Semantics and Implementation of Type Dynamic Modifications[1]

Mohammed Erradi, Gregor v. Bochmann, and Rachida Dssouli

Université de Montréal, Dept. I.R.O.
Faculté des Arts et des Sciences, CP. 6128, Succ. "A"
Montréal, (Québec) Canada H3C-3J7

Email: {erradi, bochmann, dssouli} @iro.umontreal.ca
fax: (514)343-5834

Abstract

The ability to dynamically make a variety of changes, to the persistent object-oriented language type definitions, is an important requirement for systems designers. This is known as schema evolution in object-oriented databases. In this paper,we study type modifications within a persistent object-oriented language that is particularly suitable for distributed systems modeling and specification. To ensure the consistency of the evolving system, where types dynamically change, we introduce two relations: structural consistency and behavioral conformance. While structural consistency deals with the static aspect of the evolving system, the behavioral conformance deals with the dynamic aspect of the system. Then we present a reflection based implementation, using meta-objects, to allow dynamic modifications of types and instances.

Key words: Types, Reflection, object-oriented programming, dynamic change, system evolution.

---

[1] "This research was supported by a grant from the Canadian Institute for Telecommunications Research under the NCE program of the Government of Canada".

1. Introduction

In a wide spectrum of applications, system specifications require modifications to accommodate evolutionary change, particularly for those systems with long expected lifetime. They need to evolve along with changes of human needs, technology and/or the application environment. The changes may require modifications of certain functions already provided by the system, or some extension introducing new functions. In general, evolutionary changes are difficult to accommodate because they cannot be predicted at the time the system is designed [Kram 85]. So, systems should be sufficiently flexible to permit arbitrary, incremental changes.

Software developers or database designers working with an object oriented system are frequently led to modify existing classes so that they suit their needs. This is typically achieved by adding or removing attributes, reimplementing methods, rearranging inheritance links, etc. Such modifications indicate that the existing classes are not entirely satisfactory. Recently, research in class modifications are intensively investigated in object oriented databases field [Bane 87], [Jaco 87], [Skar 87], and [Lern 90]. There, the available methods determine the consequences of class changes on other classes and on existing instances as well, so that possible integrity constraints violations can be avoided. A major concern in designing a class modification methodology is how to bring existing objects in line with a modified class.

Skarra and Zdonik [Skar 87] explore an approach in which filters are placed between instances of an older version of a class and methods that expect instances of a newer version of the class. The Orion system [Bane 87] employs screening on objects presented to an application. The representations of objects are corrected as they are used; this is effectively a late binding on the representation of objects. The Gemstone system [Jaco 87] use the conversion approach; when a class is modified, the system attempt to convert the underlying database to conform to the new class definition and thus maintain a consistent database.

While most of the existing approaches [Bane 87], [Jaco 87], and [Delc 91] address structural consistency, behavioral consistency remains only a design objective. The methodology of Skarra and Zdonik [Skar 87] goes a long way toward preserving behavior. But the problem they address go beyond type modification to versioning of types, objects, and methods. We are exploring solutions to type modification that do not require versioning. All these approaches are restricted to sequential languages, however we designed *RMondel* [Erra 91]i.e., a reflective version of *Mondel*) that is a concurrent object-oriented language suitable for distributed systems modeling and specification, and supports type dynamic modifications. In distributed systems, objects' behaviors are of extreme importance, so behavioral consistency need to be carefully addressed. The existing approaches are all static in that classes can not be changed when the system is operating. In this paper, we present our approach to ensure both structural and behavioral consistencies for dynamic type and instance modifications.

The paper is organized as follows. Section 2 gives an overview of the original *Mondel* language and its important characteristics. Section 3 introduces the structural consistency and behavioral conformance relations. In Section 4, we provide a framework for type modifications, in terms of a set of invariants that correspond to the static semantics rules of the underlying language. We also define the semantics of each type change. Further, through illustrative examples we introduce the allowed behavior modifications where the behavioral conformance relation holds. In Section 5 we address the impact of type modifications on existing instances. Section 6 discusses *RMondel* and the reflection based implementation issues. Conclusions are drawn in Section 7.

2. Mondel Overview

We have developed *Mondel*: An object-oriented specification language [Boch 90] with certain particular features, such as multiple inheritance, type checking, rendezvous communication between objects, the possibility of concurrent activities performed by a single object, object persistence and the concept of transaction. *Mondel* is particularly suitable for modeling and specifying applications in distributed systems. *Mondel* has a formal semantics, expressed by means of a translation into a state transition system. An object is an instance of a type definition (i.e., called class in most object-oriented languages) that specifies the properties that are satisfied by all its instances. Each *Mondel* object has an identity, a certain number of named attributes (i.e., each object instance will have fixed references to other object instances, one for each attribute), and acceptable operations which are externally visible and represent actions that can be invoked by other objects.

A *Mondel* specification corresponds to a type lattice. In such a lattice, types are linked by mean of the inheritance relation. The implementation of such a specification consists of a set of objects (i.e, instances) that run in parallel. Each object has its individual behavior which provides certain details as constraints on the order of the execution of operations by the object, and determines properties of the possible returned results of these operations. Among the actions related to the execution of an operation, the object may also invoke operations on other objects. Basically, communication between objects is synchronous, based on remote procedure call or rendezvous mechanism. An operation call is syntactically represented by the "**!**" operator. For instance in the statement *m**!** InsertCoin* (see line 27 of Fig.2.3.), "*m*" designates the called object, and *InsertCoin* is an operation defined within the type of "*m*" (i.e., the type *Machine*). In the following we discuss those aspects of *Mondel* which are necessary for the discussions of type modifications.

Each *Mondel* object is of a given type. A type definition specifies the properties that are satisfied by all instances of that type.

definition1: An object type definition t consists of an interface I_t and a behavior B_t definitions : $I_t = \{ A_t, Op_t \}$ where: A_t is the set of attributes and Op_t is the set of operations; and B_t is the behavior definition for objects of the type t.

