Introducing Objects and Parallelism to an Imperative Programming Language

A. A. Radenski

Department of Computer Science, WSSU, P. O. Box 13069

Winston-Salem, North Carolina 27110, U.S.A.

E-mail: radenski@uncecs.edu

Abstract

The problem of enhancing objects with parallelism has been in the focus of numerous research projects in the recent years, but a satisfactory and commonly accepted solution has not appeared yet. A major problematic points seems to be providing *inheritance* for parallel objects. The general objective of this paper is to contribute to a better understanding of the language design issues in the area of parallel object-oriented programming (OOP), and in particular, to design a framework for *parallel OOP with multiple inheritance*. What makes our proposed framework different from the other parallel OOP languages is its easy-to-use and efficient multiple inheritance *for parallel objects*. Our framework is *easy-to-use* because it is designed as a minimal parallel and OOP enhancement of the imperative programming paradigm - a paradigm which is relatively simple, very popular, and well understood. It is *efficient* for the same reasons, and because the implementation of dynamic binding in our proposed multiple inheritance scheme does not require run-time method tables. Dozens of known serial and parallel OOP languages employ run-time method tables which may impose significant space and time overhead, particularly in a parallel environment.

1 Introduction

Objective. Possible models for the integration of the OOP paradigm and the parallel programming paradigm have been in the focus of numerous research projects in the recent years. Language designers believe that providing objects with parallelism will substantially increase the power of the OOP paradigm. However, despite of the observation that objects seem to blend naturally and well with concurrency, the transition from serial to parallel OOP languages turns out to be a complex problem. It has been found [19] that popular parallel OOP languages are "... often compromised in important areas, including inheritance capability, efficiency, ease of use, and degree of parallel activity... Unless concurrency, synchronization, and communication are carefully integrated, a parallel object-oriented language can be inefficient and difficult to use." One of the most difficult issues seems to be providing practical *inheritance for parallel objects* [8].

The general objective of this paper is to contribute to a better understanding of the relationship between objects, parallelism, and inheritance. Our particular goal is to *define a framework for OOP which provides easy-to-use and efficient multiple inheritance for parallel objects*.

Approach. Some of the most successful OOP languages are *extensions* of statically typed imperative languages. Well-known examples in this respect are the languages from the C++ family and the object-oriented versions of Pascal, which extend C and Pascal with classes, inheritance, dynamic binding and other mechanisms needed for object-oriented programming. Other examples are given by Ada 9X and Oberon, which can be considered as object-oriented enhancements of such typical statically typed languages as Ada and Modula-2.

The transition of programmers from a sequential language to its parallel object-oriented enhancement would be easier and less frustrating, if the syntactical, semantical and terminological

changes made for the enhancement are only minimal. Designing a parallel OOP framework as a minimal enhancement of a sequential language will facilitate the reuse of existing serial software, while designing an entirely new language will be a serious obstacle to such reuse. A parallel enhancement of a serial language is easier to implement than a language designed from scratch. For these reasons, we adopt the following approach to the integration of the object-orientation and parallelism: *Find the smallest possible enhancement of the imperative programming paradigm which can support the needs of the parallel object-oriented programming paradigm with multiple inheritance*.

Minimality of extensions was a major criterion in the transition from Modula-2 to its sequential object-oriented descendant Oberon [18]. Wirth did not introduce the concepts of class and method in Oberon, but used such traditional languages features as records and procedures instead. Because traditional record types do not support inheritance, Wirth developed the mechanism of record type extension to implement single-class inheritance [16]. The object-orientation of Oberon [17], Oberon-2 [11] and Ada 9X [14] is based on the notion of type extension or its modifications. Unfortunately, Oberon does not provide support for parallelism and multiple inheritance. A major disadvantage of Ada 9X is its high complexity. Like Oberon, Ada 9X directly supports only single-class inheritance for serial objects.

