Concepts and Paradigms of Object-Oriented Programming
Expansion of Oct 4 OOPSLA–89 Keynote Talk
Peter Wegner, Brown University

Concepts and Paradigms of Object-Oriented Programming
Peter Wegner, June 1990

Abstract

We address the following questions for object-oriented programming:

What is it?
What are its goals?
What are its origins?
What are its paradigms?
What are its design alternatives?
What are its models of concurrency?
What are its formal computational models?
What comes after object-oriented programming?

Starting from software engineering goals, we examine the origins and paradigms of object-oriented programming, explore its language design alternatives, consider its models of concurrency, and review its mathematical models to make them accessible to nonmathematical readers. Finally, we briefly speculate on what may come after object-oriented programming and conclude that it is a robust component-based modeling paradigm that is both effective and fundamental. This paper expands on the OOPSLA 89 keynote talk.

1. What is it?

We introduce the basic terminology of object-oriented programming and then delve more deeply into its goals, concepts, and paradigms.

1.1. Objects

Objects are collections of operations that share a state. The operations determine the messages (calls) to which the object can respond, while the shared state is hidden from the outside world and is accessible only to the object's operations (see Figure 1). Variables representing the internal state of an object are called *instance variables* and its operations are called *methods*. Its collection of methods determines its *interface* and its *behavior*.

name: object
local instance variables (shared state)
operations or methods (interface of message patterns to which the object may respond)

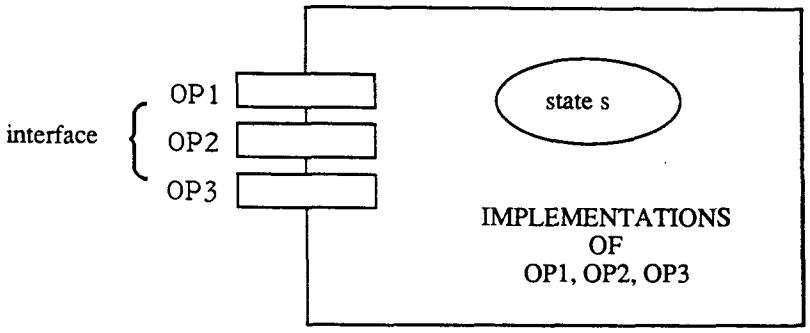

Figure 1: Object Modules

An object named *point* with instance variables *x*, *y*, and methods for reading and changing them may be defined as follows:

```
point: object
  x := 0; y := 0;
  read-x: ↑ x;  – return value of x
  read-y: ↑ y;  – return value of y
  change-x(dx): x := x + dx;
  change-y(dy): y := y + dy;
```

The object "point" protects its instance variables x,y against arbitrary access, allowing access only through messages to read and change operations. The object's behavior is entirely determined by its responses to acceptable messages and is independent of the data representation of its instance variables. Moreover, the object's knowledge of its callers is entirely determined by its messages. Object-oriented message passing facilitates two-way abstraction: senders have an abstract view of receivers and receivers have an abstract view of senders.

An object's interface of operations (methods) can be represented by a record:

(op1, op2, ... ,opN)

Objects whose operations opi have the type Ti have an interface that is a typed record:

(op1:T1, op2:T2, ... ,opN:TN)

Typed record interfaces are called signatures. The point object has the following signature:

point-interface = *(read-x:***Real**, *read-y:***Real**, *change-x:***Real** → **Real**, *change-y:***Real** → **Real***)*

The parameterless operations read-x and read-y both return a *Real* number as their value, while change-x and change-y expect a *Real* number as their argument and return a *Real* result.

The operations of an object share its state, so that state changes by one operation may be seen by subsequently executed operations. Operations access the state by references to the object's instance variables. For example, read-x and change-x share the instance variable x, which is nonlocal to these operations although local to the object.

Nonlocal references in functions and procedures are generally considered harmful, but they are essential for operations within objects, since they are the only mechanism by which an object's operations can access its internal state. Sharing of unprotected data within an object is combined with strong protection (encapsulation) against external access. The strong encapsulation at the object interface is realized at the expense of modularity (and reusability) of component operations. This captures the distinction within any organization or organism between closely integrated internal subsystems and contractually specified interfaces to the outside world.

The sharing of instance variables by methods can be described by the let notation:

```
let x := 0; y := 0; in
  read-x: ↑ x;
  read-y: ↑ y;
  change-x(dx): x := x + dx;
  change-y(dy): y := y + dy;
endlet
```

This let clause specifies an anonymous object with a local environment of instance variables accessible only to locally specified operations. We can think of objects as named let clauses of the form "*object-name: let-clause*". In Lisp notation the object-name can be associated with an object by a *define* command of the form "*(define object-name let-clause)*", or by an assignment command of the form "*(set-q object-name let-clause)*" (see [ASS], chapter 3).

Let notation was originally introduced for functional programming and required local variables such as x, y to be read-only. Our let notation differs fundamentally from functional let notation in allowing assignment to local instance variables.

1.2. Classes

Classes serve as templates from which objects can be created. The class *point* has precisely the same instance variables and operations as the object *point* but their interpretation is different: Whereas the instance variables of a *point* object represent *actual* variables, class instance variables are *potential*, being instantiated only when an object is created.

```
point: class
  local instance variables (private copy for each object of the class)
  operations or methods (shared by all objects of the class)
```

Private copies of a class can be created by a make-instance operation, which creates a copy of the class instance variables that may be acted on by the class operations.

```
p := make-instance point;        -- create a new instance of the class point, call it p
```

Instance variables in class definitions may be initialized as part of object creation:

```
p1 := make-instance point (0, 0);  -- create point initialized to (0,0), call it p1
p2 := make-instance point (1, 1);  -- create point initialized to (1,1), call it p2
```

The two points p1, p2 each have private copies of the class instance variables and share the operations specified in the class definition. When an object receives a message to execute a method, it looks for the method in its class definition.

We may think of a class as specifying a behavior common to all objects of the class. The instance variables specify a structure (data structure) for realizing the behavior. The public operations of a class determine its behavior while the private instance variables determine its structure.

1.3. Inheritance

Inheritance allows us to reuse the behavior of a class in the definition of new classes. Subclasses of a class inherit the operations of their parent class and may add new operations and new instance variables.

Figure 2 describes mammals by an inheritance hierarchy of classes (representing behaviors). The class of mammals has persons and elephants as its subclasses. The class of persons has mammals as its superclass, and students and females as its subclasses. The instances John, Joan, Bill, Mary, Dumbo each have a unique base class. Membership of an instance in more than one base class, such as Joan being both a student and a female, cannot be expressed.

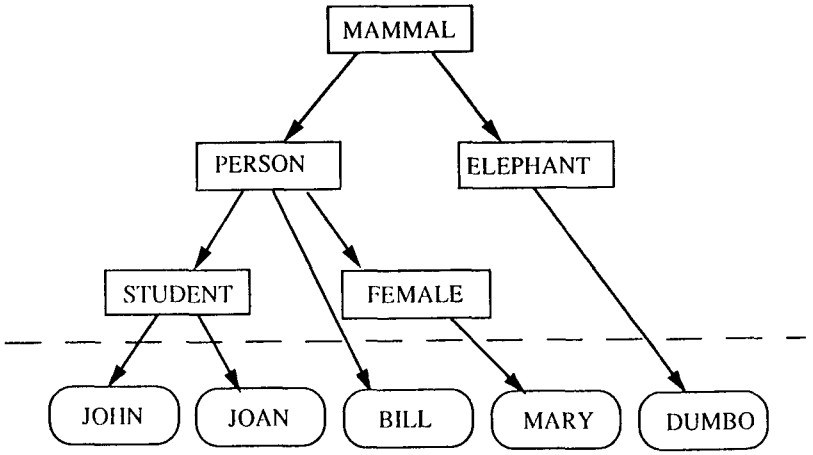

Figure 2: Example of an Inheritance Hierarchy

Why does inheritance play such an important role in object-oriented programming? Partly because it captures a form of abstraction, which we call *super-abstraction*, that complements data abstraction. Inheritance can express relations among behaviors such as classification, specialization, generalization, approximation, and evolution. In Figure 2 we classify mammals into persons and elephants. Elephants specialize the properties of mammals, and conversely mammals generalize the properties of elephants. The properties of mammals approximate those of elephants. Moreover, elephants evolved from early species of mammals.

Inheritance classifies classes in much the way classes classify values. The ability to classify classes provides greater classification power and conceptual modeling power. Classification of classes may be referred to as second-order classification. Inheritance provides second-order sharing, management, and manipulation of behavior that complements first-order management of objects by classes.

1.3.1. An Alternative View of Objects, Classes, and Inheritance

We introduce the Common Lisp Object System [CLOS] to provide the reader with an alternative view of what object-oriented programming is. CLOS has an alternative view of the relation between classes, methods, and objects, defining classes solely by their instance variables:

(defclass classname (superclass list) (list of instance variables))

Methods are separately defined by *defmethod* specifications for already defined classes:

(defmethod method (list of classes) (method definition))

CLOS allows a class to have multiple superclasses (multiple inheritance) and a method to have multiple classes. Because methods can have multiple classes they cannot be defined within just a single class. Instead, all methods having a given method name are grouped together in a single *defgeneric definition*:

(defgeneric methodname (list of method definitions with given methodname))

Instances of classes are created by a *make-instance* command, which creates an object of a specified class. The Lisp set-q command associates a name with a newly created object:

(set-q inst-X (make-instance class-C)) -- create object of *class-C*, call it *inst-X*

When a CLOS object receives a message to execute a method, it consults the generic definition for that method to determine its context and environment of execution. This contrasts with Smalltalk-like languages which simply look for the method in their class definition.

Generic functions are conceptually useful in grouping together similar operations (polymorphic operations) on related classes, such as refresh for labeled, bordered, and colored windows. But it is their mechanism for acting on multiple classes that requires a radical shift in program structure. CLOS searches for methods by their method name as the primary key; the classes with which methods are associated are a secondary key. In contrast, traditional object-oriented programs use the class of an object as the primary key to find methods.

The price of this added flexibility is a loss of security and encapsulation. A late method definition for a class may change the behavior of already created instances. Methods referring to instance variables in multiple classes weaken encapsulation, just as a person working for multiple masters may give away the secrets of one to another.

The requirement that all methods owe their primary allegiance to a single class and operate at execution time on a single object greatly strengthens encapsulation. Weakening this requirement increases flexibility but can create unmanageable (spaghetti-like) object structures. Spaghetti-like structures may be necessary in intelligent organisms like the brain, but are considered harmful in software engineering, just as the goto has been considered harmful. This is the first of many tradeoffs between structure and flexibility considered in this paper. CLOS adopts the view that programmers should be provided with powerful tools and trusted to write well-structured programs, while software engineers place less trust in the programmer by enforcing constraints that may result in a loss of freedom and flexibility.

1.4. Object-Oriented Systems

Objects are related to their clients by a *client/server* relation. The contract between an object and its clients must specify both the object's and the client's responsibilities [WW]. The contract may be specified by *preconditions* that define the responsibilities of clients and *postconditions* that specify the object's responsibilities in responding [Me].

Objects have global software management responsibilities (to support flexible object composition and system evolution) that complement their local behavioral responsibilities to other objects. Object management is realized by classes that allow objects to be treated as first-class values and by inheritance that facilitates the reuse of interface specification through incremental modification and enhancement.

The object-oriented paradigm supports self-description of systems through *metaobject protocols*, and the description of applications by extension (specialization) of inheritance hierarchies for system description [GR]. It is *closed* under self-description. It supports three kinds of abstraction: *data abstraction* for object communication, *super-abstraction* (through inheritance) for behavior enhancement, and *meta-abstraction* as a basis for self-description:

data abstraction → *encapsulation, object communication*
super-abstraction → *object management, behavior enhancement*
meta-abstraction → *self-description*

The universality of objects as a representation, modeling, and abstraction formalism suggests that the object-oriented paradigm is not only useful but also fundamental.

2. What Are Its Goals?

Programs in procedure-oriented languages are action sequences. In contrast, object-oriented programs are collections of interdependent components, each providing a service specified by its interface. Object-oriented program structure directly models interaction among objects of an application domain. The software engineering goals of object-oriented programming include:

Technology of software components
Object-oriented software libraries
Capital-intensive software technology
Object-oriented programming in the very large (megaprogramming)

2.1. Software Components

Splitting a large task into components is a time-honored method of managing complexity, variously referred to as "divide and conquer" and "separation of concerns". Computers have hardware components while data structures have array and record components. Software components arise in decomposing computational problems into subproblems that cooperatively determine the solution to the complete problem. Software components include functions, procedures, and objects as well as concurrently executable components such as processes, actors, and agents.

In the 1960s, functions and procedures were the main kinds of software components. Simula 67 [DMN] introduced objects, classes, and inheritance, as well as object-oriented simulation of real world applications. The idea of data abstraction was introduced in the early 1970s in languages like CLU [LSAS] and incorporated into an interactive environment in Smalltalk 80 [GR]. Concurrent modules such as monitors [Ha,Ho2] and communicating sequential processes [Ho3, Mi] were introduced in the 1970s. Ada [DoD], developed in the late 1970s in response to the software crisis, has functions, procedures, packages, tasks, and generic software components. The variety of software components had, by the late 1970s, become unmanageable.

Object-oriented programming reintroduces systematic techniques for managing software components. *Objects* provide a high-level primitive notion of modularity for directly modeling applications. *Classes* facilitate the management of objects by treating them as *first-class values*. *Inheritance* supports the management of classes by organizing them into hierarchies.

Functions, procedures, and objects differ in their functionality but share certain properties. They are *server modules*, having an *interface* that specifies the services provided to clients and a *body* that implements these services.

2.1.1. Functions, Procedures, and Objects

A function f expects a parameter x as its argument and produces a result $f(x)$, as illustrated in Figure 3. The set of permitted arguments is called its *domain*, while the set of results is called

Figure 3: Function Modules

its *range*. Each x in the domain of f determines a unique value $y = f(x)$ in the range of f. A function f whose domain of arguments has the type T and whose range of results has the type $T1$ has the following interface specification:

function *f (x:T)* **result** *T1;*

Functions calls in expressions return a value that may be used in computing larger expressions. Pure functions have no memory; the result $f(x)$ of calling the function f is independent of previous calls of f and depends only on the current value of x. This simplifies function specification, but limits their ability to model the real world.

Procedures may alter their environment through side effects. They have an interface that specifies the number and type of parameters and the dependence of output on input parameters. They may have side-effects through nonlocal and pointer variables, as shown in Figure 4. Since the effect of a procedure may depend on interactions not controlled by its interface, the specification of procedures is less tidy than for functions.

Objects may have an untidy internal structure but present a tidy interface to their clients. They consist of a collection of procedures glued together with instance variables. They are a higher level of abstraction than functions or procedures, more directly representing entities of the real world. Traditional objects communicate by call/return messages and can be described by their stimulus/response behavior independently of their representation. However, Simula supports communication by coroutines that do not require returning to their caller. Concurrent objects support a variety of communication and synchronization protocols (see section 6).

2.2. Object-Oriented Libraries

Libraries are repositories of software components that serve as reusable building blocks for software development. Programming languages serve as *glue* for combining (composing) library components. Component-based software technology aims to construct programs almost entirely out of predefined components, with minimal use of glue. This generally requires domain-specific components tailored to a specific project or application. The ideal is to find for each application the *joints* that allow components to be designed for assembly into products with minimal glue. However, some applications are inherently more decomposable than others, and there is no guarantee that such joints will necessarily exist.

The paradigm for constructing programs from software components is not manufacturing large numbers of identical components but rather engineering customized products out of prefabricated components. The analogy of a plumber or a constructor of prefabricated houses is more

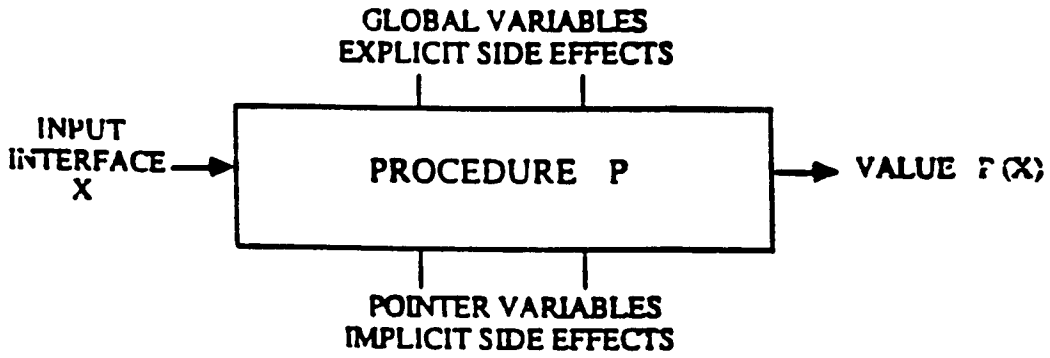

Figure 4: Procedure Modules

appropriate than that of an assembly line. Just as designing a kitchen from a set of off-the-shelf cabinets is simpler than asking a carpenter to build cabinets from scratch, so designing a program from off-the-shelf software components should be simpler than starting from primitive instructions. But designing a *complete* set of software components for an application is harder than designing a complete range of interlocking kitchen cabinets. Sometimes the process of configuring systems from components can be automated, as in automated computer configuration programs like XCON. But for one-shot applications the processes of reusing library components, designing new components, and configuring the system are intermixed.

Libraries in procedure-oriented languages have actions (procedures) as their software components. Procedure-oriented programs usually have liberal doses of glue in the form of statements of an underlying programming language. The seamless composability of procedures with programming language statements reduces the incentive for "pure" composition of procedural components, since the glue blends in so naturally with procedural components. Seamless blending of glue with components is harder for objects and the incentive for pure composition is correspondingly greater.

Components of object-oriented libraries are classes from which objects may be created. The properties of components, the notion of composition, and the nature of glue are very different. There are two levels of composition: declarative composition of classes (for example by inheritance) and execution-time composition of objects. Objects are composed by specifying module interconnections in a *module interconnection formalism* (MIF). Composition of objects cannot be specified by the imperative composition of actions. It requires declarative composition of interfaces, specification of module interconnections, and redrawing the boundary between public and private information.

Procedure-oriented libraries have traditionally been *flat* repositories of independent (unrelated) software components. The work of composing independent procedures into programs is entirely the responsibility of the library user. Object-oriented libraries contain hierarchically organized collections of classes whose patterns of sharing are determined by inheritance. Much of the work of component composition is inherent in the class hierarchy. Class libraries can express relations among behaviors such as specialization, abstraction, approximation, and evolution (see section 5.3). They include metaobjects that specify operations on classes. They support virtual classes (incomplete behaviors) whose undefined attributes must be supplied in a subclass before objects having the behavior can be instantiated. They support reusability during design by incremental behavior specification and during maintenance by incremental behavior modification. The behavior encapsulated in classes can be reused in two ways: by creating instances and by subclasses that modify the behavior of parent classes.

Class names in object-oriented libraries are generally global so that objects of any class can be created in any other class, and can then interact with objects of the creating class. Unrestricted availability of classes increases their reusability and provides nondiscriminatory uniformity of service so that all clients are treated alike. But this assumption is not appropriate for programming in the very large (see section 2.4), where different subsystems may be developed according to different conceptual frameworks and have different type systems and data representations. Homogeneous systems of components that share a common type system are appropriate for the design and engineering of specific products, but component-based software technology must also support conceptually distributed federated systems with heterogeneous components having different conceptual frameworks, languages, and type systems. The problem of communication among heterogeneous components has not been addressed by object-oriented programming.

The idea of libraries of reusable software components dates back to the earliest days of computing in the 1940s and 1950s. Why has this obvious idea proved so difficult to implement? Part of the reason is that libraries are shared resources that serve many masters. Modules reusable in diverse environments are difficult to design. Flexible reusability may conflict with the

requirement of efficiency. Library design is as difficult as the design of an environment or database, since at least the following questions must be answered:

What kinds of components can the library contain?
functions, procedures, objects, classes, generic modules
compilers, operating systems, databases, libraries

What kinds of clients will use the library?
programmers, managers, other programs

How are components created, modified, loaded, linked, and invoked?
what hooks are provided to link modules into their environment?
how varied is the environment of computation?

What relations among modules can be expressed?
hierarchies, inheritance, versions, composition, modification

The economic payoff of libraries increases dramatically as their domain of use (reuse) is restricted. Reuse of single-project libraries over their project life cycle is much greater than that of general-purpose libraries over multiple projects. The personal library of a single individual receives much greater reuse than than public libraries for all programmers. Libraries can be classified by the breadth of their domain of use:

General-purpose libraries: no restriction on domain of use or range of users
Limited-domain libraries: application generators, nuclear reactor codes
Special-purpose libraries: restricted to a single project, application, or user

This classification of software components by their intended use is reflected below in our discussion of different kinds of reusability.

2.3. Reusability and Capital-Intensive Software Technology

Capital goods are resuable resources whose costs are amortized over their uses. Reusability subsumes both physical and conceptual capital formation. Capital goods like the lathe and the assembly line are reusable resources that enhance industrial production. Compilers, operating systems, and software components are reusable capital goods that enhance software productivity. Reusable resources subsume industrial capital goods but also include less tangible software resources and conceptual resources, like education, that enhance the reusability of people. Programming in the very large (see section 2.4) is essentially synonymous with capital-intensive software technology.

Reusability may be justified both economically because of increased productivity and intellectually because it simplifies and unifies our understanding of phenomena. It derives its importance from the desire to avoid duplication and capture commonality among inherently similar situations. The assertion that we should stand on each other's shoulders rather than on each other's feet is a plea for both intellectual and economic reusability.

The initial economic motivation for general-purpose computers was the reusability of computer hardware. Critical computing resources like the central processing unit may be reused a million times per second, while less critical resources like the computer memory may be reused for different programs and data. The changed economic balance between hardware and software has altered our perceptions of what is capital-intensive. Reusability of *semantically neutral* hardware is now taken for granted and reusability of *semantically specific* software components

has become critical in increasing the problem-solving power and productivity of computing.

Technologies that rely heavily on capital (reusable) goods are called capital-intensive technologies. The process of developing capital goods is called capital formation. Capital formation in software technology is dependent on concepts, models, and software infrastructure rather than on physical plant. To understand the processes of software capital formation, we distinguish among different forms of reusability.

2.3.1. Varieties of Software Reusability

Software reusability has many different forms, each with a different economic payoff:

Interapplication reusability
reusability of software components in a variety of applications

Development reusability
reusability of components in successive versions of a given program

Program reusability
reusability of programs in successive executions with different data

Code reusability
reusability of code during a single execution of the program

Systems programs such as compilers and operating systems provide a great deal of economic benefit through inter-application reusability that justifies the large effort (sometimes hundreds of person-years) required to produce them. Interapplication reusability of application packages for linear algebra, mathematical programming, or differential equations is also economically important. But smaller-granularity libraries have been less successful in contributing materially to interapplication reusability. We conclude that interapplication reusability is worthwhile for system and large-granularity application components, but not for small-granularity general-purpose application components.

Components written for a given program are rarely reused in other programs but may be reused hundreds or even thousands of times during development and enhancement of a given program. Reusability of application software may well be the single most important form of reusability, but should be distinguished from interapplication reusability, since its goal is to support an integrated collection of special-purpose software components rather than general-purpose components reusable in other contexts. In developing new versions of application software, it is usually concepts and design strategy rather than the code of components that is reusable.

Special-purpose libraries that contain software components for a specific application domain have become a central tool in project management. They are a primary means of maintaining managerial visibility and control over the status and progress of a project, and are also invaluable for debugging and project maintenance. The use of libraries for tracking progress within a project, isolating errors during debugging, and localizing the effect of program changes reduces the cost of program development much more than interproject libraries.

Program and code reusability are important, but do not benefit from modularity to the same extent as development reusability. Program reusability is enhanced by user-friendly interfaces, while the economic payoff of code reusability is enhanced by efficiency of execution.

Reusability of special-purpose libraries during program development and maintenance is the dominant justification for modularity in language and application design. This conclusion contradicts the folk wisdom that libraries should be as general as possible and support

interapplication reusability. In principle, domain-independent object-oriented systems like Smalltalk support both interapplication reusability for system classes and development reusability of application-specific classes within a single inheritance hierarchy [GR]. But in practice, the efficiency costs of such interapplication reusability are too high, and object-oriented system software has a non-object-oriented system structure to promote efficiency.

2.4. Object-Oriented Programming in the Very Large (Megaprogramming)

The word "large" in "programming in the large" connotes extension in several dimensions (see Figure 5). The most obvious is program size. In its narrowest sense, programming in the large connotes the management of large programs through modularity, interface consistency checking, and other software tools for managing physical size.

A second dimension is that of time. Large programs take a long time (two to ten years) to develop. Once completed, the useful life of successful programs is greater than their development time, say fifteen years. Long-lived programs raise management issues such as version control, management of change, persistent data, and volatile personnel.

A third dimension is that of people with diverse conceptual frameworks and orrganizational structures who must work cooperatively in developing large programs. This raises a further set of management issues such as heterogeneity, compatibility of types and data representations, distributed and federated databases, integration of concepts and code of multiple investigators, cooperative development of documents and programs, etc.

A fourth dimension is that of the educational and technical infrastructure supporting large programs. Educational infrastructure in schools and the workplace is critical to the success of projects for programming in the large. Technical infrastructure includes hardware, interfaces, documentation, and testbeds for simulating field conditions of the system.

Programming in the very large connotes applications that are large in all of the above senses, while traditional programming in the large is concerned primarily with largeness of size. What is the impact of largeness in time, people, and infrastructure on traditional component-based software engineering? The brief answer is that extent in time requires greater attention to persistence and the management of change; diversity of people requires the management of conceptually heterogeneous frameworks and distributed collaborative components; and infrastructure requires high-performance computers, national communication networks, and support of education, training, and the human environment.

Programming in the very large requires the extension of single-user sequential systems to support large, long-lived programs having extension in space, time, and people. Object-oriented programming provides a framework for the seamless management of large hardware/software/application systems. Object-oriented programming in the very large extends traditional object-oriented programming as follows:

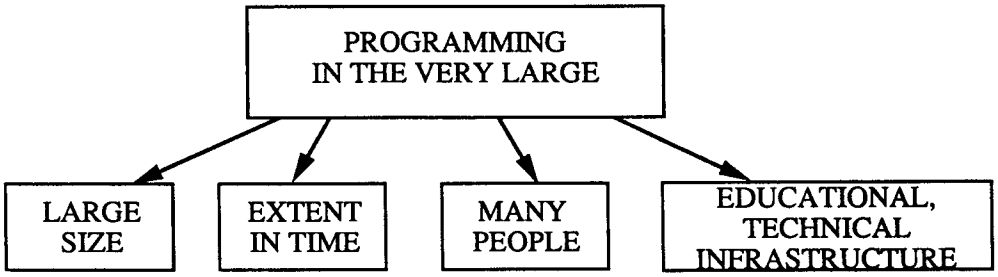

Figure 5: Dimensions of Largeness

Object-oriented programming in the very large
= object-oriented programming + concurrency + persistence
= object-oriented programming + multicomputers + multipeople

Programming in the very large is a primary goal of the Federal High-Performance Computing Program [HPC], a $2 billion five-year plan proposed by the Office of the President's Science Advisor that addresses both hardware and software issues. The National Science Foundation workshop on a *National Collaboratory* organized around a *National Research and Education Net* to support collaborative laboratory environments [Wu] is concerned with managing the new dimensions of largeness made possible by advances in computer technology.

