# Towards a Framework
# For Comparing Object-Oriented Systems

*Shyhtsun F. Wu* *

Columbia University

Department of Computer Science

New York, NY 10027

July 9, 1989

MS Thesis

(Technical Report CUCS-438-89)



Committee: *Gail E. Kaiser* and *Soumitra Sengupta*

**Abstract**

Choosing a suitable *object-oriented system* for a particular application domain is very hard, at least in part because the object-oriented research community has not yet converged on what is meant by "object-oriented system". In order to solve this selection problem, a comparison framework is proposed to evaluate a range of object-oriented systems. We then use this framework to consider three different systems, MELD, MARVEL, and VBASE, which represent several quite different design choices for object-oriented systems. We also show the advantages as well as disadvantages about such a framework, and discuss future extensions.

*My advisor, Prof. Gail E. Kaiser, is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun, and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

# Contents

[80] J.F. Winkler. The Integration of Version Control into Programming Languages. In *International Workshop on Advanced Programming Environments* (1986) 231-250.

[81] Y. Yokote, and M. Tokoro. Concurrent Programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming* (1987) 129-158.

[82] M. Young, R. Taylor, and D. Troup. Software Environment Architectures and User Interface Facilities. *IEEE Trans. on Software Engineering*, Vol. 14, No. 6., June (1988) 697-708.

[83] S.B. Zdonik. Version Management in an Object-Oriented Database. In *International Workshop on Advanced Programming Environments* (1986) 405-422.

# List of Figures

# 1 Introduction

Although object-oriented systems clearly have an important role to play in the next generation of software engineering, two tough questions have been raised in the research community:

1. *What is an object-oriented system?* The difficulty of this question is the disagreement over the essential characteristics of object-oriented systems, for example, the problem of "delegation versus inheritance" [73].

2. *How to choose the right object-oriented system?* This question differs from the first one in that several conflicts might disappear after we know the application domain and the hardware environment. For example, we would like to use an object-oriented system with very good *extensibility* for developing experimental software environments [78]. For another example, if our underlying hardware architecture is purely shared-memory, then the performance of multiple inheritance and delegation will not be so different from each other [14].

Several efforts have tried to propose a common object-oriented computational model and provide a focus for the database and programming language communities. Most of these consider only one object-oriented system with some specific application domains [54], or only one or two characteristic features among several object-oriented systems [73]. The results of these works are still far from answering the first question, because the former approach is only applicable to those domains, and the latter one doesn't address a big picture of object-oriented systems, although they both might help greatly in moving us toward the final answer.

Since not many object-oriented systems are available, very little work has been done for the second question: *how to choose a suitable object-oriented system.* Usually, people will randomly choose one object-oriented system after hearing the buzzword "*object-oriented*" and reading some commercial advertisements. This is very dangerous for software development for the following reasons:

- The features in the language might not be sufficient to support this application domain, so most software engineers will either find ad hoc solutions, which may detract from the ideal object-oriented development methodology, or just simply try another system;

| *reasons* |
| --- |
| conceptual naturalness |
| encapsulation, reusability, and extensibility |
| manipulating parallel and distributed environment |
| powerful controllability of versions |

Figure 1: Why object-oriented will reduce software cost

- In the early stages of software development, the requirements of the target system are hard to understand, so a prototype system will be useful to conquer this problem. If the object-oriented system we choose doesn't support *reusability* and *extensibility* very well, it may increase the software cost immensely;
- The language features could be changed after future convergence on question 1, and then many of existing modules may have to be rewritten. But if we have good interface control, we can reduce the impact of such changes.

The reasons listed above are far from complete, but make clear that a comparison method with a *formal framework* is necessary for answering the second question. In addition, the results of using this framework may help to answer question 1 (from a scruffy approach).

In this thesis, I proposed an experimental framework for comparing object-oriented systems. I derived this framework from my experience using three different object-oriented systems, MELD, MARVEL, and VBASE. I then explain why the selected features in my framework are important for all object-oriented systems. The correlations among these features are also described. Then, I show the result of comparing these three object-oriented systems, which are quite different in some features that an object-oriented system might have. Finally, future work regarding the comparison framework and the three object-oriented systems is also discussed.

## 1.1 Why Object-Oriented?

Using object-oriented systems can reduce the cost of software development in four different ways (Figure 1). First, for both programmers and system designers (or project managers), it will be much easier to use object-oriented software than relational software because of its conceptual naturalness for treating complex and irregular objects [42]. To support such conceptual naturalness, most object-oriented systems have *classes* or *abstract data types.*

Some also support system-defined and user-defined exception handling objects, which can be used to tolerate software faults [11]. Second, following classical software engineering principles, object-oriented systems usually support encapsulation, reusability, and extensibility [53]. This is done by having definition architecture, inheritance relation, interface control, and modularization. Third, object-oriented systems are suitable for operation in a parallel and/or distributed environment, because each object can be treated as a computation unit with local data, and thus multiple granularities of parallelism can be developed among the units [38]. For this purpose, some object-oriented systems support various kinds of inter-object communication to increase the concurrency among objects. Some other systems develop the concurrency inside an object, while yet others have transaction mechanisms to support more general concurrency control. Finally, some object-oriented systems have proposed powerful controllability of versions [83]. This includes *persistent object base systems*, *database version control*, and *software version control*. Most of these works were motivated by the shortcomings of relational database systems or tree-structured file systems.

## 1.2 Motivation

Although there are four good reasons to use object-oriented systems, it doesn't mean we will achieve all the advantages when using a particular system. Most systems support a partial set of the nice features mentioned in the previous section. This is due in part to the existence of several conflicts in language design. Examples includes the conflict between multiple inheritance and concurrency, the conflict between encapsulation and inheritance, and the class versus prototype argument.

A lot of work has been done to resolve each of those conflicts individually. The most typical solution is one *new language feature* replacing the old one and solving one conflict. For example, *behavior abstraction* [33] has been introduced to solve the conflict between inheritance and concurrency for *actor base systems* [1]. One example of behavior abstraction in Kafura's paper is illustrated in Figure 2. Suppose we want to add a new method *get_rear()* to the class bounded_buffer to produce a new class called *extended_buffer*. Using behavior abstraction, we can easily achieve this by defining the new behavior for *extended_buffer* (Figure 3).

But unfortunately, this nice solution doesn't work well when the target system is open, i.e., an actor can modify its own behavior at runtime. Behavior abstraction requires strong

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

Figure 2: Bounded Buffer

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

Figure 3: Extended Buffer

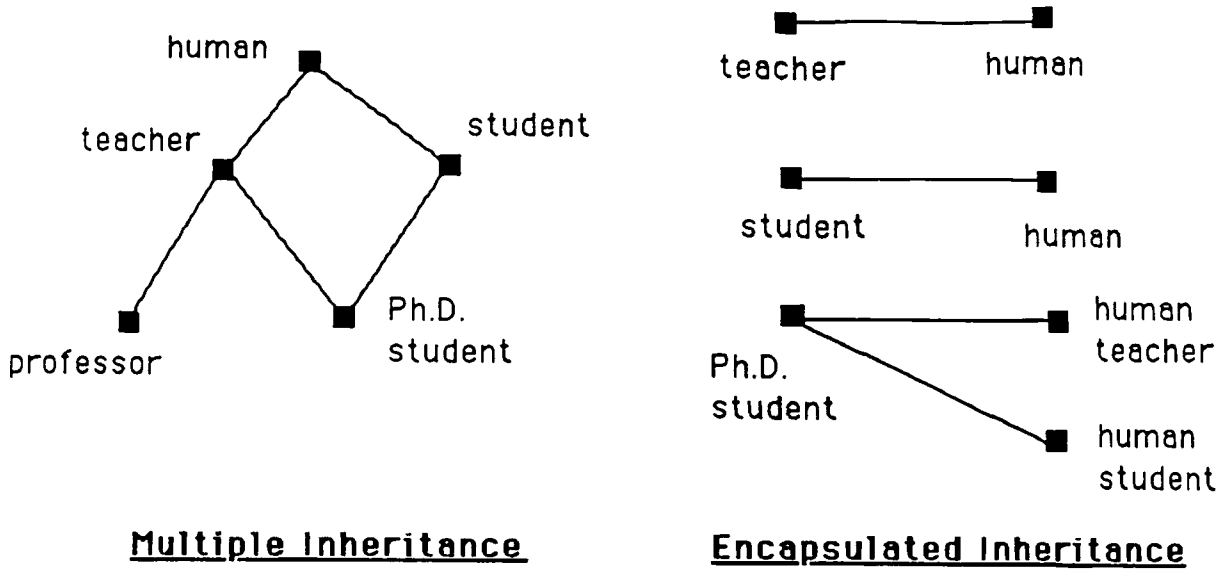


Figure 4: Inheritance and Encapsulated Inheritance

type checking, which restricts the flexibility of the interrelations among objects.

Another example is the conflict between inheritance and encapsulation. Snyder [67] proposed "*encapsulated inheritance*", which provides a second interface for inheritance, to increase the reusability of objects. The problem here is we will loss the conceptual naturalness of objects. In real world, inheritance hierarchy usually can be described as a partial ordering graph, but encapsulated inheritance just break this graph into a set of ordering pairs (Figure 4). Furthermore, it might also bring up conceptual inconsistency (Figure 5).

There are three problems with this general approach. First, the introduction of a new solution often compromises some nice aspects of the old one. Second, whether or not new conflicts are introduced by the new features is often unknown until there has been significant experience using the revised language. Finally, in most practical cases, we have to consider all the language features to choose a suitable object-oriented system from those available, while those loosely-coupled results haven't shown us a big and clear picture.

The goal of this thesis is to build up a comparison framework for a range of object-oriented systems. Instead of finding out *how to support the important features of object-*

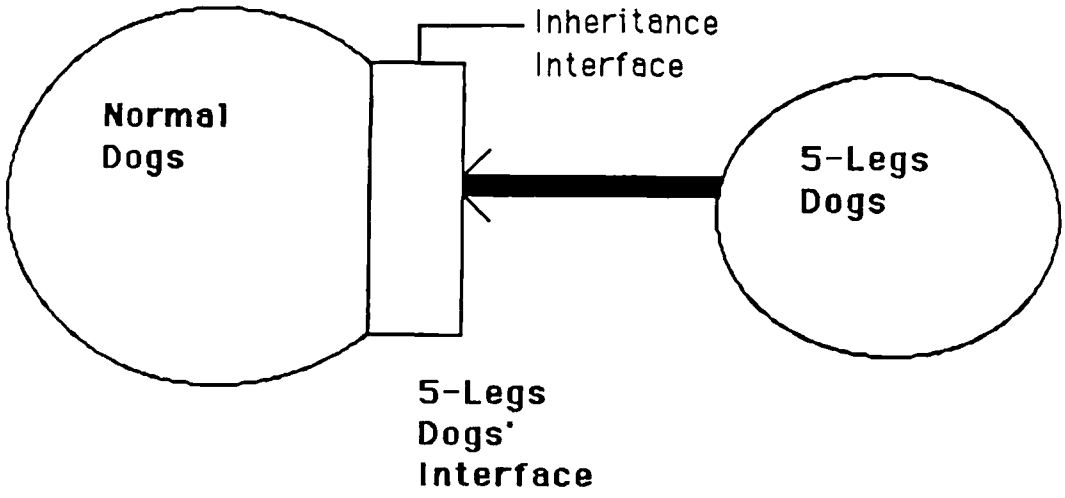


Figure 5: Conceptual Inconsistency in Encapsulated Inheritance

*oriented systems individually*, we would like to consider *"what are the important features and how do they interrelate"* from the perspective of application. Then, we can merge the individual results with this framework, and get a theoretical as well as practical method for evaluating object-oriented systems.

## 2 Background

In this section, we review a set of features that an object-oriented system might have. We classify the features into four categories, *definitional architecture, control architecture, data architecture*, and *concurrency control:*

**Definitional Architecture:** Definitional architecture mainly concerns how to define objects and the relations among objects. It includes class definition, inheritance relation, query language, and various relations.

**Control Architecture:** In object-oriented systems, objects can be treated as computational units with external interfaces and internal algorithms manipulating their local data. The questions are how to manipulate the local data and how to communicate with other objects efficiently and safely.

**Data Architecture:** Although most object-oriented systems are file-less, we still must consider the ways that objects are stored in memory and how to implement persistent

objects.

**Concurrency Control:** Several granularities of parallelism has been incorporated into object-oriented systems. The purpose of concurrency control is to make sure that parallel tasks can be performed safely, in the sense that data is accessed consistently.

In the following subsections, these winds of language features will be explored in detail.