The design of our framework for parallel OOP is guided by three main theses. The *first thesis* is that record types can be enhanced to contain bodies for the purpose of encapsulation and information hiding. Record types enhanced with bodies are an adequate substitute for classes. Such record types provide support for dynamic method overriding and time-efficient dynamic binding. The *second thesis* is that the concept of record extension can be redefined in terms of record nesting and use of incomplete designators. The new concept provides easy-to-use and

efficient multiple inheritance. The *third thesis* is that separately executable procedures with synchronization guards can adequately support parallelism in a OOP language based on extensible record types. These three theses are developed in details in Section 2. Section 3 discusses work performed by others in the area of parallel OOP and outlines some advantages of our approach.

Our model for parallel OOP is specified in an Oberon-like syntax for concreteness and clarity but applies to diverse sequential languages which support record or structure types.

Contributions. What makes our proposed framework different from the other parallel OOP languages is its easy-to-use and efficient multiple inheritance for parallel objects. Our framework is easy-to-use because it is designed as a minimal parallel and OOP enhancement of the imperative programming paradigm - a paradigm which is relatively simple, very popular, and well understood. Our framework is efficient for the same reasons, and because the implementation of dynamic binding in our proposed multiple inheritance scheme does not require run-time method tables. Typically, dozens of known serial and parallel OOP languages employ run-time method tables which may impose significant space and time overhead, particularly in a parallel environment [19]. We know of only one OOP language, Oberon [17], which inheritance scheme does not require run-time method tables; however, Oberon is not parallel and does not support multiple inheritance.

2 A Framework for Parallel OOP with Multiple Inheritance

Classes as Types. In our parallel OOP framework, a *class* is implemented as a pointer type bound to a record type with procedure component(s), and an *object* is a dynamic variable of such record type. A *method* is a procedure which belongs to a dynamic record and manipulates data components of the same record. Such a procedure must be explicitly connected to the record type and considered local to that type. We employ a special linguistic construct, the *record body*, to relate procedures to record types.

For example, Figure 1 defines *Class* as a pointer type bound to a class description record, *ClassDesc*. *Method* is specified as a procedure in the class description body. An object is a dynamic record variable, *obj^*.

Note that the dynamic record *obj^* has *public* components (*data*, *method*), and *private* components (*hiddenData*, procedure *Method*). By definition, implicit dereferencing is used to denote public components of dynamic records. Thus, *obj.data* is a correct designator for *obj^.data*. On the other end, *hiddenData* is not visible out of the body and the designator *obj.hiddenData* is illegal.

The statements from the body of the dynamic variable *obj^* are executed when the variable is created by a call of the predeclared procedure *NEW*. Thus, the call *NEW (obj)* allocates memory for *obj^* and then executes the statements *obj.method := Method; hiddenData := 0* from the record body. These statements install *Method* and initialize the *hiddenData* record component. (A *static* record variable is created when the block containing its declaration is activated.)

A *message* is be straightforwardly implemented as a call of a procedure field, as for instance *obj.method(0)*.

As shown by the above example, a *record type* consists of a public part, and optionally, a body. A *record public part* is a collection of variable declarations, including *variables of procedure types*; a record public part may also contain *constant procedure declarations*. The *public components* of the record type are traditionally called *fields*. As with regular records, the record public part determines the name and the type of each field. A *record body* may include declarations of constants, types, variables, and procedures, all called *private components*. A record body is also allowed to contain statements for the purpose of assigning initial values to the record variables (public and private).

By definition, the *scope* of an identifier which denotes a public component extends from the point of its declaration to the end of the public part of the record, and includes the whole body, if any. Such an identifier is also visible within component designators. An identifier of a private component is not visible outside of the body. Besides, a variable identifier declared outside of a record type is not visible in its body, unless it is explicitly *imported* by the body or by the record public part.

Single and Multiple Inheritance as Type Extension. The concept of *type extension*, as defined in [16], applies to record and pointer types and is used to implement what is known as single class inheritance in standard OOP terminology. We have *redefined* record extension in terms of regular record nesting and incomplete designators. Our approach to record extension is syntactically simpler but supports *multiple class inheritance*.

