INTRODUCTION TO THE SPECIAL ISSUE OF THE SIGPLAN NOTICES ON THE OBJECT-ORIENTED PROGRAMMING WORKSHOP

Organized by IBM T. J. Watson Research Center and Brown University

held at

IBM Yorktown Heights, June 9-13,1986

Peter Wegner

Brown University

and

Bruce Shriver

IBM T. J. Watson Research Center

Overview

The objectives of the workshop were to bring together researchers in object-oriented programming, review current research in this area, and explore the role of object-oriented programming in the design and realization of complex systems. The 35 participants presented about 30 papers on languages, concepts, applications and theory of object-oriented programming. In addition, there were five break-out sessions on the following topics: Multiparadigm programming; Object-oriented databases; Inheritance; Concurrency; and Type theory.

The papers in this issue are a representative sample of about half the presented papers but have been restricted in length for this SIGPlan Notices presentation. We have, however, included a complete set of abstracts from the meeting at the end of the papers in this issue. We have grouped the papers in this issue into three categories:

Object-oriented systems, including extensions to concurrency, knowledge representation, graphical programming, and actors (7 papers)

Concepts, including type-based versus instance-based inheritance, integration with block-structure, and layered system description (8 papers)

Theoretical issues, including integration with functional and logical programming, and type theory (3 papers)

Object-Oriented Systems

Stroustrup in *An Overview of C++* describes the extension of *C* to include data abstraction, inheritance, and other features that support improved security and programming methodology. *C++* provides improved type checking, a class construct for data abstraction, an inheritance mechanism

by means of derived classes, virtual functions that permit definition of a common calling interface for functions later defined in subclasses, and a flexible information hiding mechanism. The extensions were constrained by a desire for compatibility with *C* and by efficiency considerations. Programs not using an extension do pay for it. This paper is practically important in that *C++* may well have more users than any other object-oriented language. It is also an important example illustrating how to take an insecure and permissive language like *C* and augment it with features that supports object-oriented programming without sacrificing efficiency, flexibility, or portability.

CommonObjects is an extension of *Common Lisp* in the same sense that *C++* is an extension of *C*. It is representative of a new generation of object-oriented languages that build on the experience of *Smalltalk* and *Zetalisp*. *CommonLoops* (Xerox) and *Trellis/Owl* (DEC) are two other such languages in this class that were presented at the meeting but are not included in the papers herein. *CommonObjects* provides stronger encapsulation for inherited types than these other languages, restricting access of a subtype to its supertypes to the same abstract interface as that presented to users. Snyder examines language design issues to realize such abstract interfaces. For example, multiple inheritance from two classes with similarly-named instance variables results in separate copies of such variables with abstract interfaces but only a single copy with non-abstract interfaces. This paper not only presents the features of *CommonObjects* but also a comparison with corresponding features in *CommonLoops*, *Trellis/Owl*, and *C++*.

Kahn indicates how *Concurrent Prolog (CP)*, although not itself object-oriented, may serve as a target language for object-oriented source languages. The semantics of *CP* are described in a way that makes clear the relation to object-oriented concepts. Atomic formulae in *CP* are processes with a literal semantics close to that of actors. They are created for processing just a single message and then reduce to another process. If we think of the sequence of processes that handle a stream as a single process and the stream as a state variable shared among these processes, we get something similar to a traditional object. The transformation is similar to that which transforms tail recursion for a sequence of recursive processes to iteration of a single process with shared state variables. The preprocessor *Vulcan* allows the user to specify programs in terms of the notation of object-oriented programming using classes and methods and translates such programs into programs in *CP*. The source language for *Vulcan* is versatile, supporting concurrent objects, and message passing that is more general than the call/return mechanisms of *Smalltalk* and *Loops*. This paper is significant both for its specific realization of a mapping from object-oriented to concurrent logic programming and for the insights that it gives concerning the nature of the mapping.