## 2.1 Definition Architecture

Every object-oriented system offers users a set of techniques to define objects and the relations among objects. The definitions might be changed occasionally, or reused with a few differences. So, selecting a set of definitional techniques correctly is a primary concern for object-oriented systems. In this section, we consider four important features for definition: *data abstraction (objects), abstract data types (classes), knowledge sharing, query languages,* and *various relations.*

### 2.1.1 Data Abstraction and Abstract Data Types

*Data abstraction* [76] is fundamental to object-oriented systems. It provides *encapsulation* for hiding the internal structure and implementation of an object. The hidden information can be accessed only through some specific external interfaces. Therefore, the object can be reused or reaccessed through these unchanged interfaces, even if the internal structure has been modified or we find a better way to reimplement the object.

In order to bring those objects into existence, basically two ways – prototyping and classification – have been proposed. Most object-oriented systems support *abstract data types* or *classes* [17], which allow users to classify the abstracted data. Based on the class definition, a new object can be created with the internal structure, the implementation, and most essentially the external interfaces. Defined by the class template, another way is to create objects by existing objects called *prototypes* [44]. Because no concept of set or group of objects exists and every object actually stands alone, we lose some power of abstraction. But, the combination of *prototype* and *delegation* does provide a better mechanism for some special situations. We will discuss this issue in section 3.2.

Another important problem is how to define the external interfaces. One scheme offers only one interface, for other objects (clients) to access its local data only indirectly through methods [76]. Here subclasses see all internal structure. Another scheme not only offers the interface for clients to access data, but also supports an explicit interface for inheriting

classes to access its internal structure [67]. Yet another system supports multiple interfaces for different usages [28].

### 2.1.2 Knowledge Sharing

*Knowledge sharing* [14] is one of the most important features in object-oriented languages supporting the reuse of object descriptions and the refinement of object definitions. The first advantage of knowledge sharing is the increasing in *modularity* rather than redefining the whole object or class. Second, the introduction of classification with inheritance results in a hierarchically structured knowledge model, making knowledge searching more efficient. In constract, if we just have a set of class definitions, in most cases, we have to go through all the classes to find what we really need. Finally, knowledge sharing might imply sharing code efficiently.

In different object-oriented systems knowledge sharing is implemented in different ways, for example, delegation, single inheritance, multiple inheritance, copy, and recipe-query. Basically, there are two reasons to have so many knowledge sharing methods:

**Flexibility:** Do we keep only one or more than one copy of the executable for methods? If we allow all the subclasses of a class to share the executable[1], then we save a lot of memory space, and it will be much easier to modify the internal structure of a class at runtime, because we keep only one copy of a method in one class hierarchy. In inheritance, the link between subclass and superclass is *hard-wired* after the compiling phase, but the introduction of meta-class[2] with one copy executable might make the link more flexible, which might be modified at runtime by the clients of the meta-class. In delegation, everything is *soft-wired* to provide a uniform communication protocol: the delegation to the *proxy* of an object is performed by message passing, and therefore, global type checking is not necessary.

**Efficiency:** For efficiency, we might want to keep several copies of the executable, if the objects are distributed in a network environment. But if the underlying hardware model is shared-memory, then the performance difference between keeping one copy and multiple copies is small.

[1] Sharing the executable means for one piece of code (a module or a procedure), we only keep one copy. Some object-oriented systems have a preprocessing step that duplicates those pieces into several copies for different classes.

[2] meta-class: class for class objects.

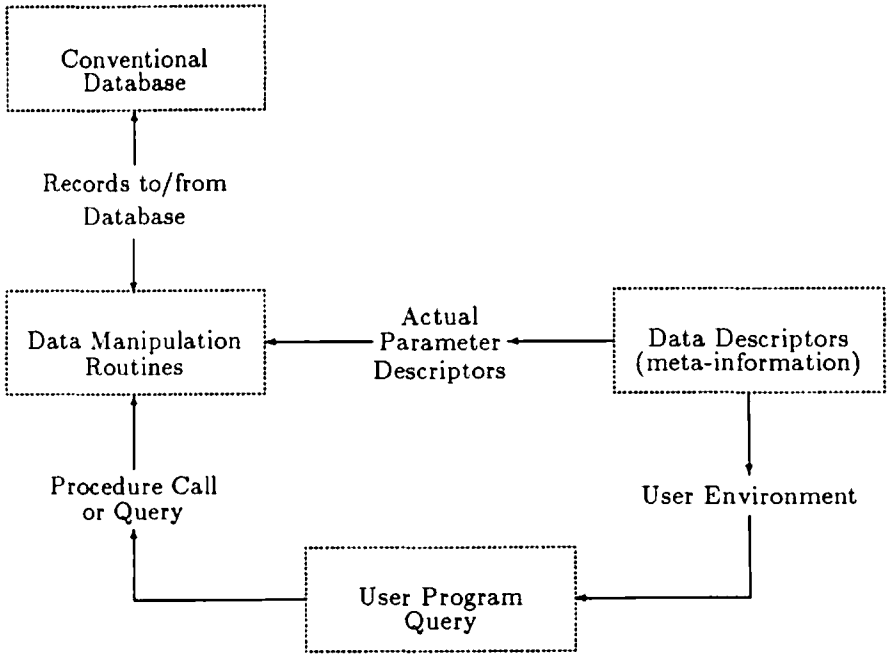


Figure 6: The conventional data base query model

### 2.1.3 Query Language and Query Processing

In order to avoid referring to objects through their object identities, predicate-based queries in an object-oriented database environment is necessary. Unfortunately, the conventional SQL-like query language is not enough to support object-oriented queries since an object-oriented data model has classes, nested classes, objects, nested objects, multiple inheritance and finally relations. It will also break the encapsulation of objects because the user must understand the internal *table* structure. In [6], a version of query languages for the *ORION* database system is derived from the fundamental differences in the semantics of queries in two models. From an application point of view, Garlan in his thesis [26] proposed a nice query language, *views*, to integrate software tools to access the database through a uniform interface.

In [7], both conventional and object-oriented query models are discussed (Figure 6) and (Figure 7).

### 2.1.4 Various Relations

Most object-oriented languages do not support *relations* directly. When relations are needed in some cases, they are usually implemented in an ad hoc fashion, which might detract

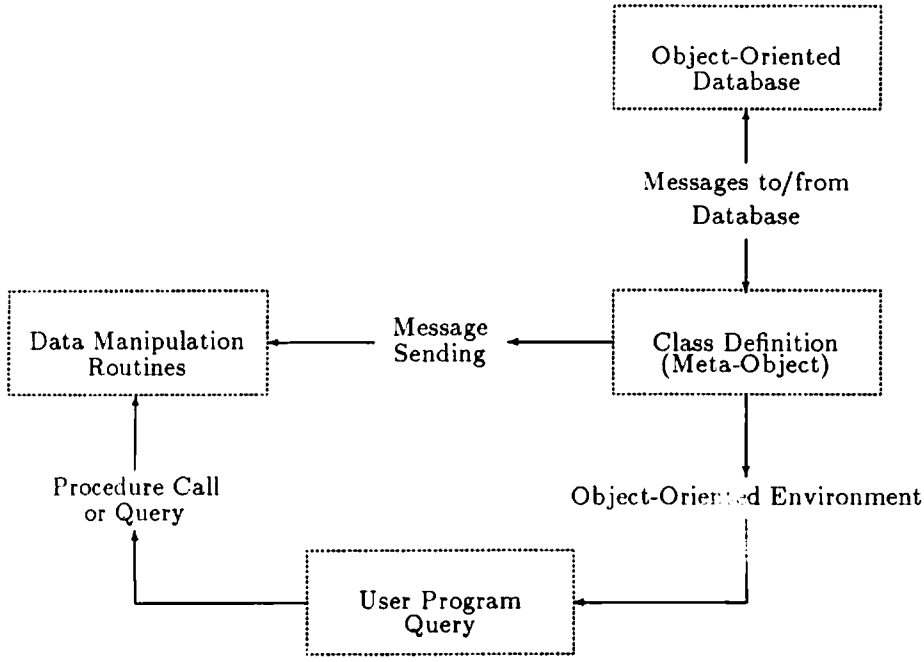


Figure 7: The object-oriented data base query model

from the abstraction of the relations. An *object-relation* model, which combines the object-oriented model with the entity-relationship model from database theory, has been proposed to be useful for designing and partitioning systems containing many interrelated objects [65]. In [12] [13] [34], *constraints* are used to maintain the consistency of the relations inside an object as well as among objects. VBASE[5] supports *inverse relations*, which can be used to easily maintain many kinds of complex relations in relational models.

## 2.2 Control Architecture

Although the hardware machine and physical communication network really do the work, the programming language used to control them represents a virtual machine which may be quite different. Traditional programming languages, such as FORTRAN or even assembly languages, represent the real hardware conceptually, so the users don't have to worry about anything else other than the architecture proposed by Von Neuman. For some modern programming languages, the programmers have to understand a different architecture, *control architecture*. For example, *data driven computation* is the control architecture for *dataflow languages*, and *production system* is the control architecture for *rule-base languages*.

Control architecture includes several issues:

- What is the language paradigm?
- How to control the program in a distributed environment?
- How does the conceptual computer handle exceptions?
- How to control the object interfaces?
- What is the input/output interface?

### 2.2.1 Language Paradigms

From the view point of programming language design, we can classify languages into several paradigms, for example, procedural language or declarative language, which represent different writing styles as well as two conceptual computational models.

Recently, constraint-based invocation (CBI), the integration of rule-based and object-oriented programming paradigms, has been proposed [74]. It is very useful in many cases, for example, to control nondeterministic execution in object-oriented systems, or to resolve the conflicts caused by multiple inheritance. One argument against CBI stated that because of the class-less construct, CBI may be too dynamic to control and we lose the information from type hierarchies.

### 2.2.2 Message Passing among Objects

As stated before, object-oriented computation presents a pseudo distributed computing environment, *i.e.*, the language should be able to represent the message passing[3] among distributed units, *objects*. Object-oriented distributed operating systems have been an important role in next generation operating systems, which extending conventional operating system (for example, UNIX) from function call to message passing. For example, NEXUS Distributed Operating System [71] is structured as a collection of objects with some system-defined types, and the main function of the NEXUS kernel is to support IOC (Inter-Object Communication). Other works include Choices [66], Clouds [18], and Mach [32].

[3] In real hardware, we always consider two different models, shared-memory architecture and pure-message-passing architecture. The conceptual object-oriented model should be message-passing only (the updating of the status of objects should be done through some specific communication channels), but the inheritance mechanism is actually a conceptually shared-memory model, because one class can share the knowledge of its ancestor classes. This is why encapsulation will conflict with inheritance, i.e., it is hard to encapsulate the inherited information.

### 2.2.3 Exception Handling

In traditional languages, exceptional handling is implemented by either system-defined or user-defined exceptional conditions. In object-based systems, we can also implement it in the same way. The issues are:

- How to describe the exceptional condition?
- What is the structure or hierarchy of system-defined exceptional condition?
- How to trigger these conditions?

For example, in Choices [66], the exception class hierarchy is built-in for both exception handling and hardware exceptions. User-defined exception classes inherit the system hierarchy.

### 2.2.4 Interface Control

With very little help, an experienced programmer can understand the interrelations among a small number of program modules. But in general, we are dealing with a huge number of objects as well as classes, which together constitute a very large software system. Therefore, describing and maintaining interfaces between modules in such a large system is very important.

Basically, we can describe the functionalities of interface control as follows:

**Elegant Definition:** Easily defining an easy-to-understand interface;

**Consistency Maintenance:** Type checking when linking two interfaces;

**Impact Limitation:** Prototyping systems will become easier.

There are several trade-offs among these concerns. For example, consistency maintenance implies information-rich interface while impact limitation implies information-poor interface [78].

### 2.2.5 I/O Interface Design

The user interface management for object-oriented systems has been addressed by several researchers [82] [56]. Three problems are currently considered as the most important:

**Integration with Software Tools:** We consider how users can easily interact with the programming environment we built;

**Browsing Interface for Persistent Object Base:** How users or programmers can understand the architecture of the object base, and manipulate the objects;

**Protocol between Human and Computer:** One idea is to treat the user as an object, then we can ask "will the users be satisfied by standard inter-object communication protocol."

## 2.3 Data Architecture

We have considered the conceptual model and definitional architecture, but sometimes for performance issues, it would be better if the user could understand the real hardware and explore the capability of the real hardware environment. We have several points concerning this:

- What kind of hardware are we using?
- How to store objects efficiently?
- How about persistence?

