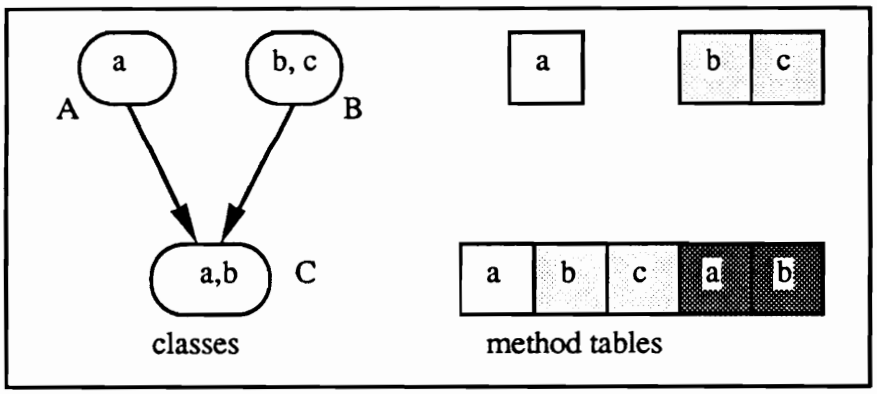


Figure 15 - Method Table with Separate Entries

Another possible solution is to modify the way a method is addressed in a method table. For the method table of a class, the compiler computes the offset value for each superclass's table component included in the method table. Each method dispatch needs to

adjust the table base address before indexing by a method offset. While this scheme is workable, it has several disadvantages which renders it undesirable. First, the scheme is inefficient in space. Second, the extra addition operation required for dispatching each method slows the binding process at run time. These two run-time overheads are significant enough to warrant a new solution.

5.4 A Hierarchy Partitioning Example

In this section, we briefly describe the new late binding technique called *hierarchy partitioning* by an example. A more detailed description of the technique is provided in Section 5.6. An example multiple inheritance hierarchy is shown in Figure 16. Consider the method table of class I. The method table created by the separate entry scheme described in Section 5.3 and the one created by hierarchy partitioning are shown in Figure 17(A) and 17(B), respectively. For the example used here, the new scheme uses only half the space required by the separate entry scheme. A description of the method table of class I created by the hierarchy partitioning scheme is now presented.

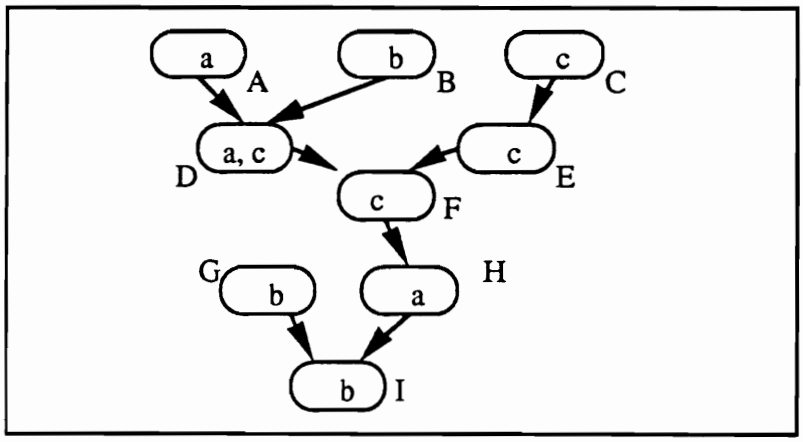


Figure 16 - A Multiple Inheritance Hierarchy

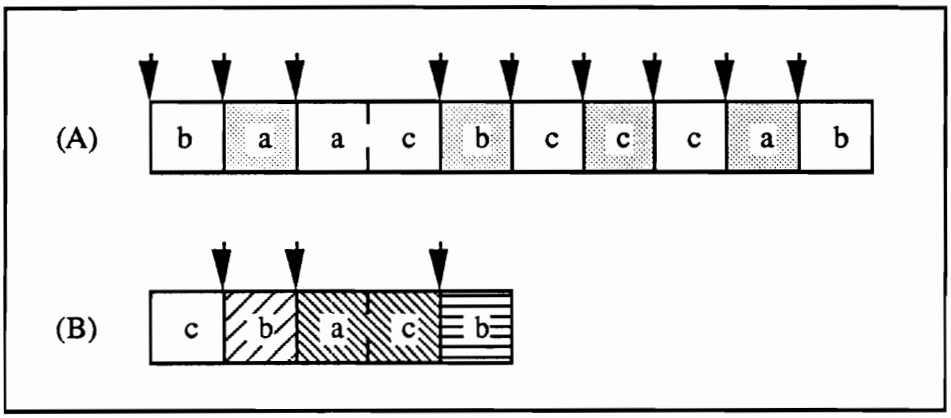


Figure 17 - Method Table for class I

In order to create the method table of class I, we first partition the multiple inheritance hierarchy into single inheritance hierarchies. The particular hierarchy shown in Figure 16 is divided into four partitions, shown in Figure 18. The next step is to create a method table for each of classes I, H, B, and E, which are the lowest class in each partition. Each partition is considered to be an independent single inheritance hierarchy which has no connection to other partitions.

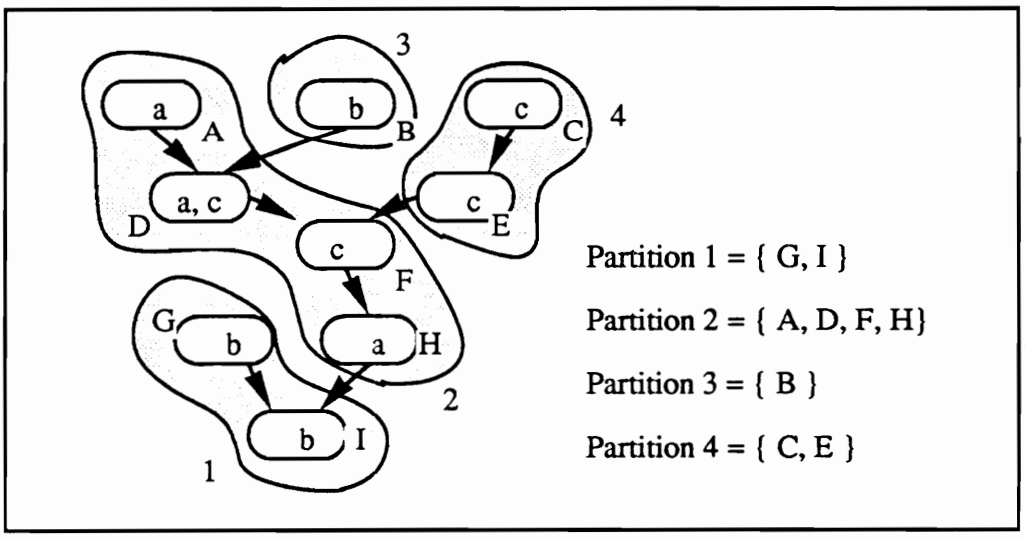


Figure 18 - Partitions for Class I

Figure 19(A) shows the method tables for these four partitions. The method table for each partition is called a *component table*. Note that in each component table, methods with the same name share the same slot as in single inheritance. For example, the component table for partition 1 contains only one entry for method b while b is defined by both G and I.

The method table of class I is constructed as a composition of the component tables. Figure 19(B) shows the final method table of class I, which contains four component tables, namely, those for partitions 1, 2, 3, and 4, in that order from right to left. The offset of the component table for a partition in the final method table is called the *table offset* for the classes in the partition. In Figure 19(B), classes C and E have table offset 0[1] while classes A, D, F, and H have table offset 2.

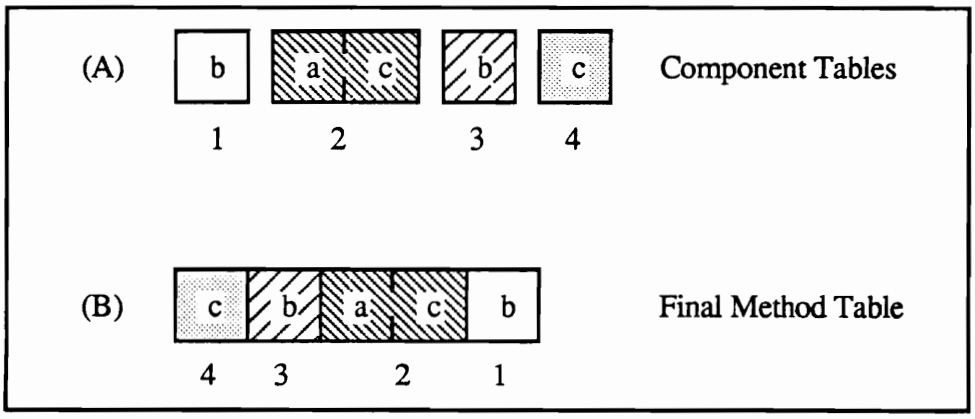


Figure 19 - Method table of class I

5.5 Translation with Hierarchy Partitioning

The offset of a method defined by a class is computed as its offset in the component table for the partition of which the class is a member. For the example shown in Figure 16, the method a in classes A, D, F, and H has index 0 while method c in A, D, F, and H has index 1. In the method dispatching at run time, the classes belonging to the same partition

[1]The index of the first entry of a table is zero.

use the same table offset for indexing into the method table. The entry for the method f stored in a method table x is located at

starting location (x) + method table offset (f) + index(f)

In the hierarchy partitioning dispatching scheme, the operation of adding the method table offset to the starting location of a method table is performed during an assignment operation rather than during each method dispatch. This shift in the time at which an adjustment of method table address is performed allows the overhead of an addition incurred at each method dispatch to be amortized over the number of method invocations applied to the same object. Furthermore, the class hierarchy partitioning requires the addition operation only for a small percentage of assignments. Before a more detailed description of the method dispatching technique is presented, a model of how a method table created by the partitioning technique is used in method dispatching at run time is presented.

The run-time data structures associated with an object are shown in Figure 20. In this example, an object variable contains a reference to a data structure called an *object descriptor*. An object descriptor is a data structure consisting of two addresses. One is the address of the method table of the class of which the object is an instance, and the other is the address of the actual data segment for the object.

In the following sections, we consider translation of language constructs which are relevant to the implementation of inheritance. Translation of assignments and method invocations will be described.

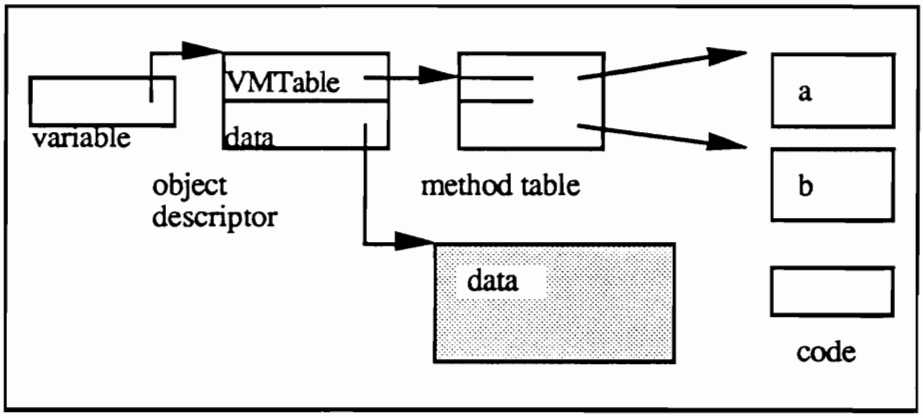


Figure 20 - Run Time Data Structures Associated with an Object

5.5.1 Translation of Assignments

For simplicity, only an assignment where both the source and the target are variables is considered. The case where the source is an object rather than a variable is identical. If b and c are variables of the same type and c is to be assigned to b, the translation of the assignment is straightforward. The execution of this statement causes the variable b to become another reference to the object currently named by the variable c. Both b and c become references to the same object descriptor after the assignment. In this case, it is sufficient to translate the assignment into an instruction which copies the value stored in variable c into variable b at run time.

A more interesting situation occurs when b and c are of different types. Let B and C denote the class of b and c, respectively. If c is to be assigned to b, the subtyping rule of an object-oriented language dictates that B must be a superclass of C. The situation is further divided into two cases depending on whether B and C are members of the same partition created during the hierarchy partitioning for the class of which the object named by variable c is an instance. If B and C are members of the same partition, the assignment is translated into a copy instruction as in the case when b and c are of the same type. This occurs because classes B and C have the same method table offset in the method table to be used. Since the

field (VMTable) of the object descriptor already contains the correct address for the starting location of the method table component, variables b and c can share the same object descriptor. After this statement is performed, both b and c are references to the object descriptor previously named by c.

If B and C are members of different partitions, the value of the VMTable field in the object descriptor needs adjustment, so the object descriptor for variable c cannot be used for variable b. In order to compute the starting location of the method table component for class B, one must know the distance between the offset of the method table component for class B and the one for class C in the method table of the object which is currently named by the variable c. While the distance between two method table offsets can be computed at compile time for any two classes in a class hierarchy, the problem is how to determine the actual class of the object currently named by the variable c. Due to the subtyping of an object-oriented language, the object currently named by c can be an instance of not only class C but also any of its subclasses. Hence, the actual type of the object named by a variable cannot be known until run time.

Fortunately, a theorem, to be introduced later, enables one to use the partitions of the class hierarchy for constructing the method table of class C instead of the the actual class of the object under consideration. Hence, the distance between the method table offset of class B and that of class C is completely determined at compile time. Furthermore, the overhead of computing this value is small since a table of method table offsets can easily be created during the process of partitioning the class hierarchy.

The translation of the assignment statement requires the creation of a new object descriptor be created for variable b. The descriptors for b and c have different values for the starting

location of method table indexing. Let d denote the distance between the method table offsets of classes B and C. Then, the following actions are required at run time.

```
b->VMTable = c->VMTable + d;
b->object = c->object;
```

5.5.2 Translation of Invocations

Four different kinds of method invocations are found in existing object-oriented languages. The first, probably the most difficult case, is the invocation of a method on a variable. The second is when a method is applied to the pseudo variable *self*. The third is an invocation of method on the pseudo variable *super*. The fourth case is when a method being invoked is qualified with a class name. Among these, the last case may be viewed as a variation of the third case. A description of a translation scheme for each of these follows.

5.5.2.1 Invocation on a variable

Invocation of a method f() on a variable x takes the following syntactic form:

```
x.f();          // invoke f() on x.
```

Let x be a variable of class X. The above expression is an invocation of method f on the variable x, more precisely, on the object currently named by variable x. The code to be executed is determined by the type of the object named by x. Since the type of the object is unknown until run time, statically binding the invocation to any specific definition of f() at compile time is not possible. Hence, the binding of f() must be delayed until run time when the actual object named by x is known, i.e., late binding is necessary.

With hierarchy partitioning, locating the code of the method being invoked takes only a single index into a method table. On encountering x.f(), the compiler computes the offset of f() within the method table of X. Let d denote the offset of f(). At run time, the address of the code for f() is located at index d from the point whose address is stored in x->VMTable. By the time the above expression is executed, the object descriptor pointed to by variable x has the correct address of the method table component for X within the method table of the actual object currently named by x. Hence, the above expression is translated into a call to a function whose address is located using the value of x->VMTable as the base address and the offset of f() within X's method table as the offset in a relative addressing. The action can be expressed in C language syntax as follows:

```
(*x->VMTable[d])();
```
[1]

5.5.2.2 Invocation on self

While the pseudo variable self has a variety of names in different languages, every language supporting inheritance has the concept of self. In most object-oriented languages, a method invocation on self is abbreviated to a call to the method without an explicit specification of a keyword corresponding to self. Invocation of a method f() on self takes the following form:

```
self.f()
```

In the hierarchy partitioning scheme, self is a reference to an object descriptor for the sender object. Suppose that the above expression is encountered in the definition of method

[1]Translation of an invocation involves binding the receiver object. Hierarchy Partitioning can also be used for efficient binding of instance variables. In order to focus on method dispatching, the issue of binding variables is momentarily ignored. The use of Hierarchy Partitioning for variable binding is discussed later in a comparison with the C++ implementation scheme.

u() in class X. The expression is translated into a call to the function whose address is located in the method table of self via relative addressing (indexing) which uses self->VMTable and the offset of f() in X's method table as the base address and the offset , respectively. Let d denote the offset of f(). The invocation is translated into the following:

```
(*self->VMTable[d])();
```

5.5.2.3 Invocation on super

The following is an invocation of f() defined by a superclass of the sender object.

```
super.f()
```

The method to be executed is the one inherited from a superclass. The rules set forth by the inheritance model being used determine the parent which appears in the path to the class containing the definition of f() being inherited. The compiler needs to find the offset of f() in the method table of the parent class at compile time. At run time, the code for f() is located in the method table of the parent class. The above expression is translated into an indexing operation in which the address of the method table and the table offset are known at compile time. Let M, T, d denote the addresses of the method table of the parent class, the table offset, and the index of f(). Then the translation of the above invocation looks like

```
(*(M+T)[d])();
```

5.5.2.4 Invocation qualified with a class name

A method invocation qualified with a class name is found in C++ and Trellis/Owl. The above expression is an invocation of f() provided by the interface of class B.