[*end of definition1*]

2.1. *Mondel* statements

Objects behaviors are specified using *Mondel* statements. In the following, we consider a subset of *Mondel* language as shown in Fig.2.1. This *Mondel* subset will be considered for the modification of objects' behaviors. For a full description of *Mondel* , we refer the reader to [Boch 90].

- Attr **!** OpName	: call of the operation "OpName" on the object refered by "Attr".
- **accept** OpName **do** Stat **end**	: acceptance of an operation. "Stat" can be one of the statements listed here.
- **return**	: the end of a rendezvous.
- ProcName	: procedure instantiation.
- Stat1 **;** Stat2	: sequential composition.
- **choice** Stat1 **or** Stat2 **end**	: either "Stat1" or "Stat2" is executed.
- **Parallel** Stat1 **and** Sat2 **end**	: "Stat1" and "Stat2" are executed in parallel (pure interleaving).
- **Loop** Stat **end**	: cyclic behavior can also be defined implicitly, by recursive procedure call.

Fig.2.1. subset of *Mondel* Statements

Mondel has a formal semantics which associates a meaning to the valid language sentences. The formal semantics of Mondel was defined based on the operational approach. In this approach an abstract machine simulates the real computer role. The meaning of a specification is expressed in terms of actions made by the abstract machine. We more particularly applied the technique of Plotkin [Plot 81] where state/transition systems are taken as machine models. The *Mondel* formal semantics is the basis for the verification of *Mondel* specifications [Barb 90b], and has been used for the construction of an interpreter [Will 90].

2.2. Objects structure

Each *Mondel* object has the following aspects:

- An identity: Objects obtain a system wide identifier when they are created. The identifier of an object serves as a reference to it and is used to refer to the object when it is passed as an actual attribute to a newly created object, or as a parameter or a result of an operation.

- Attributes: An object type may include a certain number of named attributes. This means that each object instance of that type will have a fixed references to other object instances, one for each attribute. An attribute may be declared non-visible; by default, an attribute is visible which means that any object "knowing" the object may also access its attributes. It may also be declared internal, which means that it is defined by the internal behavior of the object; otherwise it must be provided as effective parameter when the object instance is created.

- Operations: They define the functions and procedures that the object can accept during execution. The operations are externally visible and represent actions that can be invoked by other objects. An object may have internal procedures which can be called from within the object behavior.

- Typing: *Mondel* supports strong type checking based on the declared object types. Generic types (i.e. with type parameters) are also supported. Therefore the type consistency of the effective parameters of operation invocations and object instantiations can be checked by a compiler.

- Behavior: It provides certain details as constraints on the order of execution of operations by the object, and also determines properties of the possible returned results of these operations. Among the actions related to the execution of an operation, the object may also invoke operations on other objects.

- Inheritance: Types can be related to each other by means of the inheritance relation. *Mondel* allows a form of multiple inheritance where a given type may inherit from several supertypes as long as the inherited properties are without conflicts.

2.3. Example

In the following we show an example using *Mondel* language. This example will be used through the paper. Let us consider a vending machine which receives a coin and delivers candies to its user. In this example, we suppose that the machine delivers only candies. We distinguish two types of objects: the type *Machine* and the type *User*, as shown in *Mondel* specification of Fig.2.3. The relation between the *Machine* and the *User* is expressed by the fact that the user knows the machine. Such a relation is modeled by the attribute “*m*” defined in the *User* type.

The user is initially in a *Thinking* state, and when he decides to buy a candy he inserts a coin. After the coin has been accepted, the user enters the *GetCandy* state. Then the user pushes the machine's button to get a candy. Once the candy is delivered, the user enters the *Thinking* state again. The machine is initially in the *Ready* state, ready to accept a coin. Once a coin is inserted, the machine accepts the coin and then enters the *DeliverCandy* state. After the user has pushed the button of the machine, the latter delivers a candy and becomes *Ready* to accept another coin. Fig.2.2 shows the main states and transitions diagrams of the vending machine example.

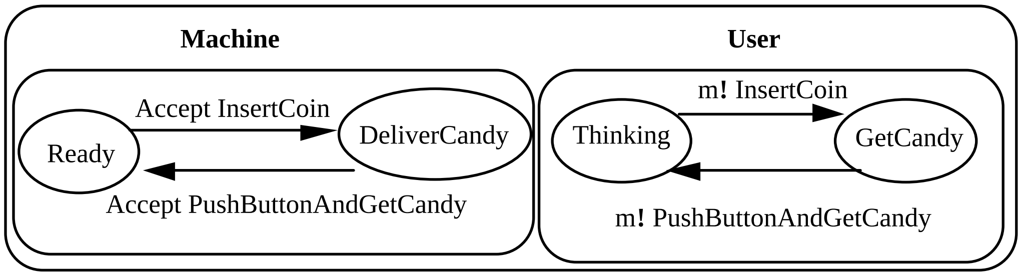

Fig.2.2. State/transition diagram of the vending machine example

Note that object operations model the occurrences of events. The behavior of the vending machine system is defined as the composition of interacting objects (i.e., *Machine* and *User* objects). The object types are specified using a state oriented style [Viss 88]. Each object internal state is modelled as one *Mondel* procedure. We interpret the operations of an object of type *Machine*, as follows:

InsertCoin : the machine object is ready to accept a coin from the environment, and when the coin is inserted the machine changes its state to become ready to deliver a candy.

