PB - 209

K.S. Thomsen: Multiple Inheritance

TRYK: RECAU (06) 12 83 55

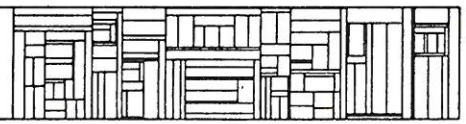
ISSN 0105-8517

Multiple Inheritance, a Structuring Mechanism for Data, Processes and Procedures

Kristine Stougård Thomsen

DAIMI PB - 209
April 1986

DATALOGISK AFDELING
Bygning 540 - Ny Munkegade - 8000 Aarhus C
tlf. (06) 12 83 55, telex 64767 aausci dk
Matematisk Institut Aarhus Universitet



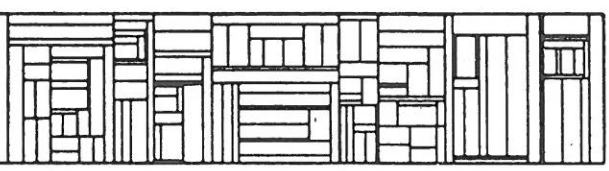
ISSN 0105-8517

Multiple Inheritance, a Structuring Mechanism for Data, Processes and Procedures

Kristine Stougård Thomsen

DAIMI PB - 209
April 1986

DATALOGISK AFDELING
Bygning 540 - Ny Munkegade - 8000 Aarhus C
tlf. (06) 12 83 55, telex 64767 aausci dk
Matematisk Institut Aarhus Universitet



Abstract

A motivation is given for the use of multiple inheritance as a general structuring mechanism for data, processes and procedures, and an object oriented programming language that incorporates such an inheritance mechanism is outlined. Objects in this language combine the notions of abstract data structures and processes. Classes and procedures are organized in multiple inheritance hierarchies. The main contribution of this paper is the introduction of a coroutine like strategy for combining multiple action parts of objects. Coincidence of named properties from different classes are treated by combining all versions of the property. The inheritance mechanism on procedures offers an elegant way of combining a number of inherited operations with the same name.

CONTENTS

1 INTRODUCTION 1

2 MOTIVATION 1

3 LANGUAGE PRESENTATION 8

3.1 Classes and Objects 8

3.2 Types 11

3.3 Communication 12

3.4 Control structures 13

3.5 Multiple action parts 15

3.6 Inheritance on routines 21

3.7 Modification of inherited properties 23

4 RELATIONS TO OTHER WORKS. 34

5 CONCLUSION. 37

6 REFERENCES. 38

1 INTRODUCTION

Inheritance, including multiple inheritance is getting widely acknowledged as a valuable structuring mechanism for data classes that describe objects containing data and associated operations as for instance in Smalltalk [9]. The first programming language that included inheritance was Simula [8]. The purpose of this paper is to show that inheritance on processes and procedures is equally appropriate and to present a language with this facilily.

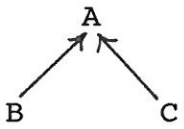
In section 2, a summary of traditional inheritance and its advantages is given and a similar mechanism for processes and procedures is motivated. In section 3, a new language is outlined. The language is object oriented in the sense that it is based on objects and classes that are organized in a multiple inheritance hierarchy. Compared to most other object oriented languages, this language extends the notion of objects to be autonomous communicating processes. That is, besides containing data structures and associated operations, objects perform an autonomous process that may be involved in a parallel computation with other objects or may just act as an administrator of the data structures in the object. The language presentation contains examples that illustrate the usefulness of inheritance in relation to the process aspect of objects. Section 4 contains an overview of relations to previous work.

2 MOTIVATION

In this section it will be argued that inheritance is a valuable structuring mechanism for both data and processes. First a brief summary of the essence of traditional inheritance on data classes will be given and the advantages will be pointed out. Then an analogous notion of inheritance on processes is outlined and motivated.

Inheritance is based on the ability to form hierarchies of classes. Traditionally, classes are templates from which objects that contain data structures and associated operations can be generated. In the literature, the class hierarchies have various names: is-a hierarchies, generalization/specialization hierarchies and inheritance hierarchies. In this paper, the following textual and graphical notation will be used:

```
class A ...;
class B is-a A ...;
class C is-a A ...;
```



B and C are said to be subclasses of A, and A is said to be a superclass of both B and C. An object generated from a class A is said to be an instance of A or an A-object.

If A introduces the declarations D1,...,Dn and B introduces the declarations E1,...,Em, then the is-a relation between B and A implies that B inherits D1,...,Dn such that B for the purpose of this section can be considered equivalent to the class that contains all the declarations D1,...,Dn and E1,...,Em. B is considered a specialization of A since all B-objects have at least all the properties described in A and are therefore also A-objects.

Example:

```
class Person;
  var name,address: text;
  procedure change-address;
  ...
end;
```

```
class Student is-a Person;
  var student-number: integer;
      courses: list of course;
  procedure add-course;
  ...
end;
```

Moreover, in some languages a subclass B can modify declarations inherited from a superclass A by introducing some restrictions on the declarations from A that all B-objects must satisfy.

Inheritance hierarchies offer an excellent structuring mechanism for data classes when many classes with many details are involved. The mechanism of inheritance has therefore primarily been acknowledged within the database and artificial intelligence

areas, but it is gradually becoming clear that inheritance has much wider application [4].
The general advantages of inheritance are:

- Better conceptual modelling.
  Since specialization hierarchies are very common in everyday life, direct modelling of such hierarchies makes the conceptual structure of programs easier to comprehend. See [3], [4] and [17].

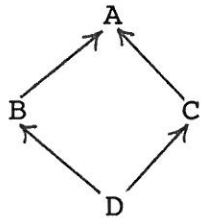
- Factorization.
  Inheritance supports that common properties of classes are factorized - that is, described only once and reused when needed. This results in greater modularity and makes complicated programs easier to comprehend and maintain since redundant description is avoided.

- Stepwise refinement in design and verification.
  Inheritance hierarchies support a technique where the most general classes containing common properties of different classes are designed and verified first, and then specialized classes are developed top-down by adding more and more details to existing classes.
  A detailed discussion of the technique used in design is given in [3]. Verification is discussed in [21].

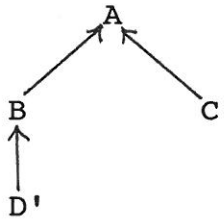
- Polymorphism.
  The hierarchical organization of classes provides a basis for introduction of parametric polymorphism in the sense that a procedure with formal parameter of class C will accept any C-object as actual parameter, including instances of subclasses of C.