B::f()

For the purpose of translation, the expression can be viewed as a variation of a method invocation on super. The only difference is that the class whose method table should be used for dispatching is already given. Translating the above expression is essentially the same as the method invocation on super where the parent class is B.

5.6 Hierarchy Partitioning

This section describes why and how a class hierarchy is partitioned. As shown in the example in section 5.4, method tables based on Hierarchy Partitioning significantly reduces the use of run-time memory. This reduction is due to the sharing of method entries in each component table. Another advantage is the saving of execution time by significantly reducing the need for recomputing the method table base address. Performing adjustment of the base address of the method table during assignment reduces the overhead of adding an offset at each method dispatch. In addition, few assignments require an adjustment of the base address since adjustment is needed only if the classes of the source and the target variables are members of different partitions.

5.6.1 Characteristics of Partitioning Algorithm

Partitioning a class hierarchy is done in such a way that each of the resulting partitions is a single inheritance hierarchy. Observe that in each partition, two methods have the same name if and only if one is a redefinition of the other. This observation enables one to use the common offset technique of single inheritance for the method table of each partition. In each partition, any two methods with the same name are assigned the same index. In other words, in the component table of each partition, methods with the same name share the same entry. The final method table is constructed as a composition of the method tables of the single inheritance hierarchies.

While several partitioning algorithms are possible which satisfy the above requirement, the algorithm should have another property. The partitioning algorithm should allow a new method table to be built incrementally from the method tables of its superclasses.

A partitioning algorithm which meets these criteria is a variation of a depth first search traversal of a directed acyclic graph. In order to apply the algorithm, a graph for the given class hierarchy needs to be created first. We call the graph an *inheritance graph*. In the inheritance graph, a node labeled by a name exists if and only if a class with the name exists in the class hierarchy. An arc from node X to node Y exists in the graph if and only if Y is an immediate superclass of X in the hierarchy. Obviously, an inheritance graph is a directed acyclic graph.

5.6.2 Traversal Algorithm

5.6.2.1 Partitioning

The algorithm for hierarchy partitioning performs a depth first traversal in an inheritance graph, visiting adjacent nodes following every arc starting from the leftmost arc. A node without any outgoing arcs (i.e. classes with no superclasses) is called a *marker*. In the traversal, each marker node represents the end of a partition. Hence, all nodes which are visited after the last marker node belong to the partition of the next visited marker node. This algorithm is called the *traversal algorithm*.

Figure 21 shows the inheritance graph for the hierarchy shown previously in Figure 16, and a traversal on the graph using the traversal algorithm. A number beside a node denotes the order in which each node is visited. Gray nodes in the graph denote markers. Applying this algorithm to the class hierarchy shown in Figure 16, results in the partitioning shown in Figure 18. The numbers of partitions have been assigned so that a partition visited earlier has a smaller number. Hence the order of visits is 1, 2, 3, and 4.

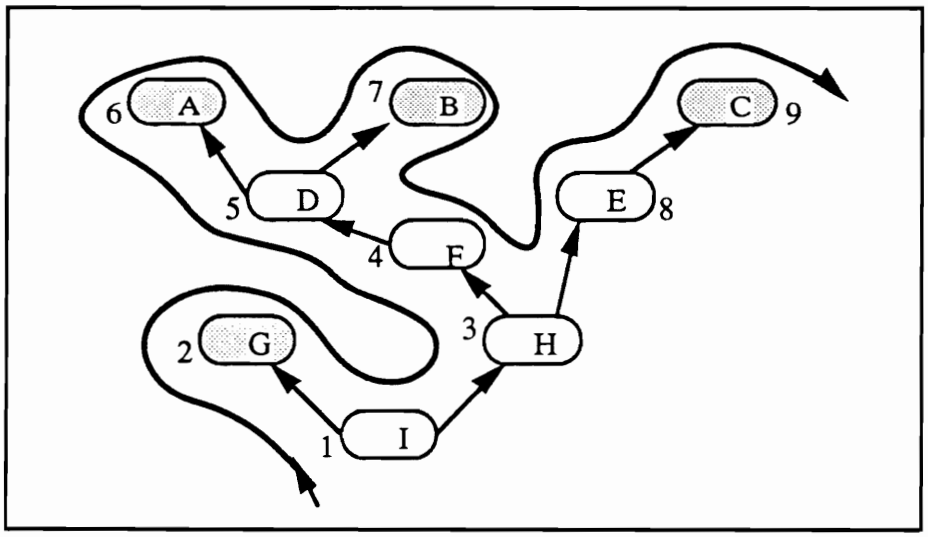


Figure 21 - Inheritance Graph and Traversal

5.6.2.2 Concatenation

A method table is created as a combination of the component tables for the partitions identified in the partitioning step. Continuing with the above example, one first creates a method table for each partition as if the partitions were disconnected. Figure 19(A) shows the method table for each partition. The final method table of class I is a concatenation of these tables. The tables are concatenated in the reverse order of visitation, in order to keep the distance between two component tables within the method table of a class the same in the method tables of its subclasses, thereby allowing the common offset technique to be used for method dispatching. In the example, table 4 comes first and then 3, 2, and 1 from left to right.

5.6.3 Incremental Algorithm

In fact, the traversal algorithm is not explicitly needed to create the method table of each class. Supposing that the method table of class H is already constructed, consider the method table of class I. The method table of class H is shown in Figure 22. Observe that the method table of class I in Figure 19(B) contains the method table of class H. In general, the method table of a new class can be built on those of its immediate superclasses, incrementally. We call this algorithm the *incremental algorithm*.

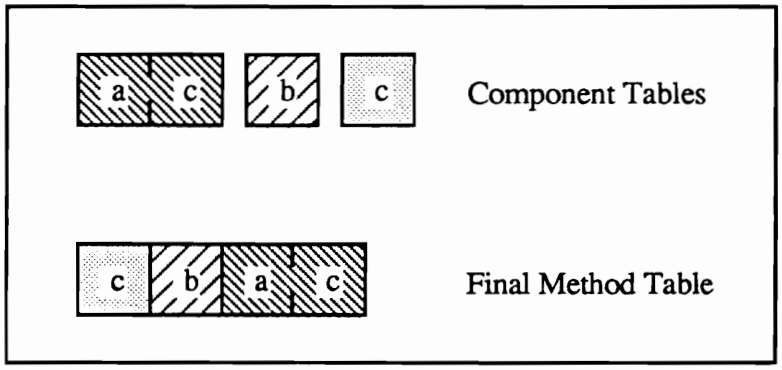


Figure 22 - Method Table of Class H

Since a class may belong to no more than one partition[1], the additional work in constructing a method table of a new class is to identify the superclass to whose partition the new class will belong. Once such a superclass is chosen, the method table of the new class can be constructed as a concatenation of the method tables of its other immediate superclasses and a component table combining the method table of the chosen super class and the methods defined by the new class. The choice of such a superclass is always unique with the above algorithms: the leftmost child in the inheritance graph. Hence, little additional work is needed to create the method table of a new class.

Now the theorem which was used in Section 5.5 is introduced. Let P be one of the partitions created in a Hierarchy Partitioning for class X. P is said to be a *partition for* X. Furthermore, if X is a member of P then P is said to be the *partition of* X. Note that partitions created by the traversal algorithm and the incremental algorithm are identical.

Theorem:

> **Two classes A and B are members of the same partition for class X if and only if they are members of the same partition for any subclass of X.**

The proof of this theorem is obvious since by the definition of the incremental partition algorithm, each partition for a class X remains the same as a partition for a subclass Y, except the one which contained X. In this case, the partition of X is a subset of the partition of Y. Hence, the theorem follows.

---

[1]Assume an inheritance graph is a tree. A more general inheritance graph where a node in the graph may have more than one parent is known as repeated inheritance [Meyer 86]. An inheritance graph with repeated inheritance can be converted to a tree by replicating the class being repeatedly inherited. The technique is described in [Snyder 86A]. As is shown later in this section, Hierarchy Partitioning works for repeated inheritance as well.

For example, consider the partitions obtained when we use Hierarchy Partitioning for class H in the hierarchy shown in Figure 16. The set of the partitions created is

{ {A,D,F,H}, {B}, {C,E} }.

Compare this with the partitions for class I, a subclass of H, as shown in Figure 18:

{ {G,I}, {A,D,F,H}, {B}, {C,E} }.

Repeated Inheritance

The Hierarchy Partitioning technique also works in the case of *repeated inheritance* [Meyer 88][1].

Theorem:

> **Let k denote the index of method f in the method table of a class X. Then, f has index k in the method table for each partition containing X.**

Proof of this theorem is also obvious if we consider that indices of methods in a partition do not change when a new class is added to the partition.

An example of repeated inheritance is shown in Figure 23. Class D inherits twice from A, once through B and once through C. According to the above theorem, method *a* will have index 0 in both B's and C's method tables as shown in the figure. In general, the hierarchy partitioning technique works for any multiple inheritance hierarchy.

One limitation of the approach is that the final method table contains duplicates in the case of repeated inheritance. For example, the method table in Figure 23 contains *a* twice. However, other techniques which completely remove duplicates are not currently known. It

[1]The resulting sets are no longer partitions in a strict sense since they are not disjoint. However, the term partition will be used for this case, since it causes little confusion.

seems that the common offset technique is used for repeated inheritance, duplicates can not be completely eliminated.

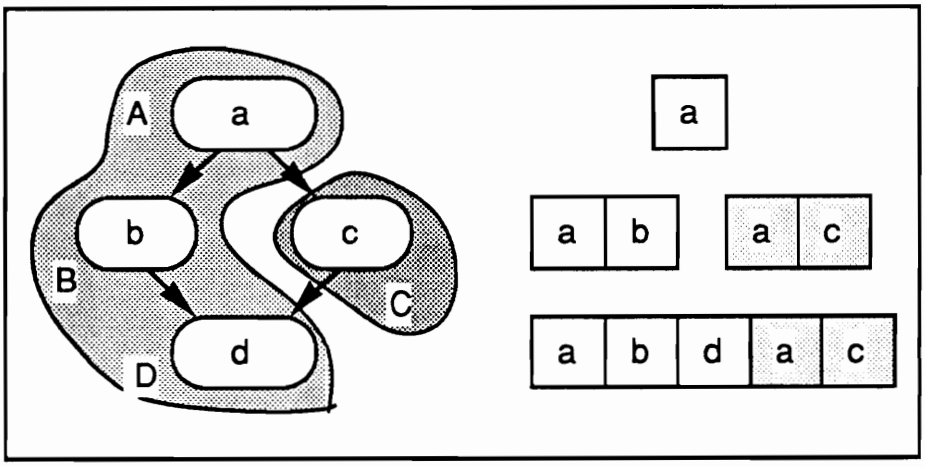


Figure 23 - Repeated Inheritance

Partitioning Strategies

When incrementally building a method table, there are various strategies for selecting the partition of the new class. One strategy is to choose a partition which results in the longest hierarchy chain. This strategy may have some positive effect on reducing the frequency of adjusting table base addresses. Another strategy is to choose a parent which results in the smallest method table. A parent whose component table contains the largest number of methods which are also defined by the new class is selected. While these heuristics may be useful, there does not exist an optimal algorithm since the decision made by a class may turn out to be suboptimal when its subclasses are introduced later. Figure 24(A) and 24(B) illustrate this point. As shown previously in Figure 12, the partitioning for class C is optimal, since no duplicate entries exist in the method table of C. However, this decision results in a suboptimal partitioning for class D, a subclass of C, as illustrated in Figure 24(B).

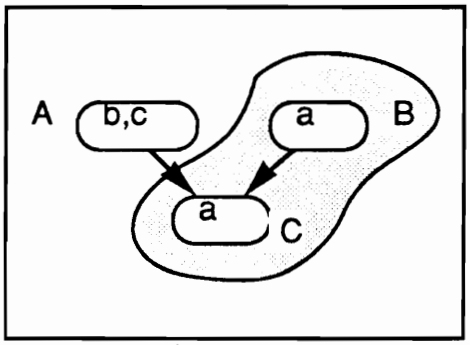


Figure 24(A) - Optimal Partitioning for Class C

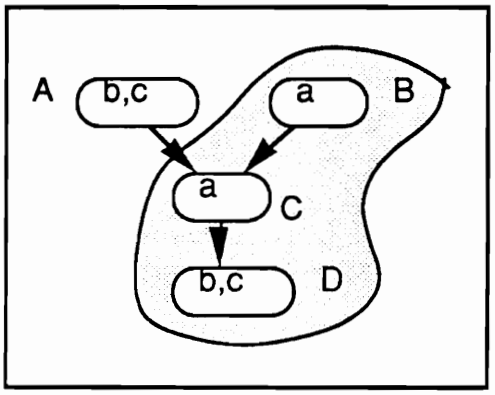


Figure 24(B) - Suboptimal Partitioning for Class D

5.6.4 Performance Prediction

An implementation of multiple inheritance based on hierarchy partitioning is expected to make an efficient use of space. An example which substantiates this expectation was shown in Figure 17. The efficiency approaches that of single inheritance as the number of classes which have only single parents increases. The extreme case occurs when the class from which the method table is created is the lowest class in a single inheritance hierarchy. Note that Hierarchy Partitioning pays no additional overhead in either space or time in such a case. In practice, the use of single inheritance is dominant even when the language being used supports multiple inheritance. The additional space overhead caused by the support of

multiple inheritance is small. In the next section, Hierarchy Partitioning is shown to be more space efficient than other reported techniques.

For an analysis of the execution time overhead, a rough estimate of the average execution time overhead per method dispatch is made. The expectation is that the effect of additional execution time overhead caused by the operation of adjusting the base address of a method table is negligible. This expectation is based on the following observations.

1. Typically, two or more operations are applied to the same object.

2. An adjustment of the base address is needed only when the class of the object being assigned is in a different partition from the class of the target variable.

3. Assignment from a subclass variable to a superclass variable is less common than assignment between two variables of the same class. Note that only the former case can result in the overhead of an addition operation.

An example which supports the first observation is that the user of an object usually bases its next message on the current state of the receiver. For example, a user of a stack object needs to check if the stack is empty before it issues a pop request.

The second observation has already been made in Section 5.5. Without a rigorous analysis, we claim that the number of assignments which require the adjustment of the table base is less than half of the total number of assignments from a subtype variable to a supertype variable. In fact, even this may be a conservative assessment. Observe that many assignments between variables of classes in two different partitions are not type compatible

and so assignments between them are not allowed. For example, an assignment between a variable of class D and one of class C, as shown in Figure 18, is not type compatible.

The third observation is rooted in experience in writing programs in object-oriented languages. While subtype polymorphism is a powerful tool, its use is rather limited. The main use of subtype polymorphism is to replace the case-like explicit decision construct by moving the decision point into the receiver object. In order to put this issue in perspective, consider the frequency of case statements and assignments in programs written in a language like Pascal which does not support subtyping.

Other factors may also affect the run time overhead favorably. In contrast to conventional languages which achieve side effects with assignments, object-oriented programming achieves side effects by object encapsulation. A method invocation usually results in change in the state of the receiver object. When an assignment is used, often the target variable in the assignment is of a primitive type such as integer. Most assignments are replaced by passing an object reference in a method invocation. While argument passing involves a similar issue to assignment, parameter subtyping is not used frequently. For example, consider that C++ does not even support parameter subtyping.

A further hypothesis is that the use of single inheritance is dominant even when multiple inheritance is supported. Under certain assumptions, one can predict that the increased run time overhead will be within 10 percent of the time for a method dispatch in a fast single inheritance implementation.

5.7 Comparison to Other Late Binding Techniques

In this section, a comparison of Hierarchy Partitioning with other dispatching techniques found in the literature is made. Most earlier schemes for method dispatching in object-oriented languages were intended for dynamically typed languages. These techniques typically perform an unbounded method search at run time and so suffer from a significant overhead in execution time. More recently, Rose [Rose 88] described table based dispatching techniques for dynamically typed languages. Without static type checking, method dispatching is slow. Since hierarchy partitioning is intended for a statically typed language, we will consider only those dispatching techniques proposed for statically typed object-oriented languages with multiple inheritance. The criteria used in the comparison include overheads in run time memory use, execution time, compile time, and support of incremental compilation. Three recent proposals are reviewed using these criteria.