PushButtonAndGetCandy : The machine object waits for the button to be pushed after a coin has been inserted, and the machine delivers a candy.

```
type Machine = object with
operation
  InsertCoin;
   PushButtonAndGetCandy;
behavior
   Ready
where
   procedure Ready =
       accept InsertCoin do
             return;
          end;
        DeliverCandy;
   endproc Ready

 procedure  DeliverCandy =
   accept PushButtonAndGetCandy do
      return;
   end;
   Ready;
endproc DeliverCandy

endtype Machine
```

```
type User = object with
 m: Machine;
behavior
   Thinking
where
  procedure Thinking =
        m! InsertCoin;
        GetCandy;
  endproc Thinking

  procedure  GetCandy =
    m! PushButtonAndGetCandy;
    Thinking;
   endproc GetCandy

endtype User
```

Fig.2.3. *Mondel* specification of the *Machine* and *User* types

3. Consistency relations

The change of the structure and behavior of types and/or named objects, must be done without resulting in run-time errors, blocking, or any other uncontrollable situation. So the semantics of type changes should ensure that a modified system (i.e., executable specification) remains consistent. For this purpose our interpretation of system consistency consists of the composition of two relations. The first relation maintains a *structural consistency*, while the second is concerned with *behavioral conformance*. In the following we introduce the basic definitions that are useful for type modifications.

definition2: An object type interface $I_{t'}$= { $A_{t'}$, $Op_{t'}$ } is *compatible* with another object type interface I_t = { A_t, Op_t } if and only if:
- The type t' has (at least) all the attributes defined for the type t (i.e., $A_{t'}$ □ A_t); the inherited attributes may be more specialized.
- The type t' has (at least) all the operations defined for the type t (i.e., $Op_{t'}$ □ Op_t), where the operations result must be *compatible* and the input parameters must be inversely *compatible* [Blac 87].

[*end of definition2*]

definition3: An object type t' is *structurally consistent* with an object type t if and only if: $I_{t'}$ is *compatible* with I_t, where $I_{t'}$ and I_t are the interfaces of t' and t respectively.

[*end of definition3*]

If we ignore operations parameters, our interpretation for the *behavioral conformance* relation that we note *conform*, will be similar to the *extension* relation defined for LOTOS specifications [Brin 86]. LOTOS is an internationally standardized formal description technique designed for the specification of OSI protocols and services. We introduce the *conform* relation as follows:

definition4: The behavior defined for the object type t' *conforms to* the behavior defined for the object type t if the following properties are satisfied:
property1. Any object of type t' does what is explicitly allowed according to the type t (but it may do more).
property2. What an object of type t' refuses to do (i.e., blocking), after any behavior that is explicitly specified, can be refused according to the type t (an object of type t' may not "refuse more").

[*end of definition4*]

It is important to note that for many authors the concept of inheritance is only concerned with the names and parameter types of the operations that are offered by the specified type as, for instance, in *Emerald* [Blac 87] and *Eiffel* [Meye 88]. However, there are other important aspects to inheritance which considers comparing the dynamic behavior of objects [Amer 87], [Amer 89], including constraints on the results of operations, the ordering of operation execution, and the possibilities of blocking [Boch 89].

Our interpretation of inheritance can be defined by taking into account the dynamic behavior of objects as follows:

definition5: An object type t’ **inherits** from an object type t if and only if :
$I_{t'}$ is *compatible* with I_t.
and t’ *conforms to* t .
[*end of definition5*]

The constraints defined by the above definition, will be used to ensure the system consistency while the system changes.

4. Type definition Modifications

We are mainly interested in the modifications of a system S which lead to a consistent system S' using an incremental approach. The incremental approach consists in building the system S' by successive enhancements to the existing system S. Such a system consists of a type lattice, where nodes represent types and edges represent the inheritance relationship. So the modification of the system corresponds to the modification of the types and/or the type lattice.

In this section, we present our framework for both object type structure and behavior modifications. For the structure modification we introduce a set of properties called invariants that must be preserved to ensure *structural consistency*. Then we introduce the basic type modification primitives, and we define their semantics. For the behavior modifications we study, through some examples, the behavior modifications allowed by the *behavioral conformance* relation.

4.1. Structure modifications

The modifications of types structures must be done in a way to ensure that the lattice remains consistent, and the objects instances conform in some way to the modified types.

We define a set of invariants that must be satisfied by each type and its related types in the lattice.

4.1.1. Structural consistency

In this section, we discuss the invariants that must be preserved across *Mondel* type modifications. The invariants define mainly the consistency requirements of the type lattice, which corresponds to the static semantic rules of *Mondel*. The type lattice and full inheritance invariants are similar the those presented for ORION [Bane 87].

Type Lattice Invariant

The type lattice is seen as a directed acyclic graph, where the root is a system-defined type called OBJECT , and each node (i.e., a type) is reachable from the root. Each type in the lattice has a unique name.

Distinct Name Invariant

All attribute and operation names of a type, wether defined or inherited, are distinct.

Object Representation Invariant

Each object in the system is an instance of a type. So the object’s structure must be as specified by its type.

Full Inheritance Invariant

A type inherits all attributes and operations from each of its supertypes. Name conflict is not addressed here, but may be avoided in a similar way as in [Delc 91].

Type Compatibility Invariant

If an attribute A2 of a type T is inherited from an attribute A1 of a supertype of T, then the type of A2 is either the same as that of A1, or a subtype of the type of A1.

In order to keep a system in a consistent state, these invariants must be preserved by each type (or instance) and its related types in the lattice. The invariants are checked when an object is created, in order to maintain compatibility rules between attribute values and their corresponding types. These invariants are also checked during type updates.

4.1.2. Operations for type structure modifications

In this section, we classify all type modifications that we support in *RMondel*, and define the semantics of type modifications based on the invariants introduced above.

Basic type update primitives.

Updates are classified in three categories: Updates to the type structure which corresponds to the contents of a node in the type lattice, to a node in the type lattice, and to an edge in the type lattice. In the following we present a list of basic updates one can perform on a type specification.

1. Modifications to the contents of a node in the type lattice.
    - 1.1- Modifications to an attribute of a type.
        - 1.1.1. Add an attribute to a type.
        - 1.1.2. Drop an existing attribute from a type.
        - 1.1.3. Change the type of an attribute.
    - 1.2. Modifications to an operation of a type.
        - 1.2.1. Add an operation to a type.
        - 1.2.2. Drop an existing operation from a type.
        - 1.2.3. Change the signature of an operation.

2.Modifications to an edge of the lattice.
    - 2.1. Make a type T a supertype of type S.
    - 2.2. Delete a parent (supertype) of a type.