### 2.3.1 Matching with Hardware Architecture

Although object-oriented systems have many nice features for a software environment, the potential savings in programming effort have been curtailed by low performance in most conventional computer systems or expensive processor cost [72]. Many novel architectures have been proposed to support high performance object-oriented systems, for example, *SOAR*, *SWARD*, and *SLOOP* [46].

Some other works have considered how to efficiently implement object-oriented systems on general multiprocessor environments, which include MIMD machines, SIMD machines, Shared-Memory parallel processing machines, and distributed processors on a local area network. With these different hardware environments, we have different choices for the implementation and language features of object-oriented systems. In *SLOOP*, the notion of *virtual object space*, between the real hardware and the object-oriented computational model, is used to encapsulate the underlying hardware environment. *SLOOP* encapsulates the *virtual object space* in a type called *Domain*, which provides a set of operations for asynchronously creating and accessing objects, and for specifying groups of objects. This allows changing the *virtual object space* dynamically at run time.

### 2.3.2 Object Clustering

Object clustering is for improving the performance by exploiting the locality of the data contained in multiple objects. In conventional distributed systems, usually we improve the performance of *caching* by analyzing a sampling of the data and after some calculation, determine some parameters for the cache manager. In object-oriented systems, not only the analytical data is used, but also semantic information provided by programmers.

Another advantage of object clustering is that such clustering will make the task of changing definitions easier. This means, if we redefine an interface and want to propagate the effect, and we have clustered the objects that impact each other, then it is relatively easy to update the whole database [78].

*Dynamic grouping* [79] has been proposed to improve the performance by placing objects during the runtime. This work is built upon a virtual memory hierarchy. The *Emerald* system [9] has a similar scheme for objects, which are fully mobile and can be moved from node to node in the network, even during an invocation. For every create operation, VBASE [5] allows the invoker to specify a previously existing clustering object, and then the new object will be clustered in the same segment as the clustering object.

### 2.3.3 Persistent Objects

In a software development environment for a large project, large amounts of information might be involved. This data must live in an object base for a long time. Nestor [57] has pointed out the weaknesses of traditional file systems and conventional database systems in supporting two top level design goals for future software environments:

**Openness:** This means the ability to incorporate tools, methodologies, and technologies into the current environment as needs and opportunities arises. Usually, openness is achieved by sophisticated interface control mechanisms or supporting computational reflection.

**Integration:** We try to make all the components of an environment work together through a uniform interface, style of operation, and communication protocol.

## 2.4 Concurrency Control

Object-oriented computational model provides a natural setting for the construction of parallel programs. By conceiving a problem solution in terms of a collection of cooperating

objects, a programmer can partition a program into natural functional units, replicating these units according to the data parallelism inherent in the particular problem.

Unfortunately, in most cases, it is not easy to have such global knowledge about parallelism, instead, we have only partial knowledge. Thus, the lack of knowledge introduces the risk of developing incorrect concurrent events. Basically, there are two kinds of incorrectness: first, they can be executed concurrently but they should be synchronized under a set of constraints; second, they may not be synchronized at all because no such constraints exists.

For case 1, usually we will try to synchronize the concurrent events in order to avoid incorrect execution caused by message racing, for example, as in *Petri Nets* [62]. In case 2, the transaction mechanism is introduced.

In this subsection, the design of parallel object-oriented languages will be discussed first. Then, we raise some problems about concurrent access to persistent objects. Finally, the basic transaction mechanism is introduced.

### 2.4.1 Parallel Object-Oriented Languages

The *parallelism* in object-oriented systems can be classified as *fine grain* parallelism inside an object and *large grain* parallelism among objects. For fine grain parallelism, macro data flow execution is performed in the MELD system [34]. For large grain, we have to consider how objects communicate with each other. In [27], four types of inter-object communication are discussed:

**Monitors:** this is concerned with shared memory architecture;

**Message Passing:** this is used by most object-oriented systems, which will generate new threads by sending messages;

**Remote Procedure Call:** this doesn't generate new threads because it will wait until the answer from the remote site has arrived;

**Tuple Spaces:** [27] this allows many messages handled concurrently in one tuple space, which basically does pattern matching between messages and object interfaces.

### 2.4.2 Concurrent Access to Persistent Objects

Persistent objects maybe distributed across sites, and can be accessed by local processes and remote processes concurrently. To achieve efficient and reliable data retrieval, data

replication techniques have been developed in different forms [25].

### 2.4.3 Transaction Processing

When we have global knowledge about parallelism, we can represent the concurrency in CSP-like languages [30]. If only local knowledge or partial knowledge is available, we might apply some techniques such as *nested objects* [48] or *model of correctness* [43]. All other cases are concerned with *transactions.*

In [2], we can find some good comparisons among *two-phase locking*, *time-stamp* and *optimistic concurrency control algorithms* for record-based system. Is there any difference when we make comparisons for object-oriented database?

### 2.4.4 Nested Transactions

Nested transactions are an extension and enhancement of atomic transactions. They provide safe concurrency within transactions, enhancing both performance and modularity. They also provide for smaller grained recovery, allowing better control over transaction execution, simplifying the programming of reliable transaction systems. However, there are open problems for implementing of nested transactions on object-oriented databases.

# 3 Comparison Framework

After having those important features, we now want to have a comparison framework, which represents a conceptual structure organizing all the language features from the perspective of application. In this section, we consider the methodology about how to evaluate object-oriented systems. Then, a comparison framework is proposed. Finally, we will raise some problems about this framework.

## 3.1 Methodology

In [77], a methodology is proposed to evaluate software development environment. They addressed the shortcomings of three different approaches:

1. one specific component, but not how components interact;
2. particular environment with some specific tools available;
3. lists of questions and criteria without the detailed of how to answer the questions.

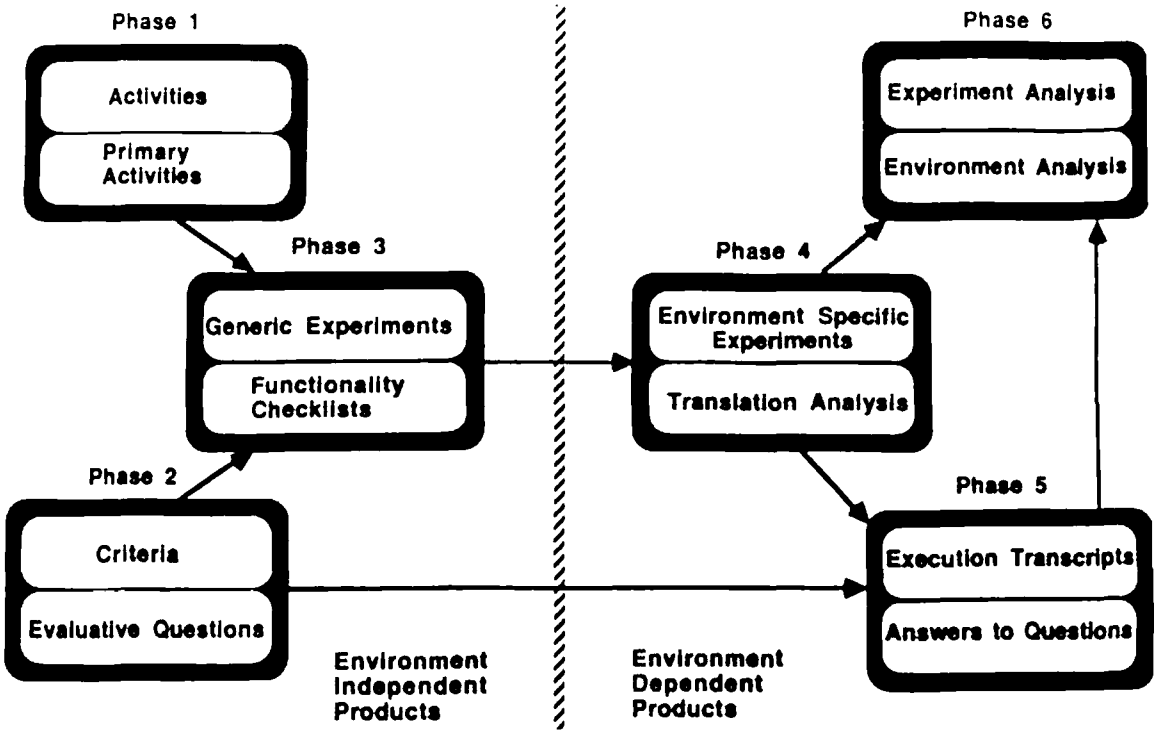


Figure 8: Products of the Evaluation Methodology

After analysing these approaches, they proposed a extensible and partially domain independent methodology to evaluate environments (Figure 8).

In this thesis, I import their methodology to the domain of evaluating object-oriented systems. It is not claimed that my work doesn't belong to the three approaches discussed in [77], but actually my comparison framework belongs to all of them:

1. get as many specific components as possible, and try to find their interactions;
2. 3 particular environment with some specific tools available;
3. lists of questions and criteria with some information and suggestions, which make it easier to answer the questions.

## 3.2 Comparison Framework

A comparison framework can be easily described as a three-step program:

- Can this object-oriented system satisfy the application-domain requirement?

- Can this object-oriented system reduce the cost in software development for this specific application domain?
- Will the run-time performance of this system be too low?

### 3.2.1 Domain Requirement

As a necessary condition, *domain requirements* should be considered first when choosing a suitable object-oriented system. The qualified systems must be able to support those requirements without too much exceptional effort or skill taken. We list several kinds of domain requirements below, which cover most application domains:

**Transaction Management:** Most object-oriented systems will offer some kinds of parallel processing in many different ways, unfortunately, in many application domains, the lack of the global knowledge about concurrency makes it hard to write a parallel program safely. So, various transaction mechanisms are necessary to handle those random concurrent events.

**Real-Time Systems:** Real-time software has performed an important role in many real life applications, for example, military software, network management software, and decision support systems. Distributed data acquisition and processing with timing constraints definitely is a very tough research topic. Other problems include "what is the right abstract level for representing *real-time* in an object-oriented system, "how to support real-time in an object-oriented database systems?" and "how to build up real-time transaction?". For instance, *ENVISAGER* [20] [21] is one object-oriented languages for specifying real-time systems.

**Multiple Language Paradigms:** In some application domains, it is necessary to use rule-base programming, logic programming, or functional programming to represent the idea. This is especially true for developing artificial intelligence systems.[4]

If we want to use object-oriented languages as *process programming languages* [58], it is also necessary to support multiple paradigms, and the mechanisms for integrating those different-paradigm pieces are also important.

[4] It is not a new idea to merge object-oriented with rules. In A.I. research community, a lot of work has been done in the area of coupling rules with frame-base systems.

**Matching with Hardware:** Due to some special hardware architectures, we should consider whether an object-oriented system can fully match with the underlying hardware. As mentioned before, one specific programming language represents a conceptual computational model. This implies if the conceptual model is so different from the real hardware, then we should avoid using this language.

**Query Language and User Interface:** As we mentioned before, in most application domains, we don't want to manipulate the object identities directly, so what we need is a nice query language and friendly user interface.

**Reflective Computation:** At the early stage of the research work about reflection in object-oriented systems [75] [47], it is expected that reflective facilities will become increasing more useful in managing distributed systems. In [75], three examples of reflective programming are illustrated:

1. Dynamic concurrent acquisition (inheritance) of methods from other objects, which can be treated as one kind of very flexible code sharing behavior;
2. Monitoring the behavior of concurrently running objects, which is a dynamic concurrency control scheme with learnability;
3. Augmentation of the time warp mechanism [31] to a distributed system, which will process *undo* and *rollback* operations in a meta-class level.

This is especially for AI-oriented application.

### 3.2.2 Software Development Environment

In building a large and complex system, system requirements usually can not be known completely. Just like those requirements in the previous section, we always can only discover very few special requirements related to the application domain at the beginning. This implies we should consider those general system features very carefully after all the obvious requirements are raised.

Unfortunately, to get a general object-oriented system is just as hard as to answer the question *"what is an object-oriented system?"* or *"what is the relationship between software development environments and the software process?"* [58]. So, before solving these open problems, in this thesis, we try to derive a nearly general object-oriented system from our experience in software development environments. Four important characteristics are

proposed: *version management*, *reusability*, *extensibility* and *prototyping*, which should be supported well by a good object-oriented system [19].

**Version Management** In most object-oriented systems, *version management* is achieved by having *persistent object base* and *interface control*. The persistent object base will keep all the versions as well as the information about those versions. This task includes efficient saving, retrieving, and distributing. But the problem will arise when we try to integrate those software tools with different version identities, i.e., they might have different internal structures, method implementation, and external interfaces. So, we need interface control mechanism to solve these conflicts during the integration.