5.7.1 Color Indexed Code Technique

Dixon, et al., [Dixon 89] proposed an indexing scheme called the *color indexed code* technique. As in Hierarchy Partitioning, a method dispatch is accomplished by indexing into a dispatch table at run time. A method f has the same index in every dispatch table. In order to reduce the table size, a global conflict analysis of selectors (methods) is performed using a graph coloring technique at compile time.

In comparison to Hierarchy Partitioning, the following observations are made:

1. The color indexed technique is significantly inefficient in memory use compared with hierarchy partitioning. Each dispatch table contains many unused entries in order to place each method entry at a fixed position in every table. For example, the classes given in Figure 1 of [Dixon 89] requires a total of 28 table entries when their coloring technique is used, while Hierarchy Partitioning uses only 15 entries.

2. Another difference between the coloring technique and Hierarchy Partitioning is that the coloring algorithm cannot exploit the structure of the given class hierarchies. For example, the coloring technique cannot have the efficiency of the efficient single inheritance scheme when most classes form a single inheritance hierarchy while the efficiency of Hierarchy Partitioning approaches that of a single inheritance implementation. An example illustrating this point is given in Figure 25. While Hierarchy Partitioning uses only 6 table entries, which is the minimum for the given example, the coloring technique requires 3 different colors for each method table giving a total of 9 table entries.

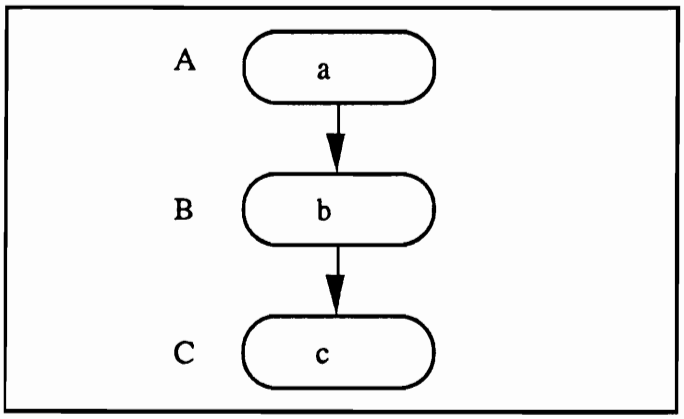


Figure 25 - An Example Class Hierarchy

3. With the coloring technique, incremental compilation becomes difficult because the coloring technique depends on a global analysis of the program. Adding a new class may result in the redefinition of dispatch tables of existing classes. This is a deficiency especially in object-oriented programming where incremental development is more pronounced.
4. The global analysis required by the coloring technique seems to incur a significant overhead in compile time. A global analysis needs to be performed every time changes occur in the set of selector names.

5.7.2 Address Map Technique

Another method dispatching technique based on an *address map* was proposed by Connor, et al. [Connor 89]. An address map is a run time table which contains a list of offsets for methods in a method table. At run time, method dispatching involves an indirection through an address map. Although the technique was proposed for multiple inheritance, it only considers subtyping. No actual inheritance, in the sense that a subclass inherits data and operations from a superclass, is assumed. The distinction between subtyping and inheritance was discussed in Chapter 2. While the technique can also be used for run-time type-checking languages, the technique is weak in exploiting the extra information provided by compile-time type-checking. The following observations can be made in comparison to Hierarchy Partitioning.

1. While the address map technique may be suitable for the implementation of subtyping with no inheritance, there is significant overhead in execution time. An indirection taken at each method dispatch slows down the binding process.

2. An address map must be created at run time. This is an expensive process especially because there is little structural relation between subtype objects and supertype objects. Hence, an assignment between a subtype variable and a supertype variable requires that an address map be dynamically allocated and its entries be filled at run time.

3. Each address map creates space overhead. An address map must be created for each pair of a subtype and supertype if an assignment occurs between variables of the two types.

4. The technique does not support code sharing.

5.7.3 Virtual Table Approach of C++

A closely related but independently developed late binding technique used by multiple inheritance C++ is described in [Stroustrup 89]. As with Hierarchy Partitioning, the technique uses a run-time table called a *virtual table* (vtbl) for each class and performs late binding by indexing into a virtual table. It seems that the virtual table approach of C++ is the best of the late binding techniques currently available for statically typed multiple inheritance object-oriented languages.

An instance of a class is a concatenation of the components called subobjects, each of which is a data structure defined by one of its superclasses or by the class itself. Each subobject contains a pointer to a virtual table. As with the Hierarchy Partitioning method table, a virtual table contains the pointers to the virtual functions supported by the the class of the subobject. An object variable is implemented as a pointer to a subobject.

A set of example C++ classes used in [Stroustrup 89] are shown in Figure 26(A). Figure 26(B) shows an instance of class C and its associated virtual tables. As shown in the figure, each entry in a virtual table contains two fields, namely, a method pointer and an offset of the subobject to which the method is applied. Which subobject a variable points to is determined by the type of the variable. As with Hierarchy Partitioning, an assignment between a supertype variable and a subtype variable needs an adjustment of a constant offset to the pointer value stored in the source variable. Method dispatching at run-time involves finding the starting location of a virtual table and indexing into the table.

```
class A { virtual void f(); };
class B { virtual void f(); virtual void g(); }
class C : A, B { void f(); };
```

Figure 26(A) - Example Class Definitions in C++

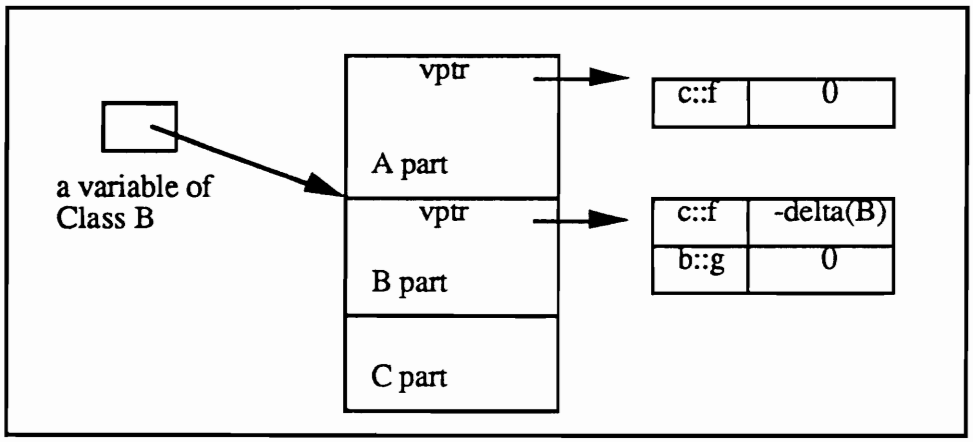


Figure 26(B) - An Instance of Class C

In comparison to Hierarchy Partitioning, virtual table uses the same amount of space for storing method pointers and data. However, Hierarchy Partitioning does not need as much space since the virtual table requires an additional field in each entry in order to store the offset of a subobject. Further, C++ must adjust the base address at each method dispatch.

A more direct comparison with the virtual table approach is possible when the issue of binding instance variables is considered. In fact, Hierarchical Partitioning is an extension of the virtual table scheme used by multiple inheritance C++. It can be shown that hierarchy partitioning is a significant improvement over the virtual table approach of C++ in both space and execution time. Figure 27 shows the representation of the same object of Figure 26 when Hierarchy Partitioning is applied in addition to the virtual table approach. The field

for offset(delta) has been eliminated from the method table. As a result, the space used for a method table is only half of that of a C++ virtual table. In addition, adjusting the object pointer at each invocation is not necessary.

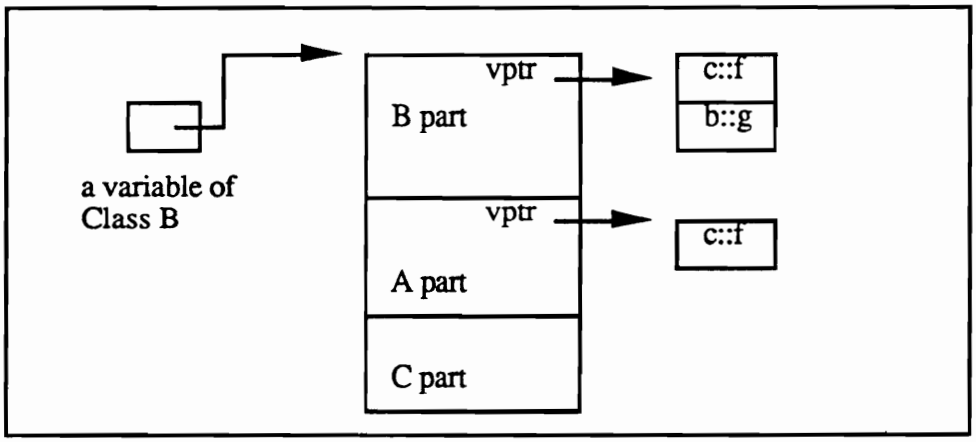


Figure 27 - Object Structure with Hierarchy Partitioning

Chapter 6 ACT++: Building a Concurrent C++ with Actors

6.1 Introduction

ACT++ (Actors in C++) [Kafura and Lee 89C] is a prototype object-oriented language designed to explore combining concurrency with inheritance. The language is a hybrid of the actor kernel language and the object-oriented language C++. The concurrency abstraction of ACT++ is derived from the actor model as defined by Agha [Agha 86]. ACT++ represents an effort to test the utility of the actor style of programming in the domain of real-time systems applications. ACT++ is intended to evolve into a language for distributed real-time applications [Kafura 88, 89].

One of the main goals in the design of ACT++ is to prototype a language which supports the actor concurrent computation model and provides software reusability through the class inheritance mechanism of an object-oriented language. Since the language is intended for exploring the actor style of programming and object-oriented programming with class inheritance, a requirement imposed on the current ACT++ design is that it should lend itself to an inexpensive implementation. The current ACT++ design extends C++ [Stroustrup 86] with a class hierarchy which provides the abstraction of the actor model of concurrency.

The remainder of this chapter is composed of the following sections. Section 6.2 is an introduction to the actor model as defined by Agha. Section 6.3 describes the design and implementation of ACT++. Some implications of the actor model are also discussed. Some modifications of the actor model are described. Section 6.4 describes the ACT++ class hierarchy which provides the actor abstraction. Section 6.5 presents some example

programs written in ACT++. Section 6.7 further elaborates on the implementation of ACT++. The classes used in the implementation are described in some detail.

6.2 The Actor Computation Model

The actor model [Agha 86] is a concurrent computation model in which computation is achieved by actors communicating via asynchronous message passing. The actor model consists of five basic elements: actors, mail queues, messages, behaviors, and acquaintances.

An *actor* is a self-contained active object. Interaction among actors occurs only through message passing. Each actor is associated with and named by a unique *mail queue*, whose address serves as the identifier of the actor. The messages sent to an actor are buffered by its mail queue. Messages buffered in a mail queue are read one at a time in strict FIFO order. If the message queue is empty when an actor is ready to receive the next message, the actor is blocked until a message arrives in the mail queue. An actor *A* may send a message to another actor *B* only if *A* knows the mail queue address of *B*. Mail queue addresses themselves may be passed as a part of a message. These last two properties allow the connectivity among actors to be reconfigured dynamically.

The *behavior* of an actor is a specification of how the actor reacts to a request specified in the message being processed. A behavior is defined by a code body called the *behavior script*. The script contains method definitions and *acquaintances*. Acquaintances are the names of other actors to which an actor may send messages. A behavior script corresponds to a class definition in other object-oriented languages while acquaintances correspond to instance variables. However, acquaintances in Agha's actor model are read-only variables. There is a good reason for the *read-only* requirement. The actor model assumes *inherent concurrency*, meaning that every statement of a behavior script is executed concurrently

except when there is a sequencing constraint required by causal ordering. The inherent concurrency would not be possible without a guarantee that no data dependency among instructions exists. Hence, the actor model provides only side-effect free operations. For example, assignment operations are not allowed as in functional programming languages.

In processing a message, an actor may perform three kinds of actions: create new actors, send messages to existing actors, and specify a replacement behavior. Actors are dynamically created at run-time as needed during the course of computation. A new actor is created by the **new** operation. The mail queue address of the created actor is returned as a result. Other actors may send messages to the new actor using this mail queue address. Since the creator of a new actor may pass the mail queue address of the new actor in a message, the existence of a newly created actor becomes known to other existing actors through message communication.

The **send** operation is the primitive used for asynchronous message passing. A message consists of the address of the target actor, the name of a method to invoke, and parameters for the method invocation. An actor is not blocked by a message sending operation since the mail queue of the receiving actor buffers incoming messages.

The last primitive operation is the specification of a *replacement behavior*. The operation, called **become**, is the most novel concept in the actor model. The difficulty in understanding the actor model comes mainly from the unusual notion of a replacement behavior.

To avoid ambiguous terminology, following entities are distinguished: actor, behavior, and thread. The following definitions deviate from Agha's terminology used in the actor model. An actor, which is identified by a mail queue address, is a combination of a mail queue and

all the activities and data structures associated with the mail queue. A *behavior script*, or *behavior* for short, determines what actions an actor may take when processing a message. A *thread* denotes a behavior script in execution. A thread has a list of acquaintances which constitute the local execution environment of the thread. The *behavior of a thread* refers to the behavior script executed by the thread. In Agha's model, a behavior receives exactly one request message.

The concept of replacement behavior means an actor may change its behavior dynamically at run time. That is, an actor may be bound to different behaviors at different points in time. These behaviors may differ in their methods and internal data structures. The only entity which does not change in the life of an actor is its mail queue address. For this reason, the *current behavior* is defined as the behavior of the most recently created thread of the actor. Similarly, the *state of an actor* is defined to be the state of the current behavior. Behaviors which are not current are called *old behaviors*. Figure 28 depicts the conceptual view of an actor.

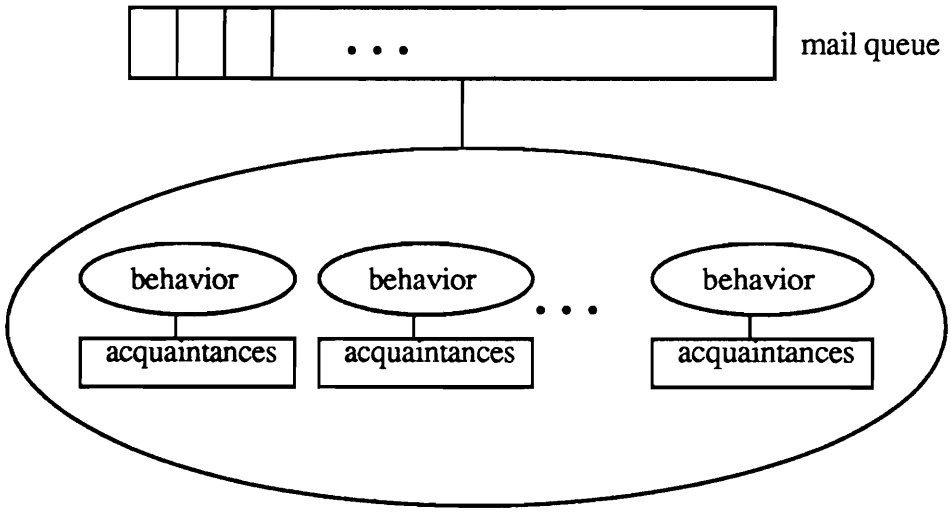


Figure 28 - Conceptual View of an Actor

The become operation is the language mechanism used for specifying a replacement behavior. The become operation requires a behavior script name and a list of acquaintances. The results of the become operation is a newly created thread. The newly created thread is the current behavior of the actor. The current behavior determines how the next unprocessed message is handled. Since a thread can specify a replacement only once in its life, there is at most one behavior waiting for a message at any time. Figure 29 is the BankAccount example written in actor style.

```
Actor BankAccount(Acquaintance currentBalance)
Method GetBalance[customer]
        become BankAccount(currentBalance);
        send currentBalance to customer;

Method Deposit[amountToDeposit]
        send currentBalance + amountToDeposit to customer;
        become BankAccount(currentBalance + amountToDeposit);

Method Withdraw[amountToWithdraw]
        if currentBalance < amountToWithdraw then
                become BankAccount(currentBalance);
                send "Insufficient balance!" to customer;
        else
                become BankAccount(currentBalance -
                                        amountToWithdraw);
                send amountToWithdraw to customer;
```

Figure 29 - A Bank Account Actor

6.3 Building Actors in C++

The become operation plays two important roles in the actor model. The first is that the become operation is the mechanism by which an actor changes its state. Since actor operations have no side-effects, acquaintances cannot be modified by assignment. An actor changes its state by becoming another behavior with different acquaintances. One may view each behavior as an object by itself which has a much smaller life-time compared with active objects in other concurrent object-oriented languages. This view naturally extends the replacement behavior to one with different methods and data structures, not to mention acquaintances whose values are different from those of old behaviors.