3. Modifications to a node of the lattice structure.
    - 3.1. Add a new type.
    - 3.2. Delete an existing type.

Semantics of type updates

In this Section we provide a description of the semantics of the basic update operations performed on types.

1. Modifications to the contents of a node in the type lattice.

1.1. Modifications to an attribute of a type.

Add an attribute A to a type T: this update allows the user to append an attribute definition to a given type definition. We suppose that the added attribute A causes no name conflicts in the type T or any of its subtypes.

Drop an existing attribute A from a type T: this update allows the deletion of the attribute A from the type T. A must have been defined in the type T; it is not possible to drop an inherited attribute.

Change the type T of an attribute A: we assume that the type T of an attribute A can be only specialized to a type T1. In other words T1 inherits from T.

1.2. Modifications to an operation of a type.
Add the operation O to the type T: This update allows the user to append the operation O to the type T. We suppose that the added operation O causes no operations name conflicts in the type T or any of its subtypes.

Drop the existing operation O from the type T: This update allows the deletion of the operation O from the type T. O must have been defined in the type T; it is not possible to drop an inherited operation.

Change the signature S of the operation O.
a) Change the type T of the parameter p in S: This update allows the change of the type T of the parameter p in S, to become T'. This update must be done according to the definition3 above which ensures that the change is allowed only if T inherits from T'.

b) Change the type T of the result, if any, of the operation O: This update allows the change of the type T of the result to become of type T'. This update is allowed only if T' inherits from T, as stated in definition3.

c) Drop an operation parameter: This update allows the suppression of an operation parameter. When parameters disappear from the operation O defined in the type T, this is an indication that the objects of type T requires less information to carry out the same service. One can assume some default value for the droped parameter.

d) Add an operation parameter: This update allows the addition of an operation parameter. When parameters are added to the operation O defined in the type T, this is an indication that the objects of type T requires more information to carry out the same service.

2.Modifications to an edge of the lattice.

2.1. Make a type T a supertype of type S: This modification is allowed only if it does not introduce a cycle in the inheritance graph. The attributes and operations provided by T, are inherited by S and by the subtypes of S.

2.2. Delete a parent S (supertype) of the type T: The deletion of an edge from T to S must preserve the type lattice invariant. This must not cause the type lattice to be disconnected. If S is the only supertype of T then the immediate supertypes of S become the supertypes of T. T does not loose the features (attributes and operations) that were inherited from the supertypes of S. T will only loose those features that were defined in S.

3. Modifications to a node of the lattice structure.
3.1. Add a new type T: If no supertypes of T are specified, then the type OBJECT (i.e. the root of the type lattice) is the default supertype of T. If supertypes are specified, then the inheritance invariants defined previously requires that all attributes and operations from the supertypes are inherited by T. The name of the added type T must not be used by an already defined type. The specified supertypes of T must have been previously defined.

3.2. Delete an existing type T: The edges from the subtypes of T are dropped using operation 2.2. The edges from T to its supertypes are aslo dropped, and T is then removed from the lattice. If T was the type (domain) of an attribute A of another type T1, then A is assigned a new type

4.2. Behavior Modifications

Our purpose for the modification of the behavior part of types definitions, is to extend the existing behavior to meet new requirements. This is similar to the notion of incremental specifications proposed for a subset of basic LOTOS language [Ichi 90]. However, LOTOS which is an internationally standardized formal description technique designed for the specification of OSI protocols and services, was not concerned with the object-oriented approach. It mainly focus on the temporal ordering of events. The idea behind incremental specifications is to obtain a new specification (i.e., a new system) by giving additional specification descriptions to the initial existent specification description.

We believe that the most important case of change, w.r.t. the incremental approach, is the addition of operations, we distinguish many possibilities of behavior definition modification according to this case. The other cases of behavior modifications such as operation deletion may be of interest for specifications designer, but these cases will not be addressed in this paper. The possibilities of behavior definition modifications, presented here, are based on the language constructs which can be involved in such modifications as described in the following sections.

The behavior of objects is to a degree dependent upon preserving *structural consistency*. For instance, when an operation is called on an object, the operation associated code (i.e.,method) to be executed is determined by the object's type or supertypes. Additionally, once the operation code is located, its implementation is dependent on the called object' structure. This structure has to be present in all objects that are instances of the type where the operation is defined. So, changes to the type interface may lead, in most cases, the user to change the behavior definition accordingly. Sometimes, one need only to change the behavior definition without changing the interface. We distinguish two categories of behavior definition change: The first category consists in changing the behavior definition according to changes in the type interface, and the second category consists in changing the behavior definition while the type interface remains unchanged.

4.2.1. Behavior changes according to interface changes.

In this section, we introduce, through illustrative examples, the allowed behavior modifications where the *behavioral conformance* relation holds. The behavior modifications are based on the language statements presented in Section 2.1. It is important to note that we consider only finite behaviors for the behavior modifications presented in this section. This restriction, to finite behaviors, allows the preservation of the *behavioral conformance* relation.

a. Sequential composition.

Suppose that we want to modify the vending machine specification given in Fig.2.3, to give a gift to its user after each purchase. We modify the type *Machine*'s interface by adding the *GetGift* operation. The code associated to the *GetGift* operation is added in the type *Machine* 's behavior definition in sequence with the existing behavior as shown in Fig. 4.1.

```
type Machine = object with
operation
    ...
    GetGift;
behavior
    ...
 procedure  DeliverCandy =
   accept PushButtonAndGetCandy do
      return;
   end;
   accept GetGift do
      return;
   end;
   Ready;
 endproc DeliverCandy
endtype Machine
```

```
type User= object with
   m: Machine;
behavior
    ...
   procedure  GetCandy =
    m! PushButtonAndGetCandy;
    m ! GetGift;
    Thinking;
   endproc GetCandy

endtype User
```

Fig. 4.1. Added operation in sequence

In this case the type *Machine* of Fig.2.3 is modified by adding the *GetGift* operation, this leads to the modified type *Machine* given in Fig.4.1. The behavior definition is modified in a way to allow the execution of the*GetGift* operation in sequence after the execution of the operations defined previously. According to the temporal constraints, the *GetGift* operation can be accepted only after a candy purchase. So, the modification illustrated above is allowed according to the *conform* relation of definition4. Any object of type *Machine* of Fig.4.1, accepts the *PushButtonAndGetCandy* operation as any object of the initial type *Machine*. If we ignore the loop defined by the recursive call of *Ready* procedure (i.e.,we consider only a finite behavior of the machine), then an object of the modified type *Machine* does not block where an object of the initial type *Machine* does not.