Tokoro and Ishikawa examine the integration of object-oriented, concurrent, knowledge-representation features in *Orient84/K*. Objects consist of a behavior part that specifies a collection of methods, a knowledge-based part that has *Prolog*-style rewrite rules, and a monitor part that controls the concurrent behavior of the object. The behavior part supports inheritance for methods. The monitor part includes facilities for event-based (demon) invocation, for protection by checking authorization of incoming messages, and for supervising a variety of message-handling protocols. A version of *Orient84/K* has been running on the VAX-11 and Sun-2 systems since 1984. This project represents a significant contribution to the integration of object-oriented, concurrent, and logic programming concepts in a single system.

GARDEN is an object-oriented language-independent graphical programming environment that supports programming in terms of pictures and multiple views. Object-orientation provides a basis for graphical display and for the support of operations on display objects. *GARDEN* is strongly-typed with multiple inheritance, light-weight processes, and database support for graphical objects. Evaluation of objects is performed by a simple execution algorithm based on the idea that the evaluation strategy for an object is stored in the object itself. The object-oriented database supports transactions, two-phase locking, garbage collection for temporary objects, and editors for constructing graphical views. A preliminary version has been implemented on the Sun, Apollo, and MicroVAX computers. Reiss explores a different aspect of object-oriented programming from the previously

discussed papers. Graphical representation and multiple views may well be a dominant feature of next-generation object-oriented systems.

Actor languages emphasize message passing and communication while object-oriented languages emphasize conceptual organization of information about application domains by classification and inheritance. Agha defines actors in terms of their ability to communicate with other actors, to create new actors, and to specify a replacement which will accept the next communication to that actor. The *Act* programming language is defined and is illustrated by a detailed presentation of a factorial program with emphasis on how a high degree of evaluation concurrency may be realized. This paper contributes to an understanding of the pure message passing paradigm represented by actors.

The actor model may either be combined with inheritance to realize object-oriented programming or may serve as the target language of a compiler or preprocessor that maps languages with inheritance into an actor language where the inheritance is realized in terms of more primitive concepts. de Jong takes the object-oriented language *Scripter* with synchronous message passing and maps it into the actor formalism with asynchronous message passing. These goals are similar to those of the previously described preprocessor for *CP* but the source and target languages are different. Another difference is the concern with optimization; both time optimization by minimizing the number of message sends and space optimization by minimizing the number of customers. The replacement of tail recursion associated with a sequence of incarnations of an actor by sharing of a single actor plays a key role in the mapping and optimization process just as for *CP*. This paper contributes further to the understanding of the relation between object-oriented and purely functional message passing systems of computation.

Concepts in Object-Oriented Programming

All of the papers discussed so far are based on prototype implementations. The remaining papers are concerned with concepts not related to any particular implementation.

Generalized objects are so called because they support direct inheritance at the level of objects without any associated concept of type. This allows us to separate the notion of inheritance from that of type and to examine its properties independently. When a generalized object with direct inheritance receives a *request* message for execution of an operation it executes it directly if it is locally defined; otherwise it sends an *inherits* message to an object that depends on the requestor and the operation requested. Inherits messages may be treated differently from request messages since they are passive requests for copies of a code body that do not modify any shared variables. Thus objects can handle an arbitrary number of inherits messages concurrently but only a single request message. Concurrent objects can implement inherits messages as asynchronous calls while request messages are implemented as synchronous (remote procedure) calls. Types can in principle be reintroduced as special kinds of objects that are repositories for methods. However, doing away with the distinction between types and instances is analogous to doing away with the distinction between sets and elements in mathematics. Nguyen and Hailpern contribute to the clear formulation of issues that arise when we replace type inheritance by direct object inheritance.

Strom examines object and process paradigms and concludes that the process model better supports very large systems where global perspective is inappropriate. Object-oriented systems consist of objects with local state and interfaces of operations that communicate by passing messages. They impose a global class structure on objects by inheritance hierarchies. The process model is similar in its support of interacting objects with state and abstract interfaces but differs in that global system structure is replaced by dynamically reconfigurable local structure. Processes may have different interfaces to different clients. Port interfaces consists of "plugs" associated with an *output port* that are dynamically connected to sockets of compatible type associated with an *input port*. The unit of resource sharing is the object in the object model and is the port (rather than the process) in the process model. Types are associated with objects in the object model but with port variables (rather than with processes) in the process model. A given port type may be implemented in different ways

in different processes. For example a type which provides resources for personal transportation may be implemented as a car in a process which also has a car maintenance port or as a horse in a process which also has a hay port. Whereas object-oriented inheritance models structural relations among classes of objects by types, thereby making it static, the process model models such structural relations dynamically by port interconnections among objects, and reduces the scope of the concept of type. In some respects this is similar to the generalized object model where inheritance is dissociated from type and associated directly with objects. This paper contributes to an understanding of the role of types, of structuring mechanisms, and of local versus global perspectives in systems for programming in the large.