Many languages with inheritance allow only single inheritance - that is, tree structured inheritance hierarchies where a class can have only one superclass. However, there are many advantages of allowing multiple inheritance - i.e. several superclasses of a class: Consider the inheritance hierarchy in fig.

a) below. If multiple inheritance is not available, an alternative hierarchy must be chosen, for instance the one in fig. b) below, where D' is like D plus an explicit description of the properties in C.



a)



b)

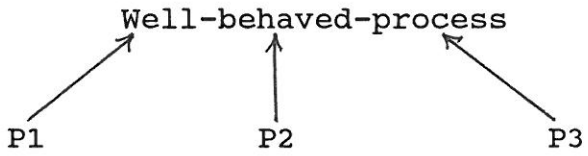
Solution b) is inferior to solution a) in several respects: b) is not a good conceptual model of a). Full factorization is not obtained in b), since the properties in C must be duplicated in D'. Finally, the full advantage of polymorphism is not obtained in b), since D' is not a specialization of C and thus D'-objects cannot be accepted as actual parameters for formals of class C. More concrete examples of multiple inheritance will be presented in section 3.

Inheritance on procedures and process classes (templates for generation of processes) can be defined in a way that is quite equivalent to inheritance on data classes: Consider two processes/procedures, P1 and P2, where P1 is-a P2. Besides declarations, processes/procedures also contain statements. The is-a relationship between P2 and P1 implies that declarations in P1 and P2 are combined in P2 in the same way as for data classes. Moreover, if P1 contains the statements S1,...,Sn and P2 contains the statements T1,...,Tm, then execution of P2 will imply execution of all the statements S1,...,Sn and T1,...Tm in some sequence. (How this sequence can be controlled is the main topic of this paper and is discussed in depth in section 3.5.) With this notion of inheritance, P2 is considered a specialization of P1 since P2 will do at least what P1 does.

Such an inheritance mechanism for process classes and procedures is a useful structuring mechanism just as it is for data classes. This will be demonstrated in the following by showing that the general advantages of inheritance also apply to processes/procedures. The examples given are rather abstract. They serve only as motivation in this section, but some of them will be concretized through examples in section 3.

Factorization

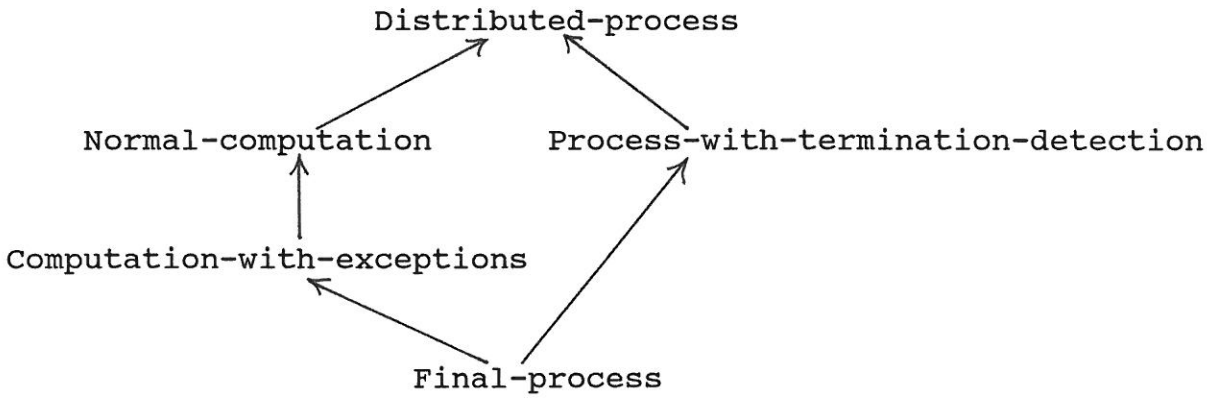
When several process classes are used in the same system, they will often have some properties in common. These properties can be factorized into one general process class that is used as a common superclass of all the needed process classes. For instance, consider a system in which all processes must be well-behaved according to some discipline of allocating resources. In such a situation, the following factorization could be made:



As for data classes, this would mean that duplication of description is avoided and a clearer conceptual structure is obtained.

Stepwise refinement

In the development of complex processes, stepwise refinement is a helpful technique that can help structuring the details and separate different concerns. For instance, normal behaviour of a process can be designed first and description of exceptional behaviour can be added in a subclass. In a distributed computation, the concerns of the distributed algorithm can be separated from the concerns of detecting termination of the distributed computation:

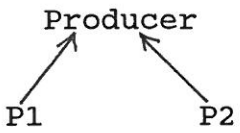


This example is too complex to be concretized in this paper, but [18] realizes the termination branch of the hierarchy by means of the inheritance mechanism presented in section 3.

Polymorphism

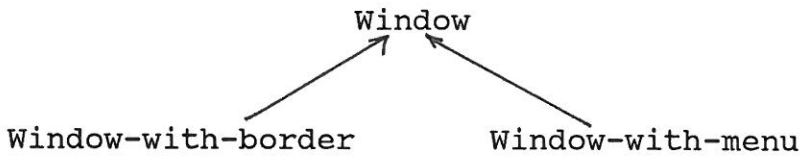
When process classes are parameterized with processes, the advantage of polymorphism is obvious. A process parameter may then be known to behave like a certain class of processes (e.g. accepting some specific message) and still the actual parameter may be any specialization of this class. As an example consider a class of consumers parameterized with a producer from which to get its data:

```
class Consumer(p:Producer);
 ...
end;
```

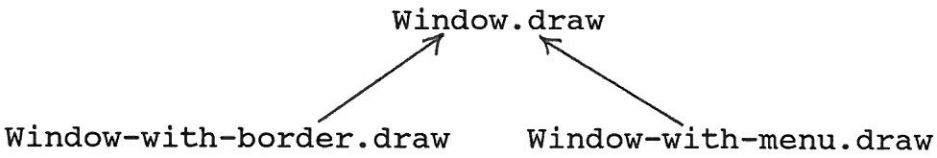


Procedure hierarchies

Most of the arguments given in favor of inheritance on processes are also valid for procedures. Moreover, it is often so that when data classes are organized in an inheritance hierarchy, then the procedures that manipulate the data can be organized in a hierarchy that is parallel to the data hierarchy. Consider as an example a number of window classes organized in an inheritance hierarchy:



The procedure draw is relevant for all three classes but in different versions, since besides drawing the window itself, a window-with-border must have its border drawn, and a window-with-menu must have its menu drawn. The three versions of the draw procedure can be organized in an inheritance hierarchy that is parallel to the hierarchy of the window classes:



The conclusion of this section is that inheritance, including multiple inheritance, can be a useful structuring mechanism for both data classes, process classes and procedures. However, the potential of inheritance on process classes and procedures has not been fully explored yet. The language outlined in the next section is meant to contribute in this direction.

3 LANGUAGE PRESENTATION

The language outlined in this section is intended as an illustration of how multiple inheritance on data, processes and procedures can be included in an object oriented language. The language presentation does not contain a complete language definition. Only language features needed in order to show the basic points of the paper are included in the presentation.