DARPA is developing a software technology plan centered on *megaprogramming* [BS], which is essentially a synonym for programming in the very large. DARPA's vision of megaprogramming (defined as *component-based software engineering for life-cycle management*) focuses on software components that are object-like rather than procedure-like, and is this respect conceptually similar to object-oriented programming. Traditional object-oriented programming may be viewed as a limited form of megaprogramming that makes specific assumptions about the structure of components and the mechanisms of component composition. It is the most developed form of component-based software engineering and can serve as a baseline for extensions to concurrency, persistence, and heterogeneity.

The software components of megaprograms are likely to be large-granularity megamodules with internally homogeneous type systems (conceptual frameworks) and heterogeneous interfaces with incompatible communication format. The distinction between internal and interface structure is analogous to that for traditional objects, but instead of worrying about instance variables and methods we must distinguish between internal glue for type-compatible objects and external glue among heterogeneous megamodules. The internal programming language within megamodules could well be object-oriented, but the megaprogramming language for the coordination and management of megamodules will have requirements that go beyond object orientation [WW1].

3. What Are Its Origins?

Programming languages have evolved from assembly languages in the 1950s, to procedure-oriented languages in the 1960s, structured programming and data abstraction in the 1970s, and object-oriented, distributed, functional, and relational paradigms in the 1980s.

3.1. Programming Language Evolution

Computer science is a young discipline that was born in the 1940s and achieved its status as a discipline in the 1960s with the creation of the first computer science departments. In the 1940s, the designers and builders of computers, such as Aiken at Harvard, Eckert and Mauchley at Penn, and von Neumann at Princeton, had little time for programming languages. It was felt that precious computing resources should be spent on "real" computation rather than on mickey mouse bookkeeping tasks like compiling. von Neumann felt that computer users should be sufficiently ingenious not to let trivial matters such as notation stand in their way. However, assembly languages and library subroutines were developed as early as 1951 [WWG], and by 1958 the success of Fortran, combined with the increased availability and decreasing cost of hardware, had convinced even skeptics of the usefulness of higher-level languages. The evolution of higher-level languages may be summarized as follows:

1954-58: First-generation languages
Fortran I, Algol 58, Flowmatic, IPL V
basic language and implementation concepts

1959-61: Second-generation languages
Fortran II, Algol 60, Cobol 61, Lisp
long-lived, stable, still widely used languages

1962-69: Third-generation languages
PlII, Algol 68, Pascal, Simula
not as successful as second-generation languages

1970-79: The generation gap
CLU, CSP, Ada, Smalltalk
from expressive power to structure and software engineering

1980-89: Programming language paradigms
object-oriented, distributed, functional, relational paradigms
characteristic structure, implementation, patterns of thought

A surprisingly large number of basic programming language concepts were developed for first-generation languages, including arithmetic expressions, statements, arrays, lists, stacks, and subroutines. These concepts achieved a stable embodiment in second-generation languages. Fortran is still the most widely used language for numerical computation, COBOL the language with the largest number of application programmers, and Lisp the most important language for artificial intelligence programming. Algol 60 has not achieved widespread use as an application language but has exerted an enormous influence on the development of subsequent Algol-like languages like Pascal, PL/I, Simula, and Ada.

The attempt by third-generation languages to displace established second-generation languages was unsuccessful, lending credence to Tony Hoare's remark that "Algol 60 is an improvement on most of its successors". PL/I's attempt to combine the features of Fortran, Algol, and COBOL resulted in a powerful but complex language that was difficult to learn and to implement. Algol 68's attempt to generalize Algol 60 was elegant but not widely accepted by users. Pascal's extension of Algol 60 was enthusiastically accepted by the educational community, but insufficiently robust for industrial use.

Simula, the first object-oriented language, was widely used as a simulation language in Europe but not in the United States. In contrast, Smalltalk, which evolved through a sequence of experimental versions in the 1970s into a stable embodiment in Smalltalk 80, has become a popular workstation language in the late 1980s as a result of well-designed system interfaces, good documentation, and aggressive marketing.

The languages of the 1970s were even less widely used than third generation languages. However, this was a period of intensive research and reevaluation of the goals of language design. The software crisis of the late 1960s led to a change of goals in language design, away from expressive power and towards program structure. At the micro level structured while statements replaced unstructured goto statements, and the term *structured programming* became fashionable. At the macro level there was greater emphasis on modularity, first in terms of functions and procedures and later in terms of objects and data abstraction.

The ideas of information hiding and data abstraction, introduced by Parnas [Pa] and Hoare [Ho1], were embodied in experimental languages like CLU [LSAS]. Language mechanisms for concurrent programming were introduced, including monitors [Ha,Ho2] and CSP processes [Ho3]. Ada, born in 1975 as a Department of Defense response to the software crisis, was rich in its module structure, attempting to integrate modularity at the level of functions and procedures with data abstraction through packages and concurrency through tasks.

The 1980s are still too close for historical evaluation. During this period there was intense research on functional, logic, and database languages, and the term *object-oriented* became a popular buzzword, rivaling the popularity of the term *structured programming* in the 1970s. Attention shifted from the study of individual languages to the study of language paradigms associated with classes of languages [We4].

3.2. Programming Language Paradigms

Programming language paradigms determine classes of languages by testable conditions for distinguishing languages belonging to the paradigm from those that do not. There are many abstraction criteria for choosing paradigmatic conditions, including program structure, state structure, and methodology. The design space for programming languages is characterized by six program structure paradigms as a prelude to the more detailed exploration of subparadigms of object-oriented programming.

Paradigms may be specified *extensionally* by languages that belong to the paradigm, *intensionally* by properties that determine whether or not a language belongs to the paradigm, *historically* by the evolution of the paradigm, and sometimes, but not always by an *exemplar*. The paradigms of Figure 6 are described by a one-line extensional, intensional, and historical description, and where applicable by one or more exemplars.

Block structure, procedure-oriented paradigm
Extensional: Algol, Pascal, PL/I, Ada, Modula
Intensional: program is a nested set of blocks and procedures
Historical: primary paradigm in the 1960s and 1970s
Examplar: Algol 60

Object-based, object-oriented paradigm
Ada, Modula, Simula, Smalltalk, C++, Eiffel, Flavors, CLOS
program is a collection of interacting objects
Simula(1967) → Smalltalk(1970s) → Many Languages(1980s)
Exemplars: Simula, Smalltalk

Concurrent, distributed paradigm
CSP, Argus, Actors, Linda, monitors
multiple threads, synchronization, communication
fork-join(1960s) → monitors(1972) → Ada-CSP-CCS(1975-80) → OBCP(1980s)
no dominant exemplar

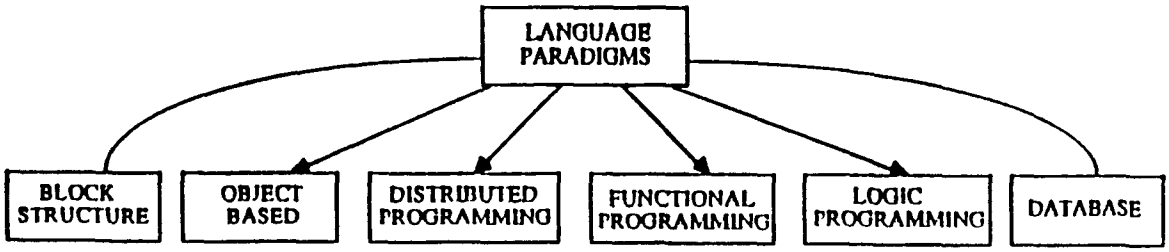

Figure 6: Paradigms of Program Structure

Functional programming paradigm
lambda calculus, Lisp, FP, Miranda, ML, Haskell
no side effects, first-class functions, lazy evaluation
Lisp(1960) → FP-Miranda(1970s) → ML(1980s) → Haskell(1990s)
Exemplars: lambda calculus, Lisp

Logic programming paradigm
Prolog, Concurrent Prolog, GHC, Parlog, Vulcan, Polka
relations-constraints, logical variables, unification
Prolog(1970s) → concurrent-logic-languages(1980s)
Exemplar: Prolog

Database paradigm
SQL, Ingres, Encore, Gemstone, O2
persistent data, management of change, concurrency control
hierarchical → network → relational → object-based
no dominant exemplar

The above paradigms are not mutually exclusive. For example, Ada is both a block-structure and an object-based language. Concurrent Prolog is both a concurrent and a logic programming language. Object-oriented database systems like Encore and O2 combine the object-oriented and database paradigms. Systems that support programming styles of more than one paradigm are referred to as multiparadigm systems.

Multiparadigm systems are in principle desirable but are notoriously difficult to realize. PL/I attempted to combine the expressive power and programming styles of Fortran, Algol, and Cobol. Ada attempted to combine procedure-oriented, object-based, and concurrent programming. In each case the result was a complex (baroque) language that weakened the applicability of constituent paradigms by inappropriately generalizing them. Because harmonious (seamless) integration of multiple paradigms has so often failed at the language level, less ambitious multiparadigm environments have been proposed, with looser coupling among paradigms by carefully controlled interactions that preserve paradigm integrity and facilitate independent program validation [Za]. Object-orientation provides a framework for multiparadigm coupling, with each class specifying a distinct (independently validated) behavior and possibly a distinct execution algorithm through a metaobject protocol (see section 6.4).

Real-world systems (large corporations or multiperson research groups) may be viewed as loosely coupled distributed systems with multiparadigm cooperating subsystems. Programming in the very large should consequently support multiparadigm problem solving through loosely coupled subsystems with heterogeneous components.

4. What Are Its Paradigms?

The object-oriented paradigm may be viewed as:

A paradigm of program structure
in terms of the characteristic program structures supported by the paradigm
A paradigm of state structure
in terms of the characteristic structure of its execution-time state
A paradigm of computation
in terms of its balance between state transition, communication, and classification mechanisms

Program, state, and computation structure are complementary abstractions that respectively capture the view of the language designer, implementor, and execution agent. They determine three different ways of abstracting from specific languages to define language classes with characteristic modes of thinking and problem solving. Robust paradigms, such as the object-oriented paradigm, are interchangeably definable by program, state, or computation structure.

4.1. The State Partitioning Paradigm

In contrast to the shared-memory model of procedure-oriented programming, object-oriented programming partitions the state into encapsulated chunks each associated with an autonomous, potentially concurrent, virtual machine.

In a shared-memory architecture action sequences, including procedures, share a global, unprotected state, as shown in Figure 7. It is the responsibility of procedures to ensure that data is accessed only in an authorized way. Processes must assume responsibility for synchronizing their access to shared data, executing entry and exit protocols to critical regions in which shared data resides by means of synchronization primitives such as semaphores [AS]. Synchronization requires correct protocols in each of the processes accessing the shared data. Incorrect protocols in one process can destroy the integrity of data for all processes.

The object-based paradigm partitions the state into chunks associated with objects, as in Figure 8. Each chunk is responsible for its own protection against access by unauthorized operations. In a concurrent environment, objects protect themselves against asynchronous access, removing the synchronization burden from processes that access the object's data.

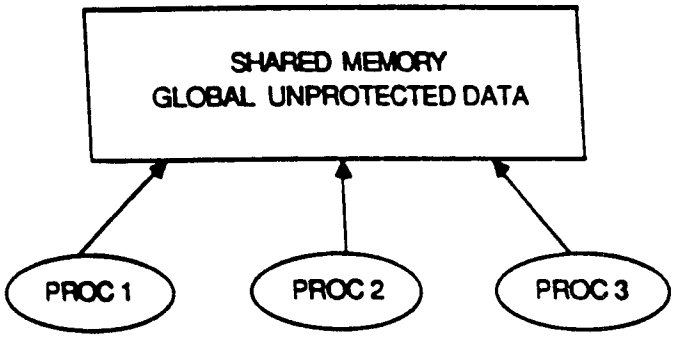

Figure 7: Shared Memory Architectures

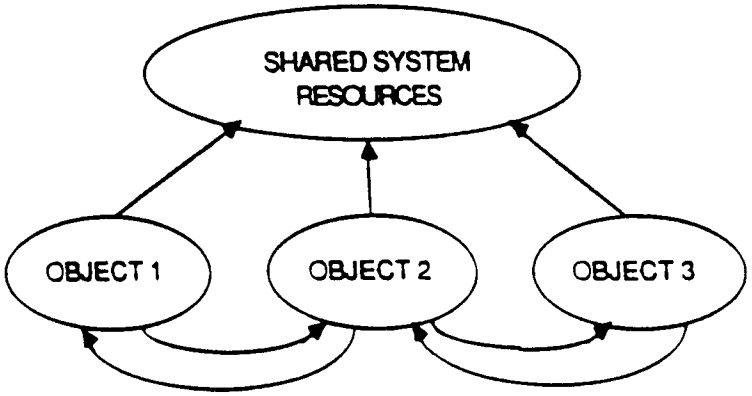

Figure 8: Object-Oriented, Distributed Architecture

Partitioning the state into disjoint, encapsulated chunks is the defining feature of the distributed paradigm. Object-based programs are logically distributed. But object-oriented systems emphasize user interfaces and system modeling, while distributed systems emphasize robust service in the presence of failures, reconfiguration of hardware and software, and autonomy of ownership and control. Object-oriented programming emphasizes object management and application design through mechanisms such as classes and inheritance, while distributed programming emphasizes concurrency, implementation, and efficiency.

In spite of these differences of emphasis, there is a strong affinity between object-oriented and distributed architectures. Because object-based programs are logically distributed, the paradigm can be extended to support physical distribution and the concurrent execution of components. In an ideal world, the distributed and object-oriented paradigm would be merged, combining the advantages of robustness and efficiency with those of user friendliness and reusability. But such multiparadigm systems may prove to be excessively complex because of their attempt to serve too many masters, just like PL/I and Ada.

Individual chunks of the partitioned state, corresponding to individual objects, have the shared-memory structure of Figure 7. An object is a collection of procedures sharing an unprotected state. The state-partitioning paradigm builds on the shared-memory paradigm as the structure of individual software components and introduces a second level of structure with an entirely different set of interaction rules for communicating among chunks of the partitioned state.

4.2. State Transition, Communication, and Classification Paradigms

Language paradigms may be classified by the degree to which they utilize state transition, communication, and classification (statements, modules, types) for computation (see Figure 9).

We may think of state transition, communication, and classification as three *dimensions* of a language design space, and of languages as occupying points of this three-dimensional space. Languages lying near an axis or a plane formed by two axes are low-level, while high-level languages have a balanced combination of these computational features.

The state-transition paradigm views a computation as a sequence of state transitions realized by the execution of a sequence of instructions. Turing machines are state-transition

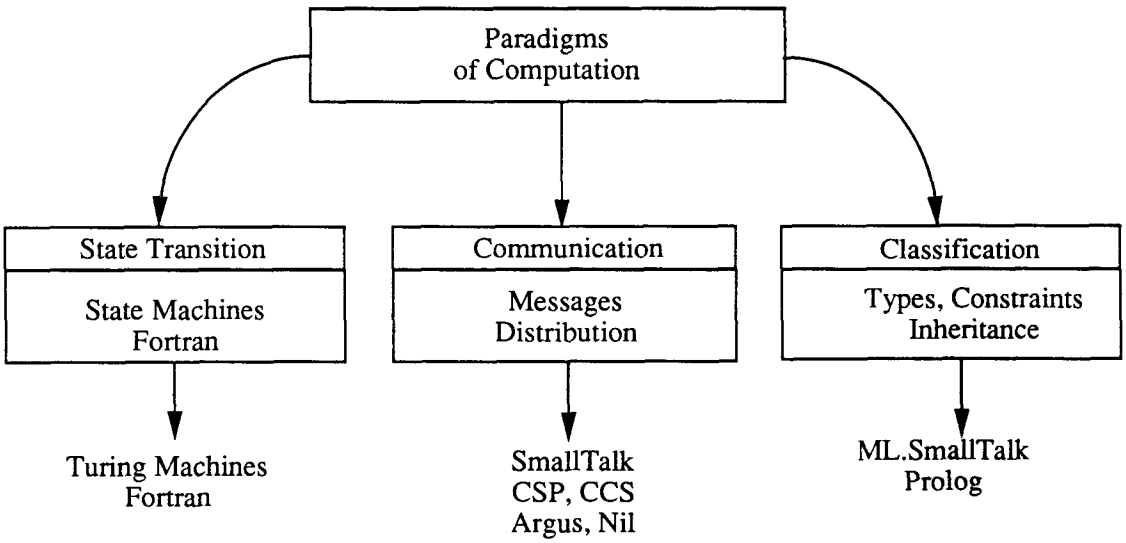

Figure 9: State-Transition, Communication, and Classification Paradigms

mechanisms in which the instructions are determined by the input symbol and current state. Stored-program computers store instructions as part of their state and move instructions to the central processing unit before executing them. Assembly languages and imperative higher-level languages embody the state transition paradigm.

The communication paradigm views communication among agents as the primary computation mechanism. The communication primitives send and receive parallel the state-transition primitives store and fetch or write and read. The communication channel, sometimes embodied by a buffer, plays the role of storage, but generally supports nondestructive send/write operations in contrast to the destructive assignment operation of the state-transition paradigm. There is a duality between communication and state-transition paradigms with channels playing the role of storage. However, computation is enriched by combining communication and state-transition paradigms with agents having both an internal state and ports with message queues.

Actors [Ag], the Calculus of Communicating Systems [Mi], Self [US], and A'UM [Yo] approximate pure communication. Although computation may at a sufficiently primitive level be viewed as pure communication (variables may be viewed as communication channels), agents generally have both internal and communication behavior. Data abstraction supports the communication paradigm by its insistence that objects be defined solely by their communication interface (independently of their internal state).

But the behavior of agents is often more naturally expressed by their state-transition behavior than by their communication (interface) behavior: knowing the inner workings of a black box often helps us to understand its behavior. Sharing among agents is most simply modeled by a state-transition model (variables with values). Actors are based on the communication paradigm, but have named (shared) mailboxes with a state. Stream-based languages such as A'UM compute by merging and otherwise operating on streams, but streams effectively have a state (communication channels are effectively variables).

The classification paradigm views a computation as a sequence of classification steps each of which serves to constrain the result. A complete computation terminates when the sequence of classification steps yields singleton elements. For example, Quicksort classifies the elements of the vector with respect to a pivot element, and repeats this process on subvectors until all are singleton elements. The game of Twenty Questions starts with a domain that is animal, vegetable, or mineral and poses a sequence of classificatory questions designed to reduce the domain to a singleton element.

Whereas Quicksort and Twenty Questions perform complete computation by classification, types classify values as a prelude to computation by other means. Types classify values by the operations applicable to them to establish a context for computation and a basis for checking that only applicable operations are applied to the type at execution time. Computation of typechecked programs may then be carried out by state transition and communication.

Object-oriented programming supports both "first-order" classification of objects into classes and "second-order" classification of classes by their superclasses. Classification plays an essentially greater role in object-oriented languages than in languages that do not support inheritance because second-order classification of classes supplements first-order classification of objects. Inheritance hierarchies provide a higher-order classification mechanism that greatly increases the expressive power of object-oriented languages.

Pure paradigms of computation are generally low-level. For example, Turing machines compute by a pure state-transition paradigm, Actors by an almost pure communication paradigm, and mathematical set theory or the predicate calculus by a pure classification paradigm. High-level languages generally use a combination of paradigms. However, different high-level languages combine state transition, communication, and classification features in different ways. Object-oriented languages combine the three paradigms in a unique way by employing state

transitions for computation within objects, messages for communication between objects, and two-level classification for the management of both objects and classes.

The success of the object-oriented paradigm is due to its seamless integration of state transition, communication, and classification. Statements, modules, and types play complementary and supportive roles in the computational process, and reinforce each other in contributing to the overall design, implementation, and maintenance of application systems.

4.3. Subparadigms of Object-Based Programming

Paradigms may be refined into subparadigms that determine entirely new modes of thinking and problem solving. Procedure and object-based paradigms are subparadigms of the generic modular programming paradigm whose modes of thinking are so different that the communities of language designers and users hardly overlap. Subparadigms of the object-based paradigm associated with Ada, Smalltalk, and Actors likewise determine nonoverlapping research communities that rarely talk to each other. Small linguistic differences yield large paradigm shifts.

By imposing restrictions on paradigms we can define subparadigms for language subclasses. We are particularly interested in robust conditions that are easily checkable at the level of language structure and also determine a programming methodology and style of problem solving, such as the following subparadigms and associated language classes (see Figure 10):

object-based languages: *the class of all languages that support objects*
class-based languages: *the subclass that requires all objects to belong to a class*
object-oriented languages: *the subclass that requires classes to support inheritance*

Object-based, class-based, and object-oriented languages are progressively smaller language classes with progressively more structured language requirements and more disciplined programming methodology. Object-based languages support the functionality of objects but not their management. Class-based languages support object management but not the management of classes. Object-oriented languages support object functionality, object management by classes, and class management by inheritance. The problem-solving style of these three language classes is sufficiently different to warrant a separate identity as distinct paradigms.

The object-based languages include Ada, CLU, Simula, and Smalltalk. They exclude languages like Fortran and Pascal which do not support objects as a language primitive.

CLU, Simula, and Smalltalk are also class-based languages since they require their objects to belong to classes. But Ada is not class-based because its objects (packages) do not have a type

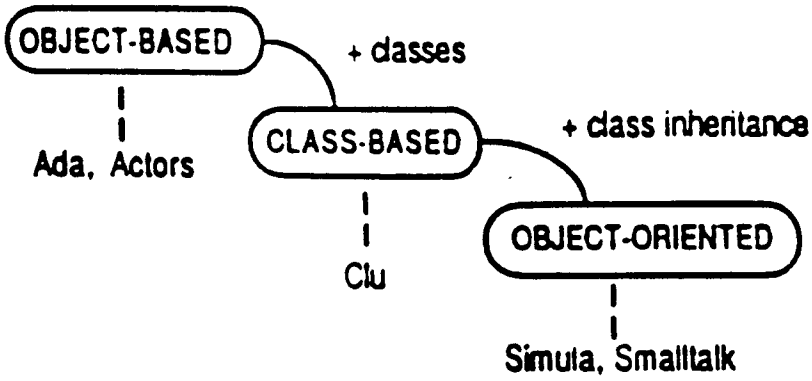

Figure 10: Object-Based, Class-Based, and Object-Oriented Languages

and cannot therefore be passed as parameters, be components of arrays or records, or be directly pointed to by pointers. These language perks are automatically available for any typed entity, but are not available in Ada for untyped entities like packages. (Note that private data types specified in a package interface have first-class values, but this is subtly different from treating the package itself as a first-class value. This small difference in package structure has far-reaching consequences for package management.)

Simula and Smalltalk are object-oriented according to our definition, since their classes support inheritance. CLU is class-based but not object-oriented, since its objects must belong to classes (clusters), and clusters do not support inheritance.

The use of the term object-oriented to denote a narrow class of languages including Simula and Smalltalk but excluding Ada and Self [Un] has sparked debate about the proper use of the term object-oriented. Our narrow definition more precisely captures object-orientedness in Simula and Smalltalk than a broader definition. Being precise helps to counter the quip that "everyone is talking about object-oriented programming but no one knows what it is". The looser view of object-oriented programming as "any form of programming that exploits encapsulation" [Ni] lends credence to the above criticism.

Our taxonomy has practical relevance because it discriminates among *real* languages like Ada, Simula, and Smalltalk on the basis of language design criteria that affect programming. Classifying Ada as object-based but not class-based or object-oriented implies that it supports the functionality of objects but not their management and determines characteristic differences in program structuring and system evolution between Ada and Smalltalk.

Ada has a rich module structure, supporting functions,. procedures, packages, tasks, and generic modules. Its packages provide the functionality of objects but, since packages do not have a type, Ada's facilities for object management are deficient. Ada was developed at a time when the design of languages with data abstraction and concurrency was not yet understood, and it does not integrate its many notions of modularity into a seamless whole. Its notion of type does not uniformly handle its rich and almost baroque module structure. Procedures and packages do not have a type and cannot be passed as parameters or appear as components of structures, while tasks (concurrent modules) are typed. The fact that sequential modules are untyped while concurrent modules are typed is somewhat anomalous. The lack of support of object management reflects deeper problems with module structure due to an inadequate language design technology. Ada was intended to be a conservative extension of well-understood technology, but the incorporation of data abstraction and concurrency into the procedure-oriented paradigm turned out to be a radical and controversial leap into uncharted territory.

There are two kinds of problems with adopting Ada as a standard: its technical limitations and its exclusion of other legitimate language cultures. We refer to these as problems of *soundness* and *completeness*. Problems of soundness will require its users to live with greater complexity and cost of system and application programs, but can be mitigated with good system tools. Problems of completeness are potentially more serious because they limit the impact of Ada technology on the wider community and cut off the Ada community from other language cultures, promoting a fortress mentality with psychological barriers against multiparadigm programming or other potentially effective software engineering practices.

5. What Are Its Design Alternatives?

The object-based design space has the following dimensions (see Figure 11):

objects
classes and types
inheritance and delegation
data abstraction
strong typing
concurrency
persistence

Any given language determines a coordinate in each design dimension and a point in the design space. We focus on design alternatives for objects, classes, and inheritance in the present section and on design alternatives for concurrency in the next section.

5.1. Objects

Functional, imperative, and active objects are respectively like values, variables, and processes (see Figure 12):

Functional objects: *have an object-like interface but not an updatable state.*
Imperative objects: *have an updatable state shared by operations in the interface.*
Active objects: *may already be active when receiving a message, so that incoming messages must synchronize with ongoing activities of the object.*

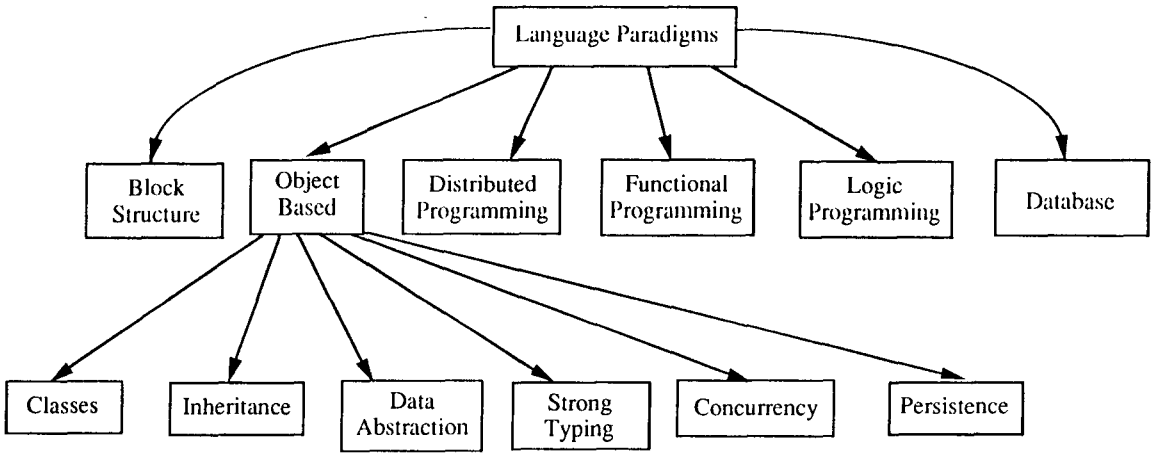

Figure 11: Dimensions for Object-Based Languages

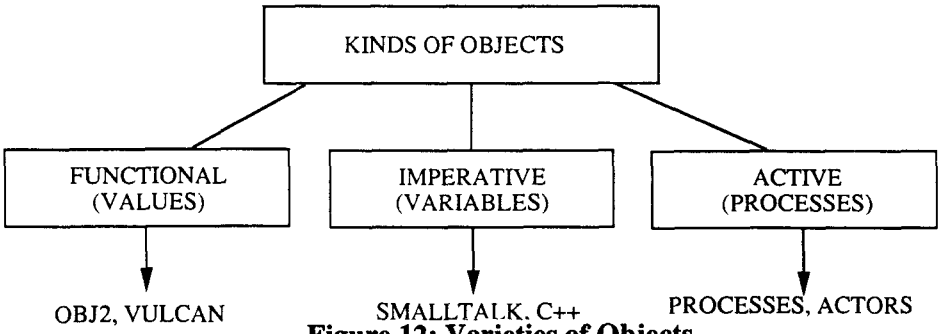

Figure 12: Varieties of Objects

Functional objects arise in logic and functional programming languages, traditional object-oriented languages have imperative objects, and active objects arise in object-based concurrent programming languages.