In some other applications, we might want to modify the class definition for one specific version of objects, which is one kind of *type evolution* [68] [52]. In this case, we can have a set of version interfaces for *types*, and thus allow us to change the type definitions dynamically through these interfaces.

Several works have been done in version management for object-oriented systems. Zdonik [83] [68] has built the version control as a required function of the database management system itself. In another approach, Winkler [80] formulates version control information as part of the program text. The benefit of the former approach is to have an elegant scheme dealing the changes of type definition, while the latter one offers the programmers a flexible environment, in which *rules* and *facts* can be used to express the program versions. Beech and Mahbod [8] proposed yet another scheme, which has both explicit and implicit version control. Their work is the combination of the former two, but they didn't consider *versions for types*.

The system features needed to support version control in object-oriented environment is described as following:

$$\boxed{\text{features for version control}} = \boxed{\text{persistent object base}} + \boxed{\text{interface control schemes}}$$

**Reusability** Reuse of existing software components has the potential to decrease the initial development time and the risk for hidden bugs [61]. In object oriented software, *inheritance* is a natural way to achieve software reuse. For example, under the mechanism of *multiple inheritance*, an new object can be defined with several properties and operations from all his ancestors. Unfortunately, a naive inheritance scheme might hurt the encapsulation of an object [67]. For this purpose, Snyder proposed an inheritance interface in superclasses, which the inheriting classes must go through. In [35], *features*, which contain

many classes, are treated as an encapsulated units, and there is no special interface in classes for multiple inheritance. We do feel the latter approach is better in that: first, it is a big load for the programmer to concern about the inheritance interface as well as the client interface; second, *features*, which are bigger than *classes*, might be more attractive to the programmer to reuse.

Another concern about reusability is *version control* because we hope we can reuse different versions of codes, and sometimes the information about those persistent objects and versions is necessary. Thus, we can describe the required features for *reusability* as the following:

$$\boxed{\text{features for reusability}} = \boxed{\text{encapsulated inheritance}} + \boxed{\text{features for version control}}$$

**Extensibility** Notkin [59] has pointed out that 40% of the total software life cycle cost is spent in the *enhancement phase*, and the extension mechanism is proposed to reduce such cost. Actually, *extensibility* and *reusability* are tightly related. If one module in the existing system is very reusable, then it should be able to merge with another system, which will extend the ability of the latter system. But, extensibility also means we can change the specification of an object, then reuse it. This kind of change will interfere with other objects and cause the reusing unsafe. So, we do need very good interface control and type evolution scheme to deal with this problem. Then, we get another equation for *extensibility*:

$$\boxed{\text{features for extensibility}} = \boxed{\text{type evolution}} + \boxed{\text{features for reusability}}$$

**Prototyping** *Prototyping* is a powerful way in developing large software systems without complete knowledge about system requirements. Usually, we will obtain some information from those prototypes by doing some experiments on them. Then, we can evaluate and improve our first design, and generate the next prototype. Prototyping will increase the performance of the first production version, and thus becomes a tool for reducing risk and cost.

The five major objectives of supporting *prototyping* [24] is

1. providing useful information,
2. producing prototypes quickly, easily, and accurately,
3. being easy to change to get different information,

4. supporting mixtures of behavioral and structural prototypes, and

5. facilitating incorporation of future improvements in prototyping technology. [5]

For *number 1: providing useful information*, the required features will be *persistent object base*, *class definition*, and *query language*. *Reusability* and *extensibility* should be enough to support *number 2: producing prototypes quickly, easily, and accurately.* As concerning about *number 3: being easy to change to get different information*, we suggest *version control*, *query language*, and *extensibility* will do it. *Number 4: supporting mixtures of behavioral and structural prototypes* should be achieved by *type evolution schemes.* Then, we claimed that *number 5: facilitating incorporation of future improvements in prototyping technology* might be supported by having *reflective computation.* Finally, we got the following equation:

$$\boxed{\text{features for prototyping}} = \boxed{\text{features for reusability}} + \boxed{\text{features for extensibility}} + \boxed{\text{impact limitation}} + \boxed{\text{reflective computation}}$$

### 3.2.3 Major Efficiency Problems

*Run time performance* is the main concern in this section.

**Message Passing:** the most expensive run-time cost for object-oriented systems would be the cost for inter-object communication. Because in normal cases, the number of objects will be huge, it is very reasonable that the number of messages is quite large.

Those messages can be classified into two categories, local messages and remote messages. Usually, we can assume that the cost of local messages is much cheaper. Based on this assumption, several efficient schemes are proposed:

[5] It is indeed an open problem to consider what will be the future improvements in prototyping, which is not stated cleared in [24], either. On thing I can think about is automatic prototyping, which means we only have to tell the system some very high level specifications, and it will generate a prototype system by *rewriting* the whole program, or more intelligently *reuse* or it extend the current system. The former approach, *automatic programming*, has been developed for several years by researchers in artificial intelligence, while very little work has been finished for the latter one. Usually, we just let human reuse and extend software. My conjecture is, in either case, *computational reflection* might be a key to develop future prototyping systems.

**Object Replication:** A lot of technologies about replicated database have been well developed. Popovich and Kaiser [63] proposed a partial-replication scheme, *facet*, which allows the programmers to distribute a single object across multiple machines.

**Locality of Objects:** Another way to reduce the number of remote messages is to place those objects intelligent so that most of the inter-object communication can be done in local sites. A typical example is *object clustering* in VBASE [5], which will not suffer from consistency control as in data replication.

**Message Compression:** Compress many messages into one big message and send it. It should be applied to some special communication schemes, for example, *tuple spaces* [49].

**Multicasting:** This supports one message sent to a lot of machines, but the problem will be how to decide the *destination address group* in compiling time, because it is inefficient to decide this group at run time [22].

**Query Processing:** The cost of one query can be separated into two sub-costs: *cost for deciding what to get*, and *cost to get them*. The first cost in object-oriented systems includes how to do query optimization, how to get information from other sites, and how to manage a set of query requests. The second one mainly concerns about communication cost, which we had discussed in *message passing*.

**Concurrency Control Algorithms:** A trade-off between optimistic scheme and locking scheme for transaction management has been discussed for years. The benefit of locking scheme is simple and fast in general cases. And, the optimistic scheme is very complex under very simple conditions, which means it will be much more complex if we consider some advanced transaction models, for example, *nested transaction* [55].

### 3.2.4 Summary

The comparison framework can be illustrated as in Figure 9, and Figure 10.

## 3.3 Interrelations

In this section, we consider important relations among the three steps in the naive comparison framework. We will neglect those relations that are very trivial or not belonging

| | |
|---|---|
| 1. Domain Requirements | Transaction Management |
| | Real-Time Database System |
| | Multiple Language Paradigms |
| | Matching with Hardware Architecture |
| | Query Language and User Interface |
| | Reflective Computation |
| 2. Software Development Environment | Version Management |
| | Reusability |
| | Extensibility |
| | Prototyping |
| 3. Major Efficiency Problems | Message Passing |
| | Query Processing |
| | Concurrency Control Algorithms |

Figure 9: The Framework for Comparing Object-Oriented Systems: I

to the research community of object-oriented systems, for example, the relation between real-time system and message passing, which should be concerned in another area.

We must notice that the relations discussed here would only apply to some of the systems. For instance, some other systems do support reusability but do not have inheritance relation, *i.e.*, they have other language features in our four background architectures to support reusability. In this case, the conflict between concurrency control and reusability may not be applicable to them.

### 3.3.1 Concurrency Control versus Reusability

The conflict between concurrency and multiple inheritance has been addressed in [4], [14], [33], [81], and [76]. The fundamental problem is the multiple inheritance mechanism will violate the concurrent structure, i.e., the concurrent relations in the ancestor classes might be destroyed for several reasons, for example, conflict resolution. In MELD, the commit-abort transaction protocol proposed in [38] might cause some problems if one class overrides two ancestor classes, and these two ancestor classes have one method with the same name, which contains one or more abort-commit statements.

The conflict between reusability and concurrency control may exist even without inheritance at all. For example, in *Actors* [1], the synchronization problem between the

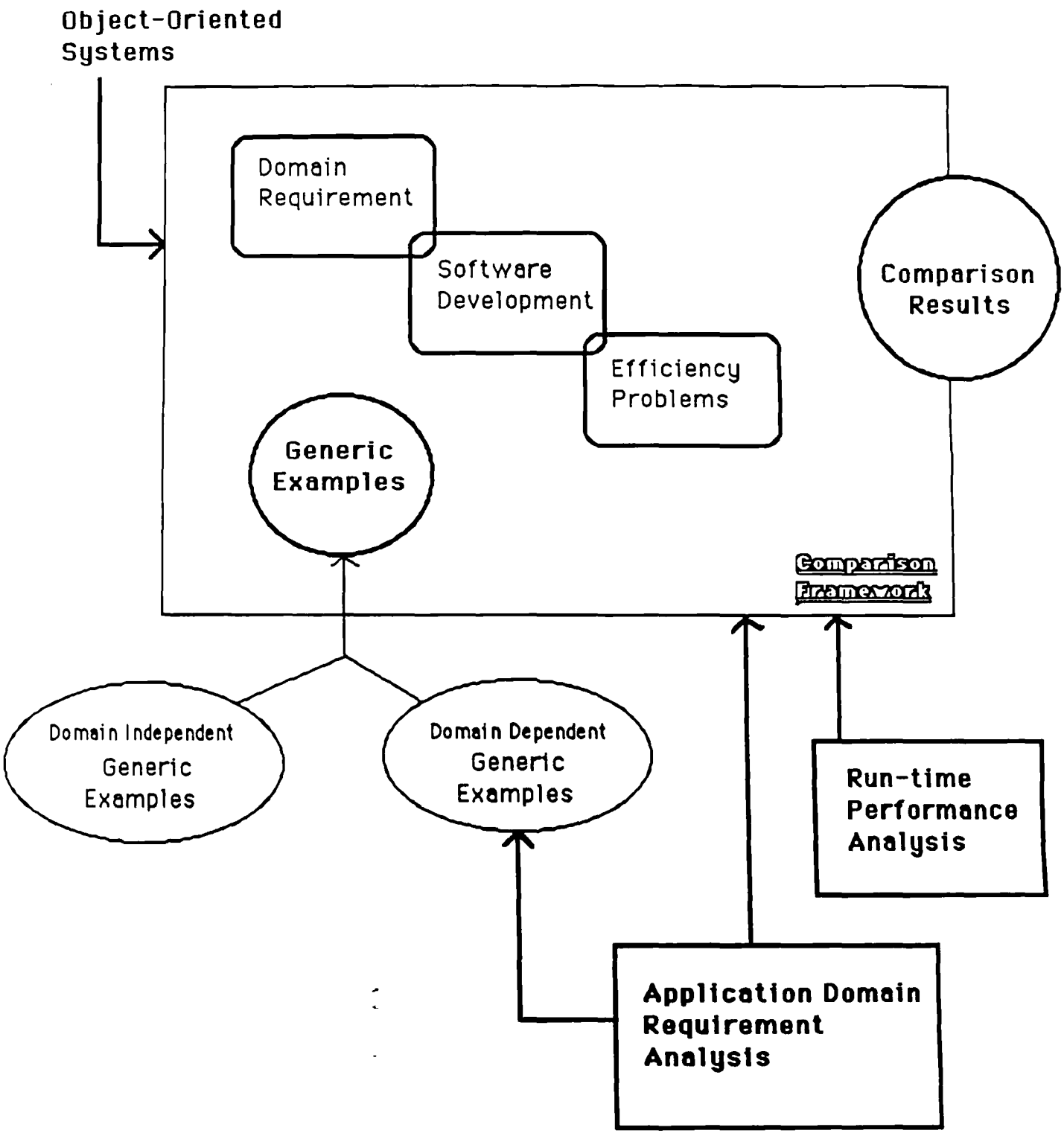


Figure 10: The Framework for Comparing Object-Oriented Systems: II

delegating object and delegated objects might raise some unexpected errors.

The critical point is the relation between *threads* and *objects*. Threads will survive across object boundaries, where objects usually are the reusable units. Therefore, for reusing one object we have to care about those threads attached to it.

### 3.3.2 Prototyping versus Interface Control

For prototyping a large software system, sometimes we just want to modify a few objects, and run it again. Such impact limitation facility is very necessary for being a nice interface control. But, we also need strong type checking for linking objects together safely. This means the interface controller has to do global type checking, but neglect those interfaces among a few objects, which definitely is a very complex task. Furthermore, for the programmers, such a complicated interface is really hard to define and might break the encapsulated objects.