By definition [13], a record type R_{ext} *directly extends* a record type R_{base}, if R_{ext} has exactly one component of type R_{base}. Further, R_{ext} *extends* a type R_{base}, if it equals R_{base}, or if it directly extends an extension of R_{base}. As specified in [16], type extension applies to pointer types as well: a

pointer P_{ext} to R_{ext} is said to be an extension of a pointer P_{base} to R_{base}.

Subclasses can be defined as extended types. Consider, for example, the declarations of a subclass and its description in Figure 2. *SubclassDesc* directly extends *ClassDesc* with component *newData*. *ClassDesc* is a direct base type of *SubclassDesc*. As specified below, nested components of *SubclassDesc* can be referenced by incomplete designators that do not contain the identifier *parent*.

Nested public components can be denoted by *incomplete designators* according to the following rules [12]. (1) It is said that *c* is a *nested component* of a record variable *r*, if *c* is a component of *r*, or *c* is a nested component of some component of *r*. (2) If the record variable *r* does not have a public component *c*, then *r.c* designates a nested component of *r* determined by a unambiguous level-order search among all nested components of *r*. For example (see Figure 2), the incomplete designators *extObj.data* and *extObj.method* are valid and stand for *extObj.parent.data* and *extObj.parent.method* correspondingly.

When the body does not immediately follow the record type declaration, the type declaration must be followed by a *body stub*, as shown on Figure 2.

Our proposed type extension mechanism provides easy-to-use multiple inheritance in the same way it provides single inheritance. Consider, for instance, the declarations of *Class0*, *Class1* and *Class2*:

TYPE *Class0* = RECORD *x: INTEGER* END;

Class1 = RECORD *parent0: Class0; y: REAL* END;

Class2 = RECORD *parent0: Class0; parent1: Class1* END;

Class2 is an extension of both *Class0* and *Class1*. *Class2* inherits *x* from *Class0*, and *parent0*, *y*

from *Class1*. Let *r* be a variable of *Class2*. The following are legal incomplete designators: *r.x* (inherited from *Class0*, same as *r.parent0.x*), *r.parent1.x* (inherited from *Class0* through *Class1*, same as *r.parent1.parent0.x*), *r.y* (inherited from *Class1*, same as *r.parent1.y*). Note that conflicts between inherited fields with the same name are easily resolved with the use of appropriate designators, such as *r.parent1.x*.

Polymorphism. Extended types are defined to be assignment compatible with their base types; variables of such base types are *polymorphic*. (Similar compatibility rules are valid for classes and subclasses in most standard OOP languages.) For instance, *extObj: Subclass* (see Figure 2) is assignment compatible with *obj: Class*. The assignment *obj := extObj* is valid and variable *obj* is polymorphic because it can point to dynamic records of different types. A *type test* can be applied to determine if the actual value of a polymorphic variable belongs to a subclass, for example *obj* IS *Subclass*. If it does, a component of the subclass can be referred to by means of a *type assertion*, such as *obj(Subclass).newData*. The program execution is aborted if a type assertion fails in run-time. The rules for type compatibility, tests and assertions are originally formulated in [17] in the case of single type extension; versions of the same rules apply without difficulties to our proposed concept of type extension.

Dynamic Binding. A subclass can *override* a method inherited from a parent class by means of a simple procedure assignment. For this purpose, the overriding method must be defined and properly installed in the body of the subclass description. Figure 3 implements this approach to method overriding for the subclass defined in Figure 2.

Let *extObj* be a pointer variable of type *Subclass*. When *extObj^* is created by *NEW (extObj)*, the statements from the body of the nested record *extObj.parent* (of type *Class*) is

executed first. In particular, the *Method* defined in the body of *Class* is initially assigned into *extObj.parent.method*. As a next step, the statement from the body of the outer record, *extObj: SubClass*, is executed. The assignment *parent.method := OverridingMethod* replaces inherited *Method* by *OverridingMethod*.

