Inheritance and Subtyping in a Parallel Object-Oriented Language

Pierre America
Philips Research Laboratories
Eindhoven, the Netherlands

Abstract

We have investigated the concepts of inheritance and subtyping in order to integrate them in a parallel object-oriented language. In doing so, we have concluded that inheritance and subtyping are two *different* concepts, which should not be confused in any object-oriented language (be it parallel or sequential). Inheritance takes place on the implementation level of classes, and it is a convenient mechanism for code sharing. It can be supported, for example, by introducing inheritance packages into a programming language. Subtyping deals with the message interface of objects, and it leads to a conceptual hierarchy based on behavioural specialization. Subtyping should take place on the basis of specifications of the external behaviour of objects, and as much as possible of these specifications should be formal. Some specific problems with introducing these two concepts into a *parallel* language are also discussed.

1 Introduction

ESPRIT project 415 aims at the development of new architectures and languages to speed up applications in the area of artificial intelligence by the use of large-scale parallelism. Subproject A of this large project is mainly carried out at Philips Research Laboratories in Eindhoven, the Netherlands. Here object-oriented programming has been chosen as the approach to follow towards this goal. A parallel architecture, called DOOM (Decentralized Object-Oriented Machine), is being developed [14], and a language, POOL (Parallel Object-Oriented Language), is being designed to program this machine. For more details on the language than can be given here, we refer to [2].

This paper is motivated by the wish to enrich the language POOL with a mechanism commonly called *inheritance*. Inheritance is regarded by many people as the hallmark of object-orientedness in programming languages [7]. We do not agree with this view, and argue that the essence of object-oriented programming lies somewhere else (see section 2.1). Nevertheless, inheritance is a very important concept and a useful mechanism to structure large systems. Therefore it would be advantageous for any object-oriented language to have a nicely integrated form of inheritance.

In section 2 we introduce the language POOL, together with the concepts on which it is based. This serves as a reference point for the rest of our discussion. The concept of inheritance and a related concept, subtyping, are introduced in section 3, where we also show that the distinction between the two is too important to confuse them, as is done in most existing object-oriented languages. For both concepts we shall indicate ways of dealing with them in an object-oriented language. Then, in section 4 we discuss the specific problems and possibilities of integrating these concepts in a parallel language like POOL.

This work is carried out in the context of ESPRIT project 415: "Parallel Architectures and Languages for Advanced Information Processing — a VLSI-directed approach".

2 The language POOL

In this section we present an overview of the language POOL, in order to introduce the concepts on which it is based, and to indicate the objectives that directed its design.

2.1 Object-oriented programming

The basic notion in object-oriented programming is of course the notion of an *object*. An object is an entity that contains some internal data, and has several procedures associated with it (these are called *methods*, following the object-oriented jargon). In general, objects are very dynamic entities: They can be created dynamically and their local data can change during their lifetime.

The important point is that the data in an object cannot be accessed directly by the outside world (i.e., by other objects): the only way of interacting with an object is sending it a *message*. A message is a request for the receiver to execute one of its methods, and such a method *can* access the local data of the object it belongs to. The fact that an object can only be manipulated through this fixed and controlled interface provided by its methods gives rise to a very powerful protection mechanism, which protects the local data of each object against uncontrolled access from every other one. This mechanism also provides a separation between the *implementation* of an object (its set of variables and the code of the methods) and its external behaviour (the way it responds to message). In this way the programming technique of abstract data types can be used in object-oriented programming.

In our opinion this protection mechanism based on objects is the basic and essential principle of object-oriented programming (cf. the locality laws of [12]). In POOL, care has been taken not to diminish the power of this protection mechanism in any way: there are no exceptions or detours around it.

In POOL, like in most object-oriented languages, objects are grouped into *classes*. All the objects in one class (the *instances* of the class) have the same structure of their internal data (the instance variables) and they have the same methods. Classes also have the task to create new instances of themselves. In many object-oriented languages (e.g., Smalltalk-80 [11]) this is modelled by viewing classes again as objects, so that the creation of new instances can be done in class methods (methods of the class object). In POOL, a different approach is chosen: A class provides its users with so-called *routines*. A routine is another kind of procedure, which can be called by any object in the system, and which is associated with a class rather than with a specific object. Routines are a suitable abstraction mechanism to encapsulate, among others, the creation and initialization of new objects.