### 3.3.3 Impact Limitation and Message Passing

To achieve impact limitation, we have to compromise some good features that interface control should have. For example, once we might have to define two client-interfaces for one object, i.e., one is for the limited object group, and another is for the rest of the world.

But if we use *tuple space* for inter-object communication, we may be able to define the impact limitation as part of tuple space. For example, we just change the interface for object $O$ (from $O$ to $O1$), and we want to limit the impact to only $\{A, B\}$. Then, we can put the pairs $(O1, A)$ and $(O1, B)$ to the tuple space. In this case, we feel it's much easier to accomplish impact limitation.

### 3.3.4 Matching with Hardware versus Reusability

If the underlying architecture is shared-memory[6], then probably we would like to save some space and let objects sharing executable code. The problem is which parts of the code we should put into the shared memory, i.e. how to cache those pieces of code together to get better performance.

If we do duplicate the reusable modules for all the clients, then we will not suffer from the performance problem, but we loss flexibility.

[6]This always means global memory access time is fast but the cost of one memory cell is very expensive.

### 3.3.5 Reusability versus Locality of Objects

As we just mentioned, to save some memory space and to achieve flexibility, we have better keep one copy of executable. Therefore, for getting better performance, we have to find out the way to locate objects. Unfortunately, most object-oriented systems didn't address this problem completely, i.e., they address the locality problem among objects, which will be retrieved at the same time, but the locality problem among classes is ignored. For example, VBASE can cluster objects together to get performance improved, but this clustering is not related to the inheritance hierarchy, which is the most important feature for reusability.

### 3.3.6 Scheme Evolution vs. Real-Time, Transaction, and Prototyping

In prototyping a software system, it is very common to modify the scheme, and basically, we can have two approaches to deal with such scheme evolution. The first approach is *screening* [68], which defer modifying the persistent store, then filter or correct the values right before they are used. It is some sort of *lazy fashion.* The second one, which is used by *GemStone* [60], is *immediate conversion*, which will change everything at one time, and keeps the consistency of the object base.

The problem of screening is the filtering and correcting process for all the messages definitely will slow down the system and is conflict with the requirements for real-time systems.

One disadvantage of immediate conversion is we might need a long time to update a lot of objects, which we might not need to use before the next updating. In case that this long update transaction will not be aborted, then we must suspend or redo a lot of other transactions at this time. Furthermore, global consistent checking will be even harder if impact limitation is required.

# 4 Case Study: MELD, MARVEL, and VBASE

## 4.1 MELD

MELD [38] is a multiparadigm language combining object oriented, module interconnection, macro dataflow, and transaction processing styles of programming. The object-oriented paradigm in MELD supports object classification, and multiple inheritance relationship as well as distributed parallelism with both synchronous and asychronous message passing among remote and local objects. The dataflow paradigm supports both medium and fine

grain parallelism among the methods within an object and among action equations within several methods. The module interconnection in Meld supports encapsulation, extensibility and reusability of much larger granularity units than classes. The transaction processing facilities are to provide superior concurrency control over the interactions both among methods applied to the same object and among different objects.

## 4.2 MARVEL

MARVEL [36] is an expert system for controlled automation of menial tasks in software engineering and development environments. The MARVEL database, which is for managing complex data in an engineering project, is mapped onto the UNIX file system as follows: MSL's (MARVEL Strategy Language) built-in entity types are divided into three categories, small types (integer, real, Boolean, string), intermediate types (text file, binary file), large types (a set of text), and also user-defined types of objects.

The classes and external relations define MARVEL's logical structure of the engineering artifacts, and rules are used as methods (or operators) to manipulate the information in the logical structure. One function of the database management system is to control the execution of these rules defined in Marvel. In Marvel, this control mechanism is different from other rule-based system in that both preconditions and postconditions are used.

## 4.3 VBASE

VBASE [5, 23], an object-oriented development environment, combines a procedural object-oriented language and persistent objects into one OODB system. Language aspects of VBASE include strong type checking for class definition (but weak type checking for method implementation), a block structured schema definition language, and the ability to type members of aggregate objects. Database aspects include relationships and inverse relationships between objects, user control of object clustering for space and retrieval efficiency, query processing and method triggering. VBASE has basically four functional components:

- TDL (Type Definition Language)
- COP (C Object Process, which has been replaced by C++ in a newer version of VBASE.)
- SQL (Query Language)
- ITS (VBASE Debugger and Browser)

# 5 Three Examples

Using the comparison framework, we can compare MELD, MARVEL, and VBASE in three different application domains: *distributed programming*, *programming environment*, and *telecommunication software*. Basically, we will generate the following results for each cases:

- Comparison based on current implementation;
- What features are weakly supported on current implementation status;
- Comparison based on current implementation plus forthcoming features;
- What problems would likely happen after adding those new features.

Then, we will consider the general language features for MELD, MARVEL, and VBASE.

## 5.1 Distributed Programming

Controlling a set of computers linked by various kinds of communication network gets more and more important now. Also mentioned in section 1, one important reason for using object-oriented systems is to manipulating parallel and distributed environment. In this section, we consider what language features are necessary to support distributed programming first. Then, we consider those three object-oriented systems individually. Finally, we raise some problems about why they can not support distributed programming perfectly.

### 5.1.1 Domain Requirements for Distributed Programming

*Distributed programming* basically involves *local processors*, *local data*, and *message communication* among those processors and data. For *controllability* reason, it is more delicate and complicated than *sequential programming*. The fundamental difficulty is in sequential programming only one *active thread* surviving at one time or a few threads executing simultaneously but being merging in a certain period of time, which implies strong *behavior predictability*. But in the domain of distributed programming, usually we have no or very little idea about *how many threads are active now*, *when this specific thread will terminate*, and *what is the interference among threads*.

In order to deal with the difficult nature of distributed programming, the programming environment is built to reduce the complexity of developing software. For example, remote procedure call make it easier to communicate two UNIX processes in a Inter-Net domain.

For another example, *nested transactions* can be used to build up a reliable and parallel systems. The requirements are:

- Inter-object communication;
- Distributed Transaction Management;
- Matching with Hardware Architecture.

### 5.1.2 MELD

The current implementation of MELD supports two kinds of message passing, *asynchronous message sending* and *procedure call*, which gives programmers flexibility to write their own style of programs. Furthermore, the design and implementation of *distributed transaction management* is being finished, which will not only support *transaction management* but the notation of *transaction objects* also gives great flexibility in building high-performance parallel systems [38]. For matching with hardware architectures, MELD didn't stress the problems except the design of *facets* [63], which considers the partial replication of objects for distributed computation on different architectures.

| **distributed programming** in MELD | current implementation | future |
|---|---|---|
| *inter-object communication* | good | unknown |
| *distributed transaction* | fair | very good |
| *matching with hardware architecture* | poor | fair |

### 5.1.3 MARVEL

In the current implementation, MARVEL is not a distributed system, nor does it support any transaction mechanism as well as any matching with hardware architectures. But MARVEL might have great potential to support these domain requirements:

**Inter-Object Communication:** *Rule firing* can be treated as one kind of message passing. For example, *pattern matching* in production systems is similar to the mechanism of *tuple space*, while *conflict resolution* strategies are those constraints for inter-object communication. If rules are distributed, it is just the case of *remote message sending* in MELD. However, it is not clear how to merge the concepts of synchronization and asynchronization into MARVEL.

**Distributed Transaction:** One important aspect of MARVEL research project is how to handle multi-agents' problem, which is still open.

**Matching with Hardware Architectures:** Although MARVEL doesn't have any features dealing with hardware, a lot of other works in parallel production systems machines might be referred by the future MARVEL research.

| **distributed programming** in MARVEL | current implementation | future |
|---|---|---|
| *inter-object communication* | poor | good |
| *distributed transaction* | none | unknown |
| *matching with hardware architecture* | poor | unknown |

### 5.1.4 VBASE

Although VBASE doesn't fully support distributed object base, it should not be a big deal to upgrade the current implementation of VBASE, probably by adding a *name server*. The problem is VBASE will only support *procedure call* (or synchronized message passing), which severely restrict the representability of a programmer. And unfortunately, it is very unlikely that a new VBASE system with *asynchronous message sending* can be extended from the old implementation, because basically COP will try to link all the possible-access procedures together with user's application program and generate a huge executable file [7].

Actually, in the trade-off between *flexibility* and *efficiency*, VBASE has chosen the latter one enthusiastically. Not only it will generate a huge binary file, but also the tightly matching with hardware architectures. The current implementation of VBASE on SUN-3 will go through the levels of *cache memory* and *virtual memory*, which make it unable to access remote object base through NFS (*network file server*). Definitely the performance it gains in the trading is very nice in a single machine [23], and we can expect that such a system should be easy to move to another hardware system with cache and virtual memory. But it is not clear that how VBASE will match with a distributed environment without changing the design methodology.

[7] approximately 2 to 4 MB for a 30-line application program in C.

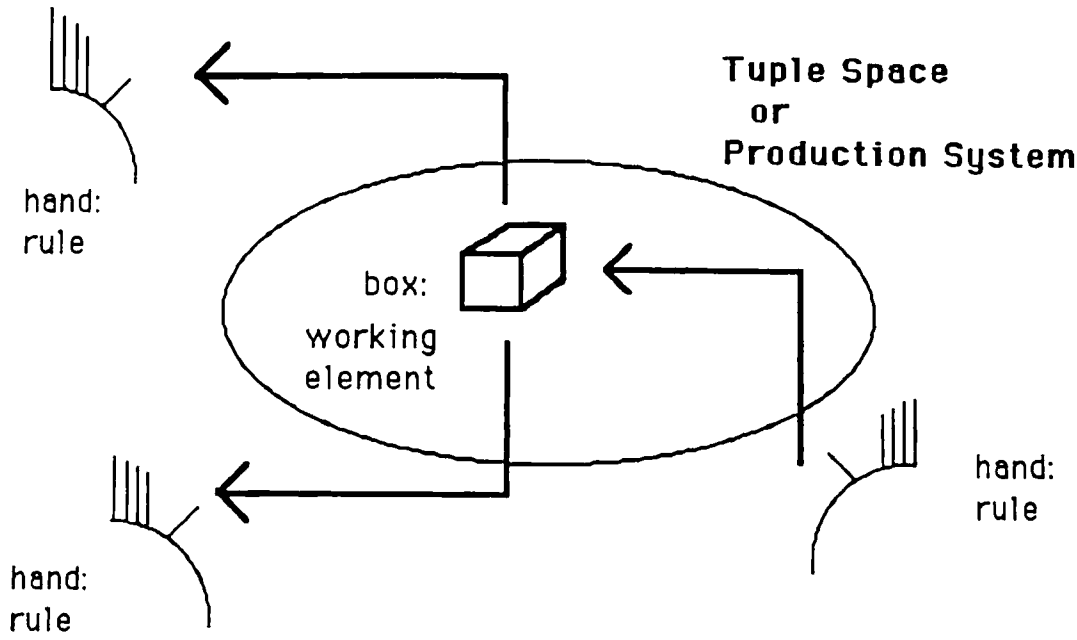


Figure 11: Box-Tossing in MARVEL

| distributed programming in VBASE | current implementation | future |
|---|---|---|
| *inter-object communication* | fair | unknown |
| *distributed transaction* | none | unknown |
| *matching with hardware architecture* | poor | unknown |

### 5.1.5 Summary

For the domain requirements of distributed programming, as considering the current implementation, MELD is preferred because it support both inter-object communication and distributed transaction well. Actually, I have implemented a Box-Tossing program (in Appendex I) using MELD.

The other two systems have severe shortcomings for supporting distributed programming. MARVEL needs distributed rule base systems as well as sophisticated transaction model [37]. VBASE, on the other hand, needs more general system design in order to control the distributed environment more smoothly. The possible implementation of Box-Tossing using MARVEL and VBASE is illustrated in Figure 11, and Figure 12.