The modification of objects based on the sequential composition of behaviors, as defined above, satisfies the consistency requirements. These requirements consists of maintaining interfaces *structural consistency* and behaviors conformance as well.

b. Constrained choice operator.

It has been shown that the choice operator does not guarantee subtyping [Rudk 91], because non-determinism can be introduced. For instance, the combination of recursion and choice may lead to a violation of the second property of definition4. Also, if two behaviors are combined by the choice operator, and these two behaviors have non-empty intersection of their initial actions, then non-determinism is introduced. In the following we distinguish two cases:

Deterministic case

We can introduce the behavior associated with an added operation using the choice composition operator. Suppose that we want to modify the vending machine of Fig.2.3 in order to allow its user to buy wether a candy or a chocolate. The *PushAndGetChocolate* operation is added in the type's interface, and the behavior associated to such an operation is introduced by mean of the choice operator as shown in Fig.4.2.

```
type Machine= object with
operation
   (same as in Fig.2.3.)
   PushAndGetChocolate;
behavior
    Ready
where
  procedure Ready =
     accept InsertCoin do return; end;
     choice
               DeliverCandy;
     or        DeliverChocolate;
     end;
   endproc Ready

 procedure  DeliverCandy =
   (same as in Fig.2.3.)
 endproc DeliverCandy

 procedure  DeliverChocolate =
   accept PushAndGetChocolate do
       return;
   end;
   Ready;
  endproc DeliverChocolate

endtype Machine
```

```
type User = object with
   m: Machine;
behavior
     Thinking
where
   procedure Thinking =
         m! InsertCoin;
         GetCandy;
   endproc Thinking

   procedure  GetCandy =
     m! PushButtonAndGetCandy;
     Thinking;
    endproc GetCandy

endtype User
```

Fig.4.2. Added operation within a choice: deterministic case

The modification illustrated above is allowed according to the *structural consistency* and *behavioral conformance* relations of definition3 and definition4 respectively. For the *structural consistency* relation, it is easy to check that the modified type *Machine* interface in Fig.4.2 is *structurally consistent* with the initial type *Machine* interface defined in Fig.2.3. For the *behavioral conformance* relation, both properties of definition4 are satisfied. So, an object of the modified type *Machine*, accepts the same operations in the same order as any object of the initial type *Machine*. Also the behavior of an object of the modified type *Machine*, does not block where an object of the initial type *Machine* does not. We conclude that the behavior, defined in the modified type *Machine*, conforms to the behavior defined within the initial type *Machine*.

Non-deterministic case

Suppose that we have a vending machine, defined by the type *Machine2* that delivers coffee as shown in Fig.4.3. The type *Machine2* is defined in a similar way as the initial type *Machine* of Fig.2.3. Let us consider that we modify the initial *Machine* in order to also provide the behavior defined by the type *Machine2*. The interface of the modified type *Machine*, shown in Fig4.3, is *structurally consistent* with both the interfaces of the initial type *Machine* and of the type *Machine2*. However, the behavior defined by the modified type *Machine*, obtained as the combination of the initial *Machine* and the *Machine2* behaviors, does not satisfy the second property of the *behavioral conformance* relation. This is because the behavior of the modified *Machine* introduces non-determinism. This non-determinism is illustrated by the existence of two branches with the same initial action (i.e., *InsertCoin* operation). The introduced non-determinism can be removed by combining the initial common actions as shown in Fig.4.4.

```
type Machine = object with
 operation
   InsertCoin;
   PushButtonAndGetCandy;
   PushButtonAndGetCoffee;
 behavior
   Ready
 where
  procedure Ready =
    choice
          CandyProc;
   or     CoffeeProc;
   end;
  endproc Ready

  procedure CandyProc=
      accept InsertCoin do return; end;
      DeliverCandy;
  endproc CandyProc

  procedure CoffeeProc=
      accept InsertCoin do return; end;
      DeliverCoffee;
  endproc CoffeeProc

  procedure  DeliverCandy =
      (same as in Fig.2.2.)
  endproc DeliverCandy

  procedure  DeliverCoffee=
   accept PushButtonAndGetCoffee do return;
   end;
   Ready;
  endproc DeliverCoffee
endtype Machine
```

```
type Machine2 = object with
operation
   InsertCoin;
   PushButtonAndGetCoffee;
behavior
   Ready
where
 procedure Ready =
      accept InsertCoin do
           return;
       end;
      DeliverCoffee;
  endproc Ready

 procedure  DeliverCoffee =
   accept PushButtonAndGetCofee do
      return;
   end;
   Ready;
 endproc DeliverCoffee

endtype Machine2
```

Fig.4.3. Added operation within a choice: non-deterministic case.

```
type Machine = object with
  operation
    ( as in Fig.4.3)
  behavior
    Ready
  where
    procedure Ready =
      accept InsertCoin do return; end;
      choice
            DeliverCandy;;
      or    DeliverCoffee;
      end;
    endproc Ready

 procedure  DeliverCandy =
        (same as in Fig.2.3.)
   endproc DeliverCandy

   procedure  DeliverCoffee=
     accept PushButtonAndGetCoffee do return;
     end;
     Ready;
   endproc DeliverCoffee

endtype Machine

type Machine2 = object with
   (as before )
endtype Machine2
```

Fig.4.4. Combination of common actions to remove non-determinism

c. Parallel composition.