2.2 Parallelism

There are several ways to introduce parallelism in an object-oriented language (see [18]). In POOL, this is done by giving objects a local activity of their own. Each object has a so-called *body*, a local process which starts executing as soon as the object is created. In an object's body it is indicated explicitly when the object sends a message and when it is willing to accept one. All the objects in one class execute the same body. Parallelism arises because, in principle, the bodies of all the objects can execute in parallel. There is no parallelism within one object.

Message passing is done in a way like the Ada rendez-vous [1]: The sender and the receiver wait for each other until both are ready to exchange the message, then the receiver executes the specified method, it returns the result (an object) to the sender, and after that both pursue their own activities in parallel again. When sending a message, the sender specifies the destination object, the method to be executed, and possibly some parameters (again objects). In order to accept a message, an object executes an answer-statement, specifying a set of methods. The first incoming message that specifies one of these methods is accepted. A select-statement, allowing conditional message acceptance, is also present in the language.

2.3 Typing

In contrast to many other object-oriented languages, POOL is not principally meant for rapid prototyping. Rather, it aims at the systematic construction of reliable and maintainable software (in this respect it resembles Trellis/Owl [15]). For this reason, we want the compiler to be able to detect as many errors as possible in a program statically, i.e., before the program is executed. Therefore POOL is a statically typed language.

Let us precisely define some terms: A *type* is a collection of objects that share some properties, notably the operations that can be performed on them (see also [8]). In object-oriented languages, the only operation that can be applied to an object is sending it a message, so the relevant information here is the set of methods that an object has available. Therefore, at this point (we shall refine this later) a type coincides naturally with a class.

Now we call a language *statically typed* if for any expression in a program, it is possible to determine statically, i.e., from the program text alone, the type of the objects this expression will represent during program execution. In POOL, like in many other statically typed languages, this is done by indicating the type of each variable, each parameter and result of a method or routine, and each constant. The type of every expression can then be determined starting from the types of its ingredients (by analysis of the program text), in such a way that it is guaranteed that whenever the expression is evaluated, the value will be an instance of this type.

In many other object-oriented languages (e.g., Smalltalk-80 [11]), this concept of type does not occur in the language; they are not statically typed. However, in most cases, the concept plays a role in the mind of the programmer, because he will make assumptions about the properties of the objects he is dealing with. Therefore, even when using such languages, it is useful to have a good understanding of the concept of typing.

3 Inheritance versus subtyping

3.1 Basic principles

The concept of inheritance was already present in the first object-oriented languages, like Simula [10] and Smalltalk-80 [11]. The basic idea is that in defining a new class it is often very convenient to start with all the variables and methods of an existing class and to add some more in order to get the desired new class. The new class (let us call it B) is said to *inherit* the variables and methods of the old one (which we shall call A). This inheritance mechanism constitutes a very successful way of incorporating facilities for *code sharing* in a programming language.

But there is more to this mechanism: It is clear that in the situation described above class B has all the variables and methods of class A. Therefore, in a way, we can consider every object of class B equally well as an object of class A: At any point where we expect an object of class A, an object of class B will satisfy our needs, because it will accept all the messages an object of class A would accept. The objects in class B can be considered as *specialized* versions of the ones in class A. To express this, class B is called a *subclass* of A, and conversely A is called a *superclass* of B.

If we compare this with the concept of typing, as introduced in section 2.3, we could see the type associated with class B as a *subtype* of the type associated with class A. More precisely, we could take a type τ comprising all the instances of class B, and a type σ which contains all the instances of the classes A and B *together*. Now τ is a subtype of σ in the sense that every element of τ is also an element of σ. (Note that, in order to avoid confusion, we shall *not* say that an instance of class B is also an instance of class A.) In this way, we can establish a one-to-one correspondence between the concepts of inheritance and subtyping.

Conceptually, if we consider objects in a program as representations of entities in the real world (for example in a database or simulation system), this makes good sense, especially if we concentrate on the variables, which contain the local data of the object. For example, if we have defined a class `Vehicle` with variables to store the owner and the maximum speed, it is convenient to define the class `Car` as a subclass of `Vehicle` so that we only have to add a variable to store the license number. An instance of class `Car` is then automatically considered as an element of the type associated with class `Vehicle`.