In particular, the treatment of name coincidences is simplified. The main contribution on this subject is to show an elegant solution to the problem of operation combination. The problem of operation combination arises when two or more operations with the same name are inherited from different superclasses and their combined effect is wanted as the result of calling the operation on instances of the subclass. The standard example of this situation is the operation initialize.

3.1 Classes and Objects

A class is a template that describes the structure and behaviour of all objects created from it (called instances of the class). Classes are used to describe both processes and data structures. A class definition contains:

1) The name of the class.

2) A list of value parameters of the class.

3) A list of superclasses from which the class inherits parameters, data attributes, operations and action parts.

4) A number of data attributes private to each instance of the class. There are four different kinds of data attributes:

   - Constants
     specified as follows:
     const x:T, where T is an ordinary enumeration type, scalar type, subrange type or sequence type. For an individual object, x will have a constant value given when the object is created.
   - Variables:
     var x:T, where T is as above. The value of x may vary during the lifetime of the surrounding object.
   - Constant references:
     const c:C, where C is a class name. Within an individual object, c is a constant reference to a specific instance of the class C. The actual value of the reference is given when the surrounding object is created.
   - Variable references:
     var c:C, where C is a class name. During the lifetime of the surrounding object, c may refer to different instances of class C.

   The term constant data attribute is used to cover constants and constant references. The term variable data attribute covers variables and variable references.

5) A number of operations attached to each instance of the class and visible to other objects. The operations are procedures and functions with ordinary value and result parameters, local variables and bodies that can manipulate the attributes of the surrounding object. Operations can be organized in an inheritance hierarchy (section 3.6).

6) An action part related to each instance of the class. The action part may manipulate the attributes, call the operations and communicate with other objects by means of synchronized operation calls (section 3.3).

A class that has one or more superclasses inherits all parameters, attributes, operations and action parts from its superclasses (and their superclasses in turn etc). Each superclass contributes only once to the class regardless of the number of paths in the inheritance hierarchy from the class to the superclass. This means that objects generated from the class will have structure and behaviour as described by the combination of all the classes in the inheritance hierarchy above the class. How this combination is formed is the main subject of this paper.

Parameters, attributes and operations that a class inherits from its superclasses are visible within the class just as if they had been locally declared.

An object has its own version of attributes, operations and action parts. The action parts of an object are executed alternately as will be described in section 3.5 . Thus, the action parts of an object can be considered as forming a single autonomous process that starts executing when the object is generated and executes in parallel with other objects. An object can only communicate with other objects through synchronized call of its operations. An object is thus a common notion for an abstract data structure and a process.

Objects are dynamically generated from classes. A variable reference declared as x:C gets a new C-object (or C1-object) assigned to it by the statement

x:=new C1(aparam) with (x1=exp1,...,xn=expn)

where C1=C or C1 is-a C. Aparam is a list of actual parameters to the class C1 and the with list supplies values to the constant data attributes x1,...xn.

3.2 Types

The language is typed and supports static type checking. However, the type system allows parametric polymorphism and thus adds flexibility and expressive power to the language compared to traditional typed languages while maintaining the security of static type checking. The first language with this kind of type system was Simula [8], but a similar mechanism is included in recent languages - e.g. Beta [13],[14] and Galileo [1].

For the purpose of this paper, the possible types are: class-types, sequence-types, scalar-types, enumeration-types and subrange-types.

Polymorphism of the type system is obtained by the partial <= relation on types. For class-types, the <= relation is the transitive closure of the is-a relation (the subclass-of relation). For sequence-types the <= relation is the <= relation on the element type. For scalar-, enumaration- and subrange-types, the <= relation is the ordinary subset-of relation.

Type compatibility related to assignment and parameterization is defined as follows:

1) Type T is assignable to type S iff T <= S.
2) Actual parameters of type T are accepted for formal parameters of type S iff T <= S.

Example

```
Class Person                    Class Student is-a Person
  ...                             ...
end                             end

x:Person

Procedure Pip(y: Person);
begin ... end;

x:= new Student;                legal because of polymorphism
Pip(x);                                    do.
```

An object can only be manipulated in accordance with the type with which it is declared, although it may actually belong to more specialized types as well. For instance, in procedure Pip above, only Person properties of the parameter y can be assumed since y is declared to be of type Person.

The type system gives the flexibility that procedures written to manipulate objects of type T can also be used to manipulate all objects of more specialized types.

3.3 Communication

Objects communicate with each other by means of a synchronous rendezvous mechanism somewhat similar to Ada's [16] and Beta's [14]. Communication is described by means of request statements:

```
partner ! procedure-name ( actual-parameters )
```

and accept statements:

```
partner ? procedure-name
```

where partner is a reference to an object and may be omitted from the accept statement.

Execution of a request/accept statement within an object implies waiting until a matching accept/request statement gets executed within another object. The two objects synchronize and the accepting object executes the denoted procedure with the actual parameters supplied by the requesting object. Afterwards the objects continue independently.

An accept statement in object A matches a request statement in object B if

- partner specified by B is a reference to A, and
- partner in A is either absent or a reference to B, and
- procedure-name in both denote the same procedure in A.

3.4 Control structures

The language includes an alternative and a repetitive command similar to CSP's [12]. The alternative command has the form:

```
if
   guard-1 -> statement-list-1
   ...
/guard-n -> statement-list-n
fi
```

and the repetitive:

```
do
   guard-1 -> statement-list-1
   ...
/guard-n -> statement-list-n
od
```

where a guard can contain a boolean expression without side effects and an accept or request statement. The boolean expression or the communication in a guard may be absent, but not both.
A guard G with a boolean expression B and a communication statement C is said to be

- closed if B is false,
- indetermined if B is true and communication is not yet ready through C,
- open if B is true and communication is ready through C

Absence of B or C makes the value of the guard depend only on the present part.

The semantics of the alternative if command is:

```
evaluate all guards
if all guards are closed then skip
else if an open guard exists
        choose an open guard
     else wait until a communication is ready
             in one of the indeterminate guards
          choose an open guard
     perform communication and statement-list
     related to the chosen guard
```

This is equivalent to the CSP semantics except that closed guards do not imply failure. The semantics of the repetitive do command is to execute the corresponding if command repeatedly until all guards are closed (equivalent to CSP).

Before continuing the language presentation, we will look at a simple example that illustrates the features already introduced. In general, the examples of this paper are kept on a minimum size, and parts of the examples that are not essential to the understanding of the examples are just sketched by means of sentences in natural language or just "...".

Example 1

This example shows a class that implements an abstract data type. The action part of the class serves the purpose of controlling the sequence of operation calls to instances of the data type. The example can be thought of as part of a university database.

```
Class Student;
  ...
   operation enrolment-for-examination;
   begin ... end;

   operation examination;
   begin ... end;

   operation dispensation;
   begin ... end;

   action
     do true ->
       ? enrolment-for-examination;
       if
           ? examination  -> skip
         / ? dispensation -> skip
       fi
     od
end;
```

The action part of a student object ensures that enrolment takes place before the actual examination and that the student either takes the examen or gets a dispensation from it, but not both. Moreover, another enrolment cannot take place before the previous examen is over or dispensated from.