As considering the future of these three systems, it really depends on how their current

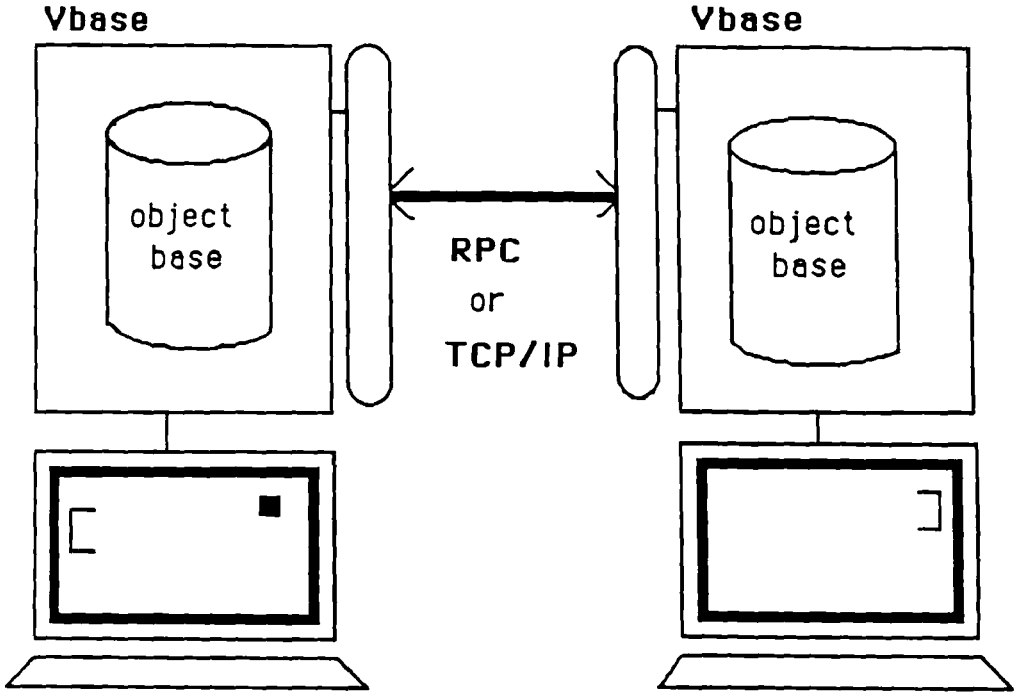


Figure 12: Box-Tossing in VBASE

problems can be solved. For example, VBASE can have two different interfaces to access both local objects and remote objects. That means the system can detect what type of object file is requested and decide which interface we will go through, which will reserve flexibility without losing too much efficiency. However, it is not clear how we can have different styles of message passing based on the current design.

## 5.2 Programming Environment

In order to build a programming environment, we actually have to consider every possible situations in software development process. Although object-oriented systems themselves have been treated as very good programming environments, because theoretically they support *version management*, *reusability*, *extensibility* and *prototyping*, we do feel we need more facilities to reduce the cost of software development. We will explain the special requirements for programming environments. Then, we will see how MELD, MARVEL, and VBASE can fit those requirements.

### 5.2.1 Domain Requirements

One significant difference between building programming environments and building other software systems is we need great *flexibility.* Usually, we will need different environments to support different software projects. Even in a single project, various environments might be suitable for various development stages. This implies we need to change from one environment to another one dynamically, but still accessing the same object-oriented database. From our experience, it is to hard to build up such an environment deterministically, instead, rule-base programming has shown itself as an attractive facility to achieve such a goal.

Another big challenge in programming environments is how to support *programming in many.* Although it is still an open problem to define the process model for this task, we do feel *long term transactions* will play an important role here.

Yet another requirement for programming environment is good query language and friendly user interface. Dealing with a delicate and complicated software project, the programmers usually need plenty of information about the project as well as visual support to understand *the hierarchy of the current project*, *the structure of the object base*, and *the facilities that the current programming environment supports.*

- Multiple language paradigms;

- Long term transaction;
- Query language and user interface.

### 5.2.2 MELD

Although MELD has *action equations* which can be linked dynamically by their data dependent relations, we claim that this data flow paradigm in MELD doesn't help us to build a flexible programming environment. The point is when moving from one to another different environment, we need to reconsider all the data dependency between the environment and the object base.

MELD does try to support various kinds of transactions, which will be manipulated by different concurrency control mechanisms. This suggests for different projects we might want to use different schemes, and the transaction objects will handle all these cases. Furthermore, in tradition transaction model, the conflict information usually is not available for the users, but in MELD, user might be able to access those information kept inside those transaction objects through a special object interface offered by the system manager. This means besides abort and redo, we might want to do something else to resolve the conflicts among long term transactions.

MELD doesn't support any particular query language or user interface so far. For the future, *views* [26] is considered seriously as the candidate to be integrated with MELD.

| **Programming Environment** in MELD | current implementation | future |
|---|---|---|
| *Multiple Language Paradigms* | poor | unknown |
| *Long Term Transaction* | fair | very good |
| *Query Language and User Interface* | poor | good |

### 5.2.3 MARVEL

MARVEL is developed as a tool integration language, and it merge two different paradigms: *object-oriented* and *rule-base*. This easily makes MARVEL satisfies the first requirement, except we need more powerful rule base model with a sophisticated conflict resolution mechanism in the future. Recently, a browser environment for MARVEL has been implemented by Sokolsky [69], which fulfills the third requirement partially. Unfortunately, MARVEL doesn't support long term transactions at this moment, and even no clear idea about this problem exists.

| **Programming Environment** in MARVEL | current implementation | future |
|---|---|---|
| *Multiple Language Paradigms* | good | very good |
| *Long Term Transaction* | none | unknown |
| *Query Language and User Interface* | good | unknown |

### 5.2.4 VBASE

Apparently, VBASE is not terribly good for building programming environment in the first two domain requirements. First, for getting efficiency, VBASE will spend a lot of time (5 to 30 mins) to generate a hard-wired executable file. So, it will slow down the performance a lot, when the programmers want to change the environment very often, which is a very common case in that a programmer will involve several projects simultaneously. But in MARVEL, to reload a set of strategies, which will build another environment, will only take a few seconds. Second, it doesn't support long term transaction so far. We believe is a minor problem because VBASE also has *abstract types* and *storage types* [16], which can be extended to the concept of transaction objects as in MELD.

*SQL* and *ITS* tools actually give very strong support for the third domain environment, which make VBASE still be a promising system for programming environment.

| **Programming Environment** in VBASE | current implementation | future |
|---|---|---|
| *Multiple Language Paradigms* | poor | unknown |
| *Long Term Transaction* | fair | very good |
| *Query Language and User Interface* | very good | very good |

### 5.2.5 Summary

For current implementation, both MARVEL and VBASE are preferred, because they support at least two domain requirements for programming environments. MELD might be better after the *views* language is implemented, because it supports a nice query language with user interface and also the mechanism for building various environments.

For the future, we feel that all of them have hard problems to conquer. For MARVEL, we are dealing with the implementation of long term transaction. VBASE needs more flexible structure to support dynamic environment. There are also a lot of unseen questions for MELD after merging *views* as a part of the system, for example, how we can make safe environment changes in higher view levels that will not generate incorrect or unexpected

data dependency relations in the lower action equation levels.

## 5.3 Telecommunication Software

A *telecommunication software system* is usually separated into two modules, *computation model* and *information model.* The former usually deals with real telecommunication problems, for example, *resource management*, *protocol verification*, *network simulation* and *monitoring.* The latter will consider how to support the former to solve those problems easily. In this section, we will only consider the information model which can be implemented in object-oriented systems.

### 5.3.1 Domain Requirements

*Timing* is the first requirement in developing telecommunication software. For example, in *MAGNET II* system [50], there are three classes of traffic and each of them associated with different kinds of time-delay and losing rate. To support such timing-constraint message passing, definitely the programmers should be able to represent their timing concept in their programs naturally. Furthermore, we need *real-time transactions* for the network shared by several managers.

To manage the network resources, the information model must offer the computation model some performance data. But there are two difficulties: first, the data are distributed and hard to get in very short time; second, even in one local node, the information is too much to process. So, usually, we will like to keep a statistical database, which is active and updated immediately when the status of the network is changed. This implies we need some facilities to support statistical database.

The third requirement for network monitoring is *rule-base programming. Knowledge-based monitoring* of telecommunication [50] [29] [21] has been treated as a promising approach to managing the complex domain.

- Real-time systems;
- Statistical database;
- Multi-paradigm programming language.

### 5.3.2 MELD

MELD doesn't support real-time construct, and the design of real-time MELD is still in

the very early stage. The requirement of statistical database, however, can be achieved in MELD by using *constraints*, which will make an object active. Finally, as we mentioned before, MELD doesn't have rule-base construction.

| **Telecommunication Software** in MELD | current implementation | future |
|---|---|---|
| *Real-Time System* | none | unknown |
| *Statistical Database* | good | unknown |
| *Multiple Language Paradigms* | poor | unknown |

### 5.3.3 MARVEL

The only difference between MARVEL and MELD in supporting *telecommunication software* is that MARVEL support rule-base programming very well.

| **Telecommunication Software** in MARVEL | current implementation | future |
|---|---|---|
| *Real-Time System* | none | unknown |
| *Statistical Database* | good | unknown |
| *Multiple Language Paradigms* | good | unknown |

### 5.3.4 VBASE

VBASE doesn't support these three requirements very well.

| **Telecommunication Software** in MELD | current implementation | future |
|---|---|---|
| *Real-Time System* | none | unknown |
| *Statistical Database* | poor | unknown |
| *Multiple Language Paradigms* | poor | unknown |

### 5.3.5 Summary

None of the three systems are suitable for developing telecommunication in current implementation status, because we do feel that the concept of timing is very necessary in this domain.

## 5.4 Software Development Environment

After considering *domain requirements* for each application domains, we should evaluate MELD, MARVEL, and VBASE from those general language features for software develop-

ment environment. Those features are *version management*, *reusability*, *extensibility*, and *prototyping*.

### 5.4.1 Version Management

MELD: MELD has very simple persistent object base without any interface control mechanism or dynamic type checking. Everything will be done at compiling time. Currently, another version of MELD will be implemented very soon with both name server and object identities.

MARVEL: MARVEL supports both persistent object base and very simple multiple version on top of UNIX file system. One program merging MARVEL with *RCS* has been implemented. If we can implement MARVEL in MARVEL, then we can integrate *RCS* with the *meta-marvel*, which will be a object-oriented system with excellent version management scheme.

VBASE: VBASE has a nice persistent object base system, but it doesn't support version control explicitly [45]. In the future, the storage type of VBASE might be upgraded as a good approach to dealing with version control.

| Version Management | MELD | MARVEL | VBASE |
|---|---|---|---|
| *Current Implementation* | none | fair | fair |
| *Future* | unknown | very good | good |

### 5.4.2 Reusability

MELD: *Features*, *classes*, and *action equations* are multiple granularities of reusable units in MELD. *Features* provide large granularity of reusable units, where *action equations* provide small reusable blocks. Like other object-oriented systems, MELD also support *multiple inheritance* to reuse the class definition.

MARVEL: MARVEL supports two kinds of inheritance relation: *data inheritance*, which can inherit the class definition, and *behavior inheritance*, i.e. every rule defined in MARVEL will be inherited by all the objects applicable to this rule. If more than one rules can be satisfied for some object at the same time, the conflict resolution mechanism will deal with this problem.

VBASE: VBASE supports the basic single inherit[illegible] with *abstract object state*, which means the definition of inheritance hierarchy as w[illegible] as the behavior of those operations and the implementation of an object are totally separated to support *encapsulation* of object definition.

| Reusability | MELD | MARVEL | VBASE |
|---|---|---|---|
| *Current Implementation* | very good | good | good |
| *Future* | unknown | unknown | unknown |

### 5.4.3 Extensibility

MELD: MELD doesn't support schema evolution at all.

MARVEL: MARVEL supports static schema evolution [69]. and the design of *dynamic schema evolution* is still in a very early stage.

VBASE: VBASE supports type checking as well as object migration using *immediate conversion* scheme. The only minor problem is the current implementation of TDL take a really long time to do *type evolution*, and it also grasps almost all the CPU resources at that period of time.

| Extensibility | MELD | MARVEL | VBASE |
|---|---|---|---|
| *Current Implementation* | fair | good | very good |
| *Future* | unknown | very good | unknown |

### 5.4.4 Prototyping

MELD: MELD doesn't support general reflective computation, but it does support reflection in transaction management. There is a system defined class *transaction*, which includes different concurrency control schemes. The final product of MELD should allow programmers to reuse and modify those system classes, and produce their own concurrency control schemes. Furthermore, they should be able to access to the transaction object through the client interface for some advanced application domains.

MELD didn't support impact limitation either, because MELD doesn't have multiple version facility now.

MARVEL MARVEL doesn't have reflective computation so far. The design of *meta-marvel.* which will fully support reflective computation, is still in a very early stage. And, although MARVEL will keep multiple versions for one object, it doesn't support *impact limitation* at all. However, as we discussed before, MARVEL might have some facility similar to *tuple space*, which is considered as one message passing scheme suitable for impact limitation.