It has been shown, for a subset of basic LOTOS language,that the behavior B obtained by the combination of two behaviors B1 and B2, using the parallel operator, satisfy the following properties [Ichi 90]:

- B *extends* B1 and B *extends* B2.

In a similar way, *Mondel* objects behaviors satisfy such properties w.r.t. the *behavioral conformance* relation of definition4. So, the behavior B obtained by the combination of two objects behaviors B1 and B2, using the parallel operator, satisfy the following properties: B *confroms to* B1 and B *confroms to* B2.

One kind of behavior modification is the combination of two behaviors using the parallel operator (i.e., pure interleaving). The following discussion, illustrate these properties through the vending machine example. Suppose that we have two machines, the initial one delivers candies (see Fig.2.3) and the second one (i.e., type *Machine2*) delivers coffee as shown in Fig.4.5. We want to modify the initial machine by combining its behavior with the behavior of the second machine, using the parallel operator. The obtained machine (modified initial machine) should behave like both machines, it should deliver either candies and coffee.

```
type Machine = object with
  operation
    InsertCoin;
    PushButtonAndGetCandy;
    InsertCoin2;
    PushButtonAndGetCoffee;
  behavior
    Ready
  where
    procedure Ready =
      parallel
                CandyProc;
        and     CoffeeProc;
      end;
    endproc Ready

    procedure CoffeeProc=
         accept InsertCoin2 do return; end;
         DeliverCoffee;
    endproc CoffeeProc

(These procedures remains as in Fig.4.3)
    procedure CandyProc= ...
    procedure  DeliverCandy = ...
    procedure  DeliverCoffee= ...

endtype Machine
```

```
type Machine2 = object with
operation
    InsertCoin2;
    PushButtonAndGetCoffee;
behavior
    Ready
where
  procedure Ready =
        accept InsertCoin2 do
             return;
         end;
        DeliverCoffee;
   endproc Ready

 procedure  DeliverCoffee =
   accept PushButtonAndGetCofee do
      return;
   end;
   Ready;
 endproc DeliverCoffee

endtype Machine2
```

Fig.4.5. Parallel composition (pure interleaving).

The modification of objects based on the parallel composition of behaviors satisfy the consistency requirements. These requirements consists of maintaining interfaces *structural consistency* and *behavioral conformance*.

4.2.2. Behavior change while the interface remain unchanged

Another aspect of type modifications is performance enhancement. These modifications has no impact on the interface of the modified object type. In this case only the implementations of the operations, are modified. The modified object behavior provides the same services, through its interface, as the old behavior (i.e., before modification). These modifications should not lead to behaviors that blocks more than the old behavior. In other words, these modifications should maintain the consistency requirements.

5. Repercussions of type changes on existing instances

Transforming all instances whose type has been modified seems like the most natural approach for dealing with change propagation. In this section we will analyze the impact of each type modification on existing instances. In order to maintain the consistency between types (specifications) and their instances (implementations), instances need to be converted (physically updated) so that their structure matches the description of the the type they belong to.

5.1. Impacts on the instances structure

1.1. Modifications to an attribute of a type.

1.1.1. Add an attribute to a type: Adding an attribute to a type leads to the logical addition of the attribute to all instances of the type and to those of the subtypes inheriting the attribute. A nil value is given by default to the added attribute.

1.1.2. Drop an existing attribute A from a type T: This implies the deletion of the attribute A from all instances of the type T and from those of its subtypes. Let us note here that removing an attribute may lead to additional problems regarding the dynamic behavior of the affected instances. For example, an object instance I may call an operation O accepted by the object refered by the attribute A, if the behavior of I was not changed according to the deletion of A, then an execution problem arises. This is because A becomes undefined within the behavior of the instance I.

1.1.3. Change the type T1 of an attribute A defined within a type T: We have seen that the type T1 of an attribute A can only be changed by T2 a specialization of T1 . So instances of the type T are not affected by this change because the operations accepted by the instances of T1 remain accepted by those of T2.

1.2. Modifications to an operation of a type: There is no impact on the existing instances of the type T. Operations appear only in the type definition. However, the modifications of an operation of a type (addition and/or suppression of an operation, and the change of the signature of an operation) may have an impact on the dynamic behavior of the existing instances of the type T and those of its subtypes.

In order to allow for dynamic type modifications, we develop a technique that uses meta-objects. More details on our implementation technique will be given in Section 6.

5.2. Impact on the instances dynamic behavior

According to type modification, the impact of such modifications on the existing instances dynamic behavior needs to be carefully addressed. The main question is that the instances (implementations) should conform to their types (specifications). To ensure behavioral conformance, the instances behaviors should be modified according to the modifications of the behavior defined by their types. So when and how can we make instances behaviors conversion? To answer this question, we introduces the concept of meta-object. To each object, we associate a meta-object which is responsible for monitoring and modifying the object's behavior. More details on the proposed technique will be given in the following section.

6. Reflection based implementation

To support the dynamic modification of objects structure and their behavior, we developed *RMondel*, a reflective version of *Mondel*, to provide a framework for the construction of flexible systems specifications [Erra 91]. In order to allow for the construction of dynamically modifiable specifications, we need to have access, and to be able to modify type definitions during run-time. So types are instances of *TYPE*, which is a system predefined object, as shown in Fig.6.1. Note that *TYPE* provides primitive operations for type modifications. More details on reflection in *RMondel* are the subject of a forthcoming paper.

```
type TYPE = OBJECT with
            TypeName          : string;
            Statdef           : Statement;
operation
  New  : OBJECT;
  <: (t : TYPE): boolean; {(see Fig.3)}
    AddAttr (A:Attribute);
    DelAttr(A: AttrName);
    AddOper(O:Operation);
    DelOper(O:Operation);
    AddStat(S:Statement);
    DelStat(S:Statement);
    ...
invariant
{ We define here, the constraints which must hold to maintain the system in a consistent state.These constraints define
the consistency requirements of the type lattice which corresponds to the static semantics rules checked by the Mondel
compiler.}
behavior
{ We specify here, in which order the operations, provided by an object of type TYPE, can be executed and what the
possible returned results are. }
endtype TYPE
```

Fig.6.1. The definition of *TYPE* object

In order to manage types evolution and to maintain consistency between types and their instances, we define two types of meta-objects which inherits from the *INTERPRETER* type, as shown on Fig.6.2.