The second important use of the become operation is as a synchronization mechanism. Besides the synchronization provided by message passing, the become operation is the only explicit synchronization primitive provided in the actor model. The consistency of the state of an actor is maintained by the use of the become operation. The combination of buffering provided by the mail queue and the become operation offers a synchronization mechanism which is similar to a *serializer* proposed in [Hewitt and Atkinson 79]. The become operation serializes the invocations of methods of an actor because the specification of a replacement behavior signals the mail queue to let the next request message enter the protected region of the actor. For example, consider an actor whose state is modified by the current behavior. The actor must prevent its state from being accessed by other invocation threads until the modification is complete. The mail queue must block other requests until the modified state becomes available. Specifying a replacement behavior is the mechanism for informing the mail queue that the current behavior has finished computing the state of the actor, and so the next thread can thus be created.

A significant result of the become operation is the potential for concurrency inside an actor. Intra-object concurrency naturally arises when the current behavior continues after it specifies the replacement behavior. In this case, two or more threads proceed concurrently.

6.3.1 The Actor Model in ACT++

The effectiveness of the instruction level fine-grain concurrency of Agha's actor model is predicated on the availability of a large number of processors with extremely fast interprocessor communication. Instruction-level concurrency is not effective in a system of a relatively small number of loosely coupled processors. Since the long-term application domain of ACT++ is distributed real-time systems, the fine-grain concurrency offered by the actor model is not desirable. Hence, instruction-level concurrency is not a concern of ACT++. An ACT++ behavior in execution is a sequential process called an *ultra-light process*. The term ultra-light derives from the fact that the code size and the life-time of a behavior is equivalent to that of a procedure. The decision to remove the instruction-level concurrency from C++ has several significant impacts.

Removal of instruction-level concurrency makes actors in our model more compatible with imperative language constructs. Method definitions are no longer subject to the requirement of side-effect free operations. Acquaintances, which are called instance variables in ACT++, can be written to. Hence, an actor may change state by directly modifying its instance variables with assignment operations. There is no loss of intra-actor concurrency at the level of behaviors. Concurrency in ACT++ comes from two sources: creation of a new actor and specification of a replacement behavior. The former results in inter-object concurrency, while the latter yields intra-object concurrency.

Another significant impact is that ACT++ encourages actors to be more coarse-grained objects. The become operation in Agha's actor model forces the data structures contained in

actors to be very small because an actor specifying a replacement behavior must enumerate all the acquaintances to be included in the local execution environment of the replacement behavior. In the primitive actor model, programming can quickly become complex even with a small increase in the number of acquaintances. Hence, programmers are discouraged from gathering related data items together into one actor. Agha's modeling of a stack, where each item of a stack is modeled as an actor, reflects this phenomenon.

The coarse actor granularity in ACT++ is also more compatible with class inheritance. Since a subclass typically needs its own instance variables in addition to inherited instance variables, the number of instance variables in a class increases as the class hierarchy grows deeper. If all instance variables must be explicitly specified in the become operation, a class hierarchy of more than one or two levels becomes unwieldy.

Objects

Two kinds of objects are distinguished in ACT++, namely *active objects* and *passive objects*. An active object proceeds concurrently with other objects. An active object, possessing its own thread of control, is an actor. All objects that are not actors are passive objects. A passive object does not have its own thread of control. The invocation of a method of a passive object is performed using the thread of the requesting object. The sender of a message to a passive object is blocked until the invocation is complete. Since a passive object has no mechanism to control invocations of its methods, concurrent invocations of its methods may cause the state of the object to become inconsistent. Hence, a passive object must be known to only a single actor. The reference to a passive object cannot be exported to other actors. Passive objects are used to build an actor, being private to the actor. All shared objects are actors. This restriction is enforced in the current implementation only by convention.

Class

Each object is an instance of some class. A class defines the properties of its instance objects. A class may inherit from another existing class by declaring itself a subclass of the existing class. A subclass may redefine, restrict, or extends the definition of its superclass. An active object is an instance of a class which is a direct or indirect subclass of a special class called *ACTOR*. A passive object is an instance of a class which is not a subclass of the ACTOR class.

Actor

The ACT++ language supports the primitive operations of Agha's actor kernel language: **new**, **send**, and **become**. By declaring a class as a direct or indirect subclass of the built-in class ACTOR, one can define a behavior script, which is also called an *actor class* in ACT++. A new actor is created using a New[1]operation. The New operation creates a pair of a new mail queue and an instance of the specified actor class. The address of the mail queue is returned as the result.

A behavior script is written using the actor primitives and the imperative language constructs of C++. The instructions in a behavior script are executed sequentially. An actor may specify itself as the replacement behavior by specifying ***self*** as the argument to the become operation. In this case, the replacement behavior is created using a copy of the instance variables of the current behavior.

Message Passing

Invoking methods of passive objects has the semantics of a function call and will not be discussed further. Invoking a method of an actor is via asynchronous message passing.

[1]To distinguish the actor new operation from the new operation of C++, we write the first character in upper case.

Asynchronous message passing in ACT++ is supported through predefined objects called *mail boxes*. Two types of messages are distinguished in ACT++, namely, *request messages* and *reply messages*. A request message is used to invoke a method while a reply message is used to deliver the result of a method invocation. Two types of mail boxes are provided to support the two different kinds of messages: *Mbox* and *Cbox*. Mbox models the mail queue of the actor model while Cbox allows for programming with *futures* [Lieberman 87].

Request Messages and Mbox

A *request message* is used to send a request to another actor. A request message consists of the name of a method to be executed by the receiving actor and arguments for method invocation. A request message is sent by a **send** operation. The send operation requires as arguments the target Mbox and the message being sent.

Request messages are buffered in the Mbox of the receiving actor. An actor may refer to its own Mbox using the pseudo-variable self. Each behavior of an actor can process only one request message in the Mbox at a time. Since a thread is created only if the mail queue has a message for the thread, the operation of receiving a request message from the sending actor is implicit in a behavior script. Once the execution of a behavior begins, the message to be processed is available and is read by an implicit **receive** operation on self. A behavior is not blocked by the **receive** operation performed on self since the thread of the behavior is not created until a message becomes available in its mail queue.

Reply Messages and Cbox

If the sender of a request message wants to receive the result of a method invocation, the sender may provide a Cbox name (see below) in the request message. The reply operation is used to transmit a reply message containing the result. The name of a Cbox specified in a

request message is called the *reply destination* [Yonezawa et al. 87]. Since an actor knows the reply destination when it accepts a request message, the reply destination need not be explicitly provided by the programmer in the **reply** operation. If the sender, *A*, does not provide its own Cbox in a request message, *reply forwarding* occurs. The reply is not delivered to the actor *A*: instead, the reply is directly delivered to the actor that sent the request message to the actor *A*.

A Cbox in ACT++, named after the Cbox structure of Concurrent Smalltalk [Yokote and Tokoro 86], implements "waiting by necessity" [Caromel 88]. The Cbox is a special mailbox created to receive the reply resulting from an initiated operation. An actor accepts a message from a Cbox using the **receive** operation. If a reply is available in the Cbox, the reply is immediately delivered to the actor. Otherwise, the **receive** operation blocks the caller until a reply arrives. An actor can check if the reply has arrived in a Cbox using the **in** operation on the Cbox. The in operation returns a boolean value of true if a reply message exists in the Cbox, else the value false is returned.

The Cbox was designed for three reasons. One reason is to prevent the an actor which needs to receive a reply from being split into multiple fine-grained continuations. Cbox provides a feedback communication channel which allows the sender of a request message to receive a reply message directly from the server of the request without creating separate actors to implement a continuation waiting on a reply message.

The second motivation is to reduce the complexity in defining a behavior which receives the results of the operations which an actor initiates. Using a single mail box for both request messages and reply messages requires a server actor to have some a priori knowledge of the client actor. The server must know what method of the client to invoke in order to pass the reply to the client. The a priori knowledge can be minimized if the server actor may

assume that every client actor provides the same method to receive a reply. However, the script of a sender of a request becomes complex if more than one reply is to be received. For example, the *function-apply* actor requiring two inputs, as defined in [Agha 86], must create continuations in order to maintain an explicit state transition to differentiate the two inputs received using the same method. The Cbox mechanism simplifies the sender/receiver protocol by eliminating the need to define a separate method for receiving a reply message.

The third reason is to allow for an efficient implementation of an actor which cannot proceed until a reply is received. The *call* and *reply* constructs in Agha's SAL language [Agha 86] abstract away the details of the protocol for receiving a reply by requiring a specialized service from the compiler. The compiler is responsible for translating a *call* expression into a continuation which waits for the reply. However, in the primitive actor model, the initiating actor becomes inactive by means of an *insensitive actor* until the reply is forwarded by the continuation. An actor, *A*, becomes insensitive by assuming a behavior which forwards all incoming messages to an auxiliary buffer actor. This forwarding continues until the message arrives which contains the result of the initiated operation. At this time, *A* sends the buffer actor a message directing the buffer to retransmit all queued messages back to *A*.

The Cbox mechanism is more flexible than insensitive actors because an actor may create many Cboxes before waiting on any of them. Alternatively, a Cbox may be used for receiving the same type of reply messages sent by multiple actors. Implementation of and-synchronization and or-synchronization are directly supported by the Cbox mechanism. An or-synchronization is implemented by reading only one reply message from a single Cbox to which multiple reply messages may be delivered. Attempts to read a message prior to the delivery of the message causes the reader to block. The actor will be resumed if a reply from any sender is received. An and-synchronization of n events may be achieved by the

action of reading reply messages from a Cbox n times consecutively. The actor reading the messages is suspended and resumed repeatedly until all n messages are received by the Cbox[1].

It is also possible to create a customer actor to implement a continuation. One can create a customer actor by passing a Cbox as an argument of the initial behavior used by a New operation. The creator of a customer actor can have the reply for a request message sent to the customer by specifying the passed to the customer as the reply destination of the request message.

6.3.2 Development Strategy for ACT++

For an experimental language like ACT++, one is faced with the decision of whether to implement a translator for the new language. Since the primary use of ACT++ is exploratory, the detailed syntax for the language was considered of less importance than the general abstractions provided by the language. Therefore, ACT++ is built upon an existing object-oriented language syntax. The mechanisms for creating class hierarchies found in object-oriented languages make it possible to create abstractions that support actor concurrency without changing the definition of the base language. As explained below, C++ was chosen as the base language.

An implementation based on a language like C++, which supports class inheritance has several advantages. First, the compiler and other tools for C++ are directly available for programming in ACT++. Committing the effort required to develop a translator for premature language designs may be deferred. The compiler development effort can be

[1] A further optimization may be achieved by extending the semantics of receiving from a Cbox. A possible extension to the current design is to provide a Cbox operation which takes the number of events and suspends the actor until all of the specified number of replies are received by the Cbox.

postponed until the language design has proven to be useful and stable. Second, since the base language supports class inheritance, the inheritance mechanism can be conveniently incorporated into ACT++. Although experiments revealed that this expectation was only partly met by the current design [Kafura and Lee 89B], the class inheritance mechanism of the base language is provided to ACT++ programmers. Third, experimenting with the ACT++ language will identify deficiencies in the design. Evolutionary changes in the design are more easily accommodated with the modularity and reusability encouraged by class hierarchies. The reasons for choosing C++ as the base language are as follows: C++ is efficient, strongly-typed, and widely available. The strong support of overloading provided by C++ turned out to be a powerful tool for syntactic extension of the language. C++ is also the base language for the operating system kernel being designed by others for the execution of ACT++ programs [Kafura 88].

6.4 Class Hierarchy

The major difficulty in combining C++ and actors lies in incorporating the asynchronous message passing mechanism into C++, which currently has no notion of either message passing or concurrency, but does insist on strong typing. The class hierarchy used in the current implementation is shown in Figure 30. Only the most important aspects of this hierarchy are given here. A more detailed discussion of these classes is provided in Section 6.6. An ACT++ programmer need only know about three classes: ACTOR, Mbox, and Cbox. Other classes found in the hierarchy are implementation details and are transparent to an ACT++ programmer. While ACT++ specific operations, with the exception of the New operation, are directly supported as methods of the three classes, some operation names were beautified with a spoonful of syntactic sugar. The additional operations supported by ACT++ are summarized in Figure 31. Each of these operations is described below.

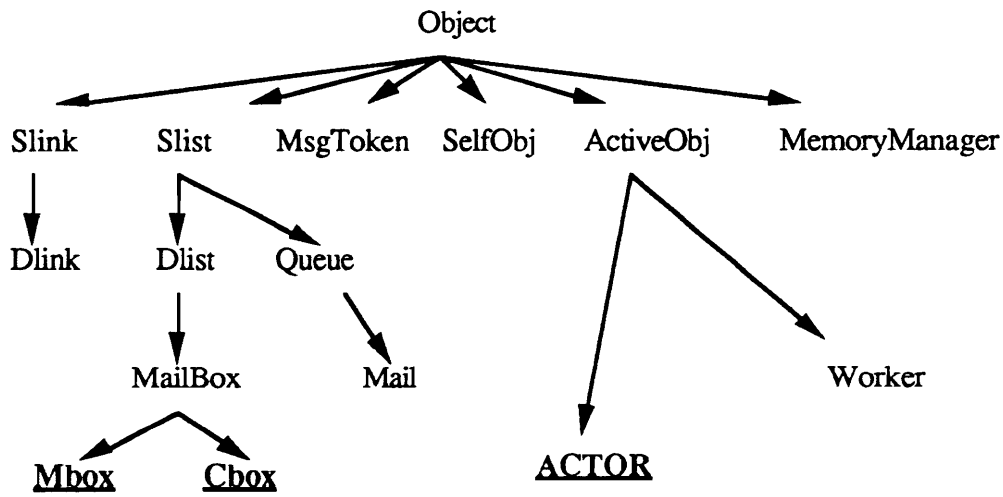


Figure 30 - ACT++ Class Hierarchy

The **New** operation for creating a new actor is a function which takes the name of an actor class as an argument. Ideally, the New operation should be provided as a class method of each actor class. However, since a class in C++ is not a run-time entity unlike in Smalltalk-80, the New operation is implemented as a conventional function without being a member of any class. The **become** operation is a method of the ACTOR class. The operation is overloaded to allow an actor to specify self as the replacement behavior. The **send** operation is implemented as a method of the class Mbox. For cosmetic reasons, the syntax for sending a message is actually written as a series of stream-like output operations (<<). For example, the following statement sends to an Mbox called myMbox a message consisting of aMethod, replyDestination, and two method invocation parameters, namely, firstInteger and secondInteger:

```
myMbox << aMethod << replyDestination << firstInteger << secondInteger;
```

An actor can read a request message from its Mbox using the operation >> on the pseudo variable self. Since the method name argument of a request message is used by the mail

queue for selecting the method to invoke, the method name is not read by the >> operation. Similarly, the reply destination argument is not read by >>. The >> operation is used only to read the values of the parameters of the method. Each method of a behavior script begins with a >> statement. For example, continuing the example above, the definition of the method aMethod of an actor myMbox uses the following syntax for reading a message from the Mbox:

```
self >> firstIntegerArg >> secondIntegerArg;
```

Operation	Class	Meaning
New	none	Create a new actor
become	ACTOR	Specify a replacement behavior
<<	Mbox,Cbox	Send a message to a mail box
>>	Mbox,Cbox	Receive from a Cbox or Self
reply	Cbox	Send a reply message
in	Cbox	Check if a Cbox is not empty

Figure 31 - Summary of ACT++ Operations

The class Cbox supports the **reply** method for transmitting a reply message. Since the reply destination is known when a request message is processed, the Cbox name is provided implicitly by the language. The following statement sends a reply message containing 23 to the reply destination:

```
reply(23);
```

Two other methods are provided by a Cbox: **in** and **receive**. The method **in**() is used to check whether or not the reply has arrived in a Cbox. The receive operation denoted by >> reads a reply from a Cbox. The use of these methods are illustrated in the following code fragment:

```
if (myCbox1.in()) then
        myCbox1 >> anInteger
else
        myCbox2 >> anInteger;
```

If myCbox1 has not yet received a reply, then myCbox2 is read, in which case the actor executing the statement may be blocked depending on the presence or absence of a reply message in myCbox2.

An or-synchronization in which the actor waits for an event, either a or b, to occur is illustrated in the following examples:

```
eventCbox >> anyEvent;
do-after-synch-stuff;
```

An and-synchronization in which an actor waits for all three events to occur is illustrated as follows:

```
eventCbox >> event1 >> event2 >> event3;
do-after-synch-stuff;
```

6.5 Example Programs

Two example programs written in ACT++ are presented in Figures 32 and 33. The first program which computes 20! is an example of actors with no acquaintances. Such actors are called *unserialized* actors. Since no time-dependent errors can occur in invoking an unserialized actor, the language assumes that an unserialized actor has an implicit become operation at the beginning of its script so that the actor can immediately serve the next request in the mail queue. The second example, a cruise control problem, illustrates a *serialized* actor. The behavior script of a serialized actor must specify a replacement behavior at some point in its computation.