3.5 Multiple action parts

As already mentioned, an object gives rise to an autonomous process that starts executing when the object is created and executes concurrently with other objects. The process originating from one object consists of all the action parts described in the class of the object and all its superclasses. In the following, we will focus on the control aspect within a single object.

The different action parts of an object may have different priorities associated with them (not to be discussed in detail) and the strategy is to execute the action parts alternately according to their ability to proceed and their relative priority. Control shifts from one action part to another at well-defined points in the action parts called changeover points. That is, the action parts are executed in a kind of programmer controlled interleaving or coroutine sequencing.

Changeover points are specified by means of the symbol "*". Each changeover point has a socalled resumption condition associated with it. The resumption condition expresses under what circumstances the action part is able to continue execution. Changeover points may appear in the following contexts:

1) *(B) may be used as a statement, where B is a boolean expression explicitly denoting the resumption condition of the changeover point.

2) * may be used as a prefix of a request or accept statement meaning that there is a changeover point immediately before the communication. The resumption condition is that the communication is ready - i.e. that a matching accept/request statement is currently being executed by another object.

3) * may be used in an alternative or repetitive command as follows:

```
if *                do *
  ...                 ...
fi                  od
```

The semantics of an if* command can be expressed using the notation in 1) above for a changeover point:

*(existence of an open guard)
choose an open guard
perform communication and statement-list

Note that the if* command will always imply choice of a guard as opposed to the if command. If* also waits if all guards are closed, since some guard may become true later, not only as a result of the arrival of communications but also as a result of the alternation with other action parts of the same object that change the values of some boolean expressions.

The semantics of do* is an infinite repetition of the corresponding if*.

If* and do* are generalizations of the when and cycle statements in Distributed Processes [11].

4) An implicit changeover point is placed after the last statement of each action part. After such a changeover point, the action part is terminated and not reconsidered for resumption.

Each action part of an object has a local program counter lpc. When the object is created, all action parts are executed until their first changeover point in a top down sequence in the inheritance hierarchy. This starting convention enables proper initialization of common variables. Hereafter all lpc's are located at the first changeover point of their action part. The alternation mechanism between the different action parts of the object then proceeds as follows:

While not all action parts have terminated do

1) Wait until the resumption condition of some action part is satisfied.
2) If several are satisfied, one of the action parts with highest priority and satisfied resumption condition is chosen. If no priorities are specified, default is that a subclass has higher priority than its superclass.

3) Resume execution of the chosen action part until its next changeover point.

Before showing examples of alternation, a few comments on the mechanism are appropriate. Obviously, the general boolean expressions as resumption conditions (*(B) and boolean expressions in guards in if* and do*) will be expensive to implement. The boolean expressions must be re-evaluated each time the action part, which is suspended at such a changeover point, is considered for resumption. However, the mechanism is very expressive as will be shown later. Moreover, it is only paid for when actually used since * as prefix to communications can be used in guards to program if and do commands with changeover points that depend on communication only and not on the full value of the guards:

```
if
  B1, *C1 -> S1
  ...
  Bn, *Cn -> Sn
fi
```

(We require that either all communications in guards of an if or do command are prefixed with * or none of them are.)

The semantics of such an if (or do) command is the same as the ordinary if (or do) command (section 3.4), except that the "wait until a communication is ready in one of the indeterminate guards" is changed to a changeover point: "*(a communication is ready in one of the indeterminate guards)". B1,...,Bn are not re-evaluated when checking such a resumption condition that only involves readiness of communications. Note, that this means that when using an if or do command with communication changeover point, then the value of Bi is not guaranteed to be true after choice of the i'th guard, since there is a changeover point between the evaluation of Bi and the choice of the guard. If Bi is required to be true after choice of the guard, then Bi should not involve variables that may be changed by other action parts in the same object. If such variables are involved, then if* or do* should be used.

To illustrate the alternation mechanism and its usefulness, we will now look at some examples.

Example 2

Suppose we have a class of processes that make heavy use of some resources and must acquire exclusive access to them during their computation. This class of processes can be described in a general class:

```
class Heavy-process(params);
  ...
action
  request all required resources;
  *(true);
  release all the resources;
end;
```

Subclasses of class Heavy-process describe well-behaved processes that request resources before using them and finish by releasing them:

```
class Computation is-a Heavy-process;
  ...
action
  perform some computation;
end;
```

An instance of Computation will have its two action parts executed alternately by first executing the request, then the computation and finally the release. The resumption condition in Heavy-process is trivially true, but the default bottom-up priorities of the action parts implies that the action part of class Heavy-process is not resumed until the action parts of all subclasses have terminated (reached their final implicit changeover point). The example shows that an action combination strategy like the one obtained by means of "inner" in Simula [8] is covered as a special case of the alternation strategy.

Example 3

This example involves multiple inheritance and shows some of the additional power of the alternation mechanism compared to the "inner" mechanism of Simula and Beta. The example is an extension of example 1.

```
class Person;
  const name:text;
  var address:text;
  operation new-address(in a:text);
  begin address:=a end;
  action
    do
      *?new-address -> skip
    od
end;
```

```
class Student is-a Person;
  ...
  operation enrolment(...);
  begin ... end;
  operation examination(...);
  begin ... end;
  operation dispensation(...);
  begin ... end;
  action
    do true ->
      *?enrolment;
      if
        *?dispensation -> skip
      / *?examination  -> skip
      fi;
    od;
end;
```

```
class Lecturer is-a Person;
  ...
  operation assign-course(...);
  begin ... end;
  operation release(...);
  begin ... end;
  action
    do true ->
      *?assign-course;
      *?release;
    od
end;
```

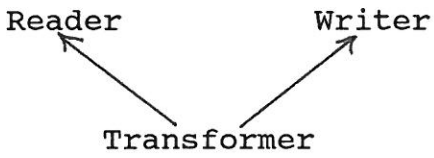
```
class Teaching-assistent is-a Student,Lecturer;
end;
```

An instance of Teaching-assistent executes its three action parts alternately with changeover points at each accept statement and alternative command. The sequence in which the action parts will be executed will depend on requests from other objects (e.g. users of the database) to perform operations, since the resumption conditions in all changeover points are ability to communicate - that is, existance of requests from other objects. An instance of Teaching-assistant will thus be able to behave alternately as a Person, a Student and a Lecturer. Moreover, some

restrictions on the sequence of accepts are enforced. In the Student part, the restrictions are the same as they were in example 1. Similarly for the Lecturer part, an assignment to a course must have taken place before release from it is possible, and a release is necessary before assignment to another course is possible, expressing that lecturers teach only one course at a time.

