Type Evolution in a Reflective Object-Oriented Language

Mohammed Erradi , Gregor v. Bochmann and Issam A. Hamid[1]

Université de Montréal, Département d'Informatique et de Recherche Operationnelle,
CP. 6128, Succ. "A" Montréal, (Québec) Canada H3C-3J7.
Email : {erradi, bochmann} @iro.umontreal.ca

[1] Now with Tohoku Geijitsu Kouka University, Miurasa Miurasa nen Yamagata-shi,
Shokoo Kaikan, Yamagata-shi, Nanoka Machi 3-1-9, 990, Japan.
Email: hamid@cc.tohoku.ac.jp

Abstract

This paper describes the design of the reflective concurrent object-oriented specification language RMondel. RMondel is designed for the specification and modeling of distributed systems. It allows the development of executable specifications which may be modified dynamically. Reflection in RMondel is supported by two fundamental features that are: *structural reflection (SR)* and *behavioral reflection (BR)*. Reflection is the capability to monitor and modify dynamically the structure and the behavior of the system. We show how the features of the language are enhanced, using specific meta-operations and meta-objects, to allow for the dynamic modification of types (classes) and instances using the same language. RMondel specifications can be modified by adding or modifying types and instances to get a new adapted specification. Consistency is checked dynamically at the type level as well as at the specification level. At the type level, structural and behavioral constraints are defined to preserve the conformance of types. At the specification level, a transaction mechanism and a locking protocol are defined to ensure the consistency of the whole specification.

Keywords. Software evolution, Type evolution, meta-level architecture, reflection, object-oriented programming, dynamic modifications.

1. Introduction and motivations

The object oriented approach is known by its flexibility for system construction. This is partly due to the inheritance property which permits class reuse and incremental construction of systems. We have developed a new object-oriented specification language, called Mondel [Boch 90] that has important concepts as an executable specification language to be applied in the area of distributed systems. The motivations behind Mondel are: (a) writing system descriptions at the specification and design level, (b) supporting concurrency as required for distributed systems, (c) supporting persistent objects and transaction facilities, and (d) supporting the object concept. Presently, our language Mondel has been used for the development of executable specifications of problems related to network management [Boch 91] and OSI directory system [Boch 92].

In a wide spectrum of distributed applications, software systems require modifications to accommodate evolutionary change, particularly for systems with long expected lifetime. In general, evolutionary changes are difficult to accommodate because they cannot be predicted at the time the system is designed. Therefore, systems should be sufficiently flexible to permit arbitrary, incremental changes. The evolution of such systems is necessary to accommodate the evolution of requirements and design decisions during the software development and maintenance process. We believe that software system modifications are most of the time incremental. They consist of adding new functionalities or extending some existing ones. In this paper, we consider that an executable specification of a system is an implementation model of such system. Therefore, we examine a method of supporting extensions of distributed systems specifications in the context of the object-oriented specification language Mondel. A difficult and important issue is that of making modification dynamically, without interrupting the processing of those parts of the specification which are not directly affected. There has been little suggestion as to how such dynamic modification should be specified, managed, and controlled.

To achieve our goal that is the construction of dynamically modifiable specifications, we define a reflective object oriented language called RMondel (Reflective Mondel) which is based on the Mondel language. Recently, reflection has gained wider attention as indicated by the first and second workshops on reflection and meta-level architectures in object-oriented programming [Work 91] held in conjunction with OOPSLA'90 and 91. A language is called reflective if it uses the same structures to represent data and programs. In conventional systems, computation is performed only on data that represent entities of an application domain. In contrast, a reflective system contains another type of data that

represent the structural and computational aspects of itself. The original model of reflection was proposed in [Maes 87] following Smith's earlier work [Smit 82], where a meta-object is associated with each object in the system to represent information about the implementation and the interpretation of the object. Metaobjects for object-based concurrent systems must represent not only an object's methods and state, but also object communication procedures [Wata 88]. To define a reflective architecture, one has to define the nature of meta-objects and their structure and behavior. In addition, one has to show how the handling of objects communications and operations lookup are described at the meta-level.

A Mondel specification consists of a type (class) lattice where nodes represent types and edges represent the inheritance relation. To allow for the construction of dynamically modifiable specifications, we need to access and modify types during execution-time. Therefore, we developed RMondel that uses meta-objects to provide facilities for the dynamic modifications of types. Reflection in RMondel is supported by two fundamental features which are: *structural reflection* (*SR*) and *behavioral reflection* (*BR*). For the *SR* we consider that a type is an object and types are instances of other types. For *BR*, a meta-object, called interpreter object, is associated with each object at creation time. An interpreter object deals with the computational aspect of its associated object. Specialized or different versions of interpreters may be defined for monitoring the behavior of objects, or for dynamically modifying their behaviors.

In RMondel, modifications are explicitly specified by an agent external to the specification. The specification may be modified by the application of modification transactions. In addition to the means for specifying and performing changes, it is also necessary to provide facilities for controlling change in order to preserve specification consistency. Consistency is checked dynamically at the type level as well as at the specification level. At the type level, structural and behavioral constraints are defined to preserve the conformance of types. At the specification level, a transaction mechanism and a locking protocol are defined to ensure the consistency of the whole specification. Structural consistency concerns mainly the compiling constraints, i.e., checking dynamically the static semantics rules of the language in use. Behavioral consistency deals with preserving the consistency of the behavior. This concerns mainly some properties of distributed systems such as blocking.