6.5.1 A Concurrent Factorial Actor

Figure 32 shows a concurrent factorial actor written in ACT++. The class name for the concurrent factorial actor is ConcFact. Since the class ConcFact is a subclass of ACTOR (line 3), the instances of ConcFact are active objects. The function main() is the driver of the program where the action of computing 20! starts. ACT++ regards the main() function as an actor without methods. In the main() function, the operation New creates a new actor whose initial behavior is defined by the class ConcFact (line 16). The New operation returns the address of an Mbox, which is bound to the Mbox variable factorial. The main() function sends a request message to factorial using << operators (line 17). The message contains the method name to be invoked, the reply destination myCbox, and a seed value n. It then receives the result from myCbox (line 18). The main() function is blocked until the Cbox receives the result. The use of a Cbox allows main() to receive the result directly from the actor factorial. Without a Cbox, an extra actor would need to be defined to receive and print the final result.

ConcFact uses another actor class called RangeProduct. The request message is read using the >> operation on self (line 24). ConcFact simply reformats the request message and

forwards the request to a RangeProduct actor (line 26). Note that ConcFact does not pass its own Cbox name in the request message sent to a RangeProduct actor. The reason is that ConcFact wants to have RangeProduct send the reply directly to main(). When RangeProduct performs a reply operation (line 42), a reply to myCbox of main() is sent since ConcFact does not change the reply destination.

The RangeProduct multiplies all numbers in the range specified by its two input arguments. The algorithm of RangeProduct is based on a divide-and-conquer technique. On receiving a request, a RangeProduct actor checks for a range containing only one number (line 32). If this condition is true, the lower limit is returned (line 33). Otherwise, the midpoint determining two sub-ranges is computed (line 35). To compute the products of these two sub-ranges concurrently, two new RangeProduct actors, rp1 and rp2 are created (lines 36-37). Each of rp1 and rp2 is sent a request message containing one sub-range (lines 39-40). This process continues until the sub-range being computed by a RangeProduct actor contains only one number.

Two sub-range products are eventually delivered to subCbox of the Range-Product actor which was created earliest. The actor multiplies the two sub-range products and sends the result to its reply destination, myCbox of the main() function. The use of a separate Cbox for each sub-range product is also possible. For this particular problem, distinguishing the two replies is not necessary. Hence, a single Cbox is used.

```
1    #include "act.h"                          // include the ACT++ kernel classes
2
3    class ConcFact: ACTOR {
4    public:
5      void compute_factorial();
6    };

7    class RangeProduct: ACTOR {
8    public:
9      void compute_product();
10   };

11   main()
12   {
13     int k;
14     int n = 20;    /* compute 20!  */
15     Cbox myCbox;
16     Mbox factorial = New(ConcFact);        // bind it to an instance of ConcFact;
17     factorial << &ConcFact::compute_factorial << myCbox << n;
                                              // send a request message
18     myCbox >> k;                           // receive a reply from factorial
19     printf("%d\n", k);
20   }

21   void ConcFact::compute_factorial()       // a method of ConcFact
22   {
23     int m;
24     self >> m;                             // read a request message
25     Mbox rp1 = New(RangeProduct);
26     rp1 << &RangeProduct::compute_product << 1 << m;
27   }

28   void RangeProduct::compute_product()     // a method of RangeProduct
29   {
30     int low, mid, high, sub1, sub2;
31     self >> low >> high;
32     if (low>=high)
33         reply(low);
34     else {
35         mid = (low+high) / 2;
36         Mbox rp1 = New(RangeProduct);
37         Mbox rp2 = New(RangeProduct);
38         Cbox subCbox;
39         rp1 << &RangeProduct::compute_product << subCbox << low  << mid;
40         rp2 << &RangeProduct::compute_product << subCbox <<mid+1<<high;
41         subCbox >> sub1 >> sub2;
42         reply(sub1*sub2);
43     }
44   }
```

Figure 32 - Concurrent Factorial Actor in ACT++

6.5.2 A Cruise Control Problem

The following is an adapted definition of a cruise control problem described in [Gehani 83].

> A cruise control maintains an automobile at a constant speed selected by the driver. The cruise control mechanism is set in action by driving the automobile at the desired cruising speed and then pushing the cruise control button. The cruise control mechanism is disengaged by depressing either the brake or the accelerator. When neither of these are depressed the cruise control mechanism must check the current speed and adjust the throttle to the requested cruising speed. It takes 0.1 second for the throttle to adjust its opening.

The modeling of the cruise control problem includes actors representing the accelerator, cruise button, brake, speedometer, throttle, and timer. A cruise control is also defined. For simplicity, only the definition of the cruise control actor is shown here. The cruise control actor may be in one of two states at any time: the cruise control is on or off. These states are defined by actor classes named CruiseOn and CruiseOff, respectively. From the description of the problem, the cruise control actor may receive a request message from the accelerator, cruise button, and brake. The cruise controller must also know the throttle actor, speedometer, and timer. The assumption is made that there exists some low level I/O mechanism which allows the actors to interact with the physical devices. The cruise button actor sends an on() message while the accelerator and the brake send an off() message to the cruise control actor. Given the previous description of the concepts of an actor program and the factorial example, one can easily follow the example in Figure 33.

```
#include "act.h"
class CruiseOff;
class CruiseOn;

class CruiseOn : ACTOR {
    Mbox mySpeedometer, myThrottle, myTimer;      // instance variables
    int cruisingSpeed;
public:
CruiseOn (Mbox aSpeedometer, Mbox aThrottle, Mbox aTimer, int aSpeed)
    { // constructor
        mySpeedometer = aSpeedometer;
        myThrottle = aThrottle;
        myTimer = aTimer;
        cruisingSpeed = aSpeed;
    }
void off()
    { // turn off cruise control
        become(new CruiseOff(mySpeedometer, myThrottle, myTimer));
    }
void on()
    { // control already on; no action required; ignore this request }
void maintain()
    { // try to maintain the specified cruising speed
        int currentSpeed, aSignal;
        Cbox speedCbox, timerCbox;
        mySpeedometer << &Speedometer::speed << speedCbox;
        speedCbox >> currentSpeed;
        myThrottle << &Throttle::adjust << cruisingSpeed - currentSpeed;
        myTimer << &Timer::wakeup << 0.1;
        timerCbox >> aSignal;
        become(Self);
        self << &CruiseOn::maintain;
    }
};

class CruiseOff : ACTOR {
        Mbox mySpeedometer, myThrottle, myTimer;
public:
CruiseOff (Mbox aSpeedometer, Mbox aThrottle, Mbox aTimer)
    {
        mySpeedometer = áSpeedometer; myThrottle = aThrottle; myTimer = aTimer;
    }
void on()
 { // turn on the cruise control
        int desiredSpeed;
        self >> desiredSpeed;
        become(new CruiseOn(mySpeedometer, myThrottle, myTimer, desiredSpeed));
        self << &CruiseOn::maintain;
 }
void off()
    { // control already off; no action required; ignore this message }
void maintain()
    { // control already off; no action required; ignore this message }
};
```

Figure 33 - A Cruise Controller

Several issues, however, are raised by this example. Messages for on() should not make any difference if the cruise control is already on. For example, the driver may press the cruise control button while the cruise control is engaged. The driver should see no visible effect. Similarly, off() and maintain() messages should be ignored if the cruise control is off. This problem is handled in the above program by dummy methods, which simply ignore the requests being processed. Two dummy methods are used: on() in CruiseOn and off() in CruiseOff(). However, this solution does not seem satisfactory since it is not feasible to have a dummy method for every unanticipated method. More importantly, some request messages cannot simply be ignored even if they are not served by the current behavior. They may need to be saved for later processing. This example illustrates a more general issue of handling request messages which are not recognized by the current behavior of an actor.

A different kind of difficulty arises when the accelerator or the brake pedal is depressed while the cruise control is on. Messages currently buffered in the mail queue must be discarded since the cruise control mechanism is disengaged. This requires that the off() message sent by the accelerator be served with a higher priority than the other buffered messages. The lack of an urgency notion in the actor model makes it difficult to model even a primitive real-time problem like the cruise control.

Although the classes CruiseOff and CruiseOn are defined as separate behaviors, defining them as full fledged classes is awkward because they are just parts of a larger behavior, say, CruiseControl. A more natural solution is to combine the two behaviors into a single class. This splitting of a coherent object definition is forced by the semantics of Agha's become operation. Splitting the behavior of an actor in this way is also undesirable from the perspective of class inheritance. One must define subclasses of both CruiseOff and CruiseOn in defining an actor using the cruise control actor. A more fundamental problem

resulting from the conflict between the become operation and class inheritance is discussed in the next chapter.

6.6 Implementation of the ACT++ Language Kernel

The classes shown earlier in Figure 30 were implemented using the AT&T C++ compiler in a Unix environment. Actors are created as light-weight processes which share the same Unix address space. In this section, a detailed description of the implementation is presented. First, an overview of the class hierarchy of the ACT++ language kernel is provided. The implementation of the kernel language primitives are discussed in detail. The implementation of the pseudo-variable self is also described. Finally, the run-time model of the language is described by following the thread of an actor which sends a message to another actor.

6.6.1 An Overview of Classes

The class hierarchy used for implementing the current ACT++ kernel appears in Figure 30. The class ACTOR extends its superclass ActiveObj with the addition of a become() method. The class ActiveObj defines the primitive behavior of all active objects. An instance of ActiveObj contains information needed by the scheduling mechanism. Instance variables for the size and the base address of the run-time stack, frame pointer, and argument pointer for the current thread are present in the ActiveObj class. Methods supported by ActiveObj include operations used to start, suspend, and resume a thread. These operations are used by the scheduling mechanism which is embodied in a class called Worker. Although the current implementation supports only a single Worker, a future design will support multiple Workers, each of which runs on a separate node in a network. The class Worker defines variables for scheduling policy information and provides methods needed to perform context switching of threads.

A MemoryManager object is responsible for dynamic storage allocation needed for running actors. Since the current implementation makes use of the C++ object creation mechanism for creating new actors, the function of the MemoryManager object is limited to allocation and deallocation of stack spaces. Future versions of the MemoryManager will incorporate automatic garbage-collection of actors. Garbage-collection of actors is known to be different from other traditional real-time garbage collectors, which are concerned about reclaiming passive data objects. The problem of garbage collecting actors is described in [Kafura 90].

The class Mail is used to create a message from a list of data items. An instance of Mail is a list of MsgToken objects. Each MsgToken object corresponds to a data item supplied in a message sending operation. The overloading of << methods allows a message to be created using a stream-like interface. The reverse operation >> is provided to allow data items to be extracted in a way similar to a stream input operation. The class Mail is not directly visible to a programmer. A programmer uses the << method of an Mbox when sending a message.

The Slink, Dlink, Slist, Dlist, and Queue classes implement generic data objects for singly and doubly linked nodes, singly and doubly linked lists, and queues. MailBox is implemented as a queue of Mail objects. Two mail boxes, Mbox and Cbox are subclasses of MailBox. An Mbox supports the method <<, which takes a method name and returns a reference to a Mail object which is newly added to the Mbox. Subsequent arguments, if any, of a message sending operation are attached to the returned Mail object using the << operation of the Mail object. The use of << operations in a message sending operation is overloaded for each type of argument. A Cbox supports the overloaded method << which allows a reply to be sent to the Cbox object. A Cbox object also provides the overloaded method >> for reading a reply from the Cbox. The method in() of a Cbox returns TRUE or FALSE depending on the availability of a reply message in the Cbox.

6.6.2 The Actor Operations

The **New** operation creates a new Mbox, and binds together the Mbox and a behavior supplied in the call. The binding is represented by setting a variable in each object to identify the other object. A pointer to the Mbox is returned.

The **become** operation changes the current behavior of an actor associated with the Mbox by resetting a variable in Mbox to point to the replacement behavior specified as the argument of the call. If a message is available in the mail queue, the replacement behavior is given the message. The become operation also modifies the execution state of the actor. Figure 34 shows the state transitions of a behavior.

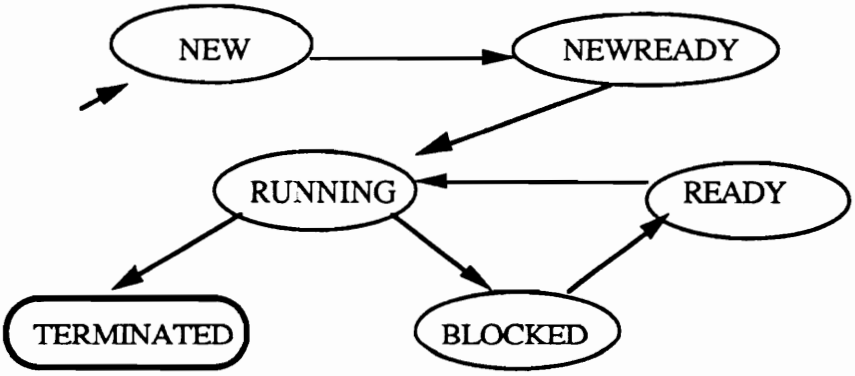


Figure 34 - State transitions of a behavior

A behavior has the NEW state when initially created. The behavior changes to the NEWREADY state if the mail queue has a request message. The behavior is given the message and placed in the ready queue of the Worker. If no message is found in the mail queue when a behavior is newly created, the behavior remains in the NEW state, which causes the behavior to wait for the arrival of a new message. When a message arrives at an Mbox, two possibilities exist. If the Mbox has a behavior waiting for a message, the

behavior is assigned the message. The behavior is then appended to the ready queue, and its state is changed to NEWREADY. Otherwise, the message is placed in the mail queue of the Mbox.

A behavior in the NEWREADY state is changed to RUNNING when the behavior is selected for execution by the scheduler. A RUNNING behavior is put in the BLOCKED state if the behavior attempts to **receive** from an empty Cbox. When a reply message arrives in the Cbox later, the behavior is moved to the READY state. The difference between NEWREADY and READY is that a behavior in the former is yet to be allocated a run-time stack while a behavior in the latter has already allocated a stack. The state TERMINATED indicates a behavior execution has been completed. The resources used by a behavior are collected for recycling by the system when the behavior enters the TERMINATED state.

The operation **reply** is implemented as a macro which is expanded to sending a message to a reply destination. The reply destination of a reply message is found in the structure self, which exists in each actor. A variable in self stores the reply destination specified in the request message currently being processed. We now describe the implementation of self.

6.6.3 Self

The pseudo-variable self is an instance variable defined in ACTOR. The variable is a reference to an instance of SelfObj which is essentially a pair of Mbox and Mail variables. The << operation applied to self invokes the << operation of the Mbox while the >> operation is directed to the Mail object. When an actor is created, the Mbox variable of self in the actor is initialized to the Mbox of the actor. The Mail variable of self is set to an instance of Mail when a message is bound to a behavior of the actor. The definition of the class SelfObj is shown in Figure 35.

```
class SelfObj : Object {
    Mbox& myMbox;                    // private instance variables
    Mail& myMail;
public:
Mail& operator<<(Method p)           // overload << to receive a
  {                                  // method name
    return (myMbox << p);
  }
  ...                                // overload << to receive args
                                     //  of a method invocation
Mail& operator>>(int x)              // overload >> to receive an
  {                                  //  integer
    return (myMail >> x);
  }
  ...                                // overload >> to receive other
                                     //  types of arguments
};
```

Figure 35 - Implementation of SelfObj

6.6.4 A Trace of an Execution Thread

In this section, a trace of the execution of the factorial actor discussed in Section 6.5 is presented in order to give a more detailed description of the implementation. Figure 36 shows the data structures related to an actor at run-time. An actor is represented by the actor's current behavior. The instance data structure of an actor class is the focal point of a running behavior. The instance of an actor class contains the self variable and a pointer to the run-time stack. The self object identifies a mail queue which represents the actor to which the behavior belongs and a request message being processed by the behavior. The mail queue is also linked to the current behavior by a pointer in its structure. This binding is changed when the current behavior specifies a replacement behavior. Associated with a mail queue are a list of request messages. Cboxes are allocated in the heap memory. The Cbox variables in the stack link a behavior to its Cboxes. Each Cbox is associated with a list of reply messages.