Example 4

Whereas example 3 involves alternation that depends exclusively on external requests, example 4 illustrates alternation that also depends on the internal state of an object. The example consists of three classes organized as follows:



An instance of class Transformer alternates between reading, transforming and writing depending on the ability to communicate input and output with devices and depending on the state of local input and output buffers.

```
class Reader(Inp: Input-device; max:Integer);
  var ibuffer: sequence of Indata;
      c: Indata;

  action
    ibuffer:=empty;
    do* length(ibuffer) < max, Inp!read(c) ->
        ibuffer:=append(ibuffer,c)
    od
end;

class Writer(Outp: Output-device; max:Integer);
  var obuffer: sequence of Outdata;

  action
    obuffer:=empty;
    do* length(obuffer) > 0, Outp!write(head(buffer) ->
        obuffer:=tail(obuffer)
    od
end;
```

```
class Transformer is-a Reader, Writer;
  var e:Indata; f:Outdata;

  action
    do* length(ibuffer) > 0 ->
        e:=head(ibuffer);
        ibuffer:=tail(ibuffer);
        f:=transform(e);
        if* length(obuffer) < max ->
            obuffer:=append(obuffer,f)
        fi
    od
end;
```

Note that example 4 makes use of the fact that checking the resumption condition in an if* and do* involves full re-evaluation of the guards including re-evaluation of the boolean expressions.

3.6 Inheritance on routines

We allow inheritance on routines in exactly the same way as on classes. A routine declaration contains:

- a name of the routine
- a list of parameters
- a list of super-routines from which parameters, local variables and bodies are inherited
- a list of local variables
- a body consisting of a list of statements.

As for action parts of classes, the bodies are executed alternately, and may have priorities associated (default is that most specialized has highest priority). For simplicity, we assume in this paper that routines can only specify changeover points for their own body, and thus we can still use the symbol "*" without introducing any ambiguities.

Example 5

In the following example, procedure Use is a general description of a well-behaved procedure that requests a printer before using it. The Use procedure can then be used as a super-procedure for procedures that actually make use of the printer.

```
Procedure Use (in p: Printer);
begin
  p ! request;
  *(true);
  p ! release
end;

Procedure Print (in f:sequence of lines) is-a Use;
begin
  do length(f)>0 ->
    p ! printline(head(f));
    f:=tail(f)
  od
end;
```

The example assumes that Printer is a class of resources with operations request, release and printline. If p1 is an instance of Printer - i.e refers to a specific printer, then the call Print(p1,file) will result in acquiring the printer p1 for exclusive access, writing the file on p1 and releasing p1.

Other examples of inheritance on routines will appear in section 3.7, including examples of multiple inheritance and value returning routines.

3.7 Modification of inherited properties

Until now, we have seen that a subclass is made from a number of superclasses by specifying additional properties that instances of the subclass must have compared to instances of the superclasses. It is also possible to make a subclass in which some of the properties inherited from superclasses are modified. Inherited operations and data attributes can be modified in a subclass, subject to some restrictions that ensure that the inheritance is conceptually a specialization hierarchy and that the type system can be statically checked.

First we consider properties inherited from only one superclass:

3.7.1 Singularly inherited properties

An operation from a superclass can be modified in a subclass to be more specialized by means of the inheritance hierarchy on procedures:

```
class A;                          class B is-a A;
  operation P;                      operation P is-a A.P;
  begin body-1 end;                 begin body-2 end;
end;                              end;
```

where A.P refers to the operation P as specified in A. The dot-notation is only available for specification of super-operations. The operation P in B is the combination of body-1 and body-2 obtained by alternation.

If additional parameters of mode "in" are specified in the subclass, default values must be given.

A constant data attribute can be modified in a subclass to have a more specialized type:

```
class A;                          class B is-a A;
  const x:T1;                       const x:T2;
end;                              end;
```

where T2 <= T1.

For a B-object, modifications in B have effect also on the parts of the B-object described in A. Generally speaking, the effect of a modification extends to all class levels of an actual object, also class levels above the level at which the modification is specified. For instance, if operation P above is called on an object c, then the actual class membership of c determines which version of P is executed, regardless of where the call appears - e.g. whether it is a request from another object or a call from some other action part within c itself. (This corresponds to the effect of method activations in Smalltalk [9] and to virtual procedures in Simula [8].)

A few comments on the relation between modification and static type checking are appropriate. The ability to modify properties in subclasses is restricted in such a way that static type checking of the language will still be possible. We want the compiler to be able to guarantee that if B is-a A then all B-objects may safely be treated as A-objects - i.e the polymorphism of the language is safe.

If B is obtained from A by just adding new properties, this is obviously possible.

If B is obtained from A by modification of existing properties, then the rule that a modification must specialize the property ensures that B-objects treated as A-objects will not violate any specifications given in A. The rule that only constant data attributes can be modified, and not variables, ensures that B-objects treated as A-objects cannot violate any type specifications of attributes given in B.

If modification of variable data attributes and parameters is wanted in the language, it should be realized by means of a separate constraint mechanism administrated at runtime like in Galileo [1] in order not to interfere with the static property of the type system.

Safety of operation calls is ensured by the rule that default values must be given for additional in-parameters to specialized operations, and by the following convention: missing actual in-parameters to a call of an operation implies use of the default parameters, whereas missing actual out-parameters are ignored (i.e. copying of value from formal to actual parameters is omitted). This implies that a call of an operation from a place that only knows the general version of the operation can also be considered legal if the actual object that owns the operation is instance of a more specialized class where the operation is specialized with additional parameters.

Now to some examples of modification.

Example 6

Example 6 shows a specialized class obtained by modification of inherited constant data attributes (and perhaps addition of new properties as well). The example can be thought of as part of a university database for planning of examinations.

```
class Examination;
  const c: Course;
        s: Student;
        l: Lecturer;
  var   t:Time;
  ...
end;
```

```
class Graduate-ex is-a Examination;
  const c: Graduate-course;
        s: Graduate-Student;
  ...

end;
```

where we assume that Graduate-course is-a Course and Graduate-student is-a Student.

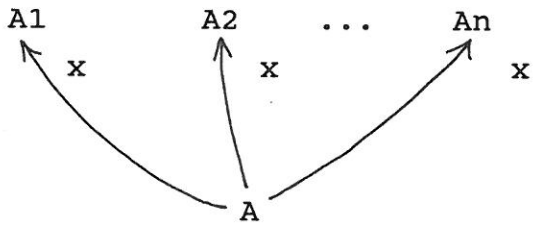
Example 7

Example 7 is part of a graphical window system and shows modification of a value-returning operation.

```
class Window;
  ...
  operation check-cursor (out inside: Boolean);
  begin
    inside:= if cursor is within the window area
             then true else false
  end;
  ...
end;

class Window-with-menu is-a Window;
  ...
  operation check-cursor is-a Window.check-cursor;
  begin
    inside:=inside or (cursor is within menu area)
  end;
  ...
end;
```

Calling check-cursor on a Window-with-menu implies that the two bodies in the two versions of procedure check-cursor are executed alternately. Since there are no explicit changeover points in any of them, the alternation degenerates to a top-down execution of the two bodies. That is, the result is obtained by first executing the top level body of the operation giving an intermediate result in the variable "inside", and then executing the other body that refines the result by means of "or". Hereafter the value of the out parameter is true if the cursor is positioned on the window itself or on its menu area.