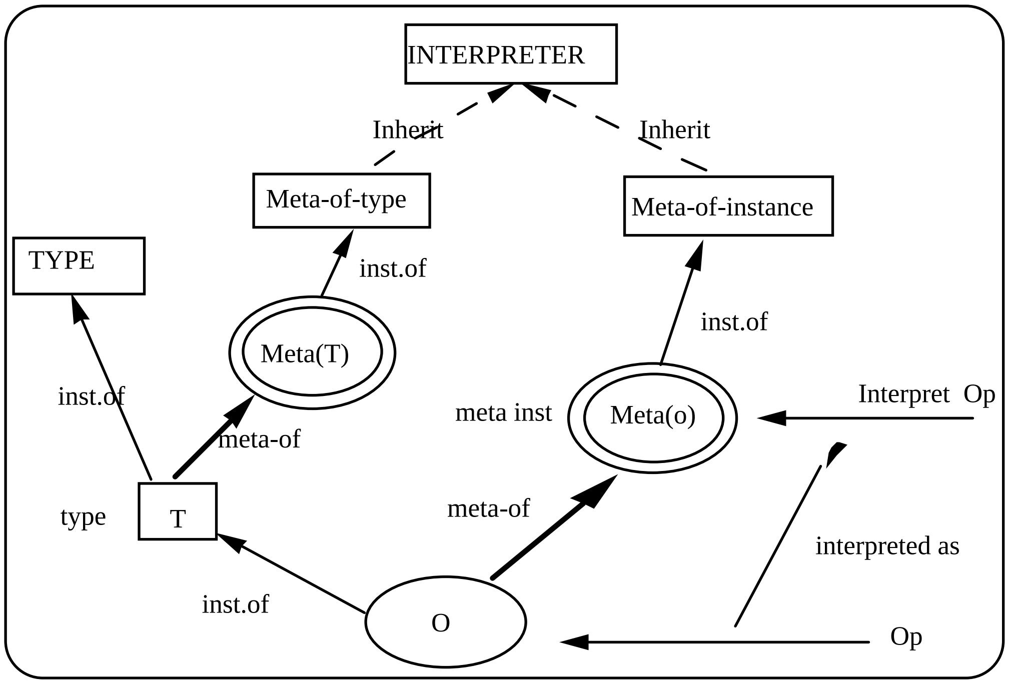

Fig.6.2. Meta-Objects for type and instance dynamic modification

The *INTERPRETER* type defines the behavior which consists of the interpretation of *RMondel* statements. An instance of the Meta-of-type type is associated to a type object, and can hold information about the type object definition evolution. An instance of the Meta-of-instance type, is associated to an object instance, and holds information about the used types within the object instance behavior. The information about the types used by the object instance behavior, will be useful for the management of the instances conversion after the object type modification.

We assume that the modifications of a type object are done as an atomic operation, this to ensure that no conversion is done until the whole modifications have been completed. We assume also, that instances are converted only when they are accessed. When an execution problem arise, due to the violation of the consistency relations, then the recovery mechanism associated with the atomic operation, will be invoked to bring the system to a state before the system modifications.

For a running(i.e., in execution) object behavior, one needs information about the types used in that behavior expressions to know if these types are changed or not. So, before accessing an object of a given type, we have to check if the object type has been changed. In the case where the object type has been changed, then the associated instances have to be converted according to the modified type.

When an object type t accepts an operation, for instance the *AddAttr* Operation, as an object t needs to change its state accordingly. After the addition of the attribute definition to the type object t, then t is considered as finishing its job. As we need, some time later, to convert the instances of t to the modified type, we define the *Convert* procedure within the meta-object associated to the object type.

The type object always holds the newest definition. We define an operation *CurrentDef* within the meta-object of the type, to return the current type definition. Also, let *OldDef* be an attribute, of the meta-object, which holds the references list of the old versions of the type. Before the usage of an instance i of a given type t, the t's meta-object has to check if there is no change in t's definition since the last use of t. If there is a change in the type, the conversion of its instances will be immediately invoked through the *Convert* procedure defined in t's meta-object. Once the conversion is done, the current definition of t is added to the *OldDef* list. The operation *CurrentDef* returns the newest definition of the type.

Two points need to be carefully addressed to ensure the system consistency according to dynamic changes.
1- Instances conversion according to the modified type.
2- The objects behaviors using the old instances, have to be able to access the converted instances.

In *RMondel,* instances conversion from old instances to new ones, can be made without changing objects identities. We have seen how meta-objects for types are defined to store the information related to type changes. We need to mange the use of types to allow the use of converted instances. We define an attribute *UsedType* in the meta-object (i.e. *Meta-of-instance* type of Fig.6.2.) of the instance the behavior of which refers to such types.When an object's behavior execution starts, this object's meta-object (instance of *Meta-of-instance*) calls the operation *Verify* on the meta-objects of the used types. The operation *Verify* is defined within the *Meta-of-type* type, to check if some used types are changed. This call, may trigger instances conversion if some types are changed. We use a pseudo Mondel to give an outline of *TYPE*, *Meta-of-type*, and *Meta-of-instance* types specifications, as shown in Fig.6.3.

7.Conclusion

We have study type modifications within a persistent object-oriented language that is particularly suitable for distributed systems modeling and specification. To ensure the consistency of the modified system, where types dynamically change, we have introduced two relations: structural consistency and behavioral conformance. While *structural consistency* deals with the static aspect of the evolving system, the *behavioral conformance* deals with the dynamic aspect of the system. For the behavioral conformance, we plan to consider operation parameters and infinite behaviors of objects are not addressed in this paper. A reflection based implementation has been presented, using meta-objects, to allow dynamic modifications of types and their instances. We have shown different cases where behavior modifications preserve the *behavioral conformance* requirements. The actual effectiveness of our approach to dynamic modification, needs to be validated. Further research is needed to address the authorization and concurrency mechanisms in a shared environment, where more than one user can access and modify types concurrently.