5.1.1. Functional Objects

Functional objects have an object-like interface, but no identity that persists between changes of state. Their operations are invoked by function calls in expressions whose evaluation is side-effect-free, as in functional languages. Programmers in Smalltalk, C++, or Eiffel might claim that functional objects lack the essential properties of objecthood. But excluding them would exclude the large body of worthwhile work on functional object-based languages.

Degenerate objects consisting of a collection of functions without a state are functional objects. Bool is a functional object with operations *and, or, not, implies*:

Bool: functional-object
operations: and, or, not, implies
equations:
A and false = false
A or true = true
not true = false
not false = true
A implies B = (not A) or B

Bool responds to messages by matching incoming expressions against the left-hand sides of each equation. If a match is found, the corresponding right-hand side is returned to the caller with appropriate substitutions. The process of expression evaluation is illustrated below:

(true implies (false or true)) $\rightarrow$
(true implies true) $\rightarrow$
((not true) or true) $\rightarrow$
(false or true) $\rightarrow$ *true*

First the message "false or true" is sent to Bool, resulting in the substitution of "true" for "false or true" in the expression being evaluated. Bool then successively matches (true implies true), (not true), and (false or true) to yield the final value "true".

5.1.2. Imperative Objects

Imperative objects are the traditional objects of Simula, Smalltalk, and C++. They have a name (identity), a collection of methods activated by the receipt of messages from other objects, and instance variables shared by methods of the object but inaccessible to other objects:

name: object
shared instance variables inaccessible to other objects
methods invoked by messages which may modify shared instance variables
end

Imperative objects have an identity distinct from their value that facilitates sharing: for example sharing by several clients of the services of a server. Two kinds of assignment are needed: $x := y$ for assigning a copy of y to x, and $x :- y$ for causing x and y to share the same object.

The object named *counter* may be specified in Smalltalk by two parameterless methods, *incr* and *total*, and an instance variable count initialized to 0:

```
counter: object
 instance variables:
  count := 0;
 methods:
  incr        count := count + 1;
  total       ↑count
```

The methods *incr* and *total* share the instance variable *count*. The reliance of operations on shared variables sacrifices their reusability in other objects in order to facilitate efficient coordination with operations of the given object. Sharing of variables is a paradigmatic specialization mechanism that sacrifices generality to achieve local efficiency. Conversely, replacing shared variables by communication channels is a paradigmatic mechanism for generalization. Objects clearly differentiate between internal specialization and external generality.

The Smalltalk instance variable *count* is an untyped variable that may contain values (object references) of varying type. Execution of *count+1* causes the message *+1* to be sent to the object whose value is stored in *count*. When the value of *count* is an integer then *+1* is interpreted as integer addition. But *count* could in principle refer to an object of the type string and + might then be interpreted as string concatenation. Because the type of instance variables is dynamically determined, Smalltalk is not a strongly typed language.

Mainstream object-oriented languages like Smalltalk have imperative objects and may be referred to as *imperative object-oriented languages*. Use of the term *object* or *object-oriented language* without qualification generally implies that objects are imperative.

5.1.3. Active Objects

Imperative objects are passive unless activated by a message. In contrast, active objects may be executing when a message arrives. Messages for active objects may have to wait in an entry queue, just as a student must wait for a busy (active) professor. Executing operations may be suspended by a subtask (procuring the student's transcript) or interrupted by a priority message (the professor receives a telephone call). Active objects have the modes: *dormant* (there is nothing to do), *active* (executing), or *waiting* (for resources or the completion of subtasks). Message passing among active objects may be asynchronous.

5.1.3.1. Actors

An actor (see Figure 13) has a *mail address*, an associated mailbox that holds a queue of incoming messages, and a *behavior* that may, at each computational step, read the next mailbox message and perform the following actions [Ag]:

create *new actors*
send *communications (messages) to other actors (its acquaintances)*
become *a replacement behavior for execution of the next message in the mailbox*

At each step an actor may *create* new actors, *send* messages to its acquaintances, and *become* a replacement behavior that processes the next message. Pipelined concurrency within an actor can occur by concurrent execution of its replacement behavior.

The nature of *become* is a key to understanding the actor model. Each computational step causes an actor's behavior to be transformed into another behavior (its replacement behavior).

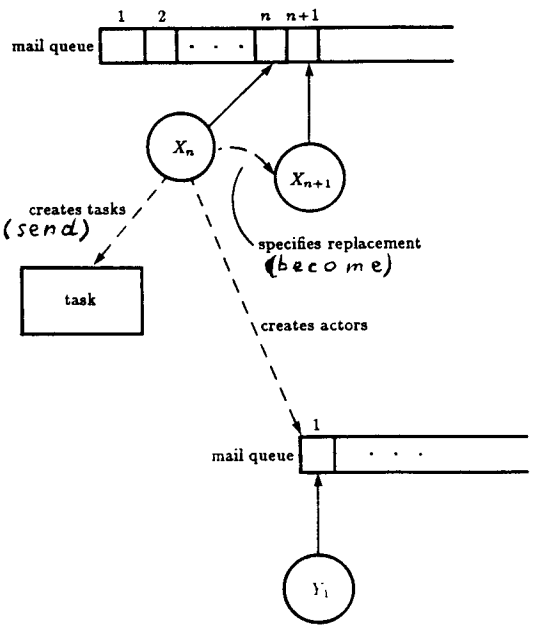

Figure 13: Computation in the Actor Model

Becoming occurs also in communicating processes [Ho, Mi] (see section 6.1.2) where "P $\rightarrow$ aQ" means "P becomes Q after executing a". An actor may *become* an entirely different behavior, but in practice often becomes behavior with the same operations but a different state. Imperative objects that *become* the same object with a different state are simply a special case. *Becoming* can support class-based languages whose objects do not change their class and volatile objects that do not have a class because their behavior can change too radically.

Newly created actors have an initially empty mailbox and a behavior specified by a *script*. The creating actor is responsible for informing the created actor of mail addresses of acquaintances and informing other actors that need to be acquainted with the newly created actor. The primitive actor computation step may dynamically reconfigure the network at each step by creating new actors and creating a replacement behavior whose local interconnection to other actors may be different from the behavior it replaces.

Higher-level languages separate creating (new objects), sending (messages), and becoming (replacement behavior) into independent computation steps. ABCL1 [Yo1] introduces three message-passing modes: *past* (asynchronous), *now* (synchronous), and *future* (see section 6.4) message passing. It replaces the primitive (create, send, become) actor computation step by object-based interpretation of messages to perform operations specified in an object's script.

Actors have an imperative mailbox mechanism for sharing the identity of actors among acquaintances and a declarative replacement behavior mechanism for executing messages within an actor. A communication paradigm motivated by sharing is combined with a functional (declarative) paradigm for execution, reflecting that communication is inherently imperative while evaluation is inherently declarative.

Concurrency is generally specified directly in terms of higher-level primitives rather than by a detour through actor models. However, actors provide a useful low-level framework for concurrent language design even if not directly used in specifying high-level languages.

5.1.4. Object Identity

Persistent objects have an identity distinct from their value, capturing a significant property of objects of the real world exemplified by the following dialog:

Smith, how you've changed. You used to be tall and now you are short. You used to be thin and now you are fat. You used to have blue eyes and now you have brown eyes.

But my name isn't Smith.

Oh, so you've changed your name too.

Can Smith be unambiguously identified by a set of attributes? Can objects be uniquely identified by their behavior? Because the answer to these questions is no, we need unique object identifiers that distinguish an object from all other objects and allow all references to the object to be recognized as equivalent.

An object's identity is logically distinct from its value (identical twins have different identities). It is distinct from the name given an object by the programmer, since an object may have many aliases. It is distinct from the address or location at which an object resides, since its address is an external, accidental attribute while its identity is an internal, essential attribute. (Note that treating someone like an object means ignoring their inner identity.)

Persistent objects that may (like Smith) find themselves in unfamiliar environments require unique identifiers. Objects that persist beyond the program in which they are created cannot rely on their creating environment for identification. Unique identifiers should be supported at the system level to allow environment-independent identification, but may be hidden from users.

Support of object identity requires operations that allow identity to be manipulated. A basic operation is testing for object identity. Another operation is coalescing the identity of two objects when we discover they are the same. This may happen in physics when we discover that the morning star is the evening star, in literature when Dr Jekyll and Mr Hyde are discovered to have the same identity, and in the game of CLUE when the murderer turns out to be the butler.

Testing for object identity may be viewed as a special case of testing for object equivalence. Object equivalence may be defined in many different ways. Having the same type or class is a form of object equivalence. Objects may be equivalent by virtue of having a specific common property (persons having the same age). A somewhat more subtle equivalence relation among objects is that of observational equivalence (having the same behavior in all possible contexts of observation). Several different criteria for observational equivalence have been proposed, but a precise criterion of what observations are legitimate has not been agreed upon. Observational equivalence is weaker than identity, identity implies equivalence but not conversely.

The need to coalesce identity (of the morning and evening star) illustrates that unique identifiers determined at object creation time do not completely capture the intuitive notion of identity, since identity can change dynamically during program execution. A second example of dynamically determined identity arises in the case of superclasses that may assume the identity of their subclasses. In this case creation-time identity is premature, and dynamic identity is determined by supplying a meaning for self-reference rather than by an externally determined unique identifier (see section 7.3). A third example is "The President of the United States" who assumes the identity of different persons at different times. The office and the incumbent may for many purposes be identified, but sometimes need to be distinguished.

Unique identifiers are just one kind of object identity (determined by object creation) rather than a self-evident characterization of the essence of object identity. Identifying objects or persons by the unique circumstances of their birth is often useful, but may be inappropriate when dynamic changes of identity can occur.

5.2. Types

Types are useful in classifying, organizing, abstracting, conceptualizing, and transforming collections of values. We present a variety of answers to the question "what is a type?", examine the distinction between types as descriptions of object structure and classes as descriptions of object behavior, and review the interaction of classes and inheritance.

Object-oriented systems facilitate treating types as first-class values since types may, like objects, be represented by finite collections of attributes and may therefore be treated as objects [Coi]. However, the view that types introduce unnecessary computational overhead and conceptual complexity must also be taken seriously. CLOS [Mo] is based on the design principle that enforcing the semantic contract of an interface should be the responsibility of the programmer rather than the language. We explore typeless delegation-based languages in section 5.3.3.2.

Types were introduced in Fortran and Algol as early as the 1950s and were extended from typed numbers to typed records, procedures, classes, abstract data types, and type inference:

Fortran: implicit integer and floating point number types
Algol 60: declarations for integer, real, Boolean variables
PL/I, Pascal: extension to arrays, records, pointers
Algol 68: extension to procedures, systematic overloading and coercion
Simula: extension to classes and inheritance
CLU: extension to abstract data types
ML: extension to inferring types of all subexpressions, polymorphic types

5.2.1. What is a Type?

The programming language answer to the question "what is a type?" depends on which of the roles played by types is of greatest interest to the questioner. The primary dichotomy is between the compiler writer's view of types as properties of expressions and the application programmer's view of types as kinds of behavior. Deeper analysis yields the following views:

application programmer's view
types are a mechanism for classifying values by their properties and behavior

system evolution (object-oriented) view
types are a software engineering tool for managing software development and evolution

system programming (security) view
types are a suit of clothes (suit of armor) to protect raw data against unintended interpretation

type checking view
types are syntactic constraints on expressions to ensure operator/operand compatibility

type inference view
a type system is a set of rules for associating with every subexpression a unique most general type that reflects the set of all meaningful contexts in which the subexpression may occur

verification view
types determine behavioral invariants that instances of the type are required to satisfy

implementer's view
types specify storage mappings for values

Each definition of type has a constituency for which it is the primary view. The system evolution view is primary for object-oriented programming, since it facilitates the incremental modification of types through inheritance.

The term ''type'' is heavily overloaded. Fortunately the term "class" is available so "type" can be used for a subset of its current meanings. We use the term type to capture the compiler writer's need to describe the structure of expressions and the term class to capture the application programmer's need to describe the behavior of values. The distinction between types and classes corresponds essentially to that between structure and behavior.

5.2.2. Types and Classes

The primary purposes of typing are the following:

specifying the structure of expressions for type checking
specifying the behavior of classes for program development, enhancement, and execution

Types are specified by predicates over expressions used for type checking, while classes are specified by templates used for generating and managing objects with uniform properties and behavior (see Figure 14). Types determine a type checking interface for compilers, while classes determine a system evolution interface for application programmers. Every class is a type, defined by a predicate that specifies its template. However, not every type is a class, since predicates do not necessarily determine object templates.

These differences in purpose and specification cause subtypes and subclasses to be derived in different ways from their parents. Subtypes are defined by additional predicates that constrain the structure of expressions. Subclasses are defined by *template modification programs* that may arbitrarily modify parental behavior, and consequently are more loosely related to their parent class. Since every class is a type, subtyping constraints may uniformly be applied to classes to constrain their structure. Subtyping heavily constrains behavior modification, while subclassing is an unconstrained mechanism that facilitates flexible system evolution.

In early programming languages a single notion of type served the needs of both type checking and behavior management. In choosing between subtyping mechanisms for strong type checking and subclassing mechanisms for the the flexible management of class behavior, many object-oriented languages have abandoned types in favor of classes.

Classes represent behavior. Inheritance facilitates composition of incomplete behavior (virtual classes) during program development and enhancement of complete behavior during system

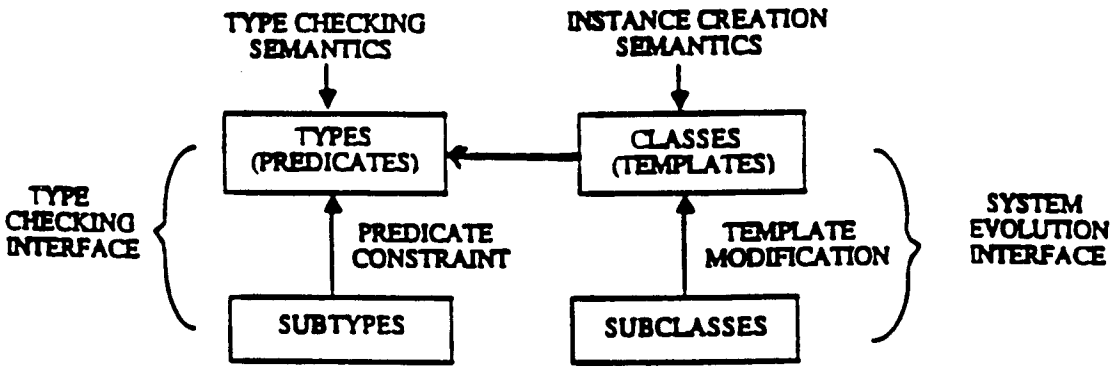

Figure 14: Types and Classes

evolution. Object-oriented database languages keep track of the set of all instances (extent) of a class and allow operations on the extent. In some languages classes may be objects and receive messages to perform actions on local class variables. Such operations on classes make the notion of class in object-oriented languages very different from that of a simple type.

Subtype inheritance in languages like Trellis/Owl [Sch] is very different from subclass inheritance in languages like Smalltalk [GR]. Subclasses may in principle define behavior completely unrelated to the parent class, but good programming practice requires subclasses to reuse the behavior of superclasses in a substantive way.

5.2.3. Interaction of Classes and Inheritance

In object-oriented languages classes have two kinds of clients; objects that access its operations and subclasses that inherit the class [Sn]. Figure 15 shows a class A with an object and an interface to a subclass B that in turn has an object and a subclass interface.

What should be the relation between object and subclass interfaces of a class? Should subclasses have greater rights than objects in accessing instance variables of the state (as in Smalltalk) or should object and subclass interfaces be equally abstract? Let's call the object and subclass interfaces its *data abstraction* and *super-abstraction* interfaces. Should the rules governing data abstraction and super-abstraction be similar, or should super-abstraction permit weaker information hiding than data abstraction.

There are good conceptual arguments for direct superclass data sharing. Letting inheritance dissolve interface boundaries supports the intuition that inheritance creates a stronger bond than mere accessibility. Viewing inherited instance variables as part of an enlarged self also suggests direct accessibility. By not insisting on information hiding, the concept of super-abstraction becomes *orthogonal* to data abstraction, with very different concerns and computational mechanisms. Data abstraction is concerned with encapsulation of data structures while super-abstraction is concerned with composition and incremental modification of already abstract behavior.

It is not unreasonable for different clients to have different interfaces to classes. Supporting different views for different clients is a desired feature of databases. There are good reasons for permitting subclasses to have a different view of parent data abstractions from that of objects

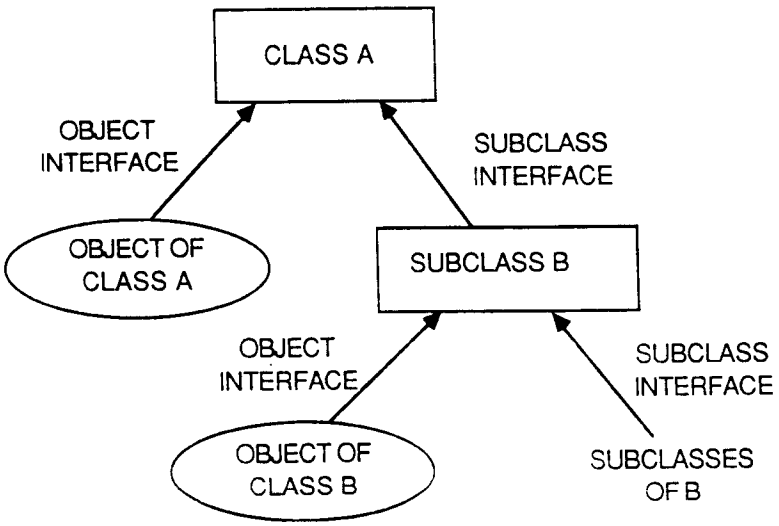

Figure 15: Object and Subclass Interfaces

directly using the abstraction. However, having a single abstract interface for all clients is also attractive, and languages such as Trellis/Owl [Sch] do precisely this.

5.3. Inheritance

Inheritance is a mechanism for sharing code and behavior. Tree structure is a general mechanism for sharing of the properties of ancestors by descendants. Just as block structure facilitates the sharing of data declared in ancestor blocks by descendant blocks, inheritance hierarchies facilitate "second-order" sharing of code and behavior of superclasses by subclasses. Multiple inheritance facilitates sharing by a descendant of the behavior of several ancestors.

5.3.1. Implementation of Inheritance

Consider a class A with instance a and a subclass B with instance b as in Figure 16. Both A and B define behavior by operations shared by their instances, and have instance variables that cause a private copy to be created for each instance of the class or subclass. The instance a of A has a copy of A's instance variables and a pointer to its base class. The instance b of B has a copy of the instance variables of both B and its superclass A and a pointer to the base class of B. The class representation of B has a pointer to its superclass A, while A has no superclass pointer since it is assumed to have no superclass.

When b receives a message to execute a method it looks first in the methods of B. If found the method os executed using the instance variables of b as data. Otherwise it follows the pointer to its superclass. If it finds the method in A it executes it on the data of b. Otherwise it searches A's superclass if there is one. If A has no superclass and the method has not been found it reports failure. This search algorithm may be defined by the following procedure:

procedure *search (name, class)*
  if *(name = localname)* **then do** *localaction*
  else if *(inherited-module = nil)* **then** *undefinedname*
  else *search (name, inherited-module)*

In order to capture the essence of inheritance (see section 7.3), the method found as a result of this search must be executed in the environment of the base class. Self-reference should be interpreted as reference to the base class rather than to the class in which the method is declared. This models the dynamic binding mechanism of Smalltalk, where the identity of inherited classes is bound to the identity of the object on whose behalf the method is being executed.

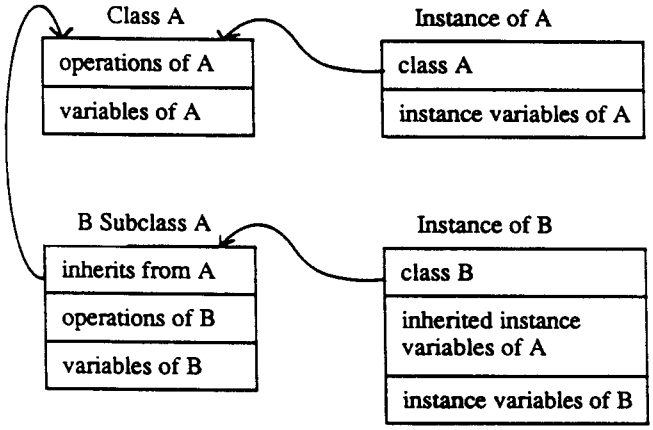

Figure 16: Implementation of Inheritance

The restriction that instances have precisely one base class may be understood in terms of considerations of implementation. Instances must know where to start looking for methods when they receive a message to be executed. Having pointers to more than one base class would make method lookup complex and possibly ambiguous. Allowing Joan to be both a student and a female in Figure 15 would considerably complicate object-oriented semantics.

5.3.2. Design Alternatives for Inheritance

We explore the following design dimensions for inheritance:

modifiability: *How should modification of inherited attributes be constrained?*
granularity: *Should inheritance be at the level of classes or instances?*
multiplicity: *How should multiple inheritance be defined and managed?*
quality: *What should be inherited? behavior, code, or both?*

A given inheritance mechanism determines a point in the design space with specific modification, granularity, multiplicity, and quality properties.

5.3.2.1. Inheritance as an Incremental Modification Mechanism

Incremental modification is a fundamental process not only in software systems but also in physical and mathematical systems. The goal of realizing small system changes by small specification changes arises in many disciplines. Incremental modification of software systems by inheritance spans two distinct notions: refinement and similarity (likeness).

"B refines A" means that B preserves and augments the properties of A, while "B like A" is the weaker relation that B and A have common properties. Refinement models behaviorally compatible extension while likeness models system evolution. Likeness is symmetric while refinement is an antisymmetric partial ordering relation and has a direction. Since both behavioral refinement and system evolution are central to managing software complexity, both should be supported in object-oriented systems [WZ].

Inheritance combines a parent P and a modifier M into a result $R = compose(P, M)$ as in Figure 17, where P, M, and R are specified by finite sets of attributes:

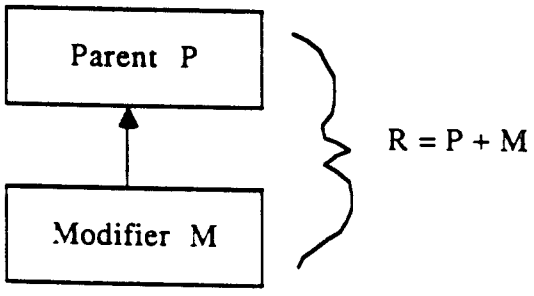

Figure 17: Incremental Modification by Inheritance

$P = (P1, P2, ..., Pp)$
$M = (M1, M2, ..., Mm)$
$R = (R1, R2, ..., Rr)$

When the attributes of M are disjoint from those of P, R has p+m attributes consisting of the union of those in P and M. For overlapping attributes those of M redefine those of P, much as identifiers declared in an inner textually nested module redefine those of an outer module. We examine four progressively more powerful mechanisms for modifying attributes of P by similarly named attributes of M (see Figure 18):

behavior compatibility: *behavior of subclass is "compatible" with superclass*
signature compatibility: *signature of subclass is syntactically compatible with superclass*
name compatibility: *superclass operation names are preserved (possibly redefined) in subclass*
cancellation: *unrestricted modification of superclass attributes by subclass*

Behavior-compatible modification requires instances of a subclass to "behave" like its parent class for all operations and arguments of the parent class. Behavior of classes may be modeled by algebras [GM], but there is no agreement on the precise notion of behavioral compatibility that should be used. For example, should operations which yield the same value when both are defined, such as addition over integers and reals, be viewed as behaviorally compatible. In [BW] we examine three notions of behavioral compatibility for subtypes:

subset subtype: *Int(1..10) is a subset subtype of Int*
isomorphically embedded subtype: *Int is an isomorphically embedded subtype of Real*
object-oriented subtype: *person is an object-oriented subtype of mammal*

Signature compatibility is defined in terms of compile-time-checkable restrictions on signatures. Types are clearly signature compatible if they have the same signature. Compatibility of signatures is preserved by horizontal extension (adding new typed components to the signature) and by vertical extension that restricts the type of existing components to a subtype.

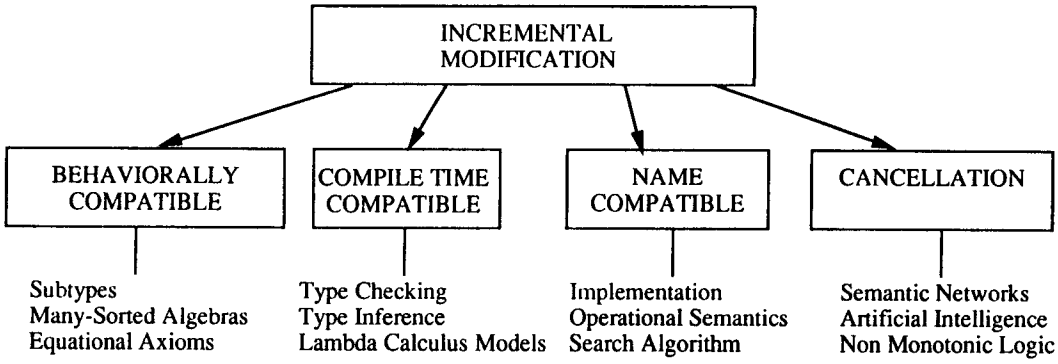

Figure 18: Varieties of Incremental Modification

If T = <op1: T1>
then <op1: T1, op2: T2> is a horizontal extension of T
and <op1: T11> is a vertical extension of T
if T11 is a subtype of T
and <op1: T11, op2: T2> is both a horizontal and a vertical extension of T
if T11 is a subtype of T

The view of types as signatures is natural for compile-time type checking. Signatures are adopted by the ML community as a basis for type inference systems, and by those in the type theory community who focus on type checking as the primary role of types in programming languages. Signature compatible modification does not distinguish among different behaviors with the same syntactic interface.

Name-compatible modification views types as templates specified by their implementation. The semantics of name-compatible inheritance is specified by the search algorithm of section 5.3.2. that looks for named operations first in the base subclass and then in successive ancestor classes. The search algorithm guarantees that names in a superclass can never be cancelled by a subclass. Implementations of inheritance in languages like Smalltalk are based on the search algorithm. Name compatibility has an operational semantics motivated by implementation.

Search algorithms usually ignore type compatibility between operations of the subclass and similarly-named operations of the superclass. Redefinition of an operation (method) is totally unconstrained; a redefined operation can have a different number of arguments and an entirely different effect from the correspondingly named operation in the superclass.