3.7.2 Multiply inherited properties



When a class A has a number of superclasses, then a data attribute, parameter or operation named x may be inherited from several superclasses, say A1,...,An. Existing approaches to multiple inheritance have mainly concentrated on how to handle such coincidences of names from different superclasses. (See section 4.)

The approach taken in this paper is to concentrate on intended coincidences where only one version of x is wanted in A and show how the inherited versions of x can be combined into one in an elegant way. Accidental coincidences, where several versions of x are wanted in A, are ignored in this paper. The possibility of combining properties in a subclass adds expressive power to the language, whereas treatment of accidental coincidences is mainly a technical matter, which of course must be taken care of by a complete language. (One way of supporting both possibilities is to do like Thinglab [5] where multiple versions is the default solution, but explicit merging into one version is possible. Choosing this approach, the mechanism in this paper could then be considered a proposal for how to perform the merging.)

For simplicity, however, we assume in this paper that multiply inherited properties must always be combined into one version.
In order for A to be a legal class, one of the following conditions must be satisfied:

1) x is a variable data attribute or parameter with the same type T in all the classes from which it is inherited. x will then have type T in A.

2) x is a constant data attribute or an operation which is modified in class A to be a specialization of all the inherited versions from the superclasses.
That is, if x is a constant data attribute with types T1,...,Tn inherited from A1,...,An respectively, then x must be modified in A to have a type T that satisfies T <= Ti for all i=1..n.
Similarly, if x is an operation, x must be modified in A such that A.x is-a Ai.x for all i=1..n. Default values must exist for all in-parameters that are not common for all the inherited versions of the operation.

3) x is a constant data attribute or an operation for which no explicit modification is present in A. In this case, an implicit modification to the most general specialization of the inherited types/operations will be automatically inserted if possible (see later), otherwise A is an illegal class.

These rules are simple generalizations of the rules for singularly inherited properties described in the previous section, and preserve the possibility of static type checking. Of course, a modification still extends to all class levels of an actual object as described earlier.

Schematical example

```
class A1;                          class A2;
  const x: T1;                       const x: T2;
  ...                                ...
  operation P;                       operation P;
  begin S1 end;                      begin S2 end;
  ...                                ...
end;                               end;

        class A is-a A1,A2;
          const x: T;
          ...
          operation P is-a A1.P, A2.P;
          begin S3 end;
          ...
        end;
```

assuming that T <= T1 and T <= T2.

An A-object will have an x attribute of type T, which thus also satisfies T1 and T2. If operation P is called on an A-object, the three bodies S1,S2 and S3 will be executed alternately.

The strategy for combining operations with the same name inherited from different classes can be thought of as a way of combining different descriptions of the same operations from different perspectives in such a way that all the descriptions contribute to the whole. In this respect, the mechanism is comparable to method combination in Flavors [20].

Before showing a more interesting example, we introduce the concept most general specialization of a number of types or a number of operations - abbreviated mgs:

1) If T1,...,Tn are scalar types, enumeration types or subranges then
   mgs(T1,...,Tn) = intersection-of(T1,...,Tn), if not empty, otherwise mgs(T1,...,Tn) does not exist.

2) If T1,...,Tn are sequence types with element types S1,...,Sn respectively, then
   mgs(T1,...,Tn) = sequence of mgs(S1,...,Sn).

3) If T1,...,Tn are classes, then mgs(T1,...,Tn) is the class that has T1,...,Tn as superclasses and no additional properties, if such a class makes sense. (If some Ti <= Tj then Tj is superflous in the list of superclasses and ignored.)
If x is a multiply inherited constant data attribute from some of T1,...,Tn with inherited types X1,...,Xm then the type of x in the class mgs(T1,...,Tn) is mgs(X1,...,Xm).
Similarly, if x is an operation inherited from T1,...,Tm then x in mgs(T1,...,Tn) will be mgs(T1.x,...,Tm.x).
If x is a multiply inherited variable data attribute with the same type in all the classes from which it is inherited, this type will also be valid for x in mgs(T1,...,Tn). Other kinds of name coincidences in T1,...,Tn will imply that mgs(T1,...,Tn) does not exist.

4) If P1,...,Pn are operations then mgs(P1,...,Pn) is the operation that has P1,...,Pn as super-operations (again ignoring Pj if Pi is-a Pj) and no additional properties. Name coincidences between names in P1,...,Pn are solved as in 3) if possible, otherwise mgs(P1,...,Pn) does not exist.

Note that the definition of mgs is recursive and that if mgs at some level does not exist, then the mgs that started the recursion does not exist. A non-existing mgs corresponds to a number of disjoint or incombinable types or operations.

Example 8

```
class Male;                    class Female;
  const sex:(m);                 const sex:(f);
  ...                            ...
end;                           end;
```

Mgs(Male,Female) does not exist since the enumeration types (m) and (f) are disjoint.

In general, mgs(T1,...,Tn) satisfies that
- mgs(T1,...,Tn) <= Ti for all i=1..n and
- if T <= Ti for all i=1..n then T <= mgs(T1,...,Tn)

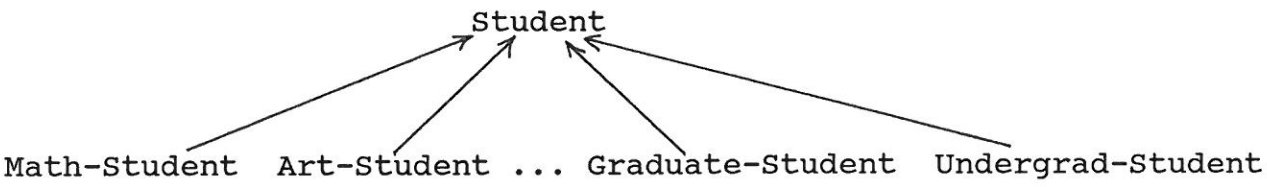
This expresses that mgs(T1,...,Tn) is the most general specialization of T1,...,Tn. Thus mgs is a kind of general intersection mechanism for types and for operations.
Since mgs participates in the <= relation, the discussion of polymorphic types also applies to mgs.
Mgs is used for implicit modification of multiply inherited properties but can also be used as an explicit type expression.