Of course, this procedure can be repeated several times. For instance, we can define a class `Truck` as a subclass of `Car`, with an extra variable to store the load capacity, we can define `Bus` as another subclass of the class `Car`, with a variable for the number of seats, and we can define `Bike` to be a subclass of `Vehicle`, adding a variable containing the number of speeds. In this way we can get a whole hierarchy of classes, which has the form of a tree:

$$\begin{tikzcd} & & \texttt{Vehicle} \arrow[dl] \arrow[dr] & \\ & \texttt{Car} \arrow[dl] \arrow[dr] & & \texttt{Bike} \\ \texttt{Bus} & & \texttt{Truck} & \end{tikzcd}$$

Moreover, it is possible to allow a new class to inherit from more than one existing class. This mechanism is called *multiple inheritance*, in contrast to *linear* inheritance. For example, a horse can be considered as an animal (having, for example, a father and a mother) and as a vehicle, and therefore the class `Horse` can be defined conveniently as a subclass of both `Animal` and `Vehicle`. In the case of multiple inheritance, the class hierarchy is not a tree any more; it becomes an acyclic directed graph:

$$\begin{tikzcd} \texttt{Vehicle} \arrow[dr] & & \texttt{Animal} \arrow[dl] \\ & \texttt{Horse} & \end{tikzcd}$$

The mechanism we have described above constitutes the basis of inheritance/subtyping as it is incorporated in most existing object-oriented languages. It has proved to be a very powerful mechanism to structure large software systems, and its presence has determined the success of object-oriented languages for a large part. This is not very surprising, because such a structuring mechanism is very useful in the development of large software systems, and while (as we have argued in section 2.1) inheritance is *not* essential for object-oriented languages, the object-orientedness *is* essential for inheritance: Only in languages where we in fact consider the internal details of only one object at a time, it is possible to replace one object by a more specialized one, which offers the same functionality as the old one, but possibly some more.

3.2 Problems with inheritance and subtyping

However useful the above mechanism is, it has some problems associated with it. One of the most evident problems is what to do when a new class wants to inherit from two existing classes that both have a variable or method with the same name. Should one of the two be chosen, and if so, which one? Or otherwise, if both are included in some way in the new class, how are they named and accessed? A symptom for the fact that the inheritance mechanism is problematic is the large variety of ways in which the basic mechanism is augmented in languages like Smalltalk-80 [11,6], LOOPS [5], Flavors [17], etcetera.

The origin of many of these problems lies in the fact that the inheritance structure, used for code sharing, and the conceptual subtyping hierarchy, originating from specialization, are *not* the same thing. They lie on different levels of abstraction in the system: inheritance is concerned with the *implementation* of the classes, while the subtyping hierarchy is based on the *behaviour* of the instances (as seen from the outside, by other objects).

For simple, record-like objects whose main function is to store data the practical difference between these two points of view are not very large. But for more complicated objects the distinction between the interface with the outside world and the internal implementation is much more important, and therefore the discrepancy between inheritance and subtyping is clear.

It may well be that in many cases the hierarchical relationships induced by inheritance and by subtyping coincide, but this is certainly not always the case: On the one hand it is very well possible in many cases to define a class that really specializes the behaviour of another class, but employing a totally different structure of variables and having different code even for methods with the same name (so we have subtyping without inheritance). On the other hand, it is also possible that, by simply adding some methods to an existing class, the behaviour even of the old methods is changed in an essential way (the new methods might assign to the variables of the object in such a way that an invariant, on which the old methods rely, is violated). In the latter case the new class cannot be said to give rise to a subtype of the old one, so we have inheritance without subtyping.

Therefore we argue that the concepts of inheritance and subtyping should be decoupled and that each should be considered on its own right (of course, the commonalities between the two should not be forgotten). This observation is not totally new (for example, it was already presented in [16]), but in exploring this line of thought further, we shall encounter some new consequences.

3.3 Inheritance

Let us first concentrate on inheritance. Now that we consider inheritance *only* as a mechanism for code sharing, we can see more clearly what it should look like. First of all, because inheritance is not the same as subtyping, it is not necessary that inheriting from a class means inheriting *everything* from that class: Since the new class is not necessarily associated with a subtype of the old one, it does not have to have all the old methods. It should be possible to inherit only a subset of the variables and methods of a given class. Of course, if a method is inherited that acts on a certain variable, that variable should also be present in the new class. A way to ensure this would be to inherit such a variable automatically with the method.