Cancellation allows attribute names of the parent to be cancelled, thereby creating classes with fewer attributes than the parent. Inheritance by cancellation arises in artificial intelligence for classes with exceptions such as the class of birds who have the attribute of flying in spite of the fact that some birds such as ostriches do not fly. The subclass of "non-flying birds" can be specified by cancelling the "fly" attribute.

Cancellation is the most flexible of the four forms of incremental modification. Remarkably, each is associated with entirely different notions of type; types as algebras, signatures, layered symbol tables, and semantic networks. These models are studied by entirely different research communities whose members rarely talk to each other.

5.3.2.2. Granularity of Inheritance

Inheritance is a behavior-sharing mechanism that complements value sharing by execution-time invocation of methods. Class-based inheritance requires all objects of a class to have the same behavior-sharing patterns. This precludes some students from inheriting the behavior of musicians while others inherit the behavior of basket weavers and still others inherit the behavior of workaholics. Uniform inheritance of several kinds of behavior by all instances of a class can be accomplished by multiple inheritance, but differential inheritance of behavior requires a finer granularity of sharing at the level of individual objects.

Behavior sharing at the level of objects is referred to as *delegation*. Objects may delegate responsibility for performing nonlocal operations to parent instances called *prototypes* that serve both as instances and as templates for behavior sharing and cloning other instances. The collection of all objects that share (delegate to) the same prototype is in some respects like a class, but there is no language-level distinction between objects and classes. The prototype is itself an object that may delegate to one or more parents, and delegation links are fair game for excution-time modification of sharing patterns.

Prototypes can contain both value and operation slots so that sharing patterns supported by delegation are very flexible. Delegation avoids the constraints on changeability and adaptability

of object behavior that results from binding objects irrevocably to a class. Delegation was studied by Lieberman [Lie] and incorporated as a design principle for the language Self [US].

The choice between modeling by prototypes or by classes and instances has its counterpart in mathematics. Mathematical set theory distinguishes between sets and instances, and between set membership (the class/instance relation) and subsets (the class/subclass relation). Programming languages that take their cue from classical mathematics distinguish subclasses from instances and use inheritance primarily for the sharing of behavior. Prototyping languages do not distinguish between behavior and values at the language level and may use inheritance for sharing both behavior and values.

Languages based on prototypes eliminate the need for classes, thereby reducing the number of primitive language concepts. But they increase semantic complexity because delegation must handle a greater variety of patterns of sharing, and instances have to serve the function of both classes and values. Using *Occam's razor* to shave the number of primitive concepts results in greater overloading of the concepts that remain. Greater flexibility of behavior sharing may be advantageous in some situations, but too much flexibility may also be harmful. The flexibility of control of the goto statement has been considered harmful and has led to the introduction of structured but redundant control primitives like *while* and *repeat*.

Classes separate concerns of structure and behavior from those of execution-time computation. During design we are interested primarily in specifying the behavior of an application domain without any specific computation. Declarative considerations of design may be separated from imperative considerations of computation, while providing executable prototypes for testing the computational consequences of behavior specifications.

Prototypes are useful for representing types having just a single instance, since there is no need to represent the type as distinct from its instance. But when there are many actual or potential instances of a type it is useful to distinguish between a type and its instances, both conceptually and at the level of implementation. The transition from prototypes to classes models the process of knowledge acquisition. When we encounter the first instance of a class, say in childhood, we may think of it as a prototype. When the second instance is encountered we may define it in terms of its differences from the first instance. But as we encounter many instances we develop an abstract notion of the class by abstracting the common properties of instances. Thus prototype systems represent a primitive substrate for initially organizing a domain of discourse, but are replaced by typed systems as robust abstractions in the domain are identified. By the time we are ready to develop application programs for a domain, we usually understand it sufficiently to model it in terms of classes.

Figure 19 illustrates the potential saving of delegation over inheritance for turtles which use pens to indicate their progress. Figure 19a has a pen class with coordinate attributes (x,y) and a turtle subclass with a direction attribute that inherits coordinate attributes from the pen class. There are two instances of pens with coordinates (50,100) and (100,100), and one instance of a turtle at position (50,100) with direction 90 (degrees). Figure 19b represents this information more economically by two pen prototypes and a turtle prototype. The first pen prototype has position (50,100) and the second pen prototype specifies just the x coordinate 100, since the y coordinate is obtained by delegation to the first prototype. The turtle prototype specifies just its heading, obtaining the values of both its x and y coordinates by delegation to the pen prototype.

In this example, delegation greatly reduces the stored information. But moving the pen or the turtle to a new position (x,y) requires testing the values of x,y against those of the first pen prototype to determine whether one or both values need to be explicitly recorded. The decrease of stored information is realized at the cost of greater complexity in the moving process.

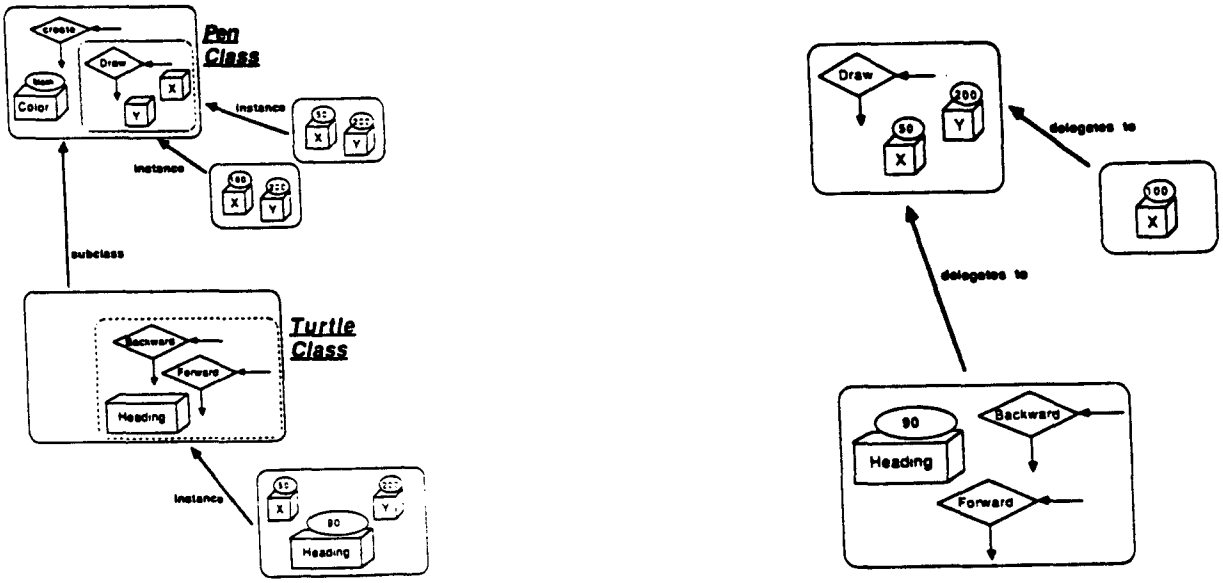

Figure 19: Classes Versus Prototypes

5.3.2.3. Single Versus Multiple Inheritance

Should inheritance be single or multiple? The real world abounds in situations with multiple inheritance. Natural inheritance is from two parents rather than one. Whenever we design a new artifact that builds on more than one previous artifact, such as designing PL/I from Fortran, Algol, and Cobol, or designing an object-based concurrent language by combining object-based and concurrent ideas, we effectively use multiple inheritance (see Figure 20).

Multiple inheritance is conceptually difficult since there are so many ways in which the inherited entities may be combined in designing a new entity. Features of inherited entities may compete in designing the new entity, just as language features from Fortran, Algol, and Cobol competed in choosing the design features of PL/I. New designs that compromise among competing inherited ideas often result in an unaesthetic and perhaps unusable hybrid. These problems of the real world spill over into computer-based multiple inheritance. It is not easy to create uniform mechanisms that usefully combine the methods of multiple inherited classes. Multiple

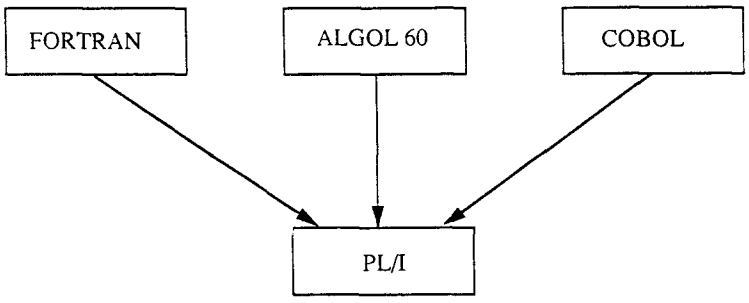

Figure 20: Multiple Inheritance of Concepts

inheritance of a class T from superclasses T1, T2, ... , TN may be specified as follows:

class T inherits(T1, T2, ... , TN) in body-of-T

Each of the types T,T1,T2,...,TN has a finite set of operations (methods). The methods invokable by objects of the class T are specified by combination of methods inherited from T1,T2,...,TN with the methods directly defined by T. In order to simplify method combination rules, some multiple inheritance systems, such as that of CLOS [Mo], flatten inheritance hierarchies, for example by ordering inherited classes T1,T2,...,TN in a left-to-right linear order. Rules of method combination for linearly ordered multiple inheritance hierarchies include:

call the first method in a specified linear ordering
call all methods in order (of the linear ordering)
execute first method that returns a non nil value
collect values of all methods into a list
compute the arithmetic sum of values
call all before demons, then the first method, then all after demons
use an argument to select one or a subset of methods

The collection of superclasses of a given class is generally not tree-structured but has joins, as illustrated in the pie hierarchy of Figure 21. The rule in CLOS for linearizing such hierarchies is to order inherited superclasses in a depth-first, left-to-right, up-to-joins order, resulting in the following linearization for the pie hierarchy:

pie apple fruit cinnamon spice food

That is, we take the leftmost predecessor of pie, proceed depth-first up to the join at the class food, then list the next branch, and list the join itself when all successors are listed. The multiple inheritance structure is thereby reduced to a single linear inheritance structure, and the problem of multiple inheritance is reduced to one of single linear inheritance.

However, as pointed out by Snyder [Sn], there are problems in linearizing multiple inheritance hierarchies because a class may get separated from its immediate parents by intervening classes that disrupt communication. In the pie example, the class pie has become separated from its parent class cinnamon by intervening classes apple and fruit which can disrupt communication by redefining methods of cinnamon. Inheritance between a class and its immediate parent may

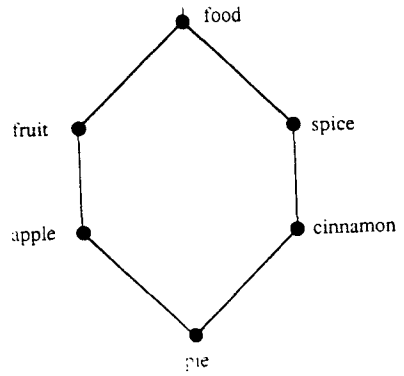

Figure 21: A Multiple Inheritance Hierarchy

thus depend on classes in an entirely different part of the inheritance hierarchy. Adding a new class may have unforeseen effects on distant parts of the hierarchy.

These difficulties may be avoided by direct modeling of the graph structure. But this leads to problems of interpretation for joins because methods of a join are automatically inherited along each of the multiple paths from the join to the lower-level class. The class pie inherits the methods of the class food along each of the two paths from food to pie.

This problem may be avoided by recognizing multiple inheritance emanating from a single node as a special case and treating it as single inheritance. But this requires global knowledge of the hierarchy and causes problems if the methods at a join are replaced by copies for each of the branches of the join, violating the principle that the behavior of the hierarchy should depend only on what is inherited and not on the structure of the hierarchy.

Instances of classes with multiple inheritance have instance variables for each of the inherited superclasses. The special treatment of joins above is consistent with having just a single set of instance variables of the join class for each object representation. A more literal interpretation of multiple inheritance would require a distinct copy of instance variables of the join class for each path in the hierarchy. The tradeoffs among alternative data structures and semantic interpretations of multiple inheritance is further discussed in [Sn], but there is no obvious solution that avoids dependence of local semantics on nonlocal structure.

5.3.2.4. What Should Be Inherited? Behavior or Code?

What is it that should be inherited? Should it be behavior or code: specification or implementation? At a recent (ECOOP 88) international panel on inheritance, the European participants felt that inheritance hierarchies should be behavior hierarchies while Americans felt that inheritance hierarchies were in practice code hierarchies. This reflects a difference between the European tradition of aesthetic programming and the American tradition of practical programming.

Behavior and code hierarchies are rarely compatible with each other and are often negatively correlated because shared behavior and shared code have different requirements. Consider a stack and a double-ended queue (deque). A deque may be viewed as a stack with additional operations and is therefore a subclass of a stack when behavior is the basis for inheritance. However, from the viewpoint of implementation, a deque needs two list pointers in its data structure while a stack needs only one. It is simpler to implement a stack in terms of a deque than a deque in terms of a stack. Thus it is easier to have the code for a stack inherit from the code for a deque than the other way round.

More generally, adding operations tends to complicate the data structure required to support the operations, and it is easier for a simple data structure to inherit from a more complex one than the other way around. Adding operations specified by a class M to a parent class P is modeled at the level of behavior by inheritance of the behavior of P by M. But the data structure of M supports a broader set of behaviors than that of P and it is practical to let the code of M be the superclass and P the subclass for code inheritance.

The differences between logical (behavior) and physical (code) hierarchies for Smalltalk 80, pointed out in [LTP], is illustrated in Figure 22. It demonstrates how considerations of code take precedence over those of behavior in the Smalltalk 80 inheritance hierarchy [GR].

5.4. Strongly Typed Object-Oriented Languages

Programming languages in which the type of every expression can be determined at compile time are said to be *statically typed languages*. The requirement that all variables and expressions be bound to a type at compile time is sometimes too restrictive and may be replaced by the weaker requirement that expressions are guaranteed to be type-consistent, if necessary by some run-time type checking. Languages in which all expressions are type-consistent are called

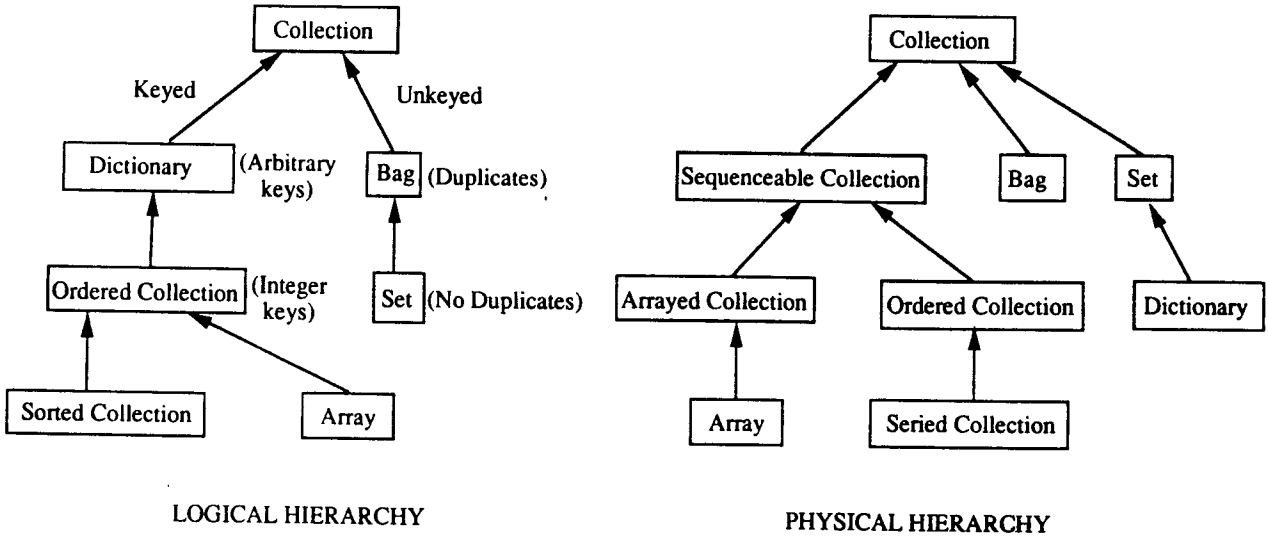

Figure 22: Logical Versus Physical Hierarchy for Smalltalk 80

strongly typed languages [CW].

The class of strongly typed object-oriented languages is the subclass of object-oriented languages in which all objects have abstract interfaces and the language is strongly typed (see Figure 23). This class is strictly smaller than the class of object-oriented languages since it excludes Simula 67 because its object interfaces are not abstract and Smalltalk because it is not strongly typed. The non-abstract classes of Simula 67 are probably inadvertent, since later versions of Simula do require classes to support abstraction. But Smalltalk deliberately supports dynamic binding at message execution time in order to enhance flexibility.

Should object-oriented languages require abstraction interfaces and strong typing? The answer to this question depends on the relative importance of structure versus flexibility. CLOS emphasizes flexibility over structure and requires neither abstraction or strong typing. However, it facilitates programming styles that support abstraction and strong typing at the discretion of system designers and programmers. Smalltalk compromises on this issue, supporting class abstraction but not strong typing. Languages like C++ are strongly typed.

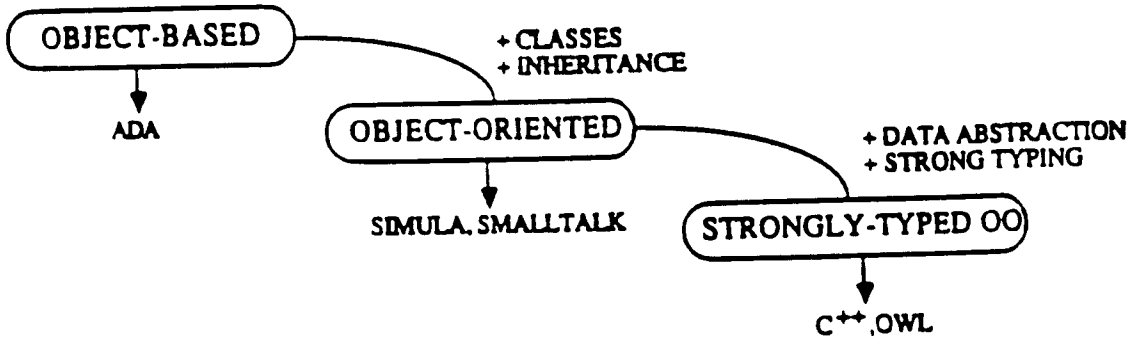

Figure 23: Strongly Typed Object-Oriented Languages

Flexible languages leave freedom and power in the hands of programmers, while structured languages do not trust the programmer with freedom and enforce stringent rules for the alleged benefit of society. Smalltalk represents an intermediate point on the scale between flexibility and structure, CLOS represents an extreme of flexibility and freedom, and strongly-typed object-oriented languages are extreme in their choice of structure over flexibility. Combining freedom with structured safeguards against its abuse is difficult in programming as it is in human societies. But it should in principle be possible, for example by supporting both *safe* and *rapid prototyping* modes in the same programming environment.

5.5. Interesting Language Classes

The seven language features at the beginning of section 4 define 128 different language classes (one for each subset of features). Some classes are more interesting than others because they contain implemented languages or support interesting methodologies. Figure 24 identifies some additional interesting language classes.

On the lower right of Figure 24 the concurrent, persistent, strongly typed object-oriented languages supports all language features. Are such languages possible, and are they desirable? What price is paid by the user in terms of efficiency and language complexity for the extra functionality of concurrency and persistence? Are the resulting objects inevitably *heavy* objects with large overheads for creation, deletion, and operation execution, or can these features be realized without paying a price in execution-time overhead when features are not used?

The left-hand side of Figure 24 shows the classless languages whose objects do not support classes. It may be argued that classes are unnecessary because they are auxiliary abstractions that capture uniformities of objects but have no counterpart in the real world. Moreover, it does not always make sense to associate an operation with a specific class, either because it may have objects of several different classes as arguments or because the operation may be applicable to several classes (friends in C++).

Classless languages may be further classified into those without delegation (inheritance) and those with delegation. Languages without classes or delegation but with concurrency include the *actor-like* languages, which sacrifice structure associated with classes and inheritance in order

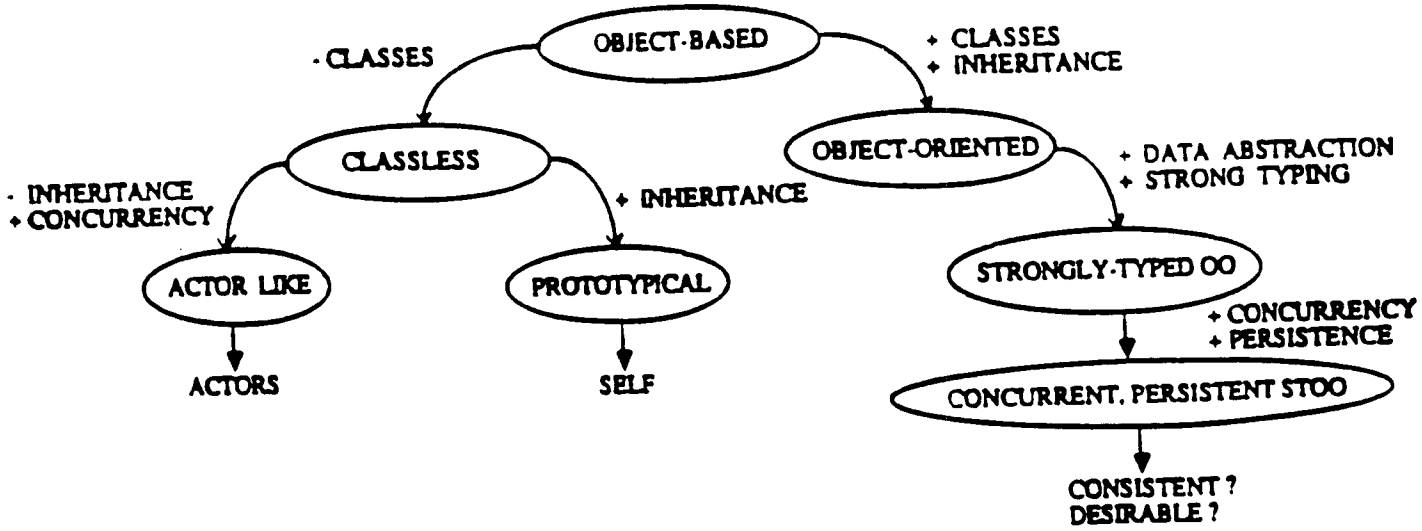

Figure 24: Interesting Language Classes

to gain flexibility for concurrency. Languages without classes but with inheritance are called prototypical languages because their objects serve both as instances and as templates from which new instances can be cloned. Prototypical languages were first studied by Lieberman [Lie], and are the basis of the experimental language *self* [Un].

5.6. Object-Oriented Concept Hierarchies

Figure 25 summarizes the relation among areas of object-oriented research discussed above. At the highest level, the study of programming language paradigms allows different approaches to language design and problem solving to be compared. Subdividing the subdiscipline of programming languages into half a dozen significant subareas is useful and provides a framework for the authors graduate-level programming language course. At the second level we list the "dimensions" of our object-based design space. Any one of the design features could be further expanded, and we choose inheritance because of its special role in object-oriented programming. The third level lists design alternatives for inheritance, each of which has been the subject of much research. The fourth level identifies four progressively more radical incremental modification mechanisms that can be realized by inheritance. The richness of research in object-oriented programming is brought home by the fact that even at this fourth, specialized level there are four very different paradigms for incremental modification associated with different conceptual models and different research communities.

Figure 25 organizes research topics in a hierarchical, object-based manner. It illustrates the power of object-oriented classification by applying it to the subject matter of this paper. It demonstrates the relation of object-oriented concepts to each other and to related research areas such as functional and logic programming.

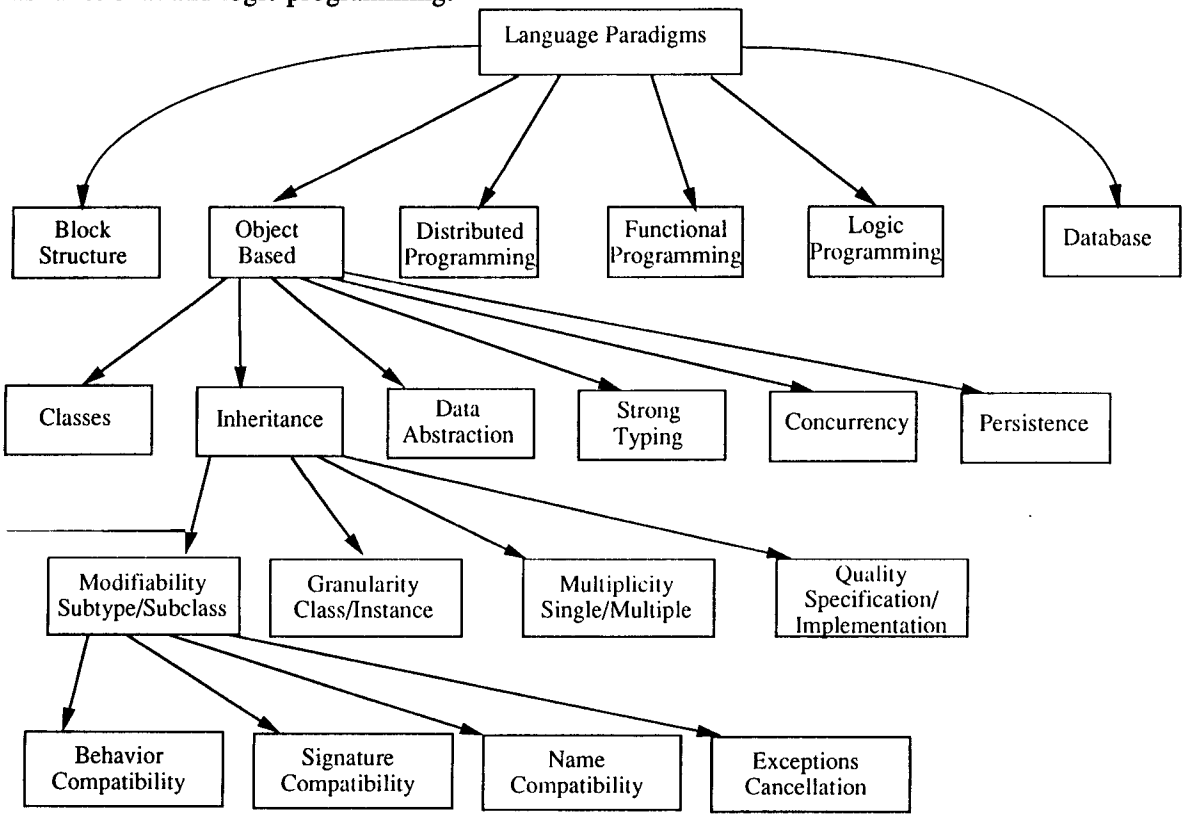

Figure 25: Object-Oriented Concept Map

The concept map of Figure 25 has more structure than meets the eye. The first level is generally thought of as an "or tree" in that paradigms are viewed as alternative programming and implementation styles. The second level is more accurately modeled as an "and tree" in that language features can coexist and we are interested in their interaction. The third level is likewise an *and* tree since we are interested in the interaction of design features of inheritance. The fourth level is again an *or* tree since the different mechanisms for incremental modification are generally viewed as alternatives. Using | to represent *or* alternatives and * to represent *and* alternatives, Figure 25 can be specified in the following manner:

paradigm → *block-structure* | *object-based* | *distributed* | *functional* | *logic* | *database*
object-based → *class* * *inheritance* * *abstraction* * *strong-typing* * *concurrency* * *persistence*
inheritance → *modifiability* * *granularity* * *multiplicity* * *quality*
modifiability → *behavior* | *compile-time* | *name* | *cancellation*

Or alternatives are related to their parents by an is-a relation: the block-structure paradigm *is-a* paradigm. The relation between and alternatives and their parents may be viewed as a has-part relation: the object-based paradigm has parts concerned with class, inheritance etc. Concept maps may be viewed as mixed is-a has-part hierarchies.