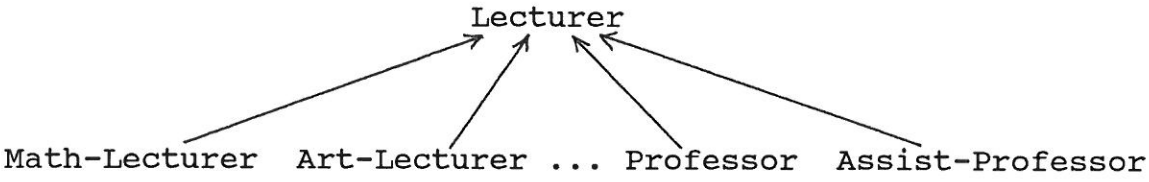
We finish the language presentation with two examples that illustrate the usefulness of our interpretation of multiply inherited data attributes and operations.

Example 9

```
class Examination;
  const s: Student;
        l: Lecturer;
  var   d: Date;
  ...
end;

class Graduate-Ex is-a Examination;
  const s: Graduate-Student;
        l: Professor;
  ...
end;

class Math-Ex is-a Examination;
  const s: Math-student;
        l: Math-Lecturer;
  ...
end;

e: mgs(Graduate-Ex, Math-Ex);
```

In the example, it is assumed that the class-types are organized as follows:

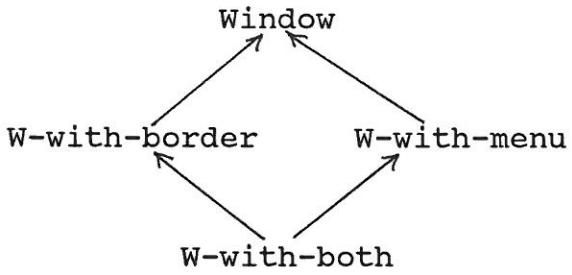




The object e is typed by means of the mgs construct and will have an s-attribute of type mgs(Graduate-Student,Math-Student) and an l-attribute of type mgs(Professor,Math-Lecturer).

Example 10

The following example illustrates how multiply inherited operations are combined. The example is an extension of the previous window example with four classes organized as follows:



The example focuses on the operations Draw, Translate and Check-cursor which are relevant for all four classes. Details that depend on the actual representation of windows are omitted.

```
class Window;
  var ul-corner: Point;
      lr-corner: Point;
      ...
  operation Draw;
  begin
    draw the window contents on the screen
  end;

  operation Translate(in x,y:Integer);
  begin
    translate window contents with the vector (x,y)
  end;

  operation Check-cursor(out inside: Boolean);
  begin
    inside:= cursor coordinates are within window area
  end;
  ...

end;
```

```
class W-with-border is-a Window;
  var border-size: Integer;
      ...
  operation Draw is-a Window.Draw;
  begin
    draw the border around the window
  end;

  operation Translate is-a Window.Translate;
  begin
    translate the border with vector (x,y)
  end;

  operation Check-cursor is-a Window.Check-cursor;
  begin
    inside:= inside or (cursor coordinates are on the border)
  end;
  ...

end;

class W-with-menu is-a Window;
  const menu: ...;
  ...
  operation Draw is-a Window.draw;
  begin
    draw the menu
  end;

  operation Translate is-a Window.Translate;
  begin
    translate menu with vector (x,y)
  end;

  operation Check-cursor is-a Window.check-cursor;
  begin
    inside:= inside or (cursor coordinates are on menu area)
  end;
  ...

end;

class W-with-both is-a W-with-border, W-with-menu;
end;

w:= new W-with-both;
...
w!Translate(a,b);
```

The Draw operation in w will be

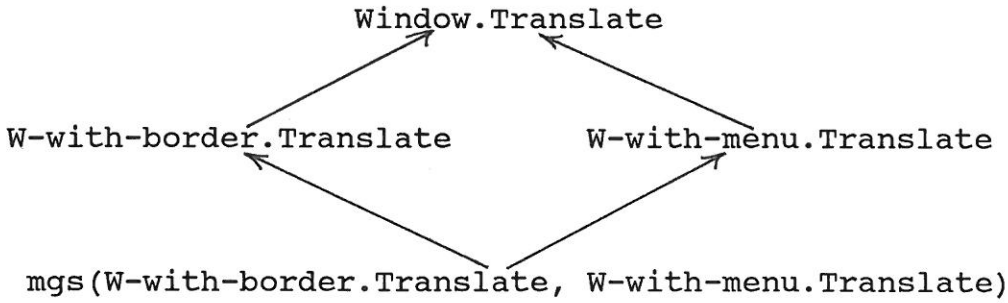
mgs(W-with-border.Draw, W-with-menu.Draw)

Similarly, the Check-cursor and Translate operations in w will be

mgs(W-with-border.Check-cursor, W-with-menu.Check-cursor) and

mgs(W-with-border.Translate, W-with-menu.Translate)

The activation of w's Translate operation will imply that Translate executes its three bodies top-down since there are only final (implicit) changeover points. The example shows operation combination by means of multiple inheritance on operations:



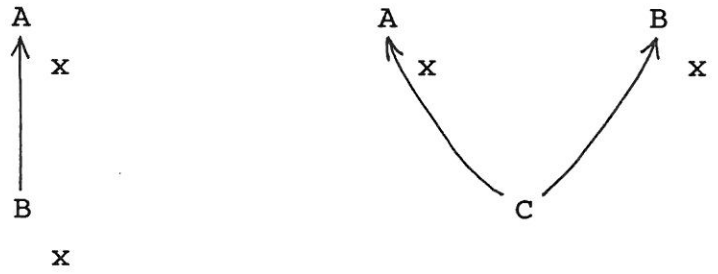
The description of the translation of the window contents is factorized into the common super-operation Window.Translate. Thus, all parts of the window will be translated exactly once with the vector defined by the parameters.

4 RELATIONS TO OTHER WORKS.

Only very few languages with inheritance on processes/procedures have been developed. Simula [8] allows single inheritance on processes and provides a simple procedure like mechanism ("inner") for combining the statement sequences. In [19] a similar mechanism for procedures is proposed and Beta [13] includes both. Taxis [15] allows multiple inheritance on transaction classes and combines the action parts in a simple sequential way. The alternation mechanism in this paper adds considerable expressive power to the inheritance mechanism for processes/procedures compared to these previous proposals. The alternation mechanism is a generalization of the alternation mechanism available between objects in Beta and the programmer controlled interleaving mechanism obtained by when and cycle statements in Distributed Processes [11].

A number of recent languages and language extensions include multiple inheritance on data classes. The main differences between the various proposals lie in the treatment of coincidence of named properties. To ease comparison, a brief analysis of different approaches is appropriate.

Name coincidences may occur either vertically (fig. a below) or horizontally (fig. b below):



a) b)

A,B and C are classes and x is the name of a data attribute or operation defined in A and B.
A name coincidence is usually interpreted in one of the following two ways:

1) x in A and x in B are two different properties and the coincidence of their names is accidental.

2) x in A and x in B are semantically the same property described at different levels of abstraction (in the vertical case) or from different perspectives (in the horizontal case). In the vertical case, any actual B-object will have all references to its x-attribute treated according to the description given in class B. In the horizontal case, the two different descriptions of x must somehow be combined into a description that can be used for x in the class C.