Hendler examines the role of *mixins* in flavors-style object-oriented systems. *Mixins* are packages of functionality that cannot be instantiated by themselves but can enrich or enhance an existing flavor. A clearer differentiation between *flavors* (types) and *mixins* (enhancements) is proposed that allows the functionality of *mixins* to be added only to instances and not to types. Thus if Person is a flavor and male is a flavor that inherits from Person and Doctor and Lawyer are *mixins* then Male-Doctor does not exist as a flavor. Instances of male doctors can be created by creating instances of Male which inherit the mixin Doctor. This allows us to model changes in the status of an object such as a person receiving a PhD or an MD without necessarily changing the type of the object. It provides a factorization of attributes of an object into static ones associated with its type and dynamic ones associated with enhancement. This factorization may also be viewed as a refinement of the notion of generalized objects which partitions attributes of objects into those inherited through the type mechanism and those inherited directly at the level of instances. This paper complements and extends both the generalized object model and the process model in its approach to dynamically changing the attributes and functionality of objects.

Borgida defines an Information System to be a computer system for modelling the real world and recommends that modelling languages for Information Systems should be object-oriented. He then makes the case for exception instances and exception subclasses as a mechanism for extending the versatility of object types to unanticipated new situations. His example of *building* as a subclass of *real estate* which is initially defined for US buildings and must later be adapted for the rare case of foreign real estate with unusual currency and addresses is a good one. Thus a Canadian house could be treated as an exception instance or, if warranted by a sufficient number of instances, as an instance of the exception subclass *Canadian Houses*, which is not behaviorally compatible with the parent class *real estate* because of redefined currency and address fields. An implementation in terms of traditional programming language exception handling mechanisms is proposed. Exception instances are an extension of abstract data types that provide flexibility in handling instances with exceptional attributes. Exception classes similarly extend object-oriented inheritance. This paper may also be viewed as an approach to injecting dynamic structure into static type inheritance hierarchies. Hendler's dynamic inheritance of *mixins* at the level of instances is potentially a mechanism for exception instances. Borgida's approach differs from Hendler's paper in considering dynamics at the level of both instances and types and in using exceptions rather than enhancement as the motivating reason for introducing dynamic flexibility into type and instance definition schemes.

Zdonik develops a technique for maintaining the consistency of objects in an object-oriented database with changing types. He presents a solution that reduces the recoding and conversion that is necessary when a type definition is changed. He distinguishes between reader's and writer's problems in accessing the properties of an object created under one version of a type from a program and written using a different version of that type. A program that reads a property value from an object may receive a value that it is not prepared to handle. A program that writes a property value to an object may write a value that the object is not prepared to handle. In order to resolve these problem, versions of a type are grouped into *version sets* and a *version set interface* is defined which is the union of all operations, properties, and constraints of versions in the set. Version handlers are defined for each version of each type which specify the action to be performed when illegal (inconsistent) actions are encountered. A version handler is a special exception handler. This approach does not auto-

matically handle type inconsistencies but instead provides a system hook and a method for programmer's to reintroduce consistency when types are changed.

Nygaard's paper succinctly summarizes the evolution of the his thoughts since the development of *Simula*. He defines *informatics* as spanning the information aspects of phenomena in nature and society. *Processes* are important objects of study in informatics and have the three qualities of *substance, measurable properties*, and *transformations*. Substance includes objects, files, records, variables etc. The measurable properties of substance are primarily the values of variables and structures. Transformations may modify both the measurable properties and substance of a process. These notions are a basis for defining the concepts of *state, attribute, reference, quantity*, and *pattern*. The notion of a *system* is defined both in general and in the context of object-oriented programming. *Declarations* for *types, classes*, and *procedures* may be unified by *pattern declarations*, as in the *Beta* programming language. Five different kinds of hierarchies are distinguished; namely action, value, substance, structure, and program execution hierarchies. Three perspectives on programming are identified; namely function oriented, object oriented, and constraint oriented, and Nygaard suggests that all three should be supported in any new general-purpose programming language. Many of these ideas are concretely reflected in the programming language *Beta*, being developed jointly with Madsen.