In our proposed parallel OOP framework, variables of procedure types implement what is known as *virtual methods* in standard OOP terminology; such methods are overridden through procedure assignments, as specified in the previous paragraph. Calls of procedure variables, such as *extObj.method(0)*, implements a easy-to-use version of what is referred to as *dynamic binding* in common OOP languages. Alternatively, constant procedures are called by *static binding* and can be only statically overridden in extended records.

Process creation and coordination. We assume that the programmer is interested in specifying parallelism, and that a parallel OOP language should provide explicit constructs for parallelism. Because OOP means programming by modelling, and because real-world objects may exist and do things concurrently, OOP languages should provide explicit support to modelling parallelism. Other researchers prefer to exclude parallelism from the language and use operating system calls instead. A third group adheres to the idea that parallelism should be transparent to the programmer, and that a parallelizing compiler should take the burden of finding and exploiting potential parallelism. While it is possible to combine the three approaches in certain proportions, our framework for parallel OOP supports parallelism explicitly. Parallelism is specified by means of *separate procedures* and *synchronization guards* encapsulated in *separate record types*.

Our parallel OOP framework includes a mechanism for explicit lightweight *process creation* based on the so called *separate procedures*. The statement part of a separate procedure consists of a regular part, and a separate part, as shown on Figure 4. When the separate part is reached, the

execution is "forked": the control is returned to the caller, and the execution of the separate part continues independently; unlike Unix, no memory copying occurs. The process defined by the separate statement part is terminated implicitly at the end of the separate part, or explicitly by a call of predefined procedure TERMINATE. The regular statement part is used by the caller and the callee as a critical region for *communication* through parameters; parameters are inaccessible in the separate statement part.

Conditional synchronization is achieved through *synchronization guards*. Synchronization guards are boolean expressions attached to record components, including procedure components. Any reference to a guarded component is suspended until the synchronization guard evaluates to true. Suspended references are serviced in a non-deterministic fashion.

Consider, for example, the implementation of a *Mailbox* class on Figure 5, with defines two methods, *Put* and *Get*. *Put* deposits a new message in the Maibox, while *Get* removes the oldest one. Messages are kept in a *box* of a limited *size* and are accessed by means of two indexes, *last* and *first*. *Put* deposits a new message at location *last* while *Get* removes the item from location *first*. *Put* and *Get* increment their corresponding indexes by *1* modulo the box *size*, and *count* the number of messages available in the mailbox. Each of the procedures *Put* and *Get* communicates a message with its caller first, then separates for independent execution. The guard $count > 0$ suspends any attempted invocations of Get when the mailbox is empty. Similarly, $count < size$ guards Put from attempts to leave a message in a full mailbox.

Separate procedures and synchronization guards generate and coordinate parallelism; they are permitted only in record types which are explicitly declared as SEPARATE, such as *MailboxDesc* in Figure 5. Separate records represent parallel objects and for them the compiler generates some necessary run-time overhead, such as the transparent semaphore discussed in the

next paragraph. For non-separate records which represent serial records this run-time overhead is not needed as is not generated.

A separate record (i.e., a parallel object) is *busy* if a procedure from the body of the same record is running or suspended, otherwise the record is *available*. A reference to a busy record is suspended until the record becomes available. This rule guarantees the *atomicity* of message passing to parallel objects; besides, it supports the implementation of such traditional *access synchronization* mechanisms as locks, semaphores and monitors. For its implementation, the compiler associates a transparent semaphore with the body of each separate record. The activation of a procedure defined in the body increments the semaphore, while the procedure termination decrements it. For each reference to a separate record, the compiler generates extra-code which eventually suspends the reference until the semaphore becomes zero.

A separate procedure defined in a record body may not call a separate procedure defined in the same body, including itself. This rule limits the number of processes within a parallel object to one - a restriction that has been adopted in the majority of parallel OOP languages for the purpose of simplicity and efficiency.