Levels are *or* subtrees if leaves are to be studied in isolation and *and* subtrees if leaves can interact. By reinterpreting *or* trees as *and* trees we can focus on interactions among concepts previously studied in isolation. For example, reinterpreting the first level of Figure 25 as an *and* tree leads to the study of interactions among paradigms and the exploration of multiparadigm languages and environments.

5.6.1. Transformations of Concept Hierarchies

Figure 25 is presented above as a passive description of previously described relations among concepts. Concept maps may also be used more actively to introduce new concepts, postulate new relations among existing concepts, and refine or restructure a set of concepts. For example, our six program structure paradigms fall naturally into two broader categories: imperative and declarative paradigms (see Figure 26).

Imperative paradigms view a program as a sequence of state-transition commands that progressively transform an initial into a final state. They are exemplified by Turing machines and von Neumann computers and by languages like Fortran that closely model the architecture of imperative computers. The block-structure, object-based, and distributed paradigms are imperative in that computation consists of assignment of a sequence of values to a state.

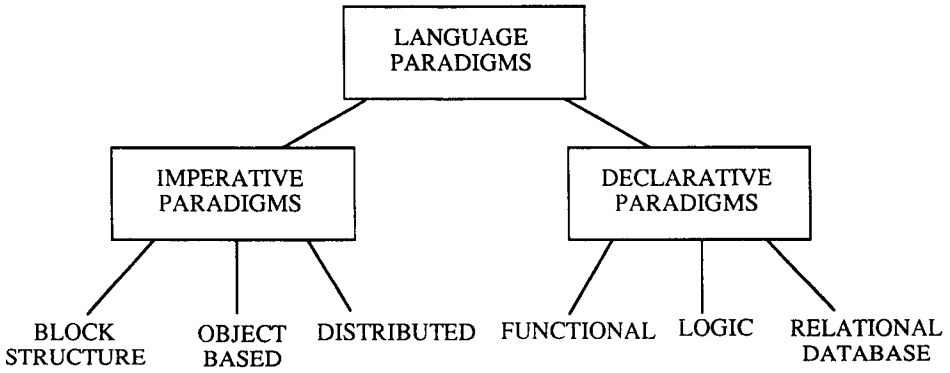

Figure 26: Imperative and Declarative Paradigms

Declarative paradigms specify (declare, describe) what is to be computed independently of how it is computed. Functional and logic programming languages respectively specify what is to be computed by functions and logical relations. The database paradigm is inherently declarative since it focuses on the description rather than transformation of data.

What is the relation between our *coarse* classification into imperative and declarative languages and the more refined classification into six language structure paradigms? Is such paradigmology useful or are we merely playing word games? We claim that Figure 26 provides insights not derivable from Figure 25. The declarative/imperative distinction corresponds to that between mathematical and inherently computational models and is associated with distinct approaches to machine architecture and programming methodology, while the six program structure paradigms yield a more detailed classification of languages into categories associated with distinct research traditions and programming styles. Although the declarative/imperative distinction is not strictly necessary for understanding the program structure paradigms, this does not make it redundant since it is concerned with a different kind of abstract distinction; namely that between descriptive and computational characterization of phenomena.

Imperative and declarative paradigms can be introduced into Figure 25 by a syntactic transformation that interposes an extra level of structure between the root and its first-level descendants.

paradigm → *imperative-paradigm* | *declarative-paradigm*
imperative-paradigm → *block-structure* | *object-based* | *distributed*
declarative-paradigm → *functional* | *logic* | *relational-database*

This kind of transformation of concept maps is useful in modeling the conceptual evolution of disciplines. We have informally explored the idea of a calculus of concepts for mixed is-a has-part hierarchies. Its transformations include expanding a leaf into a subtree (specialization) interposing a new level of structure between a node and its descendants (classification), or adding additional descendants to an internal node. Its inheritance rules include is-a inheritance for or subtrees and attribute grammar inheritance for and subtrees. Such a calculus has interesting possibilities as an object-oriented design technique, but is beyond the scope of this paper.

6. What Are Its Models of Concurrency?

Object-based concurrent programming combines the object-based and concurrent paradigms. It combines the object-based notions of encapsulation, classes, and inheritance with the concurrent concepts of threads, synchronization, and communication (see Figure 27). It combines modeling power through object-orientation with computational power through concurrency [Yo1].

How should the sequential object-based paradigm be changed to accommodate concurrency? Are concurrent objects (processes) still recognizably objects? What kinds of interfaces should processes have, and what should be their internal structure? Should concurrent objects have classes and inheritance? How are design methodology and methods of object management for concurrent objects related to those for sequential objects?

Objects mesh nicely with concurrency since their logical autonomy makes them a natural unit for concurrent execution. However, concurrent sharing is more complex than sequential sharing, requiring mutual exclusion and temporal atomicity. The interfaces, internal structure, and communication protocols of concurrent objects are more complex. Interfaces have message queues, internal structure may be concurrent, and Message protocols may be synchronous (client/server), quasi-concurrent (coroutine), or asynchronous (with futures).

The goal of both sequential and concurrent object-based programming is to directly and naturally model the real world. The real world is concurrent rather than sequential. Object-based concurrency allows real-world concurrency of applications to be naturally modeled, thereby greatly expanding our modeling power. Concurrency adds an extra dimension of complexity to programs, so that the consequences of poor program structure are likely to be much more severe than for sequential programs. Management of concurrent programs through encapsulation and software methodology is even more critical than for sequential programs.

We consider design questions for object-based concurrency in the following areas:

Process structure
Internal process concurrency
Synchronization
Message passing
Inter-process communication
Granularity of modules
Persistence and transactions

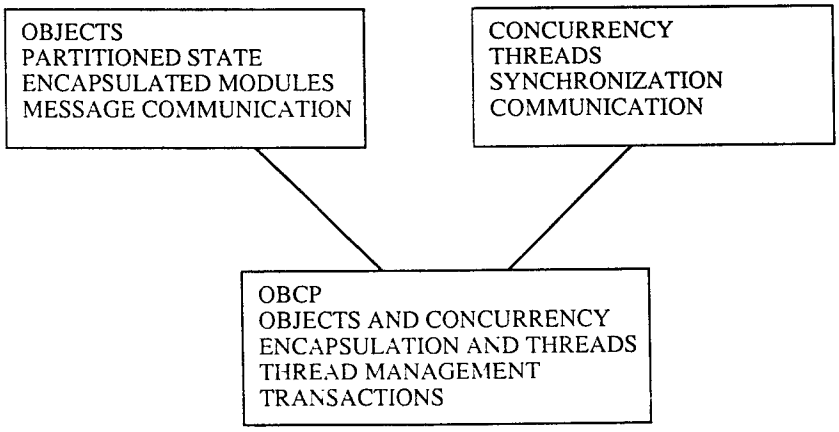

Figure 27: Combining Objects with Concurrency

6.1. Process Structure

Processes are active objects that must synchronize with messages arriving from other active objects. Design issues for processes include: protection and encapsulation, tasks versus monitors, logical versus physical distribution, and weak versus strong distribution.

6.1.1. Unprotected Versus Encapsulated Data

Consider an operating system with READ, EXECUTE, and PRINT processes, where the READ and EXECUTE processes share an input buffer and the EXECUTE and PRINT processes share an output buffer, as in Figure 28.

In the shared data model, the data in the input and output buffers are unprotected. The input and execute processes must cooperatively protect the data, for example by semaphores with request and release (P and V) operations that ensure mutually exclusive access.

In an object-based model, the input and output buffers are server processes responsible for their own protection. The input and execute (client) processes no longer need to use low-level primitives for protecting the data in the input buffer. Remote procedure calls that rely on local protection of data in called programs may be used. Ada tasks and monitors illustrate two distinct forms of interaction between calling clients and called server processes.

6.1.1.1. Tasks with Rendezvous

Ada tasks have a single thread of control that may synchronize with incoming procedure calls at entry points determined by accept statements:

Task body T
hidden local variables
sequence of executable statements that include
accept statements (entry points) synchronizing by rendezvous with remote procedure calls
accept bodies that determine communication during synchronization
endtask

Synchronization requires the calling and called threads to rendezvous in time (see Figure 29). Remote procedure calls of a calling module must rendezvous with accept statements of a called module. If a call arrives before an accept statement is ready to accept it, the calling procedure suspends and the call is placed in a queue. If an accept statement is reached before a waiting call then the task suspends and waits for a call. When a rendezvous occurs the threads of the

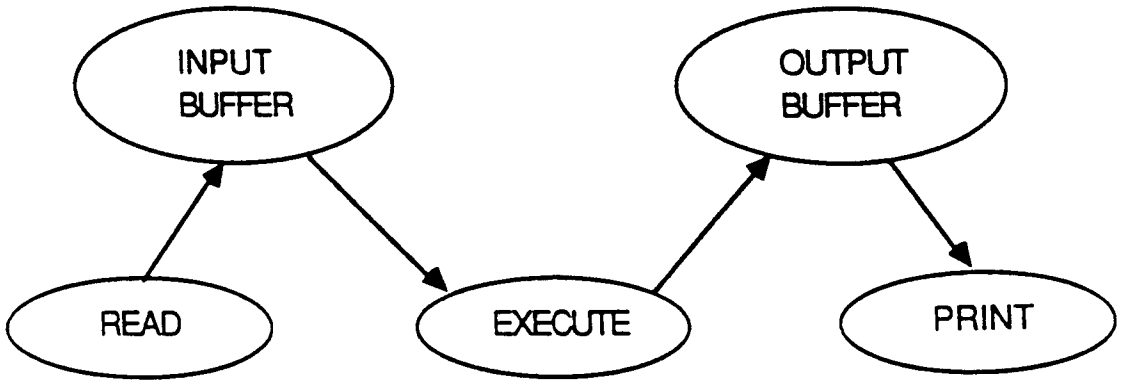

Figure 28: A Simple Operating System with Input and Output Buffers

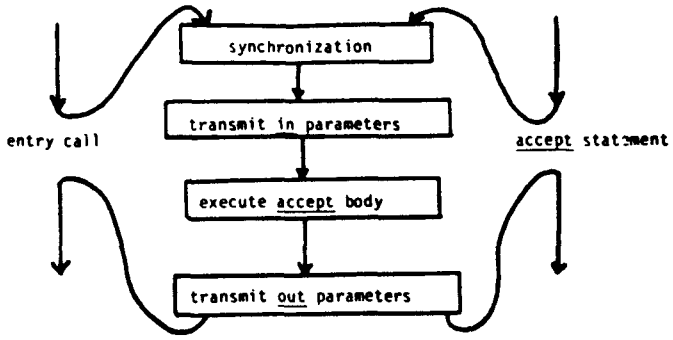

Figure 29: Rendezvous in Ada

calling and the called module are temporarily merged, input parameters may be transmitted from the calling to the called procedure, the code specified by the accept statement is executed, and output parameters may be transmitted back to the caller. On completion of the rendezvous the calling and called processes resume separate concurrent execution.

Entry points for accept statements are defined in *task specifications*. A buffer task with APPEND and REMOVE entry points may be specified as follows:

```
task BUFFER is
  entry APPEND (M: in MESSAGE)
  entry REMOVE (M: out MESSAGE)
end BUFFER
```

The task body has accept statements for accepting APPEND and REMOVE calls:

```
accept APPEND (M: in MESSAGE) do
  BUFF(IN) := M;
end
```

The code between the do and end is executed during the rendezvous. In this case no value is returned to the caller (M is an *in* parameter). However, the accept statement for REMOVE has an *out* parameter that returns the value of M to its caller as the final action of the rendezvous:

```
accept REMOVE (M: out MESSAGE) do
  M := BUFF(OUT);
end
```

We may distinguish *server tasks* with *accept* statements but no calls to other tasks and *client tasks* with calls to other tasks but no *accept* statements. The buffer task is a typical server while producer tasks that send their output to a buffer are examples of clients. Tasks that both provide a service and call other tasks act as both servers and clients.

6.1.1.2. Monitors with Internal Monitor Queues

Monitors have local variables and an interface of operations that are effectively entry points. They guarantee mutually exclusive access by clients to shared data, but their operations may suspend and later resume (like coroutines):

Monitor M
hidden local variables
operations (entry points) that include
wait commands for suspending and signal commands for resuming operations
endmonitor

Monitors have an entry queue for incoming monitor calls and wait queues for suspended monitor calls. Suspension is realized by a command **wait(condition-name)** that suspends the current thread and places it on the named wait queue, from which it may be removed (reawakened) by a command **signal(condition-name)**. When a thread suspends then a waiting reawakened or entering thread may commence execution.

A buffer with APPEND and REMOVE operations and internal wait queues EMPTY and FULL is illustrated in Figure 30. An attempt to execute a REMOVE when the buffer is empty will cause an entering thread to be placed on the EMPTY wait queue, while an attempt to execute an APPEND when the buffer is full will cause a thread to be placed on the FULL wait queue.

Monitors control the execution of threads more flexibly than tasks, allowing threads within a module to suspend and later resume. This requires greater complexity at the language level, with wait and signal commands, and greater complexity of implementation, with internal monitor queues. The system specifies priorities for the resumption of signaled threads in wait queues and incoming threads in the monitor queue: processes in wait queues usually have priority over incoming processes in the monitor queue.

6.1.1.3. Non-Determinism

Nondeterminism is a necessary feature of any client/server system because the demands made by clients for services cannot be predicted. Objects and monitors are at the mercy of clients to select the next operation to be executed. A BUFFER object cannot know whether the next operation will be an APPEND or a REMOVE.

Ada tasks accommodate nondeterministic behavior of clients through a select statement which selects among guarded waiting remote procedure calls (rpcs):

select
when guard-condition1 then accept rpc1
or
when guard-condition2 then accept rpc2
...
when guard-conditionN then accept rpcN
endselect

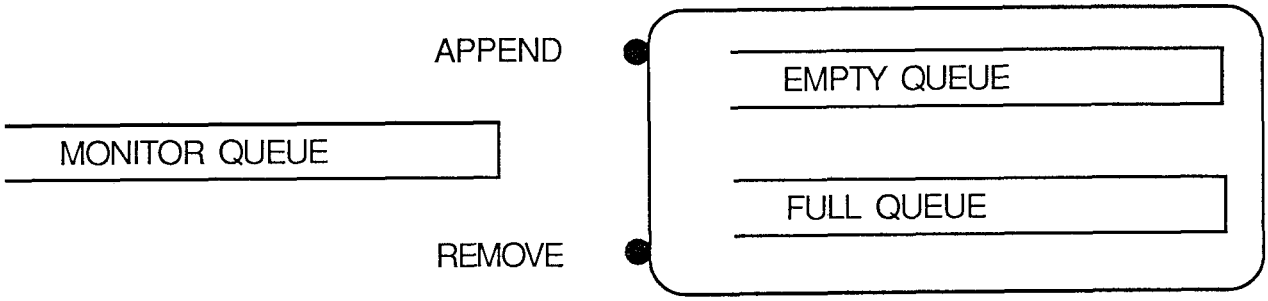

Figure 30: Monitor Queues

Select statement may be used to implement nondeterministic choice between APPEND and REMOVE operations of an Ada buffer task:

```
select
  when notfull then accept APPEND
  or
  when notempty then accept REMOVE
endselect
```

When the buffer is neither full nor empty, this statement chooses among waiting APPEND and REMOVE operations, and otherwise chooses the first to arrive. When the buffer is full executing REMOVE will unblock the buffer for APPEND. For an empty buffer executing APPEND will unblock it for REMOVE.

Ada's explicit select statement contrasts with the implicit nondeterminism of monitors, which consist of an implicit select statement with unguarded alternatives:

```
select op1 or op2 or ... or opN endselect
```

Guarded selection is realized in monitors by *wait* statements (wait(guard-condition)) that may suspend monitor operations during their execution. Monitors are more flexible than Ada tasks in allowing guards to occur not only on entering a process but at any point during execution of the process. They decouple nondeterministic entry from guard conditions that determine whether a process is ready for execution.

Nondeterminism is built into the fabric of object-based programming independent of concurrency. The nondeterminism of objects mirrors the real world where the sequence of events in which entities participate cannot be predicted. Since nondeterminism occurs even in the sequential case, the nondeterminism of object-based concurrency comes for free.

The nondeterminism of objects differs from that of traditional nondeterministic automata, where nondeterminism refers to the fact that a computational step in a given state can generate multiple next states. Objects have nondeterministic input to a computational step, while nondeterministic automata have nondeterministic output. The two forms of nondeterminism are is a sense dual and may be called input and output nondeterminism.

input nondeterminism: *the next operation on an object or process is unknown*
output nondeterminism: *the output is a nondeterministic function of the input*

Select statements arise because of the need to handle input nondeterminism of clients. They support output nondeterminism when several branches of a select statement are ready for execution. For example, if the select statement of the buffer task is reached when the buffer is neither empty or full, and both an APPEND and a REMOVE call are waiting to be executed, then output nondeterminism may be used to determine which action is actually performed.

6.1.2. Logical Versus Physical Distribution

A system of modules is logically distributed if each module has its own separate name space. Local data is not directly accessible to other modules, conversely modules cannot directly access non-local data, and must communicate with other modules by messages.

We distinguish between logical distribution defined in terms of properties of the name space and physical distribution defined in terms of geographical or spatial distribution of modules. Physical distribution usually implies logical distribution, since physical separation is most

naturally modeled by logical separation. But logically distributed systems are often implemented by shared-memory architectures for greater efficiency. Object-based systems are logically distributed but are usually implemented on non-distributed computers. The relation between logically and physically distributed processes is illustrated in Figure 31.

Logical distribution supports autonomy of software components and thereby facilitates concurrent execution. Another important benefit of logical distribution is its support of autonomous interface modules, such as multiple windows. Autonomous interface modules allow the user to pursue multiple autonomous, conceptually concurrent, interface activities. Physical concurrency is not required and usually unnecessary. But conceptual autonomy is an important property of activities in the real world, and its realization at the workstation interface is one of the most important practical benefits of object-based programming.

6.1.3. Weakly and Strongly Distributed Systems

A system is weakly distributed if its modules know the names of other modules. It is strongly distributed if its modules cannot directly name other modules. In a strongly distributed system a given module knows only the names of its own communication ports. The names of modules with which it communicates are stored as data in its communication ports, and may be viewed as pointers to ports of other modules that cannot be dereferenced to determine the name of non-local ports. A module communicates by sending a message, specified as a data value, to a local port for transmittal to a destination.

Traditional object-oriented languages are weakly distributed. Objects can know the names of other objects and send messages to methods of other objects. The weak distribution reflects that implementation is usually in a shared-memory architecture. Strongly distributed systems better capture the the reality of physical distribution where connections among modules of a network are realized by physical channels and the topology of the network may change.

In strongly distributed systems like Nil and Hermes [SY], processes can refer to their own ports but not to ports of other processes. When a process Q is created, the creating process must supply Q with initial capabilities which we here call an umbilical cord (see Figure 32). The created process stores the initial capabilities by a receive command as in the program PR below,

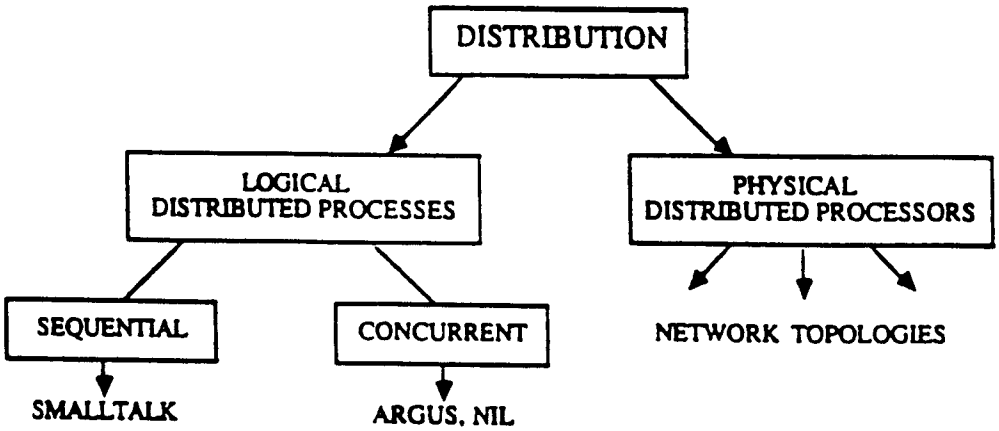

Figure 31: Logically and Physically Distributed Systems

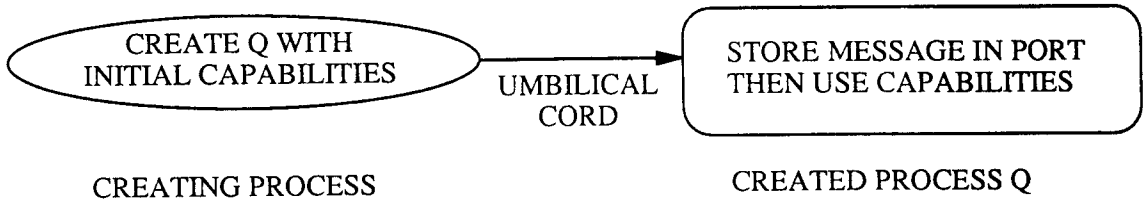

Figure 32: Creation of Strongly Distributed Processes

and can then make use of the capabilities to access its environment.

A newly created process finds its initial capabilities in the system-defined variable *init*. The program PR below for printing the message "Hello World" first executes the command "receive Port from Init", which stores its initial capabilities in the local communication port called *Port*. Then it calls the Putline component of its communication port with a message to print "Hello World". Finally it executes a return message that returns its capabilities to the caller, and allows the process to expire.

```
PR: receive Port from Init
   call Port.Putline ("Hello", "World")
   return Port;
```

Strongly distributed systems have greater overhead than weakly distributed systems but allow dynamic reconfiguration for anonymously communicating (*autistic*) processes.

6.2. Internal Process Concurrency

Processes of object-based concurrent systems may be internally sequential, quasi-concurrent, or fully concurrent, as illustrated in Figure 33.

Sequential processes are illustrated by Ada whose tasks have a single executing thread that may be suspended while waiting to receive an external communication but must run to

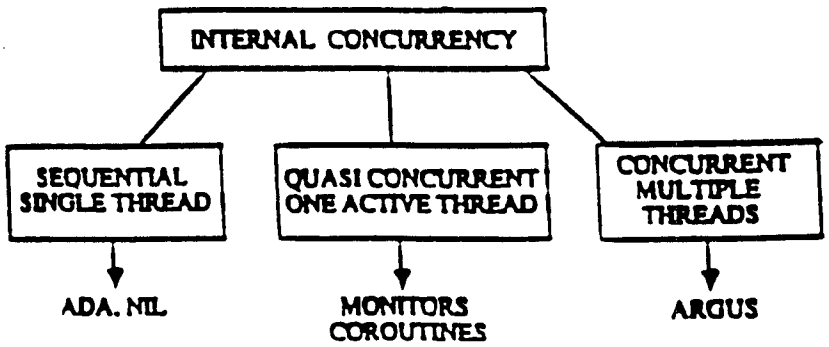

Figure 33: Internal Concurrency Within Processes

completion once the external communication is accepted. Rendezvous causes temporary merging of the incoming and executing threads until the interaction of the two threads is completed.

Quasi-concurrent execution is illustrated by monitors, which have at most a single active executing thread but may have suspended threads in one or more wait queues. The ability to suspend threads internally provides flexibility while the fact that at most one thread can be active ensures mutually exclusive access to local data. Because of these advantages object-based concurrent languages like ABCL1 and Orient 84 [YT] are based on quasi-concurrency, although their message passing protocols transcend the client/server mechanism of monitors. However, quasi concurrency gives rise to problems when the threads being executed represent transactions whose data may be accessed only when in a stable state. Suspension of threads when not in a stable state allows tampering with data that violates the conditions of transactions.

Fully concurrent processes are illustrated by Argus guardians [LS]. Guardians do not synchronize incoming threads on entry, so that entry of a thread to a Guardian simply increases the number of executing threads. Synchronization occurs later when executing threads within a Guardian attempt to access shared data.

Figure 34 illustrates an Argus mail system with MAILER, MAILDROP, and REGISTRY guardians that may be replicated at many physical locations. The Mailer is the user interface and allows users to send mail, receive mail, and add users to the mail system. The MAILDROP guardian contains a subset of the mailboxes and is invoked by MAILER to deliver mail to those mailboxes. The REGISTRY guardian specifies the maildrop for each user and must be updated when users are added or deleted from the system.

In this example, MAILER guardians may execute multiple threads freely since there is no shared data structure to worry about. MAILDROP guardians must synchronize for delivery and removal from a given mailbox but can concurrently interact with different mailboxes. REGISTRY guardians can permit concurrent lookup but must synchronize when adding or removing users. By delaying synchronization so that it occurs at the point of access to shared data rather than on entry to a concurrent module, we can realize greater and more finely grained concurrency.

6.3. Design Alternatives for Synchronization

We distinguish synchronization for unprotected and encapsulated data. For unprotected data, the work of synchronization must be performed by each process that accesses the data. Synchronization may require cooperative protocols: for example among semaphores to specify

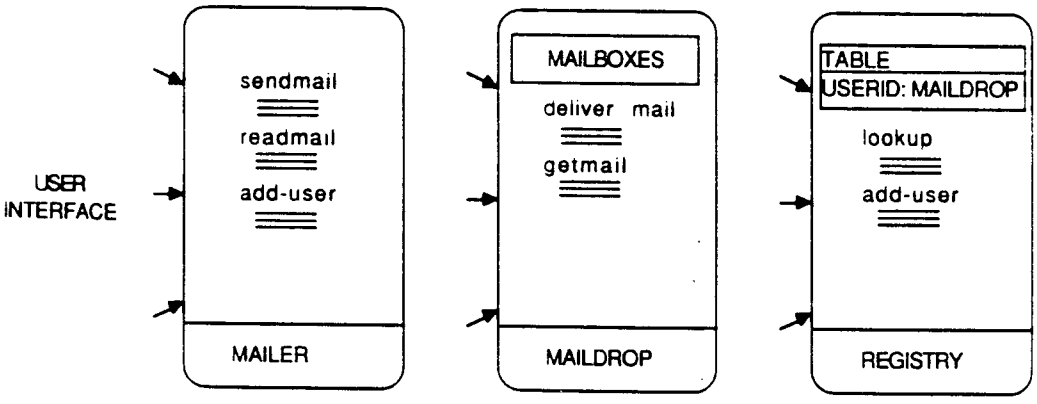

Figure 34: An Argus Mail System

mutually exclusive access to critical regions. Protected data assumes responsibility for its own protection, removing the burden from the processes that access the data. Object-based systems focus on synchronization for protected data.