Madsen contrasts *Simula* and *Beta*, which combine object-orientation and block structure with *Smalltalk* which does not and argues for the importance of block structure as a mechanism for providing locality of scope in object-oriented languages. This is illustrated with a number of examples, including tokens defined relative to a grammar, removing operator-operand asymmetry in *Simula* and *Smalltalk*, defining mutually-dependent classes, defining prototypes for classes of similar instances, and simulation of *Smalltalk* metaclasses. This paper contributes to the understanding of interactions between object-orientation and the related but orthogonal notion of block structure.

Ossher considers the specification, documentation, and representation of large, layered programs by a *grid method* which factors different kinds of structure along orthogonal dimensions. Layered programs arise when there are multiple layers of abstraction, multiple layers of security, and multiple views. Interactions within a layer are generally more frequent than between layers. The structure corresponding to different kinds of layering can be partitioned along different dimensions of a grid. The approach is illustrated for a simple data abstraction example where interactions among layers of abstraction are separated from interactions among object hierarchies. It has been used for larger programs including the analysis and documentation of *Scribe*. The paper contributes to an understanding of the relation among different kinds of program structure and of mechanisms for factoring different kinds of structure into different components.

Theory and Models

Goguen and Meseguer describe a language for integrating functional and object-oriented programming called *FOOPS*. They illustrate *FOOPS* with a bank account example and present an operational and logical semantics for *FOOPS*. *FOOPS*, like *OBJ2*, supports the definition of abstract data types that can inherit other types in three successively more permissive protection modes called protecting, extending, and using. Protecting allows the behavior of the imported type to be used without modification. Extending allows the behavior of the imported type to be extended provided the elements of the imported type retain their identity and their behavior is not compromised. Using allows the behavior of the imported type to be both extended and compromised by a many to one mapping. The paper contributes to an understanding of the relation between functional, logical, and object-oriented programming and to the specification of object-oriented concepts in terms of an algebraic operational notation such as *OBJ2*.

Bruce and Wegner examine the consequences of defining the notion of subtype as a binary relation between algebras. They develop a *weakest* notion of subtype that includes *Int is a subtype of Real, Int(1..10) is a subtype of Int* and *Student is a subtype of Person*. These three relations are respectively

called *Isomorphic Copy, Subset, and Object-Oriented.* Particular notions of subset are special cases of this general relation. Object-Oriented is of particular interest and is defined in terms of bounded quantification in the second order polymorphic lambda calculus. This paper contributes to our understanding of the modelling of object-oriented types by algebras and the lambda calculus.

Wegner defines object-oriented systems in terms of their classification rather than their communication characteristics because they are prescriptive in their classification requirements and permissive in their message passing requirements. Type inheritance extends the notion of classification from flat to tree-structured hierarchies in much the same way that Darwin extended Linnaean classification methods aimed at identification to evolutionary classification methods aimed at explanation. Object-oriented type systems determine a calculus of classes weaker than calculi for computing with values that can be modelled either algebraically or by methods of the second order lambda calculus. The paper examines the relation between algebra and calculi, the object-oriented notion of type, limitations on object-oriented specification, global versus local system perspectives, and realist versus intuitionist models of type. It tries to blend the qualitative characterization of object-oriented programming as a classification paradigm with technical understanding mathematical models that underlie object-oriented systems.

Summary

Collectively, these papers provide a snapshot of current research in object-oriented programming that touches on many interesting research issues. In preparing this overview we discovered interesting relations among papers, such as the fact that compiling from an object-oriented language into *CP* and actors are quite similar in their objective, and that three seemingly different papers were concerned with different aspects of whether inheritance should be at the level of types or instances.

Object-oriented programming clearly holds promise as a framework for programming in the large that can be extended to concurrency, databases, and knowledge bases. This workshop and this collection of papers contribute to an understanding of what needs to be done to turn this promise into a reality.