Type extension applies without restriction to separate record types and provides *single or multiple inheritance for parallel objects*. An example of a single-class inheritance is presented in Figure 6 which demonstrates how the mailbox from Figure 5 can be extended with a new method, *GetAll*. The extended *Mailbox2* uses the inherited method *Get* in order to implement method *GetAll* which reads all messages from the mailbox.

An Example of Multiple Inheritance from Parallel Classes. A uniform support for both single and multiple inheritance is provided by the original concept of type extension defined earlier in this section and illustrated by the definitions of *Class* and *Subclass*, *Mailbox* and *Mailbox2*

(single inheritance), and *Class0, Class1, Class2* (multiple inheritance). Our goal here is to specify an example of multiple inheritance involving parallel classes.

Consider the problem of defining a persistent object such as, for example, a mailbox, a tree, a graph, or a viewer, which can be saved into a specified file before the end of a program execution and later loaded form the file for a subsequent program execution. Despite of how different these objects are, they should be able to *store/load* their current state into/from a specified *file*. It is convenient to factor out the common behavior of diverse persistent objects in a special class, *Persistence*, shown in Figure 7. The virtual methods *store* and *load* are implemented in *Persistence* as simple procedure variables which do not require a run-time method table.

Given the *Persistence* class (Figure 7) and a class of non-persistent objects, such as *Mailbox* (Figure 5), multiple inheritance from both classes can be used to build a sub-class of persistent objects, such as *PersBox*, also shown on Figure 7. Note that *PersBox* provides procedures *Store* and *Load* which implement the virtual methods *store* and *load* inherited from *Persistence*. Each class extended with *Persistence* should provide its own implementation of *save* and *load* because objects of different classes are saved and loaded in *different* ways.

A persistent mailbox, *persBox^*, is created by means of the call *NEW(persBox)*. As defined earlier in this section, this call allocates memory for *persBox^*, then executes the body *MailboxDesc* of its component *mbox*, and finally executes the outer body, *PersBoxDesc*. Both body executions install *Save, Load, Put, Get*. The implementation does this by assigning pointers to these procedures into the corresponding procedure variables. Furthermore, messages to object *persBox^* are calls of procedure variables, such as *persBox.mbox.put(msg); ... persBox.pers.save; ... persBox.pers.load*; all these calls are performed directly through the corresponding pointers,

without any references to a run-time method table. Finally, the same messages can be sent using incomplete designators such as *persBox.put(msg); ... persBox.save; ... persBox.load.* These designators conveniently resemble the notation used for inherited components in conventional OOP languages.

The above example illustrates one significant advantage of our framework for parallel OOP: since virtual methods are implemented as procedure variables, their calls are as simple as procedure calls; such calls do not use a run-time method table. The inheritance mechanisms of known OOP languages require run-time method tables which deteriorate the time and space efficiency of the implementation of multiple inheritance, especially in a parallel environment.

3 Related Work and Advantages of our Approach

The main advantage of our framework for parallel OOP is that it provides easy-to-use and efficient multiple inheritance for parallel objects, while most known proposals for parallel OOP fail partially or completely to amalgamate multiple inheritance with parallelism. Although our model is specified in terms of Oberon, it is applicable to any sequential language which supports record or structure types.

The idea to use record types for the implementation of classes is adopted in Oberon[17], Oberon-2[11] and Ada 9X [14]. These languages, however, provide only single-class inheritance, while our framework supports easy-to-use multiple inheritance. As a language, Oberon does not support concurrency. Parallel objects in Ada 9X can be implemented by means of either tasks, or protected types. Task-based parallelism in Ada 9X seriously violates inheritance [9], while protected types in the same language do not permit inheritance at all.