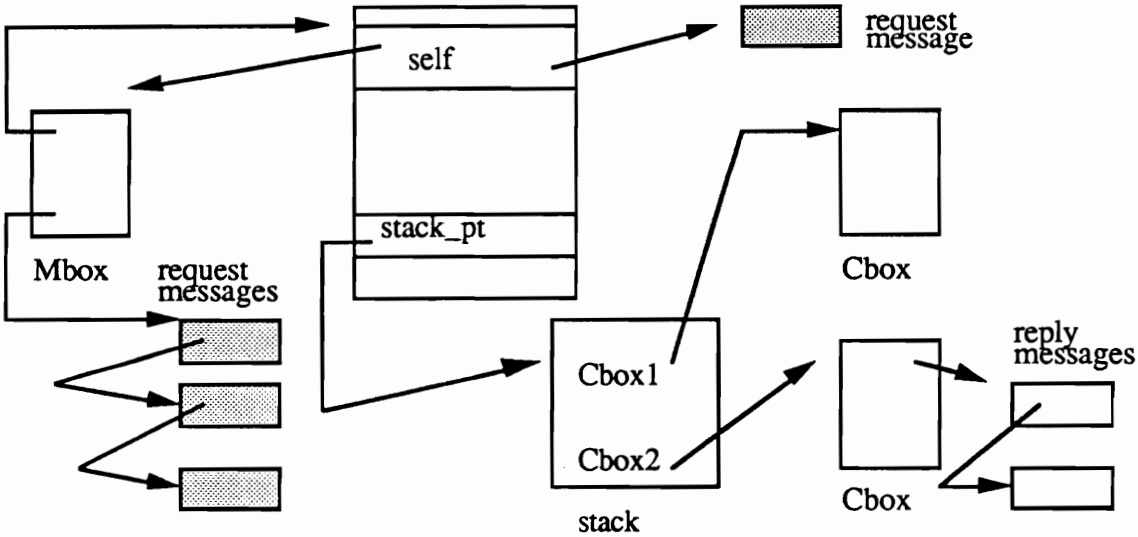


Figure 36 - Run-time data structures for an actor

The program shown in Figure 32 is used in the following discussion. The New operation (line 16) creates an Mbox and an instance of ConcFact. These two objects are bound together and the pointer to the Mbox is returned. The pointer is assigned to the variable factorial. The state of the new behavior is initialized to NEW. The next line sends a request message to factorial. In the current implementation, a method pointer is used as the method name to be invoked. The first << operation results in the creation of a Mail object, to which subsequent << operations are applied. The message consisting of a method pointer, a reply destination, and an integer is sent to the factorial actor. Since the current behavior of factorial is waiting for a message, the request message is bound to the behavior. The behavior is then placed in the ready queue of the Worker.

To allow main() to be regarded as an actor, an actor representing the thread of main() is created when the first ready behavior is put in the ready queue. From then on, main() is treated as an actor. In the program, main() executes the >> operation on myCbox (line 18). Since the Cbox has not yet received a reply, the main() actor is blocked. Register values are saved in the run-time stack while the frame pointer and the argument pointer are saved in its instance variables, defined by ActiveObj, so that the actor's state can be restored later. The state of the main() actor is changed to BLOCKED. An instance variable of myCbox is updated to contain the address of the main() actor. The Cbox myCbox is then marked as one on which an actor is waiting. The Worker (scheduler) now takes control. The Worker is implemented as a light-weight process which has its own stack. The Worker state is restored using the frame pointer and the argument pointer saved in its object structure.

The Worker schedules the next ready behavior found in its ready queue. A behavior found in the ready queue may be in either NEWREADY or READY state. Both states imply that a behavior is ready to run. More specifically NEWREADY indicates that the behavior is yet to be allocated a run-time stack, while READY indicates the behavior is resumed with its

old stack. In this example, the next ready behavior is in the NEWREADY state. The Worker first allocates a stack space with the assistance of the MemoryManager. The Worker then switches to the behavior by first saving its own state in its stack, and jumping to the entry point of the behavior. The control of the behavior automatically returns to the Worker when the behavior execution finishes. The entry point is located in the method pointer field of the request message being processed by the behavior and is retrieved via a method defined in ACTOR . Hence, line 18 results in blocking of the main() actor and the execution of the compute() method of a ConcFact actor.

The execution of the ConcFact actor causes many more actors to run concurrently. Eventually, the RangeProduct actor created by ConcFact receives replies from two actors computing sub-ranges (line 41), as discussed in Section 6.4. The RangeProduct actor sends the final result to myCbox of main() (line 42). The arrival of the reply to myCbox moves the main() actor from BLOCKED to READY, with the value in the message being assigned to the variable k. When main() is resumed by the scheduler, main() continues its execution at line 19. At this time, the value of k is available. Hence, main() prints the correct result of 20! and finishes. Since the return point for main() is not touched, the control returns in a normal way, i.e., to the operating system kernel.

Chapter 7 Inheritance in Actor Based Concurrent Object-Oriented Languages

7.1 Introduction

Is inheritance inconsistent with concurrency? The interference between inheritance and object-based concurrency has been noted by others [Briot 87,Wegner 87]. These observations center on the difficulty of locating or copying methods at run-time in systems without shared memory. However, this is not considered a fundamental difficulty because the performance penalty induced by inheritance may not always be a problem. At an extreme, the sharing problem may be avoided by allowing multiple copies of the same code and data on different nodes. [Briot 87] discusses a copy technique which is useful in this situation.

The fundamental interference between inheritance and concurrency is more deeply rooted. This difficulty can be observed in existing object-oriented languages, only a few of which support both concurrency and inheritance. The problem, to be described later, is that inheritance and concurrency control tend to interfere with each other. This interference results in concurrent object-based languages which either do not support inheritance or which do so only by severely compromising some other property. For example, one language supporting both concurrency and inheritance compromises object encapsulation [Yokote and Tokoro 86]. A second language excludes the possibility of inheriting synchronization code [Caromel 88]. A result of this restriction is limited leverage in reusability. In yet another language, inheritance had been tried but was removed later because of limited reusability [America 87A]. The same basic problem was found in the initial design of ACT++.

This chapter analyzes the approaches to inheritance and concurrency control in existing object-oriented languages and proposes a solution to the interference problem using a technique called *behavior abstraction*. In the remainder of this chapter, Section 7.2 classifies approaches to concurrency control in existing object-oriented languages. Based on this classification, an analysis of currently existing concurrent object-oriented languages is presented. Section 7.3 describes the conflict between inheritance and concurrency found in an actor based language. Section 7.4 discusses a solution to this problem.

7.2 Concurrency Control in Object-Based Languages

This section discusses the relationship between concurrency and inheritance in existing languages. The interference between the two mechanisms is observed.

An object in a concurrent object-oriented language may proceed in parallel with another object. Such an object has its own thread of control. An object with its own thread of control is called an *active* object. In contrast, an object in a sequential language does not possess its own thread of control. An object without its own thread of control is called a *passive* object. Concurrency implies the need for synchronization, without which the state of an active object may become inconsistent. Since the internal state of an object can only be accessed via method invocation, previous object-based concurrency control techniques were implemented inside the object. There are two directions in providing concurrency control. One approach centralizes concurrency control in a single procedure. This approach is called *centralized control*. The other approach distributes concurrency control among methods without a centralized procedure and is called *decentralized control*.

In centralized control, message reception is explicitly programmed using guarded commands or SELECT constructs. CSP [Hoare 78], ADA [DoD 83], ABCL/1 [Yonezawa

87], POOL-T [America 87A], and Extended Eiffel[1] [Caromel 88] belong to this category. There is a common problem in attempting to incorporate inheritance into these languages: synchronization constraints specified in the centralized procedure cannot be inherited by a subclass. This point becomes clearer when each of these languages is reviewed later in this section.

Two different approaches to concurrency control are found in languages with decentralized control. One approach uses *critical sections* and the other approach uses *interface control*. A majority of languages adopt the critical section approach. In these languages, each method is responsible for ensuring a certain condition before entering a critical section. Several languages use a locking mechanism. Each method must explicitly lock a variable before entering a critical section and then unlock the variable when exiting the critical section. Other languages use a construct similar to a conditional critical region [Hansen 72]. For example, a newer version of Concurrent Smalltalk provides a construct called *relinquish* which allows a thread to wait on a condition [Yokote 87]. In Trellis/Owl [Moss 87], the *lock block* structure automatically performs an unlock for a lock variable when its scope is exited.

Two problems exist in the approach based on critical sections. First, object encapsulation is weakened. Relying on a lock variable requires that the variable be visible to any subclass in the class hierarchy, which is a violation of encapsulation. A similar observation is made by [Snyder 86A]. Second, a method may violate the critical section protocol. For example, if explicit locking is used, a method may enter the critical section without performing the locking. This problem is compounded in a language which supports inheritance. Because

[1] The language is a concurrent extension of Eiffel [Meyer 87]. Since the language was not given a name in [Caromel 88], Extended Eiffel is adopted.

the subclass is separated from the superclass, there is a greater possibility that methods defined in a subclass may not observe the critical section protocol.

The other approach in decentralized control is based on direct control of the object interface. In this approach, called *interface control*, message reception is implicit. A method execution is initiated only when the method is allowed to access the internal state of the object. The underlying mail system delivers a message when the receiver is ready. Hybrid [Nierstrasz 87] and actor based languages such as Act2 [Theriault 83] and Act3 [Agha 86] are found in this category. For example, Hybrid provides constructs which control the availability of methods. A method may be closed temporarily so that messages for that method are not allowed to cross the object boundary. The blocked messages are processed later when the method is re-opened. In Act2 and Act3, synchronization of an active object is achieved with the become operation. This operation allows an object to change to another object, which may have a different interface and even different data structures. Neither Act2 nor Act3 supports inheritance. A serious problem occurs when adding inheritance to languages using interface control. Defining a new method in a subclass may invalidate many superclass methods.

The remainder of this section describes how the problem of combining inheritance and concurrency is manifested in the following concurrent object-based languages: POOL-T, Extended Eiffel, Concurrent Smalltalk, Hybrid, Act3, and ACT++. Each of these languages uses a different approach to concurrency control. While some of these languages do not support inheritance, they are included here since a review of these languages provides insight into the conflict between inheritance and concurrency.

POOL-T: the concurrency control approach of POOL-T [America 87A] is centralized. In POOL-T, the class definition of a concurrent object consists of a list of methods and a

separate procedure called *body* which specifies concurrency constraints. An object explicitly states its willingness to accept messages in the *body* using a construct similar to guarded command. POOL-T does not support inheritance. In fact, inheritance was tried in the initial design, but later removed. The decision to remove inheritance from POOL-T illustrates the general interference between inheritance and centralized control. The problem is that inheritance in a language with centralized control does not allow reuse of synchronization code. In centralized control languages like POOL-T, each time a subclass with a new method is defined, the *body* must be revised, since otherwise no new methods defined in the subclass can be executed. This consideration led the designer of POOL-T to choose not to include inheritance in the language.

Extended Eiffel: a concurrent extension of Eiffel proposed in [Caromel 88] supports both concurrency and inheritance using centralized control. An active object is defined as an instance of a subclass of a class called "PROCESS-POWER". A method in a class is not concerned with synchronization. Concurrency control is centralized into a single procedure called *Live* which is similar to the *body* of POOL-T. Extended Eiffel suffers from the same problem found in an earlier design of POOL-T. The method *Live* must be rewritten if a subclass adds a new method, regardless of the semantics of the method being added.

The approach of Extended Eiffel excludes the inheritance of synchronization code, and thereby severely restricts reusability. The synchronization code of *Live* may be a result of an extensive reasoning process. A subtle error may creep in during the process of copying and modifying the *Live* method. This is the very problem that inheritance intends to solve. While the separation of concurrency control from sequential action may allow a more readable definition of an object's behavior, readability can also be provided by a language which uses decentralized control. This problem is discussed in more detail section 7.4.

Concurrent Smalltalk: Concurrent Smalltalk is a concurrent extension of Smalltalk-80 [Yokote 86] which supports both concurrency and inheritance. The language uses critical sections for concurrency control. An active object, called an *atomic object*, serializes messages to maintain consistency of its internal state. Locking is used for concurrency control. An active object allows a method to be executed even when method execution is immediately blocked. In this case, the client object should block itself, terminating its current process. A provision is required in the code of the client which will send the same message again to the object when the client is restarted. Since the client is terminated and restarted, it must have a separate method which will do the retransmission.

This approach has several disadvantages. One is a weak object encapsulation. In the language, a sender must provide the method which will retransmit a message. This method obscures the readability of the program and imposes a burden on the sender. The sender is also required to understand the internals of the receiver object, violating the encapsulation principle of object-oriented languages. Another disadvantage is the use of unstructured constructs. For example, the BoundedBuffer problem described in [Yokote 86] uses a wait-signal primitive. The drawback of such a low-level primitive has been well recognized in operating systems research. A later version of Concurrent Smalltalk [Yokote 87] improves this situation by using a *relinquish* operation and the concept of a *secretary*, which is similar to conditional critical regions [Hansen 72]. This approach still has the disadvantages intrinsic to an approach based on critical sections.

Hybrid: Hybrid [Nierstrasz 87] is a concurrent object-oriented language based on decentralized control. The language provides a message queue called *delay queue* for concurrency control within an active object. Each method of an active object is associated with a delay queue. Synchronization control for accessing an object is achieved by explicitly closing and opening delay queues. Each method contains explicit statements for

controlling delay queues. A message which requests the execution of a method is blocked if the delay queue associated with the method is closed. The message is processed later when the delay queue is opened by some method.

Hybrid supports multiple inheritance. The concurrency control approach used in Hybrid presents a problem when attempting to define a subclass of an active object class. To appreciate this problem, consider adding a new method in defining a subclass. The new method may need to have its own delay queue which is not present in its superclass. The question then is how the methods of its superclass can control this delay queue. Unless the new delay queue is controlled solely by the new method itself, all superclass methods that need to open or close the delay queue must be revised so that the name of the delay queue may be referenced in their definitions.

Act3: Act3 is a concurrent object-oriented language based on the actor model as defined by Agha [Agha 86]. The language represents another approach in interface control. A main synchronization device of Act3 is the *become* operation. It is also the only synchronization primitive other than message passing operations. A become operation in a method specifies a replacement behavior, which receives the next unprocessed message. Each method execution must use a become operation to name a replacement behavior. Specifying a replacement behavior is the way an actor changes its state. In the actor model, both state change and synchronization control is accomplished using a single become operation. Act3 does not provide inheritance. A language which intends to support inheritance and the actor model of concurrency faces a fundamental problem, which is similar to that of Hybrid but more serious.

7.3 The Actor-Inheritance Conflict

In this section, ACT++ is used to illustrate the conflict between inheritance and concurrency in a language based on the actor model. Although ACT++ is used, the interference problem is not specific to ACT++ and also occurs in other languages combining concurrency and inheritance.

Using an example, the interference of inheritance and actor concurrency is now described. Consider producers and consumers communicating through a bounded buffer. The bounded buffer is modeled as an active object which is shared by producers and consumers. The buffer provides *get()* and *put()* methods to clients. Producers are actors which send put() requests when they want to deliver data items to consumers. A consumer actor sends a get() message to the buffer when the consumer needs a data item. A bounded buffer actor is empty when it is initially created. An empty buffer accepts only a put() message. If the buffer is neither empty nor full, it acts as a partially filled buffer which honors both get() and put() requests. If the buffer is full then it must accept only a get() request from a consumer. These three states are called empty_buffer, partial_buffer, and full_buffer, respectively. A possible transition sequence in the states of a bounded_buffer is

empty_buffer -> partial_buffer -> full_buffer -> partial_buffer -> empty_buffer

A subtle question now arises. What happens if the current state of an actor does not recognize the method name in a message? For example, what should be done if the next message to be processed contains a get() request while the buffer is empty? For the purpose of this section, assume that a message is put back at the end of the message queue. Figure 37 shows the definition of bounded_buffer in ACT++[1].

[1] While the primitive actor model assumes no structured types, such as array, ACT++ provides all data types of C++. For the purpose of this thesis, assume an array parameter is passed by value.

```
class bounded_buffer : ACTOR {
        int_array buf[MAX];
        int in,out;
public:
bounded_buffer()
        { in=0; out=0}
int get()
        {
                reply buf[out++];
                out %= MAX;
                if (in==out)
                        become(empty_buffer(buf,in,out));
                else
                        become(partial_buffer(buf,in,out));
        }
void put(int item)
        {
                buf[in++] = item;
                in %= MAX;
                if (in==out%MAX)
                        become(full_buffer(buf,in,out));
                else
                        become(partial_buffer(buf,in,out));
        }
};
```

Figure 37 - Definition of bounded_buffer

The syntax of ACT++ is close to that of C++. A few new constructs are added to support the actor abstraction as discussed in Chapter 6. In Figure 37, the operation **become** is used to specify a replacement behavior. A become operation takes an actor class as an argument. An actor class corresponds to a behavior script of the primitive actor model [Agha 86]. The operation **reply** is used to send a message to the sender of the message being processed. Since the class bounded_buffer is defined as a subclass of ACTOR, the bounded_buffer is an actor class whose instances are active objects, namely actors. An instance of bounded_buffer contains instance variables *in*, *out*, and the array *buf*. In C++, a method with the same name as the class name denotes a constructor. The procedure bounded_buffer() is a constructor.