Generalizing this a little, it is reasonable to say that a new class should inherit a *consistent subset* of the variables and methods of an old one. This leads to the idea of an *inheritance package* of variables and methods, which can be inherited in one piece. Such an inheritance package could be accompanied by an *inheritance interface*, listing the variables and methods that can be used freely by the inheriting class. It would be possible to include variables and methods in the package that are not in the inheritance interface. These can be used by the methods in the inheritance interface but they remain hidden from the definer of the inheriting class. In this way a certain amount of abstraction and encapsulation can be built into the inheritance mechanism. (It is even possible to completely detach inheritance packages from classes, so that inheritance packages exist independently and can be used and shared by classes.)

In fact, we get *two* interfaces for each class: the message interface (dealing with the messages accepted by instances of the class), and the inheritance interface (dealing with the inheritance packages, the code that can be inherited by other classes). This phenomenon was also observed in [16], but there inheritance meant inheriting everything from an existing class (which is, of course, an important special case of our more general mechanism).

Considering multiple inheritance, it is clearly very useful to allow a class to inherit more than one inheritance package from possibly different existing classes. However, these packages should not have conflicting methods or variables. Every method and variable should of course be present only once in the resulting class. A simple renaming mechanism would allow the combination of inheritance packages that are conflicting originally (note that with subtyping renaming is impossible because it destroys the subtype property). The variables and methods in the inheritance package that are not in the inheritance interface are assumed to be renamed in such a way that they can never cause any name conflict.

Different ways of combining methods (like in Flavors [17]) can also be used together with this mechanism. The question is only whether the additional complexity of the language is justified by the ease of programming resulting from such extensions.

3.4 Subtyping

We have seen (in section 2.3) that a type is a collection of objects sharing some properties. In section 3.1 we encountered the concept of subtyping. It is clear that for σ to be a subtype of τ, it is necessary that all elements of σ have the properties required by τ. This brings us to the issue of what information at all should be associated with a type. Let us remember that we want to use types to base our conceptual specialization hierarchy on, or more precisely that we want this hierarchy to consist of types ordered with the subtype relation. Then it is clear that a type should give us information about the *behaviour* of the objects that belong to it. This means that a type should consist of a *specification* of the behaviour of these objects. Ideally, this should be a formal specification of all the aspects of this behaviour that we are interested in. In general this will include the messages that can be sent to such an object (the methods it has available), the order in which these messages may be sent, the conditions on the values of the parameters, the value of the result, possibly also the messages the object will send itself, and perhaps even the time the object needs to perform all these actions.

Let us suppose that for each type we have a formal specification of the behaviour of its elements, in the form of a formula in some kind of logic. For example, let the type σ be specified by the formula $\phi(x)$, meaning that for every element β of σ we have $\phi(\beta)$. Now it is easy to formulate the condition that must be satisfied in order to be able to regard one type as a subtype of another type. If the type σ has the specification $\phi(x)$ associated with it and τ has the specification $\psi(x)$, then τ can be considered as a subtype of σ precisely if, for all x, $\psi(x)$ implies $\phi(x)$. This means that every object α that is a member of τ so that we have $\psi(\alpha)$, will automatically satisfy $\phi(\alpha)$ and can thus be considered as an element of σ.

Unfortunately, formal specification technology is not yet in a position that formal specifications of this kind can be included in a practical programming language. One could even say that especially for object-oriented languages there has been surprisingly little research on the formal description of program behaviour (see for example [4]). In regard of the fact that formal specification theory can contribute substantially to the understanding of mechanisms like subtyping in object-oriented programming, it certainly deserves more attention. However, even with (partly) informal specifications, it is useful to keep the above characterization of subtyping in mind. Therefore, lacking a suitable formalism to express the behaviour of objects in a type, we should at least associate with each type an informal description of this behaviour.

One aspect of this behaviour that can and should be described formally for each type, is the set of available methods, together with the types of their parameters and results. For this aspect the above subtype condition reduces to the following: In order for a type τ to be a possible subtype of σ, the following should hold: For each method m listed for σ, with parameter types $\alpha_1, \ldots, \alpha_n$ and result type β, the type τ should have a method with the same name m, with parameter types $\gamma_1, \ldots, \gamma_n$ and result type δ, in such a way that, for every i between 1 and n, α_i is a subtype of γ_i, and δ is a subtype of β.