Dynamic binding in our framework for parallel OOP is as efficient as a procedure invocation and *does not involve run-time method tables.* In contrast, dynamic binding in most

known OOP languages is based on run-time method tables and may cause considerable run-time overhead when multiple inheritance is involved. For example, the multiple inheritance scheme in C++ has caused 50% increase of the method table compared to the older single inheritance implementation [15]. Implementing efficient run-time method tables in a distributed environment is a complex task.

Dynamic binding through direct procedure invocation, without method tables, was proposed originally by Wirth and implemented in Oberon. In the framework developed by Wirth, a method can be overridden by changing the procedure assigned to a field in an extended record. Unfortunately, the overriding method must have the same type as the overridden one. Thus, the overriding method cannot operate (directly) on the extended components and may apply only to the inherited ones [12]. In order to solve this problem, Moessenbock expanded Oberon with special type-bound procedures that can be connected to a data type explicitly [11]. Type-bound procedures can be overridden for extended types; however, they are called indirectly through method tables which slows down the execution. Encapsulation is seriously violated by type-bound procedures, because they can be declared at arbitrary places in the scope of that type [13]. Information hiding is somewhat violated, since data fields of a record type are always visible in the whole module containing the record type. Similar problematic points are observed in other known OOP languages, such as C++ for example. In contrast, our record types enhanced with bodies offer time-efficient dynamic binding, excellent encapsulation and information hiding. Objects represented by extensible records do not require run-time method tables, which facilitates their implementation and migration in distributed environment.

Many researchers expand existing serial strictly-typed OOP languages with parallelism. The best known language extensions of C++, such as COOL [4] and Parmacs [1], do not furnish inheritance for parallel objects. Several comprehensive parallel enhancements of Eiffel, such as

CEiffel [8], Eiffel// [3], and the concurrent extension of Eiffel based on method guards [10], ensure inheritance for parallel objects but do not provide satisfactory reusability of synchronization code, particularly in the case of multiple inheritance. In contrast, parallel objects in our OOP framework inherit through type extension as successfully as serial ones. Even synchronization guards can be inherited and reused, because they can be specified as methods in their defining classes and inherited in subclasses.

Some researchers add parallelism to existing serial languages by means of external libraries that manage and synchronize processes; the underlying serial languages remain more or less unchanged. For example, Presto [2] extends C++ with a library of parallel programming primitives but does not furnish inheritance for parallel objects. Another project of a similar type extends Eifell with a special CONCURRENCY class [6]; likewise, Concurrent Oberon [7] expands the Oberon operating system with a set of procedures for thread programming. The main component of Charm++ [5], a recent parallel enhancement of C++ that claims to support multiple inheritance for parallel objects, is a library of functions which provides support for parallel execution. Charm++ messages to parallel objects are implemented as packets of data (C++ structures), rather than as method invocations; the latter would have the advantage to be more efficient and reliable if they were available.

The chief advantage of the external parallel programming library primitives discussed in the above paragraph is that they provide very flexible access to low level data and hardware resources. The disadvantage of this approach is that such access cannot be controlled by the compiler and may result in unreliable programs. The undisciplined and unstructured use of low-level parallelism primitives leads to unstructured parallel programs; such programs are hard to debug, maintain and modify. Therefore, low-level library primitives can be as harmful for parallel programs as is the goto statement for sequential ones. In contrast, our proposed higher-level linguistic constructs such

as separate procedures and synchronization guards are intended to stimulate disciplined, well-structured and more reliable parallel programming.

Finally, some earlier proposals for completely new languages for parallel OOP can be found in [19]. Such proposals are skipped from our overview for brevity and because our effort is aimed at adding parallelism to *existing* languages rather than designing *new* ones.

4 Conclusions

This paper defines a minimal enhancement of the imperative programming paradigm which supports the principal features of (1) the OOP paradigm, such as inheritance, dynamic binding, polymorphism, information hiding and encapsulation, and (2) the parallel programming paradigm, such as process creation and coordination. Our proposed enhancement is based on the following main features: generalized records with bodies, record extension through record nesting and incomplete designators, access guards, and separate procedures. The enhancement is *minimal* because none of its components is redundant. Indeed, removing record bodies from our framework would seriously violate information hiding and encapsulation; removing record extension would invalidate inheritance and polymorphism, removing separate procedures would make the language serial, and removing access guards would practically eliminate synchronization.