VBASE VBASE support not much reflection now, but it could be changed into another version easily, because it already had the meta-class structure. No impact limitation is available so far.

| Prototyping | MELD | MARVEL | VBASE |
|---|---|---|---|
| *Current Implementation* | fair | poor | poor |
| *Future* | unknown | good | unknown |

### 5.4.5 Summary

None of them will support *prototyping* very well in current implementation. The most important reason is they don't have very good impact limitation or *intelligent interface control.* However, MARVEL does show its potential in supporting prototyping, because:

- First, in MARVEL, there will be two levels of control knowledge: rule level and meta-rule level, which implies we can set up impact limitation in a higher level;
- Second, meta-rules imply *reflexibility*, which might be an important feature for future prototyping systems.

## 5.5 Major Efficiency Problems

### 5.5.1 MELD

The current implementation of MELD is running under an interpreter, which will process a set of action equations with data dependency relations. This makes MELD not attractive because on conventional architectures it is unlikely we can implement such a language very efficiently. So, a new version of MELD , called *MeldC*, will remove the data flow paradigm, which will make the performance much better, although we loss some fine grain parallelism.

Another efficiency about MELD is about the implementation of concurrency control algorithms. The design of MELD 's transaction management is so flexible that the user can

select one specific concurrency control scheme for his application. This imply MELD can achieve better performance in transaction management.

### 5.5.2 MARVEL

One major efficient problem of MARVEL is the *rule chaining mechanism.* In current implementation, MARVEL will build up all the links among rules when loading strategies, and after all, it just fires rules through those links. There are two drawbacks here: first, if the size of the rule base is big, then definitely the number of links will be extremely huge because $n$ rules might have $n^2$ links; second, the architecture is not very flexible.

### 5.5.3 VBASE

As mentioned before, VBASE has three efficiency problems. First, the TDL process is too slow. Second, the executable file generated by COP is too big. Another minor problem is the simple locking scheme used by VBASE will lock one whole database file, which implies that the concurrency in VBASE will not be so great.

### 5.5.4 Summary

The most important problem about MELD and VBASE is objects in different classes but inheriting the same thing do not share executable code. We can sense this question by the huge size of one VBASE executable file. Although some people claim that they get great time efficiency in return, we feel that it is not the case because too big executable file may cause main memory page fault very often, especially, in object-oriented languages, it is even harder to guess which pieces of code will be executed next time than in sequential languages, which will always follow the single thread program counter.

## 5.6 Comparison Results

The result (**Figure** 13) shows that these three systems have some severe shortcomings in supporting **three** different application domains as well as general environment features considered by software engineers. All these shortcomings are listed in Figure 14, Figure 15, and Figure 16.

| | MELD | | MARVEL | | VBASE | |
|---|---|---|---|---|---|---|
| | current | future | current | future | current | future |
| Distributed Programming: Object Communication | good | unknown | poor | good | fair | unknown |
| Transactions | fair | very good | none | unknown | none | unknown |
| Hardware Matching | poor | fair | poor | unknown | poor | unknown |
| Programming Environment: Multiple Paradigms | poor | unknown | good | very good | poor | unknown |
| Long transactions | fair | very good | none | unknown | fair | very good |
| User Interface | poor | good | good | unknown | very good | very good |
| Telecommunication: Real-time system | none | unknown | none | unknown | none | unknown |
| Statistical database | good | unknown | good | unknown | poor | unknown |
| Multiple Paradigms | poor | unknown | good | unknown | poor | unknown |
| Software Development: Version Management | none | unknown | fair | very good | fair | good |
| Reusability | very good | unknown | good | unknown | good | unknown |
| Extensibility | fair | unknown | good | very good | good | unknown |
| Prototyping | fair | unknown | poor | good | poor | unknown |

Figure 13: Comparison Results

| MELD |
|---|
| not considering the matching with hardware architecture |
| not flexible enough to reconfigurate the relations among objects at run-time |
| without good user interface |
| not supporting real-time software |
| no version control mechanism |
| not supporting type evolution |
| not efficient in time |

Figure 14: MELD's shortcomings

| MARVEL |
|---|
| without a nice paradigm for inter-object and inter-rule communication |
| without transaction mechanism |
| not supporting real-time software |
| without good version control mechanism |
| not supporting type evolution |
| not efficient in time |

Figure 15: MARVEL's shortcomings

| VBASE |
|---|
| without various message passing schemes |
| without transaction mechanism |
| not flexible enough to reconfigurate the relations among objects at run-time |
| without good user interface |
| not supporting real-time software |
| not easy to implement statistcal database |
| without good version control mechanism |
| not efficient in space |

Figure 16: VBASE's shortcomings

# 6 Conclusion and Future Work

The methodology of this work is to build up an information structure supporting the design choices in object-oriented systems. For this purpose, a simple comparison framework is introduced, and its disadvantages are also discussed. We then use this framework to compare three different object-oriented systems, MELD, MARVEL, and VBASE, in three application domains.

## 6.1 Contribution and Conclusion

The major contributions of this thesis are

- A set of topics and problems in the object-oriented research community have been reviewed and discussed, and they are organized as an experimental comparison framework, which shows a big picture of object-oriented systems;
- Some difficulties and disadvantages of building such a framework are raised and discussed, which will motivate a better comparison framework;
- The result of evaluating MELD, MARVEL, and VBASE in three different domains, *distributed programming*, *programming environment*, and *telecommunication software* is derived by using the comparison framework;

For the object-oriented research community, this framework does show a bigger picture containing many important problems in control architecture, data architecture, definitional architecture and concurrency control. For system hackers, it also addresses the most serious problems about their current systems. Some of the problems have already been raised in the thesis:

**version control and query interface:** Version control is fundamental for *reusability*, *extensibility*, and *prototyping*. We suggest an object-oriented system should not only keep multiple version, but also keep managing those versions, for example, *interface control*, *object clustering*, *type evolution support*.

**merging rules into object-oriented paradigm:** Two of the three application domains need rule base programming to deal with their hard problems. For constructing complex software system, we suggest that rules should be considered as one important language feature for object-oriented systems.

**supporting real-time software:** In order to build a real-time system, how to merge the timing concept into the system should be considered in the first place.

**message-passing mechanism:** Different message passing schemes give the programmers flexibility to represent their idea, however, using only one elegant scheme might get other advantages. For example, *tuple space* might be a good choice for *impact limitation*, and *communication efficiency*.

**various transaction models:** So far for all the three application domains, transaction mechanism is required, which implies an object-oriented system can not be very useful without a nice transaction management model dealing with various kinds of transactions.

## 6.2 Future Work

For getting a better comparison framework, two problems are considered as the future extension:

**How to generate generic experiments?** One important phase of the methodology in [77] is to generate some generic experiments and functionality checklists. In this thesis, my implementation and domain experimental examples didn't go through this phase. I seriously believe that developing a set of good generic experiments will make the comparison result more meaningful and reliable.

**How to translate those conflicts into part of the evaluation question?** As pointed out in the thesis, several conflicts will happen when we consider all these langauge features together. Before getting the solutions for those conflicts, to merge those conflicts as part of the framework will make the comparison result more accurate.

# Acknowledgments

My deepest thanks go to my adviser, Gail E. Kaiser, for her encouragement and guidance during these two years. Next, I would like to thank Soumitra Sengupta, who has always given me a lot of nice suggestions for both research and real life. I also want to thank my colleages in *frodo* group, especially Wenwey Hseush, Steve Popovich, Yaser Barghouti, and

Michael Sokolsky. Finally, I would like to thank for the support from DCC group, Yechiam Yemini, Jed Schwartz, Alex Dupuy, and Jacob Weintraub.

# A Box Tossing in MELD

```
FEATURE toss
INTERFACE:
        EXPORTS         box, window
        REMOTES         window1 : window at $WORKSTATION
                        hand1   : hand at $WORKSTATION
                        box1    : box at $WORKSTATION
IMPLEMENTATION:
OBJECT:
        box2            : box;
        hand2           : hand;
        window2         : window;
        gravity         : double := 0.12;
        get_number ()   : double;


CLASS box ::=
        active      : integer := 0;
        center_x    : double := 2028.0;
        center_y    : double := 300.0;
        speed_x     : double := 0.0;
        speed_y     : double := 0.0;
        status      : integer := 0;

METHODS:
{* set up the path between standard input and this object. *}
path (0, $self);
if (active = 1) then status := window2.box(center_x, center_y, 1);
"help" ⟼ {
        printf ("BOX:: (1) x speed = d (between 1.0 and 100.0)
                (2) y speed = d (between -20.0 and 20.0)");
        printf("Box position : (x = %lf, y = %lf) Box speed :
                (x = %lf, y = %lf)", center_x, center_y, speed_x, speed_y);
}
```

```
impel (x, y: double) ⟼ (
        if (speed_x = 0.0 && speed_y = 0.0 && (x != 0.0 || y != 0.0))
          then
              send move () to $self;
        speed_x := x;
        speed_y := y;
)
set_x_y (x, y :double) ⟼ (center_x := x; center_y := y;)
move () ⟼
{
        hand2.light (center_x, center_y);
        if (speed_x != 0.0 || speed_y != 0.0) then [
          speed_y +:= gravity;
          (center_x +:= speed_x; center_y +:= speed_y; )
          if (status = 1) then send move () to $self;
          else {
            if (status = 0) then {
              speed_y := 0.0;
              speed_x := 0.0;
            }
            else {
              if (box1.activate (center_x, center_y, speed_x, speed_y) = 1) then {
                active := 0;
                window2.box (center_x, center_y, 0);
                {* send box (center_x, center_y, 0) to window2;*}
              }
            }
          }
        ]
}
activate (cx, cy, x, y: double) ⟼ {
        if (active = 1) then return (0);
        else {
```

```
        if (window2.box (cx, cy, 1) = 1) then {
          center_x := cx;
          center_y := cy;
          active := 1;
          speed_x := x;
          speed_y := y;
        if (speed_x != 0.0 || speed_y != 0.0) then send move () to $self;
          return (1);
        }
        else {
          printf ("activate: box is not inside the window.");
          return (0);
        }
      }
}
END CLASS box


CLASS window ::=
      left       : integer := 1050;
      right      : integer := 2100;
      up         : integer := 50;
      down       : integer := 850 ;
      open_side  : integer := -1;

METHODS:
      fb_open ();
      path (0, $self);
"help" ⟼
      printf ("window = (left = %d, right = %d, up = %d, down = %d)",
            left, right, up, down);
box (x, y : double; flag : integer) ⟼ {
      box (x - left, y, flag);
      return ($self.inside (x, y));
}
```

```
hand (x, y : double) ⟼ {
        if (open_side = 1) then right_hand (x - left, y);
        else
          left_hand (x - left, y);
}
inside (x, y : double) ⟼ {
        if (x ≥ left && x ≤ right && y ≥ up && y ≤ down) then
          return (1);
        else {
          if (open_side = 1 && x > right) then return (-1);
          else {
            if(open_side = -1 && x < left) then return (-1);
            else return (0);
          }
        }
}
END CLASS window


CLASS hand ::=
        center_x : double := 2029.0;
        center_y : double := 300.0;
        speed_x  : double := -35.0;
        speed_y  : double := -2.0;
        dx       : double;
        dy       : double;
        y1       : double;
        t        : double;
        range    : double := 20;
        status   : integer := 0;
        {* 1 = grasp; 2 = toss; 0 = not both *}
METHODS:
        path (0, $self);
        send hand (center_x, center_y) to window2;
```

```
"help" ⟼ {
        printf( "Hand position : (x = %lf, y = %lf) (1) hand x = d
                (2) hand y = d (3) toss.", center_x, center_y);
}
"x*speed*=*" ⟼ {
        speed_x := get_number ($selector);
        printf( "x speed = %lf.", speed_x);
}
"y*speed*=*" ⟼ {
        speed_y := get_number ($selector);
        printf( "y speed = %lf", speed_y);
}
"hand*x*=*" ⟼ {
        center_x := get_number ($selector);
        printf( "hand x = %lf", center_x);
}
"hand*y*=*" ⟼ {
        center_y := get_number ($selector);
        printf( "hand y = %lf", center_y);
}

"init" ⟼ {
        send set_x_y (center_x - 1.0, center_y) to box2;
        send impel ((double)0.0, (double)0.0) to box2;
        status := 1;
}

"toss" ⟼ {
        if (status = 1) then {
          send signal (center_x - 1.0, center_y, speed_x, speed_y) to hand1;
          send set_x_y (center_x - 1.0, center_y)                  to box2;
          send impel (speed_x, speed_y)                            to box2;
          status := 2; {* toss *}
        }
}
```

```
light (x, y: double) ⟼ {
        if (x < center_x + range && x > center_x - range
          && y < center_y + range && y > center_y - range) then {
          if (status = 2) then return (0);
          else {
            status := 1; {* grasp *}
            box2.impel ((double) 0.0, (double) 0.0);
            printf ("I catch the box.");
            return (1);
          }
        }
        else {
          status := 0;
          return (0);
        }
}
signal (x, y, sx, sy : double) ⟼ {
        printf ("signal (%lf, %lf, %lf, %lf)", x, y, sx, sy);
        t := (center_x - x)/sx;
        if (t > 0) then {
          y1 := y + sy * t + gravity * t * t / 2.0;
          if (window2.inside (center_x, y1) = 1) then {
            send move_to (center_x, y1) to $self;
          }
          else printf ("I am not going to catch the box");
        }
}
move_to (x, y : double) ⟼ [
        if (center_x + 10 < x) then dx := 10;
        else dx := x - center_x;
        if (center_x - 10 > x) then dx := -10;
        else dx := x - center_x;
        if (center_y + 6 < y) then dy := 10;
```

```
        else dy := y - center_y;
        if (center_y - 6 > y) then dy := -10;
        else dy := y - center_y;
        if (dx != 0 || dy != 0) then [
          center_x +:= dx;
          center_y +:= dy;
          send move_to (x, y) to $self;
        ]
]
END CLASS hand
END FEATURE toss
```

# References

[1] G. Agha. *Actors:* A Model for Concurrent Computation in Distributed Systems. Technical Report 844, *M.I.T.* June (1985).