If this condition is fulfilled, we know that every object in τ will behave like an element of σ (at least with respect to the messages it accepts): It will accept each message that specifies such a method name m, listed with σ, it will be able to handle parameters from the types $\alpha_1, \ldots, \alpha_n$, like σ says it should, (it can even handle parameters from the larger types $\gamma_1, \ldots, \gamma_n$), and it will return a result of the type β, complying with σ's wishes (the result will even be a member of the smaller type δ). This is the well-known "contravariant" parameter rule described in [7] and used for example in the language Trellis/Owl [15].

However, let us remember very well that there is more to types than this condition: a type represents a constraint on the behaviour of its elements, and only a part of this behaviour is captured by the above rule. For example, a type `Stack`, with the methods **put** and **get** (having the obvious meaning), is not distinguished by the above rule from the type `Queue`, with the same methods. Yet their behaviour is clearly different and none of them should be considered as a subtype of the other. Nevertheless, both are subtypes of the type `Bag`, again with the methods **put** and **get**, where **get** is supposed to return an *arbitrary* element that was previously inserted by **put**.

In statically typed languages, the contravariant parameter condition can be checked by the compiler, and one can be sure that during the execution of a program it will never happen that an object is sent a message for which it has no method. But as we have seen, the behaviour of objects cannot yet be formalized in all its aspects, so the compiler cannot check completely that one type is a subtype of another one. This is the reason why the subtype relation should *not* be assumed *automatically* whenever the contravariant parameter condition is satisfied. To ensure that a subtype really specializes its supertype with respect to its behaviour, a certain discipline is required from the programmer: To the formal description of the available methods and their parameter and result types, an informal description of the behaviour of the elements of the type should be added, and whenever the programmer asserts a subtype relationship between two types, he should check the condition on the associated behaviour descriptions himself. The situation is like in traditional statically typed languages: the compiler can ascertain the absence of certain errors, but not of all errors.

Let us finally note that in this setting, *multiple* subtyping is a naturally occurring phenomenon: it is often the case that a type is a subtype of several, mutually unrelated types. Conflicts, like we saw above with inheritance, do not arise.

4 Integrating it in POOL

In the last section we have investigated the essential properties of inheritance and subtyping, which are valid for sequential as well as parallel object-oriented languages. In this section we shall look at the specific issues arising when we try to integrate these concepts into a POOL-like language. Now that we have seen that inheritance and subtyping are two different concepts, we can deal with each of them in turn.

Let us first talk about inheritance. It is clear that inheritance of methods and variables does not present any additional problem in a parallel language. The same scheme using inheritance packages can be used, as we described it in section 3.3. Inheriting routines (cf. section 2.1) does not make much sense, because routines can anyway be called from everywhere in the system.

The most difficult question is how to do something suitable with inheritance in the definition of the *body* (the local process of an object). There are several theoretically appealing options, like putting inherited bodies in parallel, or one after the other, or even nesting them. None of these possibilities seems very useful in practice, and it seems best not to allow inheritance of bodies. Nevertheless, there is one promising option: In general, the first part of a body will be concerned with the initialization of the variables. It seems very useful to include this part of the body in the inheritance package to which the variables belong. Then these statements will be prefixed to the body of any class that uses this inheritance package. In this way it is also ensured that the hidden variables in this package are initialized correctly.

However, as the bodies of objects become a more important part of their code, inheritance becomes a less useful mechanism for code sharing. Therefore it might well be that for parallel systems the importance of inheritance (at least in the above form) is not so great as for sequential systems.

For the integration of subtyping within a parallel object-oriented language the same kind of problems apply as in sequential languages. Of course, the formal description of the behaviour of *parallel* objects is probably even more difficult than in the sequential case. Nevertheless, there is some more work going on in this field (see for example [3,9,13]). As this work has not yet resulted in a practical way of formally specifying object behaviour (it is not even sure that it ever will), we shall have to help ourselves with informal descriptions, complemented with a formal specification of the available methods and their parameter and result types, just like in the sequential case. Let us realize that here the formal part of the description captures an even smaller fraction of the relevant object behaviour: For example, the order in which messages can be accepted and sent is much more important than in a sequential system. Failing to take these things into account can lead to nasty problems like deadlock. In this situation, where there is a greater need for formal specification, the prospects for it are worse.