Three kinds of synchronization mechanisms may be distinguished:

rendezvous: *synchronization between two threads (for sequential processes)*
condition variables: *controlled by wait and signal operations (for quasi-concurrent processes)*
locking: *synchronization between a thread and shared data (for fully concurrent processes)*

The rendezvous mechanism for sequential processes may be viewed as synchronization between two threads, namely the calling and the called thread. Synchronization is symmetrical for the two parties that synchronize. But the calling and the called threads play different roles once synchronization has occurred, with the calling thread being passive and the called thread performing the task for which it was invoked. An Ada rendezvous causes temporary merging of the calling and called threads for the purpose of executing the accept body, and subsequent forking to permit concurrent execution of the calling and called process to be resumed.

Quasi-concurrent processes employ implicit rendezvous-like synchronization to define entry to the monitor, and use condition variables (guards) to model internal synchronization. They decouple synchronization for entry and resumption of threads by having different protocols for entry and resumption, specified by different language primitives.

Fully concurrent processes involve synchronization between concurrently executing threads and local, unprotected shared data. This is realized by locking the data and temporarily restricting access to the thread on whose behalf the data was locked. The locking protocol may itself be complex when an operation requires exclusive access to multiple shared data entities. Two-phase locking, which separates the phase of acquiring locks from the phase of releasing them, reduces the likelihood of thrashing in the competition of threads for multiple shared resources. Alternative locking protocols for distributed architectures are discussed in [GT].

6.4. Asynchronous Messages, Futures, and Promises

Design alternatives for message passing include *synchronous*, *asynchronous*, and *stream-based* message passing. Synchronous message passing, which requires the sender to suspend until a reply is received, is essentially remote procedure call. Asynchronous message passing allows the sender to continue, but requires synchronization if subsequent execution depends on the reply. Stream-based message passing supports streams of messages which likewise require synchronization at message-use time to check that replies have been received.

Asynchronously computed results may be handled by data structures called *futures* created at the time of message-send. Anonymous futures allow synchronization of operators for asynchronously computed arguments, and may be assigned to variables:

```
+(future(e1), future(e2)) -- wait till both e1 and e2 are ready before adding
define(x, future(e))      -- x cannot be used till evaluation of e is complete
```

+ requires *strict* (synchronous) evaluation that blocks till its arguments and the sum are evaluated. *define* permits *nonstrict* (lazy, asynchronous) evaluation that blocks only when x is accessed for further computation.

Asynchronous *send* may be viewed as a nonstrict operation that creates a *reply-variable* at message-send time for synchronization at message-use time:

```
send(message) x(future(reply))  -- create x at send-time for synchronization at use-time
```

Future objects are used in object-based systems like ABCL [Yo1]. They are used in the *streamcall* mechanism of the MIT Mercury system [LBGSW], which allows the sender to send a stream of messages along a call stream and the receiver to send a return stream of replies (see Figure 35). Synchronization for streamcalls is realized in Argus by typed data structures called *promises* [LS1], which, like futures, are created at the time of call (message send) and can be claimed only when the promise has been fulfilled (the reply has been stored in the promise). Mercury supports synchronous, asynchronous, and stream-based message passing for heterogeneous, cooperating distributed processes:

Mercury Communication Mechanisms
call: *synchronous remote procedure call*
send: *asynchronous send with return for exceptions*
streamcall: *pipelined stream of calls and returns (Figure 35)*

6.5. Inter-Process Communication

Design alternatives for inter-process communication include two-way interconnected distributed processes, one-way interconnected client/server processes, and dynamically interconnected strongly distributed processes.

In CSP communication requires two-way naming, with the sending process knowing the name of the receiving process and the receiving process knowing the name of the sending process, as illustrated in Figure 36. Two-way naming models hard-wired interconnection between processes. However, one-way naming, where the called process need not know the names of its

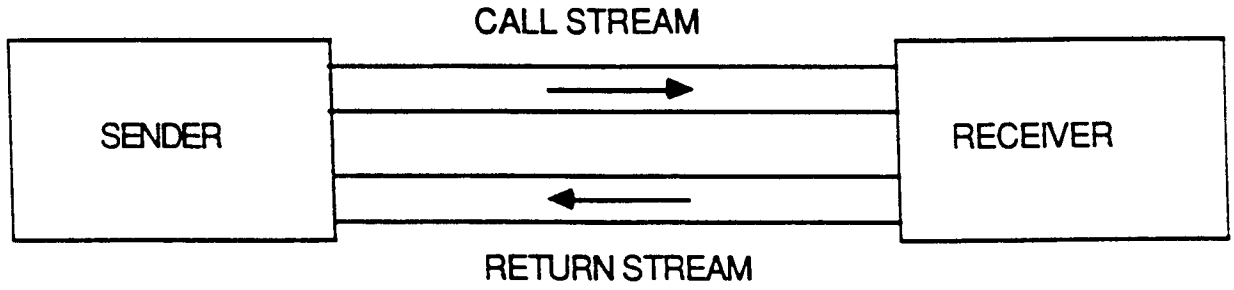

Figure 35: The Mercury Streamcall Message Passing Mechanism

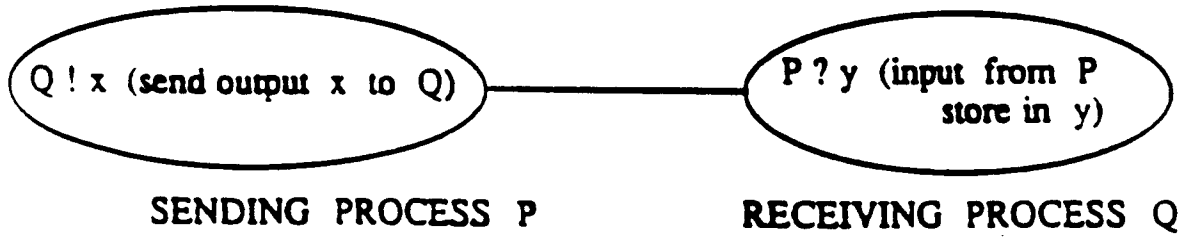

Figure 36: Two-Way Naming in CSP

callers, is more flexible and is the standard inter-module communication mechanism for procedures and synchronous message passing. Two-way naming is implemented in terms of one-way naming by passing the name of the caller as an argument to the called procedure. Ada *accept* statements use one-way naming, requiring the caller to name the called task but allowing a task to be called by anyone who knows its name (see Figure 37).

In strongly distributed processes, the names of non-local ports are stored as data in local port variables and the connection to to other processes is therefore dynamic. Channel interconnections are established by storing port values in port variables, as in Figure 38. A channel can be viewed as a cable with a plug at one end connecting it to its target and a socket at the other end into which the source can be plugged.

6.6. Abstraction, Distribution, and Synchronization Boundaries

Module boundaries in object-based concurrent systems are determined by the mechanisms for abstraction, distribution, and synchronization, as shown in Figure 39.

The *abstraction boundary* is the interface a module presents to its clients. It determines the form in which resources provided by an object may be accessed (invoked). It is an information-hiding boundary encountered by a client looking inward into a module. It limits what the client can see, hiding local data from access by the client. It is the unit of encapsulation and the fundamental unit of object granularity.

The *distribution boundary* is the boundary of accessible names visible from within an object. The abstraction boundary is encountered by a user looking inward, while the distribution

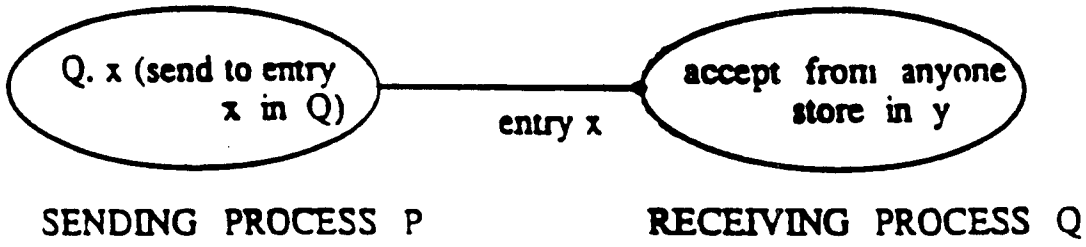

Figure 37: One-Way Naming in Ada Accept Statements

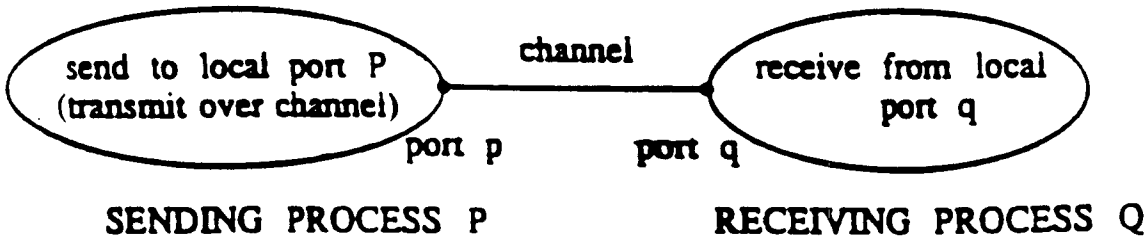

Figure 38: Dynamic Interconnection in Strongly Distributed Processes

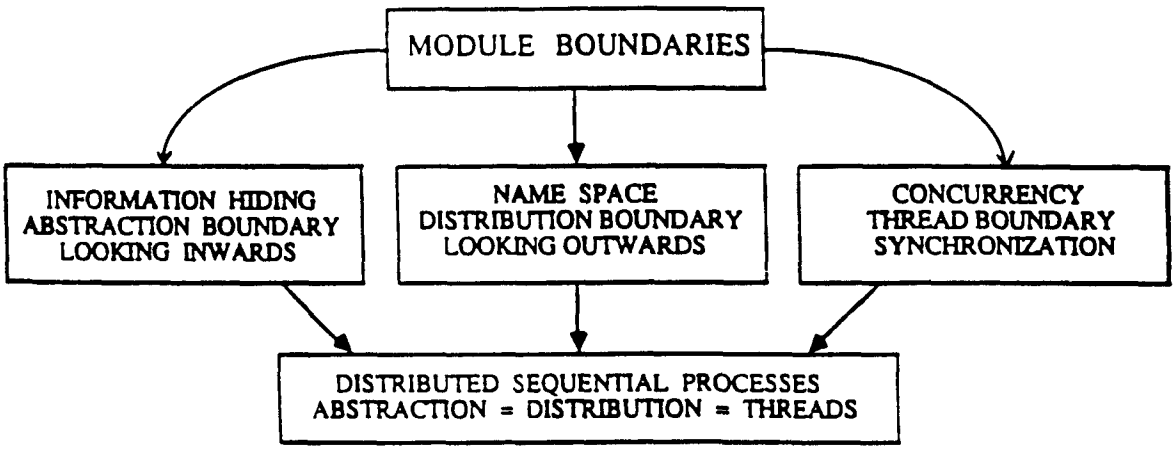

Figure 39: Abstraction, Distribution, and Synchronization Boundaries

boundary is encountered by an agent within a module looking outward. The distribution boundary may be coarser than the abstraction boundary, as in block-structure languages, or finer, when a large abstraction (say an airline reservation system) is implemented by distributed components. When the abstraction and distribution boundaries coincide, we say a module is *distributed.* Abstraction for distributed modules is symmetric: the receiver's view of the sender is as abstract as the sender's view of the receiver.

The *synchronization boundary* of a module is the boundary at which threads entering a module synchronize with ongoing activities in the module. For sequential processes the thread synchronization boundary is also the abstraction boundary. For concurrent processes the thread synchronization boundary is finer than the abstraction boundary. Conversely, the unit of abstraction can be coarser than that for concurrency, for example when the address space associated with a single thread can contain many abstract objects.

The boundaries for abstraction, distribution, and synchronization are in general independent. However, the special case when the three boundaries coincide is an interesting one. Processes for which these boundaries coincide are called *distributed sequential processes.* They are distributed because their abstraction and distribution boundaries coincide and sequential because their abstraction and synchronization boundaries coincide.

Distributed sequential processes are attractively simple. However, insisting on the same granularity for abstraction, distribution, and synchronization may reduce efficiency or expressive power. For example, large abstractions such as airline reservation systems need finely grained synchronization at the level of individual flights or even individual seats to execute efficiently. Conversely, a network of sequential computers with multiple objects at each node is naturally modeled by a unit of concurrency coarser than its unit of abstraction.

6.7. Persistence and Transactions

Data is persistent if its lifetime is long relative to the operations that act upon it: for example income tax data that persists from year to year. However, persistence is relative rather than absolute: if income tax were computed every hour (for fast moving stockbrokers) then data persistence would be measured in hours rather than years.

Figure 40 classifies modules in terms of the relative persistence of operations and data:

functions: *persistent actions on transient data*
objects: *operations and data with coextensive lifetimes*
transactions: *flexible relative persistence of operations and data*

Functions and procedures emphasize the persistence of programs for transitory data. Objects partly redress the balance by supporting coextensive persistence for operations and their data, but are not flexible in supporting variable persistence of operations and data. Databases solve the data persistence problem by entirely decoupling programs from the data on which they operate and introducing transactions for flexible temporary coupling of operations and data.

Transactions are atomic, all-or-nothing actions that either complete or abort with no effect on the rest of the program. When executed concurrently on shared data, their atomic effect can be achieved by locking sharable data temporarily to the operations of a particular transaction, so that the data and operations form a temporary entity that is dissolved when the transaction is completed. The temporary entity may be viewed as a dynamically created object that temporarily binds data to the operations of the transaction. During execution of the transaction its data is accessible only to local operations, just as for local data of objects. Transactions may be viewed as dynamically created temporal modules that supplement textual modularity of programs in space by modularity in the time dimension.

Transactions allow shared data structures to be associated with operation sets of different transactions at successive stages of their life. Conventional objects bind data structures to operations in a permanent union, while in a transaction system data structures can be promiscuous, having a variety of different partners. Operations can be equally promiscuous, having temporary liaisons with many different data structures. The relative persistence of operations and data is flexible and dynamically determined.

Are transactions object-based? The answer is no if objects are viewed as fixed combinations of data and operations. However, if the notion of object is extended to include temporary associations of data and operations, then transactions may be viewed as extending traditional object-oriented notions to include dynamically created objects.

Should persistent storage be object-oriented, or should it reflect the transitory association of operations with persistent data. It may well be that persistent storage should support loose bonds between operations and data, and transaction-like mechanisms for temporary bonding. These issues are being debated by proponents of object-based and relational databases [Ban].

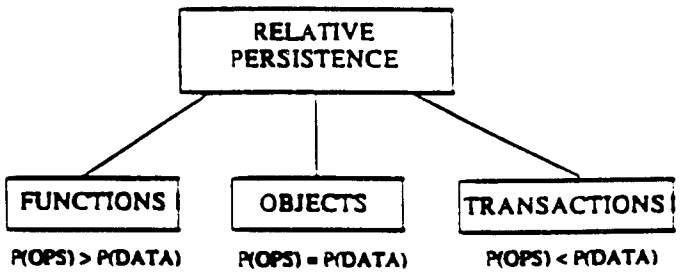

Figure 40: Relative Persistence of Operations and Data

7. What Are Its Formal Computational Models?

What is the role of mathematics in modeling computation? Is computation is simply a form of mathematics that requires troublesome computational mechanisms such as assignment and asynchrony to be banished? Or should computational models be judged by their ability to express and manage complexity, so that mathematics becomes a means rather than an end in itself. Object-oriented programming exemplifies the latter view, with rich and challenging mathematical models, but no comprehensive formal model for the complete paradigm.

Objects may be modeled by automata, but automata theory must be extended to model systems of communicating objects, especially in the presence of nondeterminism and asynchrony. **Types** are modeled by equivalence relations, algebras, and the lambda calculus. Equivalence relations capture the classification properties of types, algebras capture type behavior, and the polymorphic, typed lambda calculus models complete object-oriented type systems. **Inheritance** is modeled by composition of generators with unbound (unfixed) self-references. Taking the fixed point of a generator corresponds to binding (fixing) its self-reference. Fixed point (denotational) semantics of inheritance is developed as a novel application of fixed point theory. **Reflective systems** explore the computational consequences of treating programs as data. They provide an extra dimension of abstraction (*meta-abstraction*) that complements data abstraction and super-abstraction. Object-oriented reflective systems model the behavior of classes by metaobjects which can define class-specific reflective disciplines for debugging, interpretation etc, with default reflective behavior defined in a metametaobject.

7.1. Automata as Models of Object Behavior

The term *automaton* suggests inflexibility and mindlessness when applied to humans, but automata are nevertheless the basic formal mechanism for modeling the state transition paradigm. They provide a basis for describing Turing machines and digital computers. The stimulus/response behavior of objects can be specified by automata in terms of their inputs I, outputs O, state S, output function F, and state transition function G (see Figure 41):

$$Automaton = \langle I, O, S, F, G \rangle$$

The input alphabet determines a set of possible stimuli, while the output alphabet determines responses. The response to a given input event depends on the state. For a given input i and state s the output o=F(i,s) is determined by the output function F, while the next state s'=G(i,s) is determined by the state transition function G.

Traditional automata have their inputs on an input tape. For object-based automata the inputs (events) are generated dynamically as part of the computation. If they arrive faster than they can be handled, they are queued and handled when the automaton can accept them.

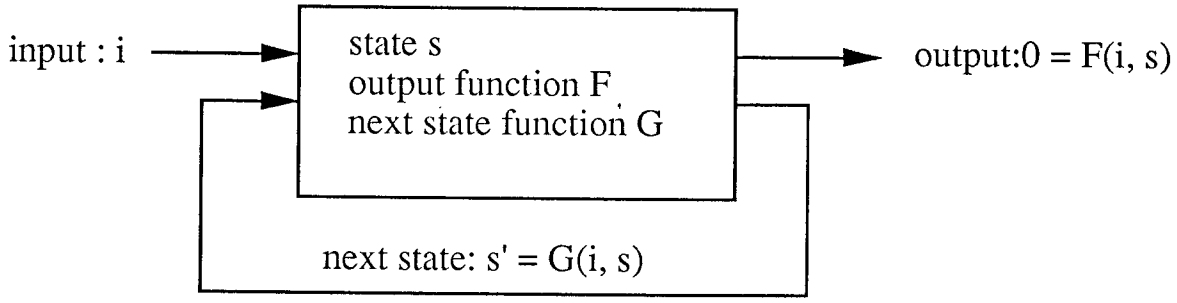

Figure 41: An Automaton with Output and Next-State Functions

The output function F captures the stimulus-response behavior of the automaton at the interface, while the next-state function G captures its internal behavior. This corresponds to the separation between observable and internal behavior of an abstract data type.

Finite automata theory was a rich research area in the late 1950s and 1960s. It included both the synthesis of automata from sequential circuits and algebraic automata theory. It was shown that any automaton could be constructed as the composition of *nand* or *and/not* logic gates, or by the algebraic composition of simple primitive semigroups. But these results assumed synchronous composition with the output at one time step being available as input at the next time step. Objects are subject to more complex and as yet incompletely understood communication protocols and composition laws.

7.1.1. Mapping Objects into Automata

An object with state s and operations (methods) f1,f2,...,fN may be viewed as an automaton which, when it receives an input fi with argument x, performs the following actions:

it returns an output fi(x, s)
it performs a state transition to a new state gi(x, s)

Objects may be viewed as automata whose input alphabet consists of input-argument pairs (fi,x). Each input may trigger both an output and a state transition. Figure 42 illustrates an object with state s, operations f1, f2, ... , fN, and state transition functions g1, g2, ... , gN. Objects generally do not separate output and next-state functions, with each operation fi having a derivative state transition function gi.

Object automata have the structure of finite automata [LP], but their states may become arbitrarily large, as in the case of unbounded buffers or stacks. The sequence of operation-argument pairs (f,x) of an automaton determines its computation history and is sometimes referred to as a "trace". The trace plays the role of the input tape of a traditional automaton.

7.1.2. Communicating Sequential Processes

The communicating sequential processes of Hoare [Ho3], Milner [Mi], and Henessey [He] model individual processes by automata and support composition of parallel processes. Henessy's example process language EPL, which reflects the process models of both Hoare and Milner, defines processes in terms of an alphabet of events and a *become* operation (→).

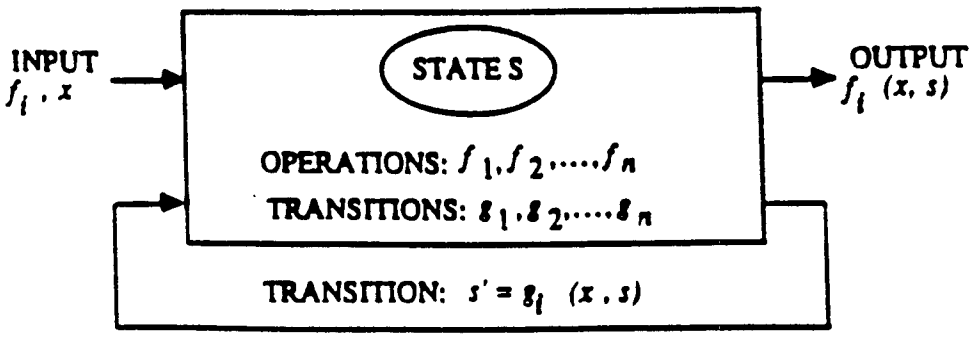

Figure 42: Objects as Automata

"P → aQ" means: "process P acted on by event a becomes Q". "P → aP" means that P can be acted on by a sequence of a-events, and "P → abP" means that P can be acted on by a sequence of "a followed by b" events. Choice by P between the events a and b is indicated by "P → aX + bY". The process "P → acP + bdP" allows a sequence of "a followed by c" or "b followed by d" events. Parameterized processes can specify choice among a set of events with similar *become* properties: P(x:B) → Q(x) means that P acting on an event x in the set B becomes Q(x).

The sequence of actions of a process during a computation is called a *trace*. Process behavior may be partially characterized by sets of possible traces. But traces are not complete behavior specifications: "P → a(b+c)X" and "Q → abX+acX" have the same set of traces but different behavior because after the action a, P can execute b or c while Q can execute only b. Behaviorally equivalent processes must continue to have the same set of traces after executing any permitted initial action sequence. Equivalence of traces is a *congruence relation* preserved by the execution of actions of equivalent automata.

Output events are indicated by primes, with a' being the output event matching the input event a. Parallel processes communicate by rendezvous between an input and a matching output. Thus "R → aR" and "S → a'S" can forever communicate by S sending a-messages to R.

Process composition of P and Q is indicated by "P | Q", and allows P and Q to execute in parallel (respond in parallel to events that drive their execution). If "X → acX" and "Y → ac'Y", then "XY = X | Y" will allow the a-event of X and the b-event of Y to execute in parallel and will then require rendezvous between the c-event of X and the c'-event of Y.

Information hiding of internal events may be realized by *restriction*. XY/{c} restricts XY so that c is internal and only a and b are visible to the outside world. Restriction makes internal actions of a process *unobservable*. Thus R|S/{a} is an *autistic* process having no communication with the outside world that forever executes the unobservable event a.

Restriction is a *concurrent data abstraction operator* that creates nonsequential processes with a more complex internal structure than traditional automata, capturing the properties of real-world agents with nonsequential internal behavior. Such agents can no longer be modeled by standard automata theory. The concept of *observable behavior* for such agents is complicated by unobservable internal actions may cause unpredictable changes of interface behavior. Milner's definition of *observational equivalence* [Mi1] in terms of bisimulation (two-way simulation) provides a basis for a precise definition of observability.

The above primitives allow the construction of composite processes with interesting behaviors and nice algebraic properties [He]. But asynchrony and nondeterminism lead to mathematical complications in specifying the behavior of large (real) programs. Moreover, the two-way naming communication protocol does not capture the behavior of client/server processes, and there are other respects in which the formalism is too simple to directly model concurrency in the real world.

Composition of real-world objects may involve multiple interdependent interconnections (Siamese twins), and restructuring of interfaces (by *unrestrict* operators) to remove internal communication barriers. The behavior of composite objects (corporations, multiperson research groups, the human body) cannot be easily or uniformly specified in terms of the behavior of their components. The behavior of a society cannot be easily predicted from the behavior of the individuals who comprise it [Min]. The problems of declarative object and process composition are much harder than those of imperative function composition. Perhaps it is too much to expect software component technology to address these issues, but we should nevertheless be aware of the large gap between models and reality.

7.1.3. Synchronous Systems

By making simplifying assumptions that eliminate asynchrony, a mathematically tractable model of concurrent computation can be developed. Synchronous systems consist of concurrently executing modules that are synchronously updated at each computational step. Figure 43 illustrates concurrently executing automata A1, A2, ... , AN in states s1, s2, ... ,sN which may be synchronously updated to states s1', s2', ... ,sN' in a single computational step. The global system state is the cross-product of component states.

How useful are synchronous systems as a practical tool for modeling concurrency? When the computation time within a module or the communication time among modules is greater than the synchronous computation interval, then synchronous systems are not realistic. But when the computation and communication time are small compared with the interval between computation steps, then synchronous systems are a good approximation. This is true of many real-time reactive systems such as control systems for elevators, chemical plants, and power stations. For such systems the model of a synchronously executing superautomaton with concurrently executing synchronous components is adequate and useful.

Synchronous systems assume that computation and communication are instantaneous, occurring at the instant of state transition. Since outputs of a given module are instantaneously accessible throughout the system, broadcasting is a more appropriate model for communication than traditional message passing. Each module may at each computation step receive instantaneously transmitted messages from any other module. We may think of the communication medium as an ether in which messages travel like light in all directions at an effectively infinite speed and may be acted on by any observer tuned in to its wavelength.

The interdependence among modules is specified by a global state transition function. Inter-module communication may be entirely different for different states, and is more flexible than for statically interconnected networks. Synchronous systems were first developed in the context of Esterel by Berry and Gonthier [BG], and are also exemplified by Statecharts [Har].

7.2. Mathematical Models of Types and Classes

Types give rise to deep philosophical and theoretical questions not only for programming languages but also for mathematics and biology [We2]. What is the relation between types and values? Are values fundamental and types simply derived value classes, or are types the basic conceptual entities and values accessible only through the interpretive filter of a type? The philosophical debate between realists who believe in the primacy of values and idealists who believe in Platonic ideals has its counterpart in mathematics and physics [We2].

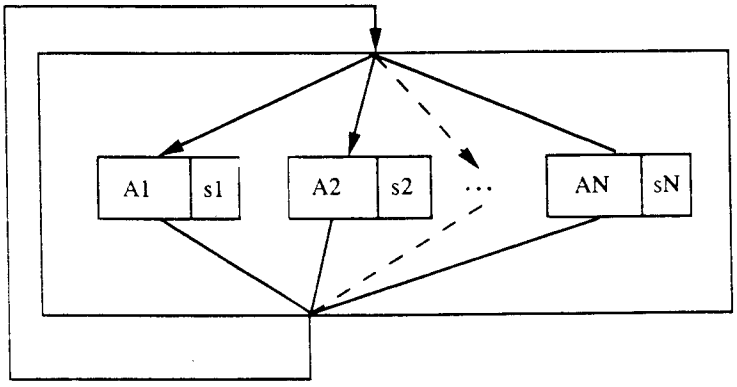

Figure 43: Synchronous Systems

Properties of types are very different from those of values. Values are transformed by computation rules (reduction rules), while types model structure or behavior not directly related to computing a result. Ideally we would like to integrate types and values into a single seamless system, but this ideal has not been realized in practice, and may be theoretically impossible.