From the programming language user perspective, our approach combines (1) the extensive experience with the imperative programming paradigm, and its traditional popularity, with (2) the power of the OOP paradigm, and with (3) the flexibility and efficiency of parallelism. From the language designer perspective, the main advantage of our framework for parallel OOP is that it provides easy-to-use and efficient multiple inheritance for parallel objects, as advocated in the previous sections.

References

1. Beck B. Shared Memory Parallel Programming in C++, *IEEE Software*, 7, No 4 (July), 1990, 38-48.
2. Bershad B., E. Lazowska, H. Levi. Presto: A System for Object-Oriented Parallel Programming, *Software - Practice and Experience*, 18, No 8 (Aug.), 1988, 713-732.
3. Caromel D. Toward a Method of Object-Oriented Concurrent Programming, *Communications of the ACM*, 36, No 9 (Sep.), 1993, 90-10.
4. Chandra R., A. Gupta, J. Hennessy. COOL: An Object-Based Language for Parallel Programming, *IEEE Computer*, 27, No 8 (Aug.), 1994, 13-26.
5. Kale L., S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++, OOPSLA'93, *ACM Sigplan Notices*, 28, No 10 (Oct.), 1993, 91-108.
6. Karaorman M., J. Bruno. Introducing Concurrency to a Sequential Lamguage, *Communications of the ACM*, 36, No 9 (Sep.), 1993, 103-116.
7. Lalis S., B. Sanders. Adding Concurrency to the Oberon System, *Proc. Intern. Conf. on Programming Languages and Comp. Architectures*, Zurich. Springer-Verlag, 1994, 328-344.
8. Lohr K. P. Concurrency Annotations for Reusable Software, *Communications of the ACM*, 36, No 9 (Sep.), 1993, 81-89.
9. Matsuoka S., A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, *Research Directions in Concurrent Object-Oriented Programming*, B. Sriver, P. Wegner (editors), The MIT Press, 1993, 107-150.

10. Meyer B. Systematic Concurrent Object-Oriented Programming, *Communications of the ACM*, 36, No 9 (Sep.), 1993, 56-80.

11. Mossenbock H. *Object-Oriented Programming in Oberon-2*, ACM Press, 1993.

12. Radenski A. Is Oberon as Simple as Possible? A Smaller Language Based on the Concept of Module Type, *Proc. Intern. Conf. on Programming Languages and Computer Architectures*, Zurich. Springer-Verlag, 1994, 298-312.

13. Radenski A. Type Extensions, and Their Support for Object-Oriented Programming With Multiple Inheritance, *Proc. 32nd Annual ACM Southeast Conf.*, Tuscaloosa, Alabama, ACM Press, 1994, 241-246.

14. Taft S. T. Ada 9X: A Technical Summary, *Communications of the ACM*, 35, No 11 (Nov.), 1992, 77-84.

15. Stroustrup B. The Evolution of C++: 1985 to 1989, *The Evolution of C++: Language Design in the Marketplace of Ideas*, Waldo J. (ed.), The MIT Press, 1993, 13-52.