To recognize the operations which are appropriate for different behaviors (eg. empty, full, partial) three classes of bounded buffer are introduced: namely, empty_buffer, full_buffer, and partial_buffer. These three classes are defined as subclasses of the class bounded_buffer. The subclass empty_buffer is the same as bounded_buffer except that it does not have the get() method. The subclass full_buffer is a bounded_buffer without put(). The subclass partial_buffer is exactly the same as the bounded_buffer class. These subclasses can be defined as restrictions of the class bounded_buffer. The definitions of the three subclasses follow.

```
class empty_buffer : bounded_buffer {
public:
        bounded_buffer::put;
};
class full_buffer : bounded_buffer {
public:
        bounded_buffer::get;
};
class partial_buffer : bounded_buffer {
public:
        bounded_buffer::get;
        bounded_buffer::put;
};
```

The first concern is that many similar classes must be defined to implement a bounded_buffer. This is a result of the natural mapping of the primitive actor model into a class-based object-oriented language. The use of the become operation implies that a different class be defined for each different interface. This is unpleasant since all of the different behaviors have almost the same methods, yet they all must be defined as distinct classes. However, the real problem occurs when a subclass with its own method is defined.

Suppose one implements a bounded buffer with a new method get_rear(), which returns the most recently deposited item, rather than the oldest one. Call this an extended_buffer. A plausible solution is to define the extended_buffer class as a subclass of bounded_buffer with an addition of a new method get_rear(). The extended_buffer should be able to inherit all other methods from bounded_buffer without change. This is not an unusual expectation

of a language with inheritance. Unfortunately, this solution does not work, as described below.

The possible behaviors of an extended_buffer are:

extended_empty_buffer	put()
extended_full_buffer	get(), get_rear()
extended_partial_buffer	get(), get_rear(), put()

Comparing these behaviors with those of bounded_buffer, reveals that extended_empty_buffer and empty_buffer have the same interface. Hence empty_buffer may be used in place of extended_empty_buffer in the new class definition. However, extended_full_buffer is different from full_buffer because of the new method get_rear() in the extended_full_buffer. Similarly, the behaviors extended_partial_buffer and partial_buffer are also different. Therefore, extended_full_buffer and extended_partial_buffer must be defined as new classes.

However, the problem does not end here. Notice that every method of bounded_buffer must be redefined in the definition of extended_buffer if the method refers to either of the two class names, full_buffer and partial_buffer. Since both get() and put() use partial_buffer, none of these method can be inherited. Hence, extended_buffer inherits no methods from its superclass. All of its methods must be implemented within its own definition! This argument equally applies to an attempt to define extended_full_buffer as a subclass of full_buffer and extended_partial_buffer as a subclass of partial_buffer. The point of this example is that no methods of the superclass can be reused in the definition of a subclass because of the conflict introduced by the concurrency requirement.

A similar interference problem exists in Hybrid. In both ACT++ and Hybrid, superclass methods are not independent of new methods being defined in a subclass. The degree of dependency is, however, higher in a language based on the actor model of computation.

7.4 Inheritance in Actors

Having described the conflict of concurrency and inheritance, a solution to this problem is now presented in the framework of an actor based language.

7.4.1 A Model of an Object Manager

Each active object (actor) is associated with an *object manager*. The object manager is responsible for protecting the object from unauthorized requests and for dispatching method invocations. An object manager is automatically created when an object is created. The object manager immediately starts and continues until the object is destroyed.

The object manager of an object protects the object by enforcing the interface of the object. Using the terminology of Hybrid, the *interface* of an object consists of all open methods. A method is *open* if the current interface of object can accept a message for the method. Otherwise, a method is *closed*. The interface of an object is dynamically changed since methods can be opened or closed during computation. Methods are closed by the object manager and opened by a method in execution, called a thread (see below). A message for a method invocation is *authorized* if the method is open. A message for a closed method is *unauthorized*.

The object manager waits for the arrival of an authorized message. On finding such a message, the object manager closes all methods and creates a thread which performs the requested method. Unauthorized messages are buffered by the object manager until their corresponding methods are opened. Closed methods may be opened by a become operation

executed by a thread. A become operation specifies a set of methods to be opened. A thread can perform the become operation only once in its life. Since a thread is created as the result of the previous become operation, no thread but the most recently dispatched thread can execute the become operation. The become operation will open at least one method; otherwise, the actor is garbage collected.

There may exist multiple threads inside an object since the become operation may be executed prior to the termination of a thread. All the threads proceed in parallel. A thread dies when the execution of the method represented by the thread is completed. Among threads, no variables are shared.

7.4.2 Implementation of the object manager

The function of an object manager is well-defined and uniform for every object. Hence, a programmer does not need to write the object manager. The object manager can be provided through compiler and run-time support. This obviates the need for concurrency control mechanisms to be centralized in one method. The object manager can be implemented either as a function of the mail queue or as a special thread. The former will result in a sophisticated mail queue while the latter is similar to a process scheduler.

The object manager will need to keep track of the object's interface, which is changed by a become operation of that object's most recently dispatched thread. This problem along with the interference problem can be solved by redefining the way a replacement behavior is specified. For this purpose, the concept of *behavior abstraction* is introduced.

7.4.3 Behavior abstraction

A *behavior name* is a handle for a set of open method names. For example, consider the bounded_buffer. The buffer actor has one of the following behaviors:

empty_buffer = { put()}

full_buffer = {get()}

partial_buffer = {get(),put()}

That is, three behavior names are defined; namely, empty_buffer, full_buffer, and partial_buffer.

A become operation specifies a replacement in terms of a behavior name. For example, "become full_buffer" is acceptable if full_buffer is a behavior name. The language should provide a convenient way for specifying behavior names. For example, defining a behavior name using a regular expression may be desirable. Note the difference in the usage of "behavior" in the primitive actor model and in the ACT++ model. In the ACT++ model, a behavior denotes a set of open methods, while a behavior in the primitive actor model means a script of an actor.

The solution to the problem of an extended_buffer which inherits from the bounded_buffer is now presented. The bounded_buffer and the extended_buffer are defined using the behavior names defined earlier in this section. Figure 38 shows a new definition of the bounded_buffer using behavior abstraction. The definition of an extended_buffer which inherits from the bounded_buffer is shown in Figure 39. The extended_buffer has three distinct behaviors. Each of these constitutes a behavior name. The following names are defined as the relevant behaviors for an extended buffer:

extended_empty_buffer = {put()}

extended_full_buffer = {get(), get_rear()}

extended_partial_buffer = { get(), get_rear(), put()}

Now consider the relationship between the behavior names of the subclass and those of the superclass. Consider the put() method, which is inherited from the superclass bounded_buffer. The new operation get_rear() does not belong to any behavior names named by put(). It is necessary to let the method put() know that get_rear() is added in the definition of the extended_buffer. This is accomplished by redefining the behavior names used in superclass methods. The redefinition is expressed by the "redefines" construct. The new definition of a behavior name is used by all superclass methods. In some cases, the redefinition of a behavior name does not change the set of methods. Such renaming may be desired to provide the object with a more appropriate name. For this reason, empty_buffer is redefined as extended_empty_buffer without changing its meaning using the "renames" construct.

```
class bounded_buffer : Actor {
        int_array buf[MAX];
        int in,out;
behavior:
        empty_buffer = {put()};
        full_buffer = {get()};
        partial_buffer = {get(),put()};
public:
buffer()
        {
                in=0;
                out=0;
                become empty_buffer;
        }
void put(int item)
        {
                buf[in++]=item;
                in %= MAX;
                if (in==(out+1)%MAX)
                        become full_buffer;
                else
                        become partial_buffer;
        }
int get()
        {
                reply buf[out++];
                out %= MAX;
                if (in==out)
                        become empty_buffer;
                else
                        become partial_buffer;
        }
};
```

Figure 38 - Bounded_buffer with behavior abstraction

```
class extended_buffer : public bounded_buffer {
behavior:
    extended_empty_buffer                          renames empty_buffer;
    extended_full_buffer      = {get(), get_rear()}        redefines full_buffer;
    extended_partial_buffer   = {get(), get_rear(), put()} redefines partial_buffer;
public:
extended_buffer()
    {
      in=out=0;
      become extended_empty_buffer;
    }
int get_rear()
    {
      reply(buf[--in%max]);
      if (in==out)
        become extended_empty_buffer;
      else
        become extended_partial_buffer;
    }
};
```

Figure 39 - Definition of extended_buffer

Using behavior names also has other advantages. First, behavior names improve program readability. With more expressive and meaningful names, a program is more readable because the next interface is denoted by the behavior name used in a become operation. Second, an active object requires that a programmer understand its dynamic run-time behavior. While the centralized approach provides an effective way to tackle this issue by separating concurrency control from sequential actions, the approach has the drawback of excluding the inheritance of synchronization code. While the ACT++ model allows inheritance, it also allows separation of concurrency. Third, the synchronization mechanism is structured because no matching primitive is needed for the become operation. This is an important requirement of synchronization primitives proposed for incremental programming. This avoids such problems as a new method defined in a subclass which forgets to signal superclass methods or fails to observe a critical section protocol. Fourth, an object-oriented design methodology is supported. The behavior names provide a level of abstraction whose granularity is smaller than data abstraction but larger than procedural abstraction. With behavior abstraction, the behavior of an object can be modeled as state transitions among behavior names. Each of these names provides a higher level abstraction which is more relevant to a programmer's conceptual view of an object. The state-transition behavior of an object is naturally expressed with the behavior abstraction.

7.4.4 Implementation of Behavior Abstraction

The mechanism of redefining a behavior name in a subclass is analogous to that of virtual function in C++ [Stroustrup 86]. Every behavior name declared in a class may be regarded as a "virtual behavior" whose meaning a subclass may override. As in a virtual function invocation, a behavior name used by an object in a become operation may denote different behaviors which are decided by the type of the object. One implementation scheme resembles the virtual function table of C++. The compiler creates a table of behavior names

for each class. The become operation is translated into specifying a set of open methods which are found using a behavior name as an index into the table.

7.4.5 Enabled Set

An extension of Behavior Abstraction called Enabled Set is described in [Tomlinson 89]. Enabled Set is used in the Rosette language, an actor based object-oriented language. As with Behavior Abstraction, Enabled Set supports the notion of a dynamically changing object interface. An enabled set corresponds to a behavior name in our scheme. An enabled set is a set of messages which may be processed in the next state. An interesting property of the Enabled Set mechanism is that enabled sets can be composed. For example, the Rosette system supports an operation which can be used to produce the union of two enabled sets.

7.4.6 Actor Concurrency and Static Typing

While the focus of this chapter is on combining inheritance with static typing and actor concurrency, a natural question at this point is how actor concurrency and static typing are related. In general, the interference between concurrency and static typing is not a serious problem. In this section, static typing and actor concurrency are considered, in particular, behavior abstraction. An argument is offered that the adapted actor model as used in ACT++ and behavior abstraction work well with static typing.

One of the fundamental principles of the actor model is the *openness* which allows an actor to specify a replacement behavior which has no relation to the current behavior. While openness provides a flexible computation model which may be supported by a dynamically typed language, such an unbounded become operation is viewed as a harmful construct, like the indiscriminate use of goto statements. Further, the unbounded become operation is a significant obstacle to be overcome in the design of a language with static typing. In the

presence of openness, type-checking of a message is not possible since an actor's behavior may change without restriction during execution.

For the above two reasons, the ACT++ form of the model modifies the way an actor may change its behavior. Behavior Abstraction assumes that the set of behaviors which an object can take are known at compile time. Under this assumption, each method invocation can be checked for type-correctness. Type checking ensures that an object is not requested to perform an unknown method.

Chapter 8 Conclusions

This thesis discussed the issues involved in designing a statically typed object-oriented programming language which supports the actor model of concurrency. The problem of combining static typing and concurrency with class inheritance was presented and a solution was offered. Static typing and concurrency were reviewed from the perspective of inheritance, which is the characterizing feature of object-oriented programming.

The feature interactions and the interferences among the three features were observed. An analysis of existing object-oriented languages and models was made to observe the interference between inheritance and static typing. The result was that in most current object-oriented languages, the feature interaction between inheritance and static typing resulted in reduced reusability and weakened type safety. The issues related to enhancing reusability and type safety were identified: subtyping, multiple inheritance, method exclusion, multiple representation, overloading. A new model of type and inheritance, called HANA, which integrates these features into a single language framework was described in detail.

The interference between inheritance and concurrency was identified by several researchers as a difficulty in sharing methods in distributed environments. In this thesis, a more fundamental difficulty in combining concurrency and inheritance was described. Existing concurrent object-oriented languages were analyzed from this perspective and a solution to this problem in the framework of the actor model of concurrency was presented. A technique called Behavior Abstraction was described in detail. The use of Behavior Abstraction was demonstrated by an example written in the exploratory language ACT++.

The work done in this dissertation may be divided into two areas: combining static typing with inheritance and combining inheritance with actor concurrency.

Integration of Static Typing and Inheritance

A new model of type and inheritance, called HANA, was presented which integrates static typing into an object-oriented language. The problems and issues related to combining static typing and inheritance were identified and resolved.

- Multiple Representation

  The support of multiple representation allows an abstraction to be implemented in several ways in the same program. HANA achieves this by adding the notion of abstract type to the model rather than separating inheritance from subtyping. Since class is a type in HANA, the implementation of a method invocation can be optimized based on the knowledge of the object structure in the presence of multiple representation. More expensive late binding is confined to the invocations on abstract type variables. If class type variables are dominant within a program, the efficiency of the program is close to the one without multiple representation.

- Method Exclusion

  HANA enhances software reusability by supporting method exclusion as an explicit type definition mechanism. An observation was made that most current type systems cannot support method exclusion without weakening type safety. Although object-oriented languages equating inheritance with subtyping cannot support method exclusion, they implicitly use method exclusion, violating type safety. An example of this is a subclass which overrides an inherited method with a different parameter type. Since the inherited method is effectively excluded from the interface of the subclass, a run-time error may occur in a type correct program. HANA

resolves this problem by providing method exclusion as an explicit type definition mechanism. HANA's subtyping rule enables method exclusion to be supported without violating subtyping constraints.

- Explicit Type Hierarchy

  Determining complete behavior compatibility requires the support of formal semantics specification techniques, which are not yet mature to be included in a programming language. While those languages separating inheritance from subtyping use interface conformance as the criteria for deciding behavior compatibility, interface conformance alone is weak as observed by other researchers. Inheritance offers a middle ground for this problem. Like a type declaration, the programmer can explicitly state the behavior compatibility between types through inheritance. HANA's subtyping based on both inheritance and interface conformance allows a type hierarchy to be explicitly created by the programmer.

- Method Name Overloading

  Overloading extends the expressive power of an object-oriented language. Overloading accepts many useful subtype relations which other type systems without overloading cannot. A new overriding rule was described in conjuction with method name overloading and an example which demonstrated its power was presented. In addition, overloading as a natural solution to the name collision problem was demonstrated. The interference between parameter subtyping and overloading was observed. The notion of binding ambiguity was characterized. A solution for binding ambiguity was offered which is based on the detection of ambiguous invocations, rather than blindly rejecting the definition of a type with binding ambiguity.

- Name Collision

  The name collision problem in multiple inheritance was described, as was the interaction between subtyping and existing approaches to name collision. The problem of an object becoming inconsistent was described. The weakness and strength of resolution by method overriding and renaming approaches were analyzed. It was shown that the mechanisms of method exclusion, overriding, and renaming are complementary to each other with respect to the power of resolving name collision. It was also shown that method name overloading offers a natural solution for many name clashes.

- Late Binding

  A fast and efficient dynamic method binding mechanism for statically typed multiple inheritance object-oriented languages was presented. A technique called Hierarchy Partitioning was described in detail. Hierarchy Partitioning is an extension of the virtual function table approach and performs with an efficiency close to that of single inheritance C++ in both time and space. One of the novel features of the technique is that it exploits the structure of the given inheritance hierarchies. When many classes have only single parents, which is often the case, the efficiency of our technique becomes approximately equal to that of single inheritance C++. Another contribution is that Hierarchy Partitioning supports incremental compilation. The technique causes little additional overhead in compile time. Late binding of an example class hierarchy using Hierarchy Partitioning was described in detail. A detailed comparison of our technique with other related techniques was provided and the Hierarchy Partitioning scheme was shown to be superior to those existing techniques.