Type expressions are generated from basic types by type constructors:

type → *basic-type* | *constructed-type*
basic-type → *Int* | *Real* | *Bool* | *Char* ...
constructed-type → *array-type* | *record-type* | *function-type* ...
array-type → *array [size] of element-type*
record-type → *record [{ident:type}+]*
function-type → *(domain-type* → *range-type)*

We are interested in the following questions about type expressions and associated types:

Equivalence: *When are two type expressions TE1 and TE2 equivalent?*
Polymorphism: *What relations between type expressions TE1 and TE2 may be expressed?*
Semantics: *Given a type expression TE what is the semantics of the associated type T?*
Behavior: *What is the behavior of type T?*
Behavior Modification: *How can the behavior T1 be modified by T2?*
Type Checking: *Does a variable x have the type T?*
Type Inference: *What can be inferred about the type of an expression e from its context?*

Different questions about type lead to different kinds of mathematical theories. Questions of type equivalence and polymorphism may be modeled by equivalence relations, questions of type behavior by algebras, and questions of type checking and type inference by the typed polymorphic lambda calculus.

The following complementary models of type each have different goals:

classification models in terms of equivalence relations
behavior models in terms of algebras
complete language models in terms of the polymorphic, typed lambda calculus

7.2.1. Types as Equivalence Classes

Equivalence relations are the fundamental classification mechanism of mathematics. By modeling types as equivalence relations, we capture their classification properties.

Simple type systems classify values of a universal value set into disjoint equivalence classes (see Figure 44). Every value belongs to precisely only one type. But there may be infinite hierarchies of types such as "(Int → Int)" and "((Int → Int) → Int)", etc.

Equivalence relations are too rigid a classification mechanism because they classify values into disjoint classes. We relax this rigidity, first by allowing partial classification in which some values remain unclassified and then by allowing overlapping value classes.

A single type may be modeled by a partial equivalence relation (PER), formally defined as an equivalence relation that is symmetric and transitive but not reflexive (Figure 45). Reflexivity holds only for values of the type, which form islands of type-equivalent values in a sea of unreflexive values of other types.

Polymorphic type systems permit values to have more than one type, corresponding to situations where values are subject to different rules of computation in different contexts of use.

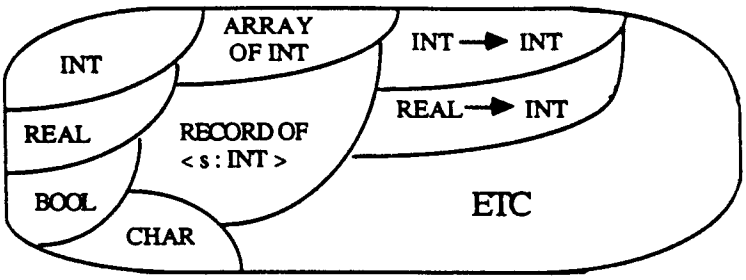

Figure 44: Types as Partitions of the Universal Value Set

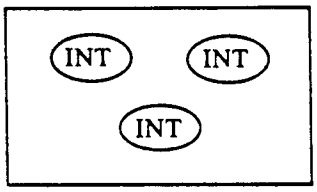

Figure 45: Islands Determined by Partial Equivalence Relations

For example, a person will behave differently in a family and a work environment, a program will behave differently in a compile-time and execution-time environment, an integer may double as a real or complex number, and a Toyota may be of the type Car and Vehicle.

Polymorphic type systems may be modeled by "compatibility relations" that are reflexive and symmetric but not transitive. Speaking a common language is a compatibility relation. If A speaks English and French, B speaks French and German, and C speaks German and Spanish, then A is compatible with B, B is compatible with C, but A is not compatible with C.

Compatibility relations determine *coverings* of their domain by overlapping compatibility classes. For example, languages cover a group of people by overlapping classes so that the number of classes to which a person belongs is the number of languages spoken. In Figure 46, the integers, reals, and complex numbers are a degenerate covering in which the complex numbers subsume the reals which in turn subsume the integers.

Classification into disjoint classes is captured by equivalence relations, classification into overlapping classes by relaxing transitivity, and focusing on a single class by relaxing reflexivity. It is quite remarkable that these simple abstract conditions provide so much flexibility in defining practical classification systems.

Eliminating reflexivity causes every value to belong to at most one class, eliminating transitivity causes every value to belong to at least one class, and requiring reflexivity, symmetry, and transitivity causes values to belong to precisely one class.

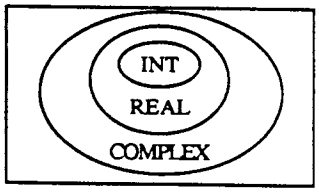

Figure 46: Covering by Overlapping Compatibility Classes

symmetric + transitive → value belongs to at most one class
reflexive + symmetric → value belongs to at least one class
R + S + T → value belongs to precisely one class

Equivalence relations specify when two values belong to the same type but say nothing else about properties of the type. In order to specify type behavior we need stronger structure on the elements of a set. Algebras impose structure on sets by means of operations, and provide a basis for modeling classes and abstract data types.

7.2.2. Types as Algebras

The data and associated operations of a programming language class or type may be mathematically modeled by an algebra:

Algebra = <set of values; operations>

Operations have an *arity* determined by their number of operands. The set S together with the operations and their arity is called a *signature*. Operations with zero, one, and two operands are referred to as nullary, unary, and binary operations.

The algebra of integers under addition and multiplication has a set I of integers, two binary operations "+,*" and two nullary operations "0,1":

Integers = <I; +,,0,1>*

To complete the definition of an algebra we must define how its operations transform its elements. This can be specified by rules, algorithms, or hardware for applying operations to values. A weaker specification of the properties of algebraic operations that constrains their behavior without completely defining it may be given by axioms:

i+j = j+i commutativity
(i+j)+k = i+(j+k) associativity
i(j+k) = i*j + i*k distributivity*
*i*1 = i multiplicative identity*
*i: i*0 = 0 multiplicative zero*

Variables in these axioms have implicit universal quantifiers: ∀ i,j: i+j = j+i

The set I is called a "sort" and the algebra of integers is called a single-sorted algebra because all operations take arguments and return values of just a single sort.

7.2.2.1. Many-Sorted and Order-Sorted Algebras

Object-oriented operations generally require arguments of several different sorts. A Type is therefore modeled by a many-sorted algebra whose signature is a set of sorts S and a set of functions fi with type Ti:

Many-sorted-algebra = <Set of Sorts: f1:T1, f2:T2, ... >

For example, a stack type is a many-sorted algebra with four sorts and three operations:

Set-of-Sorts: (Int, List(Int), Bool, Void)
Set of Operations:
*push: List(Int) * Int → Void*
pop: List(Int) → Int
empty: List(Int) → Bool

The behavior of the operations may be partially specified by axioms:

pop(push(x, stack)) = x
empty(push(x, stack)) = false

These axioms impose constraints on the stack operations that are *sound* in the sense that they are consistent with the actual behavior of stacks. Finding axioms that are *complete* in the sense that they completely specify stack behavior is more difficult, in part because stack behavior is an informal notion. The goal of algebras is to capture the semantics of behavior by a set of axioms with purely syntactic properties.

Algebras model the behavior of operations by equational axioms rather than by assignment of values to variables. They can capture the behavior of functional objects and of objects like stacks with non-destructive push and pop operations, but not of imperative objects with an updatable state.

Algebras were developed to model mathematical objects like integers and reals. Extension to many-sorted algebras is required to model classes. Further extension to *order-sorted algebras* [GM] is required to capture the ordering relations among sorts that arise in subtypes and inheritance. Algebraic models of subtype are also discussed in [BW], where notions of subtype defined in terms of subsets, isomorphic embedding, and predicate constraints are contrasted. There is little traditional mathematics on the second-order relations among algebras needed to capture the notion of inheritance and other forms of type polymorphism.

7.2.2.2. Provability, Truth, and Models

Computation models the semantic properties of the real world by purely syntactic symbol transformations. Mathematics likewise has formal languages with purely syntactic symbol transformations that aim to model semantic properties of functional application domains. The mathematical notion of proof in a formal system mirrors the notion of computation: the theorem being proved corresponds to the notion of a computational value.

The canonical formal system in mathematical logic is the predicate calculus, which provides a universal logical framework for domain-specific formalization of application areas. In contrast, algebras model specialized kinds of behavior, and equational axioms for a specific algebra may be viewed as syntactic approximations to the semantic behavior embodied by the algebra.

What do computational and mathematical models have in common? Both aim to capture semantic behavioral properties by purely syntactic transformations (computation and proof). Both are defined by languages for syntactic symbol transformation that aim to capture independently specified semantic behavior. Mathematics distinguishes between *proof theory*, concerned with methods of proof, and *model theory*, concerned with the inherent semantics of the domain being modeled. For computation, proof theory corresponds to operational semantics while model theory corresponds to denotational semantics. A primary goal of semantics is to demonstrate the adequacy of operational, computational, and proof-theoretic syntactic formalisms in capturing the semantics of the domain being modeled.

Model theory is defined in [CK] as "algebra + logic". A mathematical model maps assertions about algebraic expressions into truth values. Truth is a semantic notion distinguished from the syntactic notion of provability. *Proof theory* is concerned with the properties of proofs while model theory is concerned with whether assertions proved in a formal system are true in models of that system.

Given a formal system that defines provability and a model that defines truth, the formal system is said to be *sound* (relative to the model) if everything provable is true, and *complete* if everything true is provable. An algebra is a sound model of a behavior if properties of operations derivable from the axioms are consistent with the behavior, and is a complete model of the behavior if all its properties are derivable from the axioms. The equational axioms of an algebra capture the behavior of a model if they are both sound and complete for that model. Thus model theory establishes a relation between behavior specified by formal systems and behavior specified by an independent notion of truth.

What is the relevance of this discussion to establishing that a program captures a desired informal behavior? We cannot in practice determine soundness and completeness of algebras for informal behavior. The best we can do is to determine soundness and completeness for some formalization of the informal behavior. This leaves open the question of how informal behavior should be formalized and of how the adequacy of formalization can be demonstrated. We can never completely establish the validity of formal systems as models of informal concepts because this would be tantamount to formalizing them. But mathematical models do capture informal concepts more directly than axiom systems. There are valid analogies between informal and mathematical modeling that need to be better understood.

7.2.3. The Polymorphic, Typed Lambda Calculus

The lambda calculus [Ba] was historically developed as a formalism for computing with values, and later modified to include types. Its simplicity allows us to clearly demonstrate the interactions of types and values in a computational formalism. The polymorphic, typed lambda calculus models polymorphic types in the context of lambda calculus value reduction.

7.2.3.1. The Untyped Lambda Calculus

The syntax of lambda expressions may be specified as:

expression → *variable* | *function* | *combination*
function → *fun(variable). expression* – *abstraction*
combination → *expression (expression)* – *application*

The two primary semantic concepts are *application* and *abstraction*. Application simply applies one expression to another, and causes substantive computation (reduction) if the first expression is a function specification of the form "fun(x). E".

Abstraction converts an expression E into a function "fun(x). E", parameterizing E by a variable x that may occur in the body of E. Abstraction here means *functional abstraction* and serves to indicate that x is the varying part of the expression $E(x)$ while its remaining structure stays invariant. If $E = f(x)$, then functional abstraction parameterizes the argument x and leaves the function f invariant. By analogy, we could define a data abstraction operator *data(f). f(x)* that parameterizes the function part and leaves the data part invariant, so that different functions may be applied to the given data. We could then characterize function abstractions as having the form *data(f). fun(x). f(x)* and object abstractions as having the form *fun(x). data(f). f(x)*. That is, function abstractions supply data parameters to invariant function bodies while object abstractions supply functions (messages) to be interpreted in an object environment. However, since the lambda calculus does not formally distinguish between functions and data, just a single

abstraction operator *fun* (or *lambda*) is sufficient.

The successor and twice functions are defined by the following lambda expressions:

succ = fun(x). x+1
twice = fun(f). fun(y) f(f(y))

The primary computation rule in the lambda calculus is the reduction rule which specifies that a combination of the form "fun(x). M (N)" is evaluated by substituting N for instances of x in M, as in the following sequence of reductions:

twice(succ)(3) $\rightarrow$
fun(f). fun(x). f(f(x)) (succ) (3) $\rightarrow$
fun(x). succ(succ(x)) (3) $\rightarrow$
succ(succ(3)) $\rightarrow$ *5*

This computation applies a sequence of reduction rules to the initial lambda expression until an irreducible expression (called the value) which cannot be further reduced is obtained.

7.2.3.2. The Typed Lambda Calculus

The typed lambda calculus associates types with bound variables so that the syntax of functions is modified as follows:

function $\rightarrow$ *fun(variable: type). expression*

The typed successor function has the following form:

succ = fun(x: Int). x+1

This has the effect of prohibiting the application of succ to arguments not of the type Int, and involves type checking to determine type compatibility. The property of the untyped calculus that any expression is applicable to any other expression is not true in the typed calculus. The typed lambda calculus provides safety in ensuring that operands of combinations are compatible with their operators, but at the expense of augmenting the pure substitution rules by type checking rules that bear no relation to rules of computation.

7.2.3.3. Polymorphism

The typed lambda calculus has no mechanism for expressing structural similarity of types. The twice function must be separately specified for the types Int and Real:

twiceint = fun(f: Int $\rightarrow$ *Int). fun(x: Int). f(f(x))*
twicereal = fun(f: Real $\rightarrow$ *Real). fun(x: Real). f(f(x))*

Such structural similarity can be expressed by expressions having a polymorphic type:

twiceall = all[t] fun(f: t $\rightarrow$ *t). fun(x: t). f(f(x))*

This expression has the following polymorphic (universal) type:

$\forall$ *t. ((t* $\rightarrow$ *t) * t)* $\rightarrow$ *t*

That is, for all functions of type (t → t) and arguments t, a result of the type t is returned.

Twiceall captures the uniform structure of double application for all types t. It can be specialized by supplying a type parameter:

twiceall(Int) = twiceint
twiceall(Real) = twicereal

If twiceall is supplied with the type parameter Int, followed by the function parameter succ, followed by the integer parameter 3, it applies succ twice to 3, yielding 5:

twiceall(Int)(succ)(3) → 5

Universal types capture a particular kind of polymorphism: that associated with type parameters of generic functions. The language Fun [CW] extends the typed lambda calculus with three kinds of polymorphic types: universal, existential, and bounded types:

universal-type → ∀ variable. type
existential-type → ∃ variable. type
bounded-type → ∀ variable ≤ type. type

Existential types capture information hiding and abstract data types, while bounded types capture inheritance. Thus Fun models type systems that arise in object-oriented programming.

7.2.3.4. Type Checking and Type Inference

Type checking may be formally specified by a collection of rules (axioms) such as the following rule for function application:

given that: f has the type A → B and x has the type A
infer that: f(x) is type correct and has the type B

This rule is both a type checking rule for demonstrating that f(x) is type correct, and a type inference rule for inferring the type of f(x) from the type of f and the type of x. There is a close relation between type checking and type inference, since type checking requires type inference to demonstrate correctness. For example, to demonstrate the type correctness of "g(f(x))" where g has the type "B → C", we must use the inference that f(x) has the type B.

Type checking rules specify a compile-time semantics for types that has little to do with execution-time behavior. The typed lambda calculus has two entirely different sets of rules, one for type checking of expressions and one for computing with values. This corresponds to the difference between compiling and interpreting of programs, and is therefore quite natural. However, the resulting formalism is a hybrid of two different kinds of expressions with hybrid computation rules that no longer possesses the pristine simplicity of the pure lambda calculus.

7.2.4. Type Inference and Most General Types

Type inference systems allow the programmer to write implicitly typed expressions [BH] whose explicit type is automatically determined by a type inference system. Languages with type inference like ML support the following style of computer dialog:

input by person: 3 + 4
computer response: 7: Int

Here the computer infers that 3 and 4 have the type Int, computes the result, and informs the user that the result has the type Int. When the person specifies a function the type inference system determines the type of the function:

input by person: succ = fun(x). x+1
computer response: Int → Int
input by person: twiceall = fun(f). fun(x). f(f(x))
computer response: $\forall\ t.\ ((t \rightarrow t) * t) \rightarrow t$

To infer that *twiceall* is a universal polymorphic type, the ML type inference system must determine a most general type corresponding to all contexts of use in which the subexpression can meaningfully occur. If the subexpression does not constrain the contexts in which it may be used, the system may have to invent new most general types. The idea of types as *most general contexts* differs radically from that of types as behaviors: the mapping from syntactic contexts to semantic behaviors is not always straightforward.

The idea of type inference was developed in connection with ML [ML]. Universal polymorphic types are required in ML not only because they are useful, but also because functions like twiceall which do not reveal their context of use could not otherwise be typed.

In type inference systems the type of a type expression is the set of all contexts in which the expression can have a meaning. The idea of type as a set of contexts of interpretation is very different from the view that a type is a behavior or an equivalence class of values. Clear understanding of the intuition underlying each of these concepts of type helps us to better understand the associated mathematical models.

Languages with type inference allow users to have their cake and eat it too; reaping the benefits of type safety without the burdens of explicit type declarations. But implicitly typed values have a subtly different semantics from untyped values. They introduce polymorphic most general types beyond those needed to describe the application domain. A modest set of types chosen to model behavior may give rise to idiosyncratic contextual types that do not correspond directly to behavior. Kanellakis, Mairson, and Mitchell [KMM] have shown that the computational complexity of type inference is in principle exponential, although expressions that cause exponential type inference are unlikely to occur in practice.

7.2.5. Untyped Versus Typed Languages

What can mathematical semantics contribute to the debate between advocates of typed and untyped languages? Mathematical model theory establishes a relation between programs and their denotations that parallels their computational semantics. Type inference maps untyped into explicitly typed expressions. Figure 47 succinctly expresses relations between untyped and typed models. It provides a framework for exploring the tradeoffs between a) directly executing an untyped program and b) converting it to a typed program and executing the typed program.

Given a language in the top left-hand corner of Figure 47, the top horizontal line represents models of the untyped language, the left vertical line represents a type inference system from the untyped to the typed language, and the bottom horizontal line represents a model of the typed language. The right vertical line represents mathematical models from untyped to typed denotations. If "*type(untyped-denotation(program)) = typed-denotation(type-inference(program))*" for all programs, then the diagram commutes. However, some proposed models do not commute and Figure 47 therefore does not assume that typing of untyped denotations will always be equivalent to type inference followed by taking the typed denotation.

The first lambda calculus models were untyped, providing paradox-free denotations for untyped lambda expressions in lattice-structured domains of continuous functions [Ba, Sc]. The

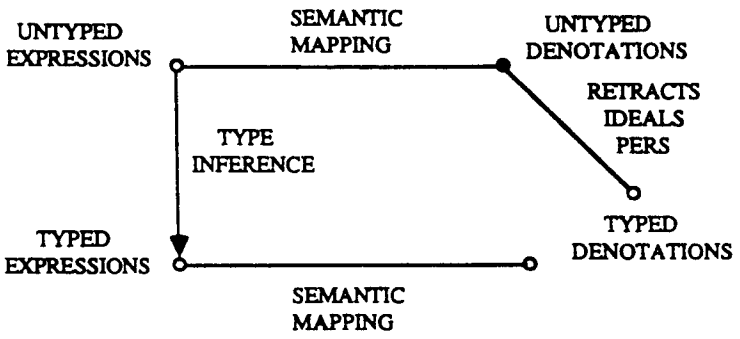

Figure 47: Relation Between the Untyped and Typed Lambda Calculus

relation between untyped and typed models is reviewed in [BH] and in greater detail in [Mey, BMM]. Although untyped models historically preceded typed models, the semantics of untyped expressions is more precisely captured by first mapping them to typed expressions than by untyped denotations. We can express semantics more simply and safely by first mapping untyped into typed expressions and restricting our denotational mapping exclusively to typed expressions. Advocates of typed languages adopt the following principle:

Typing principle: Binding expressions to a type should precede binding them to a denotation

Mathematical formalisms, including programming languages, are safer if they have an explicit notion of type. They have a simpler operational and denotational semantics than their untyped counterparts because semantics need be given only for expressions that are type compatible. *Typing before evaluation* is a divide and conquer strategy that divides the universe of values into typed categories in order to more easily conquer it by computation.

The counterarguments that typing is extra baggage extraneous to the direct behavioral properties of values, that typing prejudges and therefore restricts the contexts in which values may be used, and that untyped reflective systems are much simpler than corresponding typed reflective systems are given elsewhere in this paper.

A complete understanding of Figure 47 requires mastery of model theory for the untyped and typed lambda calculus, of type-inference algorithms for languages like ML, and of type modeling mechanisms such as retracts, partial equivalence relations, and ideals [We2]. However, the goals of untyped and typed model theory and of type inference mappings from untyped into typed languages can be understood even by nonmathematical readers.

7.3. The Essence of Inheritance

What is the essence of inheritance? What is common to the different kinds of inheritance discussed in section 4.3? What is the difference between inheriting a superclass and simply invoking its operations? Inheritance is a mechanism for sharing and reusing behavior. It is distinguished from other behavior sharing mechanisms by delayed binding of self-reference so that superclasses may merge their identity with the subclasses that inherit them.

7.3.1. Delayed Binding of Self-Reference

Inherited attributes are more essentially part of the subclass or object that inherits them than attributes that are merely used or invoked, just as eye colors are more essentially part of people than the car they drive or the house in which they live. Merging of identity is realized by late

(execution time) binding of self-references to the object on whose behalf an operation is being executed. Figure 48 shows an object X belonging to a class M with a parent class P. Late binding of self-reference in P allows it to assume the identity of each modifying subclass M that inherits it while preserving its textual independence. The identity of both M and P are bound to the object X at the time that X requests M to execute a message on its behalf.

Execution-time binding of "self" at the time the object X responds to a message can be implemented by simple pointer initialization to the base class, in this case the class M. All self-references during the execution of this operation, no matter from which superclass, are interpreted as references to this pointer. This will cause all self-references in both P and M to start searching in M for the operation to be executed and searching in P if the operation is not found in M.

7.3.2. Function Generators and their Fixed Points

This simple implementation mechanism for the execution-time binding of self-reference has a nice mathematical model in terms of fixed-point theory. The use of fixed points in recursive functions is seen in the following definition of the factorial function:

*fact = fun(n). if n = 1 then 1 else n * fact(n-1)*

Following Cook [Co], we convert the right-hand side into a nonrecursive function by treating its self-reference as a bound variable whose value is supplied later.

*FACT = fun(self). fun(n). if n = 1 then 1 else n * self(n-1)*

FACT is called the generator of fact. Generators bind the free self-reference of a recursive function so that it becomes a parameter of the generator. FACT is a functional that determines a function from integers to integers when supplied with a value for its formal parameter self. When FACT is applied to fact the factorial function is obtained.

FACT(fact) = fact

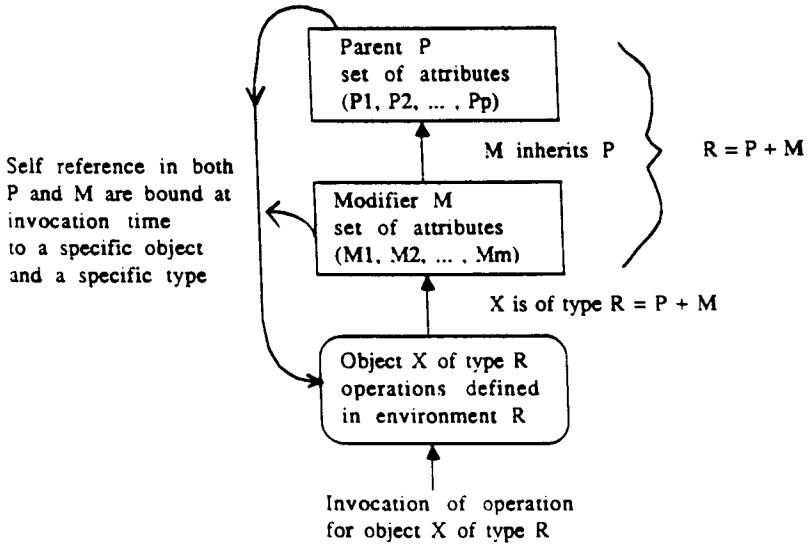

Figure 48: Execution-Time Binding of Self-References

A fixed point of a function f is any x such that f(x) = x. The factorial function is a fixed point of its generator FACT, and we may write "FIX(FACT) = fact" where FIX is the operation of taking the (least) fixed point of its argument.

The relation between the unfixed generator FACT and its fixed point is shown in Figure 49. The unfixed generator has a dangling reference, while the fixed point anchors the self-reference in the entity itself.

More generally, any recursive function of the form: "f = body with instances of f" can be converted to a nonrecursive generator: "F = fun(self). body-with-instances-of-self" with the property: "F(f) = f".

7.3.3. Generators for Classes

The above discussion of generators for recursive functions can be generalized to recursive classes. Let P be a recursive class containing occurrences of self as a free variable. By analogy with recursive functions, the generator GENP of P can be defined as "GENP = fun(self). P".

Just as in the recursive-function case, P is a fixed point of its generator GENP:

GENP(P) = P
FIX(GENP) = P

Fixing GENP binds its dangling self-reference to the class P (see Figure 50).

Generators represent recursive entities in which self-reference is a parameter. Taking the fixed point of a generator corresponds to binding its self-reference, thereby giving self-reference a particular denotation. GENP, like FACT, is a generator with a dangling self-reference. By talking the fixed point of GENP we anchor the self-reference in GENP itself.

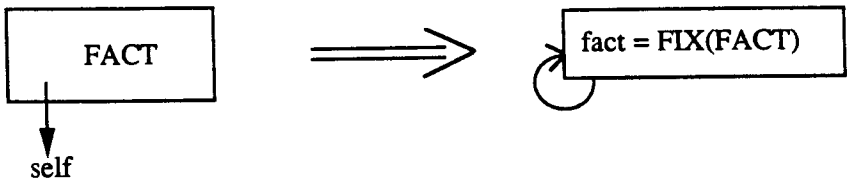

Figure 49: Function Generators and their Fixed Points

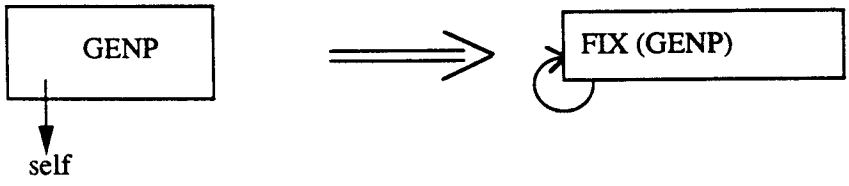

Figure 50: Class Generators and their Fixed Points

7.3.4. Inheritance as Generator Composition

Inheritance delays taking the fixed point of class generators to allow composition with subclasses. Let M be a class that inherits P and let "GENM = fun(self). M" be the generator of M. Inheritance of P by M can be defined in terms of the composition of the generators of P and M:

GENPM = COMPOSE(GENP, GENM) = fun(self). compose(GENP(self), GENM(self))
Inherits(P, M) = FIX(GENPM)

GENPM composes GENP(self) and GENM(self), and then immediately abstracts the common self-reference. This has the effect of identifying self-reference in the two generators and creating a new generator parameterized by the common self-reference. From a lambda calculus point of view this is quite a complex operation.

GENPM has two kinds of clients: subclasses that may inherit GENPM by further generator composition and objects that may FIX the generator in order to use its methods. The fixed point FIX(GENPM) determines the class "P inherited by M". It binds self-references of both M and P, so that an object of subclass M can access the methods of M and P during execution. The relation between GENPM and its fixed point is shown in Figure 51.

We can now express the difference between inheriting and using a class. When two classes use each other we take their fixed points before composing them, so that their composition is:

compose(FIX(GENP), FIX(GENM))

Composition does not commute with the taking of fixed points, so we have the inequality:

FIX(COMPOSE(GENP, GENM)) ≠ compose(FIX(GENP), FIX(GENM))

The left-hand side models inheritance of P by M, while the right-hand side models P and M symmetrically using each other as mutually recursive classes or functions.