Acknowledgement
The CITR is gratefully acknowledged for their financial support.

```
type TYPE = OBJECT with
          - - -
          PreviousVersion, CurrentVersion   : integer
          meta      : Meta-of-type;
operation
          AddAttr(A: AttrDef);
          AddOper(O:Operation);
          - - -
Behavior
  loop
      Accept AddAttr  do { behavior associated to AddAttr semantics } end;
      Accept AddOper  do {behavior associated to Addoper semantics } end;
      -the above behavior can be adapted for other kinds of change such as AddProc etc...
  end
endtype TYPE

type Meta-of-type = INTERPRETER with
          OldDef               : sequence[TYPE];
          ChangeIndicator  : var[boolean]; { true if the type has been modified }
          referent             : TYPE; { the type object for which self is meta }
          Achange             : sequence [Modification];
operation
          CurrentDef: TYPE;
          Verify;
          Propagate;
Behavior

          Accept  Verify  do
             if  ChangeIndicator then
                    - Update the OldDef list, by adding the current definition of referent to the
                      OldDef  list.
                    - Update the version number of referent (increment it).
                    - Convert ; {convert the instances according to the “Achange”}
                    - Updatechange; {the attribute ChangeIndicator becomes false}
                    - return;
          end;
  where
          procedure Convert =
                    {convert instances according to the change “Achange”.}
          endproc
          procedure updatechange =   - - -  endproc
          procedure updateOldDef =   -  - -  endproc

end Meta-of-type

type Meta-of-instance = INTERPRETER with
          UsedTypes          : sequence[TYPE];
          referent  : OBJECT; {the object for which self is meta-object }

behavior
          -check if any type t in the UsedTypes sequence was changed:  t.meta!Verify
          -update the behavior of the referent object according to the UsedTypes changes.
          -update the UsedTypes:
                    if a used type t was changed leading to a new type t'
                    then replace t by t' in UsedTypes sequence.
end Meta-of-instance
```

Fig.6.3. Meta-Objects description

References

[Amer 87] P. America, *Inheritance and subtyping in a Parallel Object-Oriented Language*, in Proceedings of ECOOP'87 (AFCET), 1987, pp. 281-289.

[Amer 89] P. America, *A Behavioral Approach to subtyping in object-oriented programming languages*, Philips Journal of Research, Vol.44, Nos. 2/3, pp. 365-383,1990.

[Bane 87] J. Banerjee, W. Kim, H. J. Kim and H. F. Korth, *Semantics and implementation of schema evolution in object oriented databases*, in Proceedings, ACM SIGMOD Int. Conf. On Management of Data, San Fransisco, CA, May 1987, pp. 311-322.

[Barb 90b] M. Barbeau and G. v. Bochmann, *Formal verification of Mondel Object-Oriented Specifications Using a Coloured Petri Net Technique.*, In preparation.

[Blac 87] A. Black, N. Hutchinson, E. Jul, H. Levey and L. Carter, *Distribution and abstract types in Emerald*, IEEE Trans. on Soft. Eng., Vol SE-13, no.1,1987, pp.65-76.

[Boch 89] G. v. Bochmann, *Inheritance for objects with concurrency*, Publication departementale # 687, Departement IRO, Université de Montréal, Septembre 89.

[Boch 90] G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval and N. Williams, *Mondel: An Object-Oriented Specification Language*, Publication departementale #748, Departement IRO, Université de Montréal, November 90.,

[Brin 86] E. Brinksma and G. Scollo, *Lotos specifications, their implementations and their tests*, Protocol Specification, Testing and Verification VI (IFIP Workshop, Montreal, 1986), North Holland Publ., pp. 349-360.

[Delc 91] C. Delcourt and R. Zicari, *The design of an integrity consistency checker (ICC) for an object oriented database system*, ECOOP'91.

[Erra 91] M. Erradi, G. v. Bochmann and I. Hamid, *Dynamic Modifications of Object-Oriented Specifications*, To appear in CompEurop'92, Int. Conf. on Computer Systems and software Engineering.

[Ichi 90]H. Ichikawa, K. Yamanaka and J. Kato, *Incremental Specification in LOTOS*, PSTV'90. pp. 185-200.

[Kram 85] J. Kramer and J. Magee, *Dynamic Configuration for distributed systems*, Trans. on Soft. Eng. Vol. SE-11, No. 4, pp.424-436.

[Lern 90] B. S. Lerner and A. N. Haberman, *Beyond Schema evolution to database reorganozation*, OOPSLA conf. Ottawa 1990.

[Meye 88] B. Meyer, *Object Oriented Software Construction*, C.A.R. Hoare Series Editor, Prentice Hall, 1988.

[Jaco 87] D. J. Penney and J. Stein, *Class Modification in the GemStone object-oriented DBMS*, OOPSLA, pp.111-117.

[Plot 81] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Aarhus University, Report DAIMI FN-19, 1981.

[Rudk 91] S. Rudkin, *Inheritance in LOTOS*, 4th. Int. Conf. on Formal Description Techniques. FORTE'91, pp. 415-430.

[Skar 87] A. H. Skarra and S. B. Zdonik, *Type evolution in an Object-Oriented Databases*, Research directions in object-oriented programming, Wegner, Bruce Shriver and Peter, MIT press, pp.393-415.

[Viss 88] C. Vissers, G. Scollo and M. v. Sinderen, *Architecture and Specification Style in Formal Descriptions of Distributed Systems*, Proc. IFIP Symposium on Prot. Spec., Verif. and Testing, Atlantic City, 1988.

[Will 90] N. Williams, *Un simulateur pour un langage de spécification orienté-objet*, MSc thesis, Université de Montréal,