Integration of Inheritance and Actor Concurrency

- ACT++

The incorporation of the concurrency abstraction of the actor model and class inheritance was described. A prototype object-oriented language called ACT++, which is based on C++ and actor concurrency, was developed. The design and implementation of ACT++ is based on a class hierarchical extension of C++. The classes in the hierarchy were described in some detail. Two example programs written in ACT++ were presented. An argument was offered that the primitive actor model needs to be adapted for efficient implementation on conventional architectures. Specifically, the actor model should allow creation of more coarse-grained actors. The problem of a coherent object being broken into small pieces was illustrated by an example. The argument was presented that request messages and reply messages should be handled differently. The Cbox mechanism was proposed for dealing with reply messages. The flexibility of the Cbox mechanism was also demonstrated.

The fundamental characteristics of the actor model are active objects, asynchronous message passing, dynamic creation of new actors, reconfiguration of actor topology, and the become operation. The formal actor model defined by Agha must be recast in the light of the intended applications. A language may be said to be based on the actor model if the language supports the above five characteristics.

- Behavior Abstraction

A fundamental interference between concurrency and inheritance was described. The occurrence of the concurrency-inheritance conflicts in existing object-oriented languages was observed, discussed, and a solution offered. The approaches to inheritance and concurrency control were analyzed and classified. In the languages

using centralized control, synchronization constraints specified in a centralized procedure cannot be inherited by a subclass. Among the languages with decentralized control, some languages compromised encapsulation for inheritance while others were unable to support incremental modification of a class. The actor-inheritance conflict caused by the become operation was described using a bounded buffer example. The technique of Behavior Abstraction which resolves the interference problem was described in detail. Behavior Abstraction allows coarser-grained actors to be defined and provides a temporal abstraction of the behavior of an object.

The interface of a passive object is fixed throughout its life. In contrast, the interface of an active object changes dynamically as shown by Behavior Abstraction. The set of open methods in each state is explicitly specified by a become operation. In current type systems, type-correctness of a method invocation is simply a guarantee that the method has been defined by the receiver object. As with other statically typed concurrent programming languages, the discussion of type-checking assumed this guarantee of existence. However, the explicit specification of the interface supported by Behavior Abstraction seems to offer a new possibility of extending the meaning of type-checking in a concurrent program. Some of the problematic situations such as deadlock and starvation may be easily detected when Behavior Abstraction is used. For example, an invocation of a method which is defined but is never enabled will lead the sender to a deadlock. Such a situation is detectable by purely syntactic analysis. While it may be questionable whether type-checking should include this kind of analysis and how much leverage can be provided, the direction seems worthy of exploration.

9 References

[Agha 86] Gul Agha, *A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[Agha and Hewitt 87] Gul Agha and Carl Hewitt, Concurrent Programming Using Actors, In Object-Oriented Concurrent Programming, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 37-53.

[Aksit and Tripathi 88] Mechmet Aksit and Anand Tripathi, Data Abstraction Mechanisms in Sina/st, OOPSLA '88 Proceedings, 1988.

[America 87A] P. America, POOL-T: A Parallel Object-Oriented Language, In *Object-Oriented Concurrent Programming*, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 199-220.

[America 87B] Pierre America, Inheritance and Subtyping in a Parallel Object-Oriented Language, ECOOP '87 Proceedings, LNCS, Vol 276, Springer-Verlag, 1987.

[Athas 88] W. C. Athas and N. J. Boden, Cantor: An Actor Programming System for Scientific Computing, Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, vol. 24, No 4, April 1989.

[Attardi 87] Giuseppe Attardi, Concurrent Strategy Execution in Omega, In Object-Oriented Concurrent Programming, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 259-276.

[Baker 82] T. P. Baker, A One-Pass Algorithm for Overload Resolution in Ada, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, October 1982, 601-604.

[Barry 87] Brian M. Barry and D.A. Thomas, Using Objects to Design and Build Radar ESM Systems, OOPSLA '87 Conference Proceedings, SIGPLAN, ACM, 1987.

[Biggerstaff 87] T. Biggerstaff and C. Richter, Reusability Framework, Assessment, and Directions, IEEE Software, March 1987, 41-49.

[Black 87] Andrew Black, Norman Hutchinson, et al., Distribution and Abstract Types in Emerald, IEEE Trans. Soft. Eng., Vol SE-13, No. 1, 1987.

[Bobrow 86] Daniel G. Bobrow, Kenneth Kahn et al., CommonLoops:Merging Lisp and Object-Oriented Programming, OOPSLA '86 Proceedings, 1986.

[Borgida 86] Alexander Borgida, Exceptions in Object-oriented Languages, SIGPLAN Notices, Vol 21, No 10, October 1986.

[Borning 82] A. H. Borning and D. H. Ingalls, A Type Declaration and Inference System for Smalltalk, Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, 1982.

[Borning 86] A. H. Borning, Classes versus Prototypes in Object-Oriented Languages, Proceedings of the ACM / IEEE Fall Joint Computer Conference, Dallas, Tx, November 1986, 36-40.

[Briot 87] J.-P. Briot and A. Yonezawa, Inheritance and Synchronization in Concurrent OOP, ECOOP '87 European Conference on Object-Oriented Programming, Springer-Verlag, June 1987, 33-40.

[Briot 89] Jean-Pierre Briot, Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment, ECOOP '89, July 1989.

[Buhr 88] P. A. Buhr, G. J. Ditchfield, and C. R. Zarnke, Concurrency in C++, Research Report CS-88-30, University of Waterloo, Waterloo, Ontario, July 1988.

[Campbell 87] R. H. Campbell, J. Johnston, V. F. Russo, Choices: Class Hierarchical Open Interface for Custom Embedded systems, Operating Systems Review 21, July 1987, 9-17.

[Canning 89] Peter Canning, William R. Cook, et al., Interface for Strongly-Typed Object-Oriented Programming, OOPSLA '89 Proceedings, 1989.

[Cardelli 84] Luca Cardelli, A Semantics of Multiple Inheritance, LNCS Vol. 173, Springer-Verlag, 1984.

[Cardelli and Wegner 85] Luca Cardelli and Peter Wegner, On Understanding Types, Data Abstraction, and Polymorphism, Computing Surveys, Vol 17, No 4, December 1985.

[Caromel 88] Denis Caromel, A Generalized Model for Concurrent and Distributed Object-Oriented Programming, Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, Vol 24, No 4, April 1989.

[Clinger 81] W. D. Clinger, Foundations of Actor Semantics, Technical Report 633, MIT Artificial Intelligence Laboratory, 1981.

[Connor 89] R.C.H. Connor, A. Dearle, R. Morrison, and A.L. Brown, An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance, OOPSLA '89 Proceedings, 1989.

[Cook 89A] William R. Cook, A Proposal for Making Eiffel Type-Safe, Proceedings of ECOOP '89 - European Conference on Object-Oriented Programming, Cambridge University Press, July 1989.

[Cook 89B] William R. Cook, Walter L. Hill, et al., Inheritance Is Not Subtyping, Conference Record of Seventh Annual ACM Symposium on Principles of Programming Languages, 1989.

[Cox 86] B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison Wesley, 1986.

[Decouchant 89] D. Decouchant, S. Krakowiak, et al., A Synchronization Mechanism for Typed Objects in a Distributed System, SIGPLAN Notices, Vol 24, No 4, April 1989.

[Diederich 87] J. Diederich and J. Milton, Experimental Prototyping in Smalltalk, IEEE Software, May 1987, 50-64.

[Delagi 88] B. A. Delagi, N. P. Saraiya, ELINT in LAMINA: Application of a Concurrent Object Oriented Language, Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, vol. 24, No 4, April 1989.

[Dixon 89] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan, A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance, OOPSLA '89 Conference Proceedings, 1989.

[Freeman 81] P. Freeman, Reusable Software Engineering: Concepts and Research Directions, Proceedings of Workshop on Reusability in Programming, ITT, Shelton, Conn., 1983.

[Gehani 83] Narain Gehani, *Ada Concurrent Programming*, Prentice-Hall, 1983.

[Gentlemen 85] W. M. Gentlemen, Using the Harmony Operating System, National Research Council of Canada Report No. 24685 National Research Council of Canada, Ottawa, Canada, May 1985.

[Goldberg and Robson 83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Graver 90] Justin O. Graver and Ralph E. Johnson, A Type System for Smalltalk, Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, 1990.

[Hansen 72] P. B. Hansen, Structured Multiprogramming, CACM, Vol. 15, No. 7, July 1972.

[Hewitt 77] Carl Hewitt, Viewing Control Structures as Patterns of Passing Messages, Journal of Artificial Intelligence, 323-364, June, 1977.

[Hewitt and Baker 77] Carl Hewitt and H. Baker, Laws for Communicating Parallel Processes, IFIP Conference Proceedings, August 1977.

[Hewitt and Atkinson 79] C.E. Hewitt and R. Atkinson, Specification and Proof Techniques for Serializers, IEEE Transactions on Software Engineering, Vol. SE-5, No 1, January 1979.

[Hewitt and deJong 83] Carl Hewitt and Peter de Jong, Analyzing the Roles of Descriptions and Actions in Open Systems, Proceedings of the National Conference on Artificial Intelligence, AAAI, August 1983.

[Hoare 78] C. A. R. Hoare, Communicating Sequential Processes, CACM, August, 1978.

[Johnson 86] R. E. Johnson, Type-Checking Smalltalk, OOPSLA '86 Conference Proceedings, 1986.

[Johnson 88] R. E. Johnson, J. O. Graver, and L. W. Zurawski, TS: An Optimizing Compiler for Smalltalk, OOPSLA '88 Conference Proceedings, 1988.

[Kafura 88] Dennis Kafura, Concurrent Object-Oriented Real-Time Systems Research, Technical Report, 88-47, Department of Computer Science, Virginia Polytechnic Institute and State University, 1988.

[Kafura 89] Dennis Kafura, Concurrent Object-Oriented Real-Time Systems Research, SIGPLAN Notices, ACM, February 1989.

[Kafura and Lee 89A] Dennis Kafura and Keung Hae Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, Proceedings of ECOOP '89 - European Conference on Object-Oriented Programming, Cambridge University Press, July 1989.

[Kafura and Lee 89B] Dennis Kafura and Keung Hae Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, Computer Journal, Vol 32, No 4, August 1989.

[Kafura and Lee 89C] Dennis Kafura and Keung Hae Lee, ACT++: Building a Concurrent C++ with Actors, Journal of Object-Oriented Programming, To appear in May 1990, also Technical Report 89-18, Department of Computer Science, Virginia Tech, 1989.

[Kafura 90] Dennis Kafura, Doug Washbaugh, and Jeff Nelson, Garbage Collection of Actors, Technical Report 90-9, Department of Computer Science, Virginia Tech, 1990.

[Knudsen 88] Jorgen Lindskov Knudsen, Name Collision in Multiple Classification Hierarchies, Proceedings of ECOOP '88 - European Conference on Object-Oriented Programming, Springer-Verlag, 1989.

[LaLonde 86] Wilf R. LaLonde, Dave A. Thomas, John R. Pugh, An Exemplar Based Smalltalk, OOPSLA '86 Proceedings, 1986.

[Lee and Kafura 89] Keung Hae Lee and Dennis Kafura, A Fast and Efficient Method Dispatching for Statically Typed Multiple Inheritance Object-Oriented Languages, TR 89-40, Department of Computer Science, Virginia Tech, 1989.

[Lee and Kafura 90] Keung Hae Lee and Dennis Kafura, HANA: A Model of Type and Inheritance for Object-Oriented Programming Languages, TR 90-7, Department of Computer Science, Virginia Tech, 1990.

[Lieberman 86] H. Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Languages, OOPSLA '86 Conference Proceedings, 1986.

[Lieberman 87] H. Lieberman, Concurrent Object-Oriented Programming in Act 1. In *Object-Oriented Concurrent Programming*, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 9-36.

[Lieberman 88] H. Lieberman, L. Stein, and D. Ungar, Treaty of Orlando, Addendum to the Proceedings of OOPSLA '87, Special Issue of SIGPLAN Notices 23, 5, May 1988.

[Liskov 77] B. Liskov, A. Snyder, et al., Abstraction Mechanisms in CLU, CACM,Vol 20, No. 8, August 1977.

[Liskov 87] B. Liskov, Data Abstraction and Hierarchy, OOPSLA '87 Addendum to the Proceedings, 1987.

[Lunau 89] Charlotte Pii Lunau, Separation of Hierarchies in Duo-Talk, Journal of Object-Oriented Programming, July 1989.

[Meyer 87] B. Meyer, Eiffel: Programming for Reusability and Extensibility, SIGPLAN Notices, Vol. 22, No 2, January 1987.

[Meyer 88] Bertrand Meyer, Object-Oriented Software Construction, Prentice-Hall, 1988.

[Moon 86] David A. Moon, Object-Oriented Programming with Flavors, OOPSLA '86 Proceedings, 1986.

[Moss 87] J. B. Moss and W. H. Kohler, Concurrency Features for the Trellis/Owl Language, ECOOP '87 European Conference on Object-Oriented programming, Springer-Verlag, June 1987, 171-180.

[Nierstrasz 87] O. M. Nierstrasz, Active Objects in Hybrid, OOPSLA '87 Conference Proceedings, 1987, 243-253.

[Papathomas 89] M. Papathomas, Concurrency Issues in Object-Oriented Programming Languages, In *Object Oriented Development*, (ed.) D. Tsichritzis, University De Geneve, July 1989.

[Parnas 72] D. Parnas, On the Criteria to be used in Decomposing Systems into Modules, CACM, Vol. 15, No. 12, December 1972.

[Rose 88] John R. Rose, Fast Dispatch Mechanism for Stock Hardware, OOPSLA '89 Conference Proceedings, 1988.

[Schaffert 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilport, An Introduction to Trellis/Owl, OOPSLA '86 Conference Proceedings, 1986.

[Snyder 86A] Alan Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages, OOPSLA '86 Proceedings, 1986.

[Snyder 86B] Alan Snyder, CommonObjects: An Overview, SIGPLAN Notices, Vol 21, No 10, October 1986.

[Stein 87] L. Stein, Delegation is Inheritance, OOPSLA '87 Conference Proceedings, 1987, 138-146.

[Stroustrup 86] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley 1986.

[Stroustrup 89] Bjarne Stroustrup, Multiple Inheritance for C++, Computing Systems, Vol 2, No 4, 1989.

[Suzuki 81] Norihisa Suzuki, Inferring Types in Smalltalk, Conference Record of the Eight Annual ACM Symposium on Principles of Programming Languages, 1981.

[Theriault 83] D. G. Theriault, Issues in the Design and Implementation of Act2, Technical Report 728, MIT Artificial Intelligence Laboratory, 1983.

[Tomlinson 88A] Chris Tomlinson and M. Scheevel, Concurrent Object-Oriented Programming, Microelectronics and Computer Technology Corporation, MCC Technical Report, ACA-ST-078-88, March 1988.

[Tomlinson 88B] Chris Tomlinson, M. Scheevel, and Won Kim, Object-Oriented Systems: Delegation vs Inheritance and Classes vs Prototypes, MCC Technical Report, ACA-ST-179-88, Agust 1988.

[Tomlinson 89] Chris Tomlinson and Vineet Singh, Inheritance and Synchronization with Enabled-Sets, OOPSLA '89 Proceedings, October 1989, 103-112.

[Ungar 87] D. Ungar and R.B. Smith, Self: The power of Simplicity, OOPSLA '87 Conference Proceedings, October 1987, 227-242.

[Wegner 87] P. Wegner, Dimensions of Object-Based Language Design, OOPSLA '87 Conference Proceedings, 1987, 168-182.

[Wegner and Zdonic 88] Peter Wegner and Stanley Zdonic, Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like, ECOOP '88 Proceedings, LNCS Vol 322, Springer-Verlag, 1988.

[Yokote and Tokoro 86] Yasuhiko Yokote and Mario Tokoro, Concurrent Programming in Concurrent Smalltalk, In *Object-Oriented Concurrent Programming*, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987.

[Yokote and Tokoro 87] Y. Yokote and M. Tokoro, Experience and Evolution of Concurrent Smalltalk, OOPSLA '87 Conference Proceedings, 1987, 406-415.

[Yonezawa 87] Akinori Yonezawa, Etsuya Shibayama, et al., Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, In Object-Oriented Concurrent Programming, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 55-89.

Vita

Keung Hae (Michael) Lee was born on September 1, 1956 in Jinchon, Choongbuk, Korea. He is married to Kyung Hee Kim and have a lovely daughter, Donna Somee Lee.

He earned a Bachelor's degree in Electronic Engineering from Sogang University, Korea, in 1980. In 1981, he joined the staff of Hyundai Engineering and Construction Company in Korea, as a programmer. In the Fall of 1984, he came to Virginia Tech to continue his education. He earned a Master's degree in Computer Science in 1986 and completed all the requirements for the Ph. D. degree in Computer Science in May 1990. Upon completion of his Ph. D., he will be working for IBM.