16. Wirth N. Type Extensions, *ACM Transactions on Programming Languages and Systems*, 10, 1987, 204-214.

17. Wirth N. The Programming Language Oberon, *Software - Practice and Experience*, 18 (July), 1988, 671-690.

18. Wirth N. From Modula to Oberon, *Software - Practice and Experience*, 18, No 7 (July), 1988, 671-670.

19. Wyatt B., K. Kavi, S. Hufnagel. Parallelism in Object-Oriented Languages: A Survey, *IEEE Software*, 9, No 6 (Nov.), 1992, 56-66.

```
TYPE Class = POINTER TO ClassDesc;
 ClassDesc = RECORD
  data: INTEGER;
  method: PROCEDURE(v: INTEGER);
 END (*ClassDesc*);

VAR obj: Class;
```

```
BODY ClassDesc;
 hiddenData: INTEGER;
 PROCEDURE Method(v: INTEGER);
 BEGIN data := v END Method;
BEGIN
 method := Method; hiddenData := 0
END ClassDesc;
```

Figure 1 Class, Object, and Method Implementation

```
TYPE Subclass = POINTER TO SubclassDesc;
 SubclassDesc = RECORD
  parent: ClassDesc;
  newData: INTEGER
 END (*SubclassDesc*);
BODY SubclassDesc; (*stub*)

VAR extObj: Subclass;
```

Figure 2 Subclass Implementation

```
SEPARATE PROCEDURE Name Parameters;
BEGIN
 regular statement part
SEPARATE
 separate statement part
END Name;
```

Figure 4 Form of Separate Procedures

```
TYPE Mailbox2 = POINTER TO MboxDesc2;
MboxDesc2 = SEPARATE RECORD
 parent: MailboxDesc;
 getAll: PROCEDURE (VAR msg: ARRAY OF MsgType; VAR num: INTEGER)
  WHEN parent.count > 0;
END; (* MboxDesc2 *)

BODY MboxDesc2;
 PROCEDURE GetAll (VAR msg: ARRAY OF MsgType; VAR num: INTEGER);
 BEGIN num := 0;
  REPEAT parent.Get(msg[num]); INC(num) UNTIL parent.count = 0
 END GetAll;
BEGIN getAll := GetAll END MboxDesc2;
```

Figure 6 Inheritance From parallel Objects as Type Extension: MboxDesc2 Extends MailboxDesc

```
TYPE Mailbox = POINTER TO MailboxDesc;
MailboxDesc = SEPARATE RECORD
  count: INTEGER;
  put: SEPARATE PROCEDURE (msg: MsgType) WHEN count < size;
  get: SEPARATE PROCEDURE (VAR msg: MsgType) WHEN count > 0;
END; (* MailboxDesc *)

BODY MailboxDesc;
  VAR box: ARRAY size OF MsgType;     last, first: INTEGER;

  PROCEDURE Put (msg: MsgType); BEGIN store[last] := msg;
  SEPARATE last := (last + 1) MOD size; INC(count); END Put;

  PROCEDURE Get (VAR msg: MsgType); BEGIN msg := store[first];
  SEPARATE first := (first + 1) MOD size; DEC(count); END Get;

BEGIN count :=0; last :=0; first :=0; put := Put; get := Get;
END MailboxDesc;
```

Figure 5 Implementation of Parallel Mailbox

```
TYPE Persistence = POINTER TO PersistenceDesc;
PersistenceDesc = SEPARATE RECORD
  file: STRING; (* file name *)
  save, load: SEPARATE PROCEDURE;
END; (* PersistenceDesc *)

TYPE PersBox = POINTER TO PersBoxDesc;
PersBoxDesc = SEPARATE RECORD
  pers: PersistenceDesc;
  mbox: MailboxDesc;
END; (* PersBoxDesc *)
VAR persBox : PersBox;

BODY PersBoxDesc;
  PROCEDURE Save;
  BEGIN SEPARATE (* ...save mbox into file *) END Save;
  PROCEDURE Load;
  BEGIN SEPARATE (* ...load mbox from file *) END Load;
BEGIN pers.save := Save; pers.load := Load; END PersBoxDesc;
```

Figure 7 Class Persistence, and Its Use in the Implementation of Persistent Mailbox

```
BODY SubclassDesc;
  PROCEDURE OverridingMethod (v: INTEGER);
  BEGIN parent.data := v; newData := v
  END OverridingMethod;
BEGIN parent.method := OverridingMethod
END SubclassDesc;
```

Figure 3 Method overriding