[2] R. Agrawal, and D.J. DeWitt. Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation. *ACM Transaction on Database Systems* 10:4 (1985) 529-564.

[3] M. Aksit, and A. Tripathi. Data Abstraction Mechanisms in Sina/st. In *OOPSLA '88* (1988) 267-275.

[4] P. America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *ECOOP '87* (1987) 234-242.

[5] T. Andrews, and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *OOPSLA '87* (1987) 430-440.

[6] J. Banerjee, W. Kim, and K. Kim. Queries in Object-Oriented Databases. Technical Report, *MCC* DB-188-87 (1987).

[7] A. Baroody, and D. DeWitt. An Object-Oriented Approach to Database System Implementation. *ACM Transaction on Database Systems* 6:4 (1981) 576-601.

[8] D. Beech, and B. Mahbod. Generalized Version Control in an Object-Oriented Database. In *IEEE 4th International Conference on Data Engineering* (1988) 14-22.

[9] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *OOPSLA '86* (1986) 78-86.

[10] G. Blair, J.J. Gallagher, and J. Malik. Genericity vs Inheritance vs Delegation vs Conformance vs. *University of Lancaster* (1988).

[11] A. Borgida. Exceptions in Object-Oriented Languages. in *Object-Oriented Programming Workshop, SIGPLAN Notices* 21:10 (1986) 107-119.

[12] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM TOPLAS* 3:4 (1981) 353-387.

[13] A. Borning. Constraint Hierarchies. In *OOPSLA '87* (1987) 48-59.

[14] J. Briot, and A. Yonezawa. Inheritance and Synchronization in Concurrent OOP. In *ECOOP '87* (1987) 32-40.

[15] P. Cointe. The ObjVlisp Kernel. In *Meta-Level Architectures and Reflection* (1988) 155-176.

[16] C. Damon, and G. Landis. Abstract Types and Storage Types in an OO-DBMS. In *Spring CompCon '88* (1988) 172-176.

[17] S. Danforth, and C. Tomlinson. Type Theories and Object-Oriented Programming. *ACM* Computing Survey, Vol. 20, No. 1, March (1988) 29-72.

[18] P. Dasgupta, R.J. LeBlanc Jr, and W.F. Appelbe. The Clouds Distributed Operating System. In *IEEE DCS '88* (1988) 2-9.

[19] A.M. Davis, E.H. Bersoff. and E.R. Comer. A Strategy for Comparing Alternative Software Development Life Cycle Models. In *IEEE Transaction on Software Engineering* 14:10 (1988) 1453-1461.

[20] J.P. Diaz-Gonzalez, and J.E. Urban. ENVISAGER: A Visual, Object-Oriented Specification Environment for Real-Time Systems. In *IEEE 4th International Workshop Conference on Software Specification and Design* (1987) 13-20.

[21] J.P. Diaz-Gonzalez, and J.E. Urban. Language Aspects of ENVISAGER: An Object-Oriented Environment for Specification of Real-Time Systems. In *IEEE International Conference on Computer Languages* (1988) 214-225.

[22] D. Duchamp. Extend: Communication Support for Transaction. Grant Proposal, *Columbia University* (1989).

[23] J. Duhl, and C. Damon. A Performance Comparison of Object and Relation Databases Using the Sun Benchmark. In *OOPSLA '88* (1988) 153-163.

[24] R.P. Gabriel. Draft Report on Requirements for a Common Prototyping System. *SIGPLAN Notices* 24:3 (1989) 93-165.

[25] H. Garcia-Molina. The Future of Data Replication. In *IEEE 5th Symposium on Reliability in Distributed Software and Database Systems* (1986) 13-19.

[26] D. Garlan. Views for Tools in Integrated Environments. Ph.D. Thesis, *Carnegie Mellon University* (1987).

[27] D. Gelernter. Generative Communication in Linda. *ACM Transaction on Programming Languages and Systems* (1985) 80-112.

[28] B. Hailpern, and H. Ossher. Extending Objects to Provide Multiple Interfaces. Technical Report, *IBM T.J. Watson Research Center* (1988).

[29] B.L. Hitson. Knowledge-Based Monitoring and Control: An Approach to Understanding the behavior of TCP/IP Network Protocols. In *SIGCOMM '88: Communications Architectures & Protocols* (1988) 210-221.

[30] C.A.R. Hoare. Communicating Sequential Processes. *Communication of the ACM*, Vol. 21, No. 8, August (1978) 666-677.

[31] D.R. Jefferson. Virtual Time. *ACM TOPLAS* Vol. 7, No. 3, (1985) 404-425.

[32] M.B. Jones, and R.F. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. In *OOPSLA '86* (1986) 67-77.

[33] D.G. Kafura, and K.H. Lee. Inheritance in Actor Based Concurrent Object-Oriented Languages. *Virginia PolyTech. Institute and State University* TR 88-53 (1988).

[34] G.E. Kaiser, and D. Garlan. Melding Data Flow and Object-Oriented Programming. In *OOPSLA '87* (1987).

[35] G.E. Kaiser, and D. Garlan. Composing Software Systems From Reusable Building Blocks. In *20th Annual Hawaii International Conference on System Sciences* (1987) 536-545.

[36] G.E. Kaiser, N.S. Barghouti, P.H. Feiler, and R.W. Schwanke. Database Support for Knowledge-Based Engineering Environments. *IEEE EXPERT* May (1988) 18-32.

[37] G.E. Kaiser. A Marvelous Extended Transaction Processing Model. To appear in *IFIP Conference* (1989).

[38] G.E. Kaiser, S. Popovich, W. Hseush, and S. Wu. Melding Multiple Granularities of Parallelism. In *ECOOP '89* (1989).

[39] P.C. Kanellakis, and C.H. Papadimitriou. The Complexity of Distributed Concurrency Control. *SIAM J. Computing* 14:1 (1985) 52-74.

[40] K.M. Kavi, and D. Chen. Architectural Support for Object-Oriented Languages. IEEE (1988) 54-58.

[41] M.A. Ketabchi, and V. Berzins. Mathematical Model of Composite Objects and Its Application for Organizing Engineering Databases. *IEEE Transaction on Software Engineering* 14:1 (1988) 71-84.

[42] H.F. Korth. Extending the Scope of Relational Languages. *IEEE Software* January (1986) 19-28.

[43] H.F. Korth, and G.D. Speegle. Formal Model of Correctness without Serializability. In *ACM SIGMOD '88* (1988) 379-386.

[44] H. Lieberman. Using Prototypical Objects to Implemen Shared Behavior in Object-Oriented Systems. In *OOPSLA '86* (1986) 214-223.

[45] L.C. Liu, and E. Horowitz. Object Database Support for a Software Project Management Environment. In *ACM SIGSOFT '88 : SDE3* (1988) 85-96.

[46] S. Lucco. Parallel Programming in a Virtual Object Space. In *OOPSLA '87* (1987) 26-34.

[47] P. Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA '87* (1987) 147-155.

[48] B. Martin. Modeling Concurrent Activities with Nested Objects. In *7th ICDCS* September (1987) 432-439.

[49] S. Matsuoka, and S. Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *OOPSLA '88* (1988) 276-284.

[50] S. Mazumdar, and A.A. Lazar. Knowledge-Based Monitoring of Integrated Networks. In ***First International Symposium on Integrated Network Management***, May (1989).

[51] D. McAllester, and R. Zabih. Boolean Classes. In *OOPSLA '86* (1986) 417-423.

[52] D. McLeod. A Learning-Based Approach to Meta-Data Evolution in an Object-Oriented Database. In *2nd International Workshop on Object-Orie Database Systems* (1988) 219-224.

[53] J. Micallef. Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* 1:1 (1988) 12-38.

[54] L. Mohan, and R.L. Kashyap. An Object-Oriented Knowledge Representation for Spatial Information. *IEEE Transaction on Software Engineering* 14:5 (1988) 675-681.

[55] J.E. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. *The MIT Press* (1985).

[56] A. Motro, A. D'Atri, and L. Tarantino. The Design of KIVIEW: An Object-Oriented Browser. in *2nd International Conference on Expert Database Systems* (1988) 107-131.

[57] J.R. Nestor. Toward a Persistent Object Base. In *International Workshop on Advanced Programming Environments* (1986) 372-394.

[58] D. Notkin. The Relationship between Software Development Environments and the Software Process. In *ACM SIGSOFT '88: SDE3* (1988) 107-109.

[59] D. Notkin. Extension and Software Development. In *IEEE 10th International Conference on Software Engineering* (1988) 274-283.

[60] D.J. Penney, and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *OOPSLA '87* (1987) 111-117.

[61] D.E. Perry, and G.E. Kaiser. Adequate Testing and Object-Oriented Programming. To appear in *Journal of Object-Oriented Programming*, November/December (1989).

[62] C.A. Petri. Concurrency as a Basis of System Thinking. In *5th Scandinavian Logic Symposium*, Janurary 1979, 143-162.

[63] S. Popovich, and G.E. Kaiser. Melding Parallel and Distributed Programming. Technical Report, *Columbia University* (1988).

[64] C. Pu, G.E. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended Activities. *IEEE VLDB '88* (1988).

[65] J. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *OOPSLA '87* (1987) 466-481.

[66] V. Russo, G. Johnston, and R. Campbell. Process Management and Exception Handling in Multiprocess Operating Systems Using Object-Oriented Design Techniques. In *OOPSLA '88* (1988) 248-258.

[67] A. Snyder. Inheritance and the Development of Encapsulated Software Components. In *Research Directions of Object-Oriented Programming* (1987) 165-188.

[68] A. Skarra, and S.B. Zdonik. Type Revolution in an Object-Oriented Database. In *Research Directions of Object-Oriented Programming* (1987) 393-415.

[69] M. Sokolsky. Data Migration in an Object-Oriented Software Development Environment. Master Thesis, *Columbia University* (1989).

[70] L.A. Stein. Delegation is Inheritance. In *OOPSLA '87* (1987) 138-146.

[71] A. Tripathi, A. Ghonami, and T. Schmitz. Object Management in the NEXUS Distributed System. In *IEEE DCS '87* (1987) 50-53.

[72] D. Ungar. The Design and Evaluation of a High Performance Smalltalk System. Ph.D. Thesis, University of California, Berkeley (1986).

[73] D. Ungar, H. Lieberman, L.A. Stein, and D. Halbert. Treaty of Orlando Revisited. In *OOPSLA '88* (1988) 357-362.

[74] M. van Biema, G.Q. Maquire, and S. Stolfo. Constraint Based Invocation: The Integration of Rule-Based and Object-Oriented Programming Paradigms. *Tech. Report, Columbia University* (1988).

[75] T. Watanabe, and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *OOPSLA '88* (1988) 306-315.

[76] P. Wegner. Dimensions of Object-Based Language Design. In *OOPSLA '87* (1987) 168-182.

[77] N.H. Weiderman, A.N. Habermann, M.W. Borger, and M.H. Klein. A Methodology for Evaluating Environments. In *ACM SIGSOFT/SIGPLAN SDE* (1986) 199-207.

[78] J. Wileden, L. Clarke, and A. Wolf. Three Techniques Supporting the Development of Large Prototype Systems. In *3rd Intl. IEEE Conf. on ADA Applications and Environments* (1988) 28-37.

[79] I. Williams, M. Wolczko, and T. Hopkins. Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy. In *ECOOP '87* (1987) 79-88.