Interpretation 1 is the simplest to implement and gives the convenience that different programmers need not worry about each others choice of names. Pie [10] has chosen interpretation 1 ex-

cept for vertical coincidence of operation names. However, from a conceptual viewpoint, interpretation 2 is much more interesting and offers additional expressive power.

The additional expressive power obtained by interpretation 2 for vertical coincidences compared to interpretation 1 is that it allows specialization of a class by specialization of some of its properties instead of just by adding new properties.

Interpretation 2 for vertical coincidence of named data attributes is provided in Galileo [1] and Taxis [15] in a form very similar to the form presented in this paper. That is, the modified version of the data attribute has a more specialized type.

Interpretation 2 for vertical coincidence of named operations is quite common in the whole Smalltalk inspired family of languages regardless of whether multiple inheritance is included or not: [9],[2],[5],[7],[19]. The mechanism was first introduced in terms of virtual procedures in Simula [8] and is very useful since specializing a class often results in a wish to specialize the operations of the class correspondingly as discussed in section 2. Most of the languages include a "runsuper" mechanism that makes it easy to define the modified version of an operation as an extension of the previous version, but the full step to inheritance on operations is not taken. This is done in Beta [13] where procedures and processes are all described by hierarchically organized patterns.

Concerning horizontal coincidence of names, interpretation 2 makes it possible to describe a concept from different perspectives that are not quite orthogonal. That is, the perspectives may depend on each other by contributing to the description of the same property.
Interpretation 2 for horizontal coincidence of named data attributes is provided in Taxis [15] like in this paper and in Galileo [1] in a restricted version where one of the inherited descriptions of the attribute must be a specialization of the

other. Thinglab [5] provides a merging facility for data attributes that gives a similar effect to the mechanism in this paper.
Interpretation 2 for horizontal coincidence of operations can be obtained in Thinglab [5], Loops [2] and Traits [7] only by explicitly modifying the operation in the common subclass and program it to do whatever combination of the inherited operations is wanted. In the Smalltalk extension in [6] a special primitive "all" is provided to ease a purely sequential combination. Thinglab [5] and the Smalltalk extension [6] require that coincidences are solved by explicit modification, whereas Loops [2] and Traits [7] choose a default version of the operation if no common modification exists. Flavors [20] includes a number of language defined strategies for automatically combining the inherited operations. Flavors has been a major source of inspiration for the work on name coincidences reported in this paper. However, the solution given in this paper is very general and has the conceptual advantages of treating data attributes and operations symmetrically and of using only an extended notion of the already acknowledged inheritance mechanism.

5 CONCLUSION.

It has been demonstrated that inheritance, including multiple inheritance, is a relevant structuring mechanism for both data, processes and procedures. An object oriented language has been outlined in which objects integrate the notions of abstract data structures and processes. Classes are organized in a multiple inheritance hierarchy that results in a coroutine like execution of the different action parts of an object. A similar inheritance mechanism on procedures is available and is shown to offer an elegant solution to the problem of operation combination that arises when several versions of an operation are inherited from different classes and must be combined in a subclass.

Acknowledgement

The work reported here has benefited from many discussions with Ole Lehrmann Madsen and Jørgen Lindskov Knudsen. Thanks are also due to Nigel Derrett, Brian Mayoh, Kirsten Nielsen and Peter Mosses for their comments during the production of this paper.

6 REFERENCES.

[1]: A.Albano, L.Cardelli, R.Orsini: "Galileo: a Strongly-Typed, Interactive Conceptual Language", ACM TODS, vol.10, no.2, June 1985

[2]: D.G.Bobrow, M.J.Stefik: "Loops - Data and Object Oriented Programming for Interlisp" Discussion Papers, European Conference on AI, Orsay, France, July 1982

[3]: A.Borgida, J.Mylopoulos, H.K.T.Wong: "Generalization/Specialization as a Basis for Software Specification", in M.L.Brodie, J.Mylopoulos, J.W.Schmidt (ed.): "On Conceptual Modelling - Perspectives from Artificial Intelligence, Databases and Programming Languages", Springer Verlag, New York, 1984

[4]: A.Borgida: "Features of Languages for the Development of Information Systems at the Conceptual Level" IEEE Software, Jan. 1985

[5]: A.Borning: "The Programming Language Aspects of Thinglab, a Constraint Oriented Simulation Laboratory" ACM TOPLAS, vol.3, no.4, Oct. 1981

[6]: A.Borning, D.H.H.Ingalls: "Multiple Inheritance in Smalltalk-80", Proceedings of the National Conference on AI, Pittsburgh, PA, 1982

[7]: G.Curry, L.Baer, D.Lipkie, B.Lee: "Traits: An Approach to Multiple Inheritance Subclassing", ACM SIGOA Conference on Office Automation Systems, June 1982

[8]: O.J.Dahl, B.Myhrhaug, K.Nygaard: "SIMULA Information, Common Base Language" Norwegian Computing Center, Oct. 1970

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

[10]: I.Goldstein, D.Bobrow: "Extending Object Oriented Programming in Smalltalk", Proceedings of the Lisp Conference, Stanford, California, Aug. 1980

[11]: P.B.Hansen: "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, vol.21, no.11, Nov. 1978

[12]: C.A.R.Hoare: "Communicating Sequential Processes",
Communications of the ACM, vol.21, no.8, Aug. 1978

[13]: B.B.Kristensen, O.L.Madsen, B.M.Pedersen, K.Nygaard:
"Abstraction Mechanisms in the Beta Programming Language",
Proceedings of the 10'th ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983

[14]: B.B Kristensen, O.L.Madsen, B.M.Pedersen, K.Nygaard:
"Multi-sequential Execution in the Beta Programming Language",
Sigplan Notices, vol.20, no.4, April 1985

[15]: J.Mylopoulos, P.A.Bernstein, H.Wong: "A Language Facility for Designing Database-Intensive Applications",
ACM TODS, vol.5, no.2, June 1980

[16]: "Rationale for the Design of the ADA Programming Language",
ACM Sigplan Notices, vol.14, no.6, June 1979, part B

[17]: J.M.Smith, D.C.P.Smith: "Database Abstractions:
Aggregation and Generalization",
ACM TODS, vol.2, no.2, June 1977

[18]: K.S.Thomsen: "Inheritance Used to Factorize Distributed Termination Detection Algorithms"
Dept. of Computer Science, Aarhus University,
DAIMI PB-210

[19]: J.G.Vaucher: "Prefixed Procedures, A Structuring Concept for Operations",
INFOR, vol.13, no.3, Oct. 1975

[20]: D.Weinreb, D.Moon: "Flavors: Message Passing in the Lisp Machine",
Massachusetts Institute of Technology,
AI Memo No.602, Nov 1980

[21]: H.K.T.Wong: "Design and Verification of Interactive Information Systems using Taxis",
TR CSRG-129, University of Toronto, April 1981