Like inheritance, it may well turn out that subtyping will play a relatively less important role in parallel than in sequential languages: As there are more aspects of object behaviour that can make a difference (e.g., the order in which messages are sent or received), it is less likely that one type is really a specialization of the other.

5 Conclusion

As we have seen in section 3, inheritance and subtyping are two different, though related, concepts, and they should not be confused. We have also developed some ideas on how to integrate them in object-oriented languages in general. In section 4 we have shown some possibilities to introduce these concepts into a parallel language like POOL. We have also observed that in a parallel language it is questionable whether the importance of inheritance and subtyping will be as great as in sequential languages.

As to the status of our own work: At the moment of this writing we have not yet completed a language design along the lines sketched above. We are certainly planning to experiment with inheritance and subtyping in the near future. However, at the moment we give a higher priority to the design of a language without these features, in which reliable programs can be written in a reasonably easy way, and which can be efficiently implemented on a parallel, decentralized machine.

References

[1] *The Programming Language Ada Reference Manual.* ANSI/MIL-STD-1815A-1983, published in: Lecture Notes in Computer Science, Vol. 155, Springer-Verlag, 1983.

[2] Pierre America: Definition of the programming language POOL-T. ESPRIT project 415A, Doc. No. 0091, Philips Research Laboratories, Eindhoven, the Netherlands, September 1985.

[3] Pierre America, Jaco de Bakker, Joost N. Kok, Jan Rutten: Operational semantics of a parallel object-oriented language. *Conference Record of the 13th Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 13–15, 1986, pp. 194–208.

[4] Pierre America: Object-oriented programming: a theoretician's introduction. *Bulletin of the European Association for Theoretical Computer Science*, No. 29, June 1986, pp. 69–84.

[5] Daniel G. Bobrow, Mark Stefik: The LOOPS Manual. Technical Report KB-VLSI-81-13, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

[6] Alan H. Borning, Daniel H.H. Ingalls: Multiple Inheritance in Smalltalk-80. *Proceedings of the AAAI Conference*, Pittsburgh, August 1982, pp. 234–237.

[7] Luca Cardelli: A semantics of multiple inheritance. In: G. Kahn, D.B. MacQueen, G. Plotkin (eds.): *Semantics of Data Types*. Springer-Verlag, Lecture Notes in Computer Science, Vol. 173, 1984, pp. 51–67.

[8] Luca Cardelli, Peter Wegner: On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, Vol. 17, No. 4, December 1985, pp. 471–522.

[9] William D. Clinger: Foundations of actor semantics. Technical report 633 (Ph. D. Thesis), Massachusetts Institute of Technology, Artificial Intelligence Laboratory, May 1981.

[10] Ole-Johan Dahl, Kristen Nygaard: Simula — an ALGOL-based simulation language. *Communications of the ACM*, Vol. 9, No. 9, September 1966, pp. 671–678.

[11] Adele Goldberg, David Robson: *Smalltalk-80, The Language and its Implementation*. Addison-Wesley, 1983.

[12] Carl Hewitt, Henry Baker: Laws for communicating parallel processes. In: *Proceedings of IFIP-77*, Toronto, Canada, August 1977, pp. 987–992.

[13] Van Nguyen, Alan Demers, David Gries, Susan Owicki: A model and temporal proof system for networks of processes. *Distributed Computing*, Vol. 1, 1986, pp. 7–25.

[14] Eddy Odijk: The Philips Object-Oriented Parallel Computer. In: J.V. Woods (ed.): *Fifth Generation Computer Architecture (IFIP TC-10)*. North-Holland, 1985.

[15] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, Carrie Wilpolt: An introduction to Trellis/Owl, *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, Oregon, September 1986, pp. 9–16.

[16] Alan Snyder: Encapsulation and inheritance in object-oriented programming languages. In: *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, Oregon, September 1986, pp. 38–45.

[17] Daniel Weinreb, David Moon: Flavors: Message passing in the Lisp machine. AI Memo 602, November 1980, Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

[18] Akinori Yonezawa, Mario Tokoro (eds.): *Object-Oriented Concurrent Systems*. MIT Press, 1987.