This model of inheritance breaks new mathematical ground in treating generators as first-class objects on which operations such as composition can be performed. It relates the formal notion of taking the fixed point and the informal notion of fixing the recursively-defined identity.

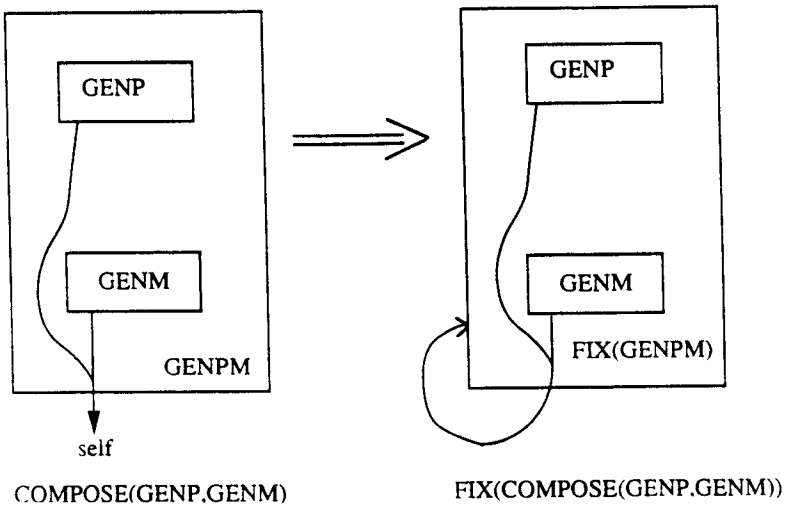

Figure 51: Fixed Points of Composite Generators

The fact that fixed points can be simply implemented by assigning a value to a pointer variable connects the abstract mathematical theory with a concrete implementation technique.

The analysis of self-reference provides a bridge to the study of the inherent nature of *self* through reflection. Inheritance allows incremental construction of composite selves, while reflection provides a computational framework for comprehensive models of selfhood.

7.4. Reflection in Object-Oriented Systems

Human reflection means thinking about ones own ideas, actions, and experiences. By analogy, computational reflection is the ability of a computational system to represent, control, modify, and understand its own behavior. Reflective systems facilitate introspective tasks such as debugging, monitoring, compilation, and execution. They support system evolution and self-reorganization. They provide an extra dimension of abstraction (*meta-abstraction*) that complements *data abstraction* and *super-abstraction*. In object-based systems meta-abstraction is captured by metaobjects, data abstraction by classes, and super-abstraction by inheritance.

Meta-abstraction is not inherently self-referential. We can represent, control, and model a system by a metadescription distinct from the system itself. Operating systems and compilers written in assembly language are meta-abstractions that allow us to control and use the resources of a computer for problem solving. In a reflective system the metadescription is a) a high-level description in the representation used for problem solving, and b) causally connected to the system in the sense that it dynamically controls as well as statically describes the application system. A reflective metadescription is part of the system being described and must describe its reflective abilities along with its abilities to perform substantive tasks.

The idea of computational reflection goes back to Lisp, which represents its programs as lists. The Lisp interpreter Apply (see Figure 52) can execute Lisp programs and therefore *understands* the structure of Lisp. When supplied with a program and its data (both represented by lists) it applies the program list to the data list to yield a result. APPLY can be modified to change its own execution behavior, for example by inserting monitoring or debugging features.

Universal Turing machines have a reflective structure very similar to that of the Lisp APPLY interpreter. A universal Turing machine can take a Turing machine representation and its data and compute the effect of applying the Turing machine to its data. It *understands* Turing-machine representations sufficiently to execute the programs they represent and can reflect on these programs in other ways, for example by printing traces of computations. Turing demonstrated certain limitations on reflection, such as the impossibility of proving that a Turing machine will halt. The Godel incompleteness result implies that reflective mathematical systems

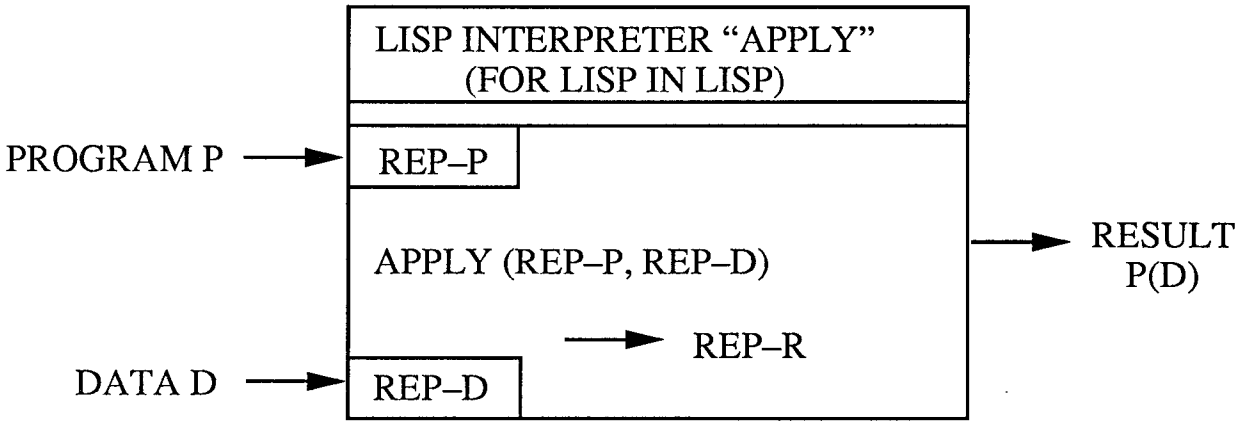

Figure 52: A Reflective Lisp Interpreter

can never completely understand themselves.

Compiler compilers are reflective models that understand compiling: they can behave like a specific compiler when supplied with a language specification and a program in that language. Reflective models of Lisp, Turing machines, and compilers have a remarkably similar structure, reflecting that reflection has a robust and simple computational model.

A reflective system has three components (M, A, D) which we can call the meta, agent, and data components. The metacomponent M is a metadescription of the behavior of agents A operating on data D. Reflective systems further require that M is causally connected to A and D in the sense that it is actually used in causing behavior. The key design task is choosing the representation of agents so they can be uniformly and naturally manipulated by the metacomponent. We represent Lisp programs by lists, Turing machine programs by tapes, and language definitions by tables. Once the representation has been chosen, the metacomponent can be defined in a straightforward manner. Viewing agents as data is referred to as *reification*, since it allows agents to be treated as objects (first-class values).

The metacomponent may be viewed as a *second-order abstraction* (in the sense of second-order functions) since it operates on agents which are (representations of) abstractions. Conversely, second-order functions are reflective, since they can act upon and understand the functions on which they operate. The ideas of being reflective, being second-order, and treating agents as first-class values are related.

Reflection is an anthropomorphic term describable by anthropomorphic notions like "self knowledge", "understanding", "behavior", etc. The term object is itself anthropomorphic so that the anthropomorphic analogies suggested by reflection fit in well with the object-oriented paradigm. Anthropomorphism models computational ideas by human analogies just as object-oriented programming models the real world by computational analogies. We make no apologies for anthropomorphism, since modeling is a respectable and even central scientific technique. In this respect we disagree with Dijkstra [Di].

Reflective object-based systems reflect on multiple agents (classes or objects) rather than just a single object. A metaobject is associated with each class or object to represent its structure and behavior. Metaobjects have information about the implementation and interpretation of classes and objects, about how it is printed, and about how new objects are created. Metaobject protocols can provide a succinct specification of the execution semantics of classes and objects. The use of metaobjects for reflection in sequential languages is discussed in [Ma] and [Fe].

Metaobjects for object-based concurrent systems must represent not only an object's methods and state, but also its message queues and evaluation method:

metaobject
  instance variables
    object state, object methods, queue, evaluator, mode
  methods
    model object behavior by acting on instance variables

The methods of the metaobject model the arrival, scheduling, and execution of messages. The modeling of queues and messages makes the reflective computation more complex than for sequential systems but still manageable. Once the basic reflective architecture has been designed, various kinds of reflective computations that modify system behavior can be specified. Examples of monitoring, dynamic object modification, and time management are given in [YW].

Object-oriented reflective systems allow each class to have its own reflective discipline for interpretation, message scheduling, debugging etc. Default behavior can be specified in a metametaobject that reflects on metaobjects and defines standard behavior when none is specified

in the metaobject. Object-based systems reflect at two levels, through metaobjects which reflect on individual classes and through a metametaobject that reflects on metaobjects and defines default behavior. Finer granularity reflection at the level of individual objects is also possible.

Reflective object-based systems provide a model for loosely-coupled multiparadigm environments, where each class is potentially a paradigm of computation and paradigms interact by message passing. For example, implementing window managers by classes, protocols for window management by metaobjects, and default window management by a metametaobject provides an implementation framework for multiparadigm, multiperson cooperative computing.

Reflective complexity provides a measure of the inherent complexity of a system. Lisp, universal Turing machines, and the lambda calculus have exceptionally low reflective complexity. As a language becomes higher-level its greater expressiveness should allow the reflective model of its high-level primitives to be succinctly expressed. But in practice, high-level primitives rarely have the reflective ability to model themselves. High-level languages therefore generally have greater reflective complexity, although well designed languages with robust primitives have much lower reflective complexity than poorly designed languages.

It is no accident that the simple reflective systems associated with Lisp and Turing machines are untyped. Typing destroys computational purity by introducing extraneous type compatibility requirements that cannot be expressed in the computational formalism of the untyped system. Reflection on both types and computation requires the reflective system to handle two very different computational mechanisms and the interaction between them. The execution-time safety of typed systems is realized at the expense of reflective simplicity.

Reflective systems facilitate both practical computation and conceptual understanding of the systems on which they reflect. They promote system understanding by both computers and humans. Metacomponents specify the operational semantics of agents. Since the difficulty of specifying operational semantics is correlated with the inherent conceptual difficulty of human understanding, reflective complexity provides a measure of conceptual complexity. The much greater reflective complexity of typed systems over untyped systems provides a measure of the immense conceptual costs of introducing types.

8. What Comes After Object-Oriented Programming?

The history of programming languages can be characterized as a progression from assembly languages in the 1950s to procedure-oriented languages in the 1960s and 1970s to object-oriented languages in the 1980s.

assembly languages (ALs)
→ *procedure-oriented languages (POLs)*
→ *object-oriented languages (OOLs)*

Object-oriented programming has become the buzzword of the 1980s, rivaling the fashionability of structured programming in the 1970s. It provides high-level structure at the level of objects, classes, and class hierarchies, complementing structured programming techniques for microstructure at the level of statements and expressions.

What will be the software engineering paradigm of the 1990s, and what buzzword, if any, will displace object-oriented programming?

Figure 53 views object-oriented programming as the ultimate buzzword. According to this view, object-oriented programming is the culmination of an evolutionary process, there is no need for further evolution, and we have reached the end of history.

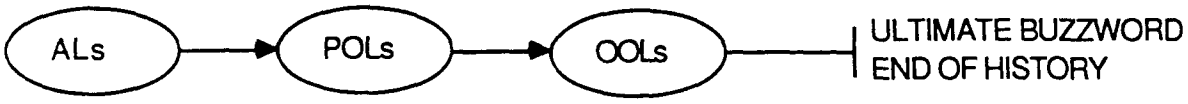

Figure 53: The Ultimate Buzzword

Figure 54 suggests that concurrency will require a return to assembly language to handle concurrent hardware and supercomputer architectures. This view may appear frivolous or even cynical, but the popularity of C as a system programming language and the primitive nature of supercomputer environments lend credence to this view. Actor languages (which happen to fit the acronym AL) are another example of low-level concurrent languages. The flexibility of languages like Lisp and efficiency of languages like C has always been seductive. Flexibility and efficiency could replace structure and modularity as the dominant requirements of concurrent software engineering. The tension between the flexibility of rapid prototyping and the structure of strongly-typed modular programming will continue to be a debating point in the future as it has been in the past. Ideally it would be nice to have ones cake and eat it too, by combining flexibility during design, efficiency during execution, and structure during system evolution.

Figure 55 lists a variety of buzzwords of the late 1980s, including some which have already been discussed. We briefly review high-performance computing, computational science, and megaprogramming.

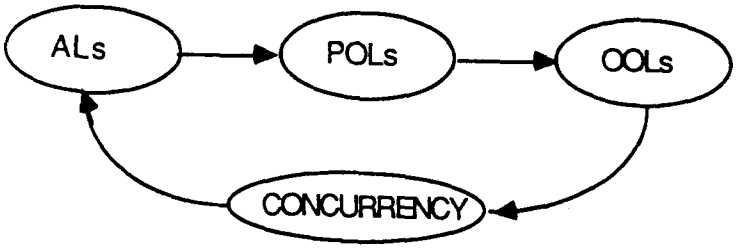

Figure 54: Regression to Assembly Language

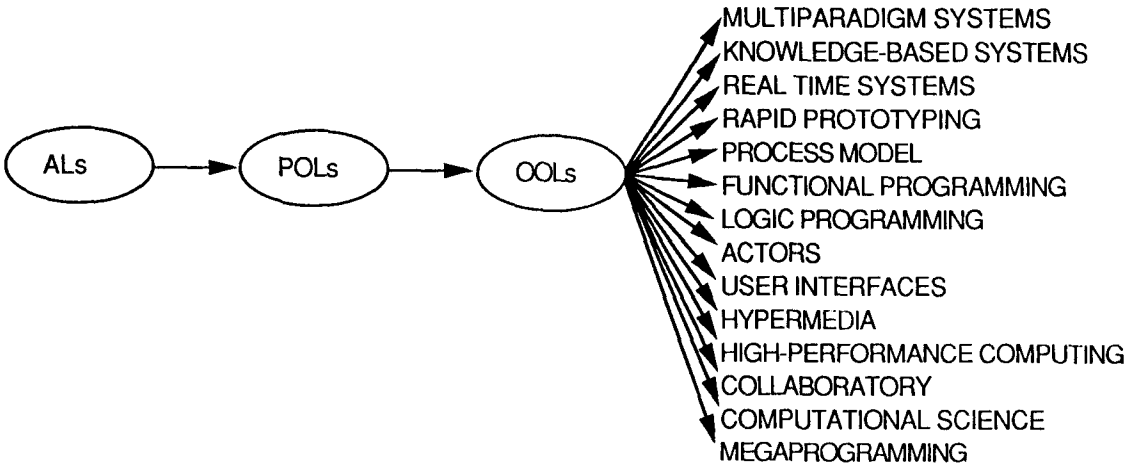

Figure 55: Scenarios for the Future

High-performance computing (HPC) is a priority of the President's Office of Science and Technology Policy (OSTP), which has proposed a $2 billion program [HPC] for hardware and software research on high-performance computing. The HPC proposal carefully balances research in hardware, software, and human resources to advance computing technology and enhance US competitiveness. If approved, it will provide substantial funding for software research and development, and an unprecedented opportunity to match software productivity on high-performance computers to their hardware productivity. The development of object-based and object-oriented environments for high-performance computers and concurrent architectures presents a challenge for the 1990s.

Computational science is concerned with complex applications such as weather forecasting. The high-performance computing proposal lists twenty grand challenges of computational science for which supercomputers are required. Skeptics view computational science as an attempt by application programmers to channel federal research funds rightfully belonging to computer science away from fundamental research into application development. However, computational science has become a buzzword to be reckoned with in part because system concerns have been incestuously overemphasized at the expense of applications, thereby leaving application development to specialists with no computer science background.

Megaprogramming focuses on component-based software engineering for life-cycle management. By viewing programming as a process of design and composition of components, it aims to simplify program structure and gain large increases in programmer productivity. Object-oriented programming is a particular kind of megaprogramming with specific kinds of components and composition mechanisms. There are many aspects of object-based component technology that we have not discussed, such as domain-specific component design, module interconnection formalisms, interface management formalisms, exception and interrupt mechanisms, real-time constraints, trusted components, etc. DARPA is planning a coordinated research program to develop a comprehensive technology of software components.

9. Conclusion

In choosing among the above scenarios we immodestly suggest that object-orientation may in fact be a persistent and even fundamental buzzword. Description of a domain of discourse by the behavioral attributes of its entities is natural for any application domain. The distinction between encapsulated internal behavior and external communication through interfaces reflects that in any organization (a bank) or organism (the human body) between interlocking internal subcomponents and explicit interfaces to the external environment. The distinction between declarative description of behavior by classes and imperative creation of instances for particular computations is also natural. Inheritance is a somewhat more specialized notion that facilitates reuse, composition, and modification of behavior specified by classes. The representation of classes by objects allows behavior to be managed and manipulated as though it were data, providing an alternative to inheritance for radical computation on behavior representations.

The object-oriented paradigm supports multiple but coordinated paradigms of thinking and problem solving. It nicely balances the state-transition, communication, and classification paradigms, combining imperative computing within objects and message passing between objects with declarative specification of object behavior. It is "closed" under reflection in the sense that metaclasses and metametaclasses are themselves objects, and facilitates independent reflective interpretation protocols for distinct objects and classes. It facilitates the harmonious integration of data abstraction, super-abstraction, and meta-abstraction in a comprehensive computing framework:

data abstraction → *encapsulation/communication* → *analysis*
super-abstraction → *composition/enhancement* → *synthesis*
meta-abstraction → *interpretation/epistemology* → *control*

Its universality as a robust representation, modeling, and abstraction technique suggests that the object-oriented paradigm is conceptually and computationally fundamental.

Object-oriented systems support the interaction of loosely coupled, distributed, concurrently executing agents. Individual agents can be modeled by automata, but the mathematics of asynchronously interacting agents is not as well developed as that for atomic sequentially executing agents. If agenthood is to be closed under composition, so that collections of asynchronously cooperating agents are modeled as agents, the notion of an agent and of observational equivalence between agents becomes more complex. Note that the modeling of societies of agents as agents is inherently difficult and is the subject of research in disciplines like sociology, economics, and cognitive science.

Object-oriented systems generally have global types and classes: classes of an object-oriented library generally do not discriminate among different kinds of clients. Programming in the very large (megaprogramming) may violate this assumption since it is concerned with systems developed by different organizations having different conceptual frameworks and addressing different application domains. Object-oriented systems provide a starting point for modeling the interaction of heterogeneous components but must be extended to support heterogeneity of data representations and type systems.

Object-oriented programming is more specific and comprehensive in its prescription for problem solving than structured programming. Structured programming is concerned with "structure" in general, while object-oriented programming focuses on a specific form of structure: that associated with objects.

The time has come to challenge those who assert that "everyone is talking about object-oriented programming but no one knows what it is". This is no longer true since there is a growing body of literature, including this paper and many of the cited references, that defines what it is and how it may be used in problem solving. Object-oriented programming is not a transitory buzzword but a fundamental modeling framework that has something fundamental to say about computation and model building, while lying on the critical path to greater software productivity.

10. References

[Ag] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press 1986.

[AS] G. R. Andrews and F. B. Schneider, Concepts and Notations for Concurrent Programming, *Computing Surveys* 15 (1), 1983.

[ASS] H. Abelson, G. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press 1985.

[AWY] G. Agha, P. Wegner, and A. Yonezawa, Proceedings of Workshop on Object-Based Concurrent Programming, *Sigplan Notices*, April 1989.

[Ba] H. P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North Holland, 1985.

[Ban] F. Bancilhon et al, Object-Oriented Database Manifesto, *Proc Sigmod 1988.*

[BG] G. Berry and G. Gonthier, The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation, INRIA Research Report #842, 1988.

[BH] H. Barendregt and K. Hemerik, Types in Lambda Calculi and Programming Languages, University of Nijmegen TR 90-4, February 1990.

[BMM] K. Bruce, A. Meyer, and J. Mitchell, The Polymorphic, Typed, Second-Order Lambda Calculus, *Information and Computation*, Spring 1990.

[Bo] G. Booch, *Object-Oriented Design with Applications*, Benjamin Cummings, 1990.

[BS] B. Boehm and W. Scherlis, Briefing viewgraphs on megaprogramming, DARPA Workshop, June 1990.

[BST] H. Bal, J. Steiner, A. Tanenbaum, Programming Languages for Distributed Computing Systems, *Computing Surveys*, September 1989.

[BW] K. Bruce and P. Wegner, Subtype Polymorphism and Inheritance in Object-Oriented Languages, *Sigplan Notices*, October 1986.

[CK] C. C. Chang and H. J. Keisler, *Model Theory*, North Holland, 1978.

[Co] W. Cook, *The Denotational Semantics of Inheritance*, PhD Thesis, Brown University, 1989.

[Coi] P. Cointe, Metaclasses are First Class: The ObjVlisp Model, OOPSLA 87.

[CW] L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys*, December 1985.

[Di] E. Dijkstra, The Cruelty of Teaching Computer Science, *CACM*, December 1989.

[DMN] O. J. Dahl, B. Myrhaag, and K. Nygaard, *Simula 67 Common Base Language*, Norwegian Computing Center, 1968, 1970, 1972, 1984.

[DoD] *Reference Manual for Ada Programming Language*, US Dept of Defense, 1983.

[Fe] J. Ferber, Computational Reflection in Class-Based Object-Oriented Languages, OOPSLA 1989.

[GM] J. Goguen and J. Meseguer, Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics, in [SW].

[GMc] N. Gehani and A. D. McGettrick, *Concurrent Programming* (Collection of Readings), Addison-Wesley, 1988.

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

[GT] G. Graunke and S. Thakkar, Algorithms for Shared Memory Multiprocessors, *IEEE Computer*, June 1990.

[Ha] P. B. Hansen, Distributed Processes, A Concurrent Programming Concept, *CACM*, 1978.

[Har] David Harel, Statecharts: A Visual Formalism for Complex Systems, *Science of Computation*, 1987.

[He] M.ennessy, *Algebraci Theory of Processes*, MIT Press, 1988.

[Ho1] C. A. R. Hoare, Proof of Correctness of Data Representations, *Acta Informatica*, 1972.

[Ho2] C. A. R. Hoare, Monitors, An Operating System Structuring Concept, *CACM*, October 1974.

[Ho3] C. A. R. Hoare, Communicating Sequential Processes, *CACM*, August 1978.

[HPC] *The Federal High-Performance Computing Program*, Executive Office of the President, Office of Science and Technology Policy, September 8, 1989.

[Hu] P. Hudak, The Conception, Evolution, and Application of Functional Programming Languages, *Computing Surveys*, September 1989.

[KC] S. N. Koshafian and G. P. Copeland, Object Identity, *OOPSLA 86*, reprinted in S. Zdonik and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufman 1989.

[Ko] R. Kowalski, Algorithm = Logic + Control, *CACM*, July 1979.

[LBGSW] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl, Communication in the Mercury System, *Proc 21st Hawaii Conference*, January 1988.

[Lie] H. Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Languages, *OOPSLA 1986*.

[LP] H. Lewis and C. Papadimitiou, *Elements of the Theory of Computation*, Prentice Hall, 1981.

[LS] B. Liskov and R. Scheifler, Guardians and Actions, Linguistic Support for Robust

Distributed Programs, *TOPLAS* 5(3), 1983.

[LS1] B. Liskov and L. Shrira, Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems, *SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[LSAS] B. H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, Abstraction Mechanisms in CLU, *CACM*, August 1977.

[LTP] W. R. Lalonde, D. A. Thomas, and J. R. Pugh, An Exemplar-Based Smalltalk, *OOPSLA 1986*.

[KMM] P. Kanellakis, H. G. Mairson, and J. C. Mitchell, Unification and ML Type Reconstruction, to be published in Festschrift for J. A. Robinson, edited by G. Plotkin and J. Lassez. MIT Press 1990.

[Ku] T. Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, 1961.

[Ma] P. Maes, Concepts and Experiments in Computational Reflection, *OOPSLA 1987*.

[McI] M. D. McIlroy, First International Conference on Software Engineering, 1969.

[Me] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall International, 1988.

[Mey] A. Meyer, What is a Model of the Lambda Calculus? *Information and Control*, 1982.

[Mi] R. Milner, *Calculus of Communicating Systems*, Springer Verlag 1980.

[Mi1] R. Milner, *Communication and Concurrency*, Prentice Hall International, 1989.

[Min] M. Minsky, *The Society of Mind*, Simon and Schuster, 1985.

[Mo] D. A. Moon, The Common Lisp Object-Oriented Programming Language, in *Object-Oriented Concepts, Databases, and Applications*, edited by W. Kim and F. Lochovsky, ACM Press/Addison-Wesley, 1989.

[Ni] O. Nierstrasz, A Survey of Object-Oriented Concepts, in *Object-Oriented Concepts, Databases, and Applications*, edited by W. Kim and F. Lochovsky, ACM Press/Addison-Wesley, 1989.

[Pa] D. Parnas, A Technique for Software Specification with Examples, *CACM*, May 1972

[Sc] D. Scott, Data Types as Lattices, *SIAM Journal of Computing*, September 1976.

[Sch] C. Schaffert et al, An Introduction to Trellis Owl, *OOPSLA 86*.

[SLU] L. A. Stein, H. Lieberman, and D. Ungar, A Shared View of Sharing, The Treaty of Orlando, in Object-Oriented Concepts, Databases, and Applications, Eds Kim and Lochovsky, ACM Press/Addison Wesley, 1989.

[Sn] A. Snyder, Inheritance and the Development of Encapsulated Software Systems, in [SW],

MIT Press, 1987.

[SW] B. Shriver and P. Wegner, editors, *Research Durections in Object-Oriented Programming*, MIT Press 1987.

[SY] R. Strom and S. Yemini, NIL, an Integrated Language and System for Distributed Programming, *Sigplan Notices*, June 1983.

[US] D. Ungar and R. B. Smith, Self: The Power of Simplicity, *OOPSLA 1987.*

[We1] P. Wegner, Capital-Intensive Software Technology, *IEEE Software*, July 1984.

[We2] P. Wegner, The Object-Oriented Classification Paradigm, in [SW], MIT Press 1987.

[We3] P. Wegner, Dimensions of Object-Based Language Design, *OOPSLA 1987.*

[We4] P. Wegner, Editor, Special Issue of *Computing Surveys* on Programming Language Paradigms, December 1989.

[Wu] W. Wulf, Towards a National Collaboratory, Report of a Workshop in March 1989, National Science Foundation Report, October 1989.

[WW] R. Wirfs-Brock and B. Wilkerson, Object-Oriented Design, A Responsibility-Driven Approach, *OOPSLA 89.*

[WW1] G. Wiederhold and P. Wegner, Towards Megaprogramming, Unpublished working document, July 1990.

[WWG] M. Wilkes, D. Wheeler, and S. Gill, *The Preparation of Programs for a Digital Computer*, Addison-Wesley 1951, revised edition, 1957.

[WZ] P. Wegner and S. Zdonik, Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like, *ECOOP 1988*, Lecture Notes in Computer Science, Springer Verlag 1988.

[Yo] K. Yoshida, A'UM: A Stream-Based Concurrent Object-Oriented Programming Language, PhD Thesis, Keio University, March 1990.

[Yo1] A. Yonezawa, *ABCL: An Object-Oriented Concurrent System*, MIT Press 1990.

[YT] A. Yonezawa and M. Tokoro, *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[YW] A. Yonezawa and T. Watanabe, Reflection in an Object-Oriented Concurrent Language, *OOPSLA 88.*

[Za] P. Zave, A Compositional Approach to Multiparadigm Programming, *IEEE Software*, September 1989.