The paper is organized as follows. In Section 2 we introduce the definitions of types and their relationships, which are mandatory for the understanding of object-oriented specifications as considered in this paper. To preserve types consistency, a set of invariants is defined in Section 3. In Section 4, we define a set of primitives which are used to modify the structure and behavior of type definitions. In Section 5, we introduce RMondel, a reflective

version of Mondel, and show how dynamic type modifications are supported. A transaction mechanism and a locking protocol which ensure the consistency of the whole specification are given in Section 6. Section 7 discusses related works. Conclusions are drawn in Section 8.

2. Type definitions and their relationships

Before addressing the problem of specification modifications, an understanding of the specification model, its components and their relationships is required. A specification is defined as a type lattice where nodes represent types and edges represent an inheritance relation. Our interpretation of inheritance considers both the structure and the behavior aspects.

Definition 1: A type t consists of an interface I_t and a behavior B_t, $t = <I_t , B_t >$.
$I_t = < A_t, Op_t >$ where A_t is the set of attributes and Op_t is the set of operations. B_t is the behavior specification of the objects of type t. o

Types' interfaces are used as a basis for the traditional inheritance scheme of object-oriented languages. Thus, a type has at least all attributes and operations defined for the more general type, where the types of the operations result must be conforming and the types of the input parameters must be inversely conforming (see for instance [Blac 87]). Based on this aspect of inheritance, we give a recurssive definition of the structural consistency relation as follows.

Definition 2: The type $t' = < < A_{t'}, Op_{t'} > , B_{t'} >$ is *structurally consistent* with the type $t = < < A_t, Op_t >, B_t >$ if:

1. $A_{t'} \supseteq A_t$. t' has at least all the attributes of t.
2. For each operation o in Op_t there is a corresponding operation o' in $Op_{t'}$ such that:
   - o and o' have the same name
   - o and o' have the same number of parameters.
   - The result type of o', if any, is structurally consistent with the result type of o.
   - The type of the i-th parameter of o is structurally consistent with the type of the i-th parameter of o'. o

The following definition introduces our notion of behavior extension. According to Mondel formal semantics, the behavior of objects is formally specified by a translation to labeled transition systems [Erra 92]. Both RMondel and Lotos have their formal semantics defined based on labeled transition systems. Therefore, If we ignore operations

parameters, our definition of the behavior extension corresponds to the *extension* relation defined for Lotos specifications [Brin 86].

Definition 3: The type t'= <$I_{t'}$, $B_{t'}$ > *extends* the type t = <I_t , B_t >, if the following properties are satisfied:
property 1. $B_{t'}$ does what is explicitly allowed according to B_t (but it may do more).
property 2. What $B_{t'}$ refuses to do (i.e., blocking), can be refused according to B_t ($B_{t'}$ may not refuse more than B_t). o

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, e.g. in *Emerald* [Blac 87] and *Eiffel* [Meye 88]. However, there are other important aspects to inheritance related to the dynamic behavior of objects [Amer 89], including constraints on the results of operations, the ordering of operation execution, and the possibilities of blocking [Boch 89]. Therefore, our definition of inheritance takes into account the dynamic behavior of objects as follows:

Definition 4: A type t' = <$I_{t'}$, $B_{t'}$ > **conforms-to** a type t = <I_t , B_t > if :
t' is *strcturally consistent* with t.
and t' *extends* t. o

If type t' conforms-to type t then we say that t' is a subtype of t and t is a supertype of t'.

3. Preserving consistency

To maitain the *conforms-to* relation, we deduce from the definitions of Section 2, a set of invariants which must be satisfied by each type and its related types in the lattice. These invariants will be used for dynamic type checking after type updates.

We denote by **<:** the *conforms-to* relation introduced in Definition 4.
Corollary: The **<:** relation is a partial order , i.e., reflexive, transitive, and antisymmetric.
Proof: evident.

Definition 5: An executable specification S is a triple < T, **<:** , O> where T is a finite set of types, **<:** is the *conforms-to* relation on T, and O is the set of objects created according to their types in T. o

(1) *Type hierarchy invariant*: The type hierarchy is a directed acyclic graph.

¢ t1, t2 [T with t1 <: t2 , then ™ t3 [T such that t2 <: t3 and t3 <: t1.

(2) *Distinct attribute names invariant*: All attribute of a type, whether explicitly defined or inherited, are distinct.

¢ attr1, attr2 [A_t, such that attr1=(a1 : t1) and attr2=(a2 : t2)

{ attri = (ai:ti) means: ai is the attribute name and ti is the attribute type }

then a1=a2 > attr1=attr2

(3) *Distinct operation names invariant*: All operation of a type, whether explicitly defined or inherited, are distinct.

¢ op1 = <$opname_1$, [p1:t1, ..., pi:ti, ...pn:tn],[r1]>,

op2 = <$opname_2$, [p1':t1', ..., pi':ti', ...pn':tn'],[r2]>[Op_t, [] means optional

then $opname_1$=$opname_2$ > op1=op2

(4) *The instance-of invariant*: Each object is an instance of a type.

¢ i [O, ¡ t [T such that i is an instance of t.

(5) *Full Inheritance invariant*: A type inherits all attributes and operations from each of its supertypes.

(5.1) For attributes:

¢ t1, t2 [T with t1 <: t2 ,

¢ attri =(ai : ti) [A_{t1},

¡ attrj =(aj : tj) [A_{t2} such that ai = aj and ti <: tj

(5.2) For operations:

¢ opi = <$opname_i$, [p1:t1, ..., pl:tl, ...pn:tn],[r1]> [Op_{t1}

¡ opj = <$opname_j$, [p1':t1', ..., pl':tl', ...pn':tn'],[r2]>[Op_{t2}

such that $opname_i$ = $opname_j$ opi and opj have the same name.

(the covariant rule holds)

r1 <: r2 the result of opi conforms to the result of opj.

and (the contravariant rule holds)

pl = pl' for l=1,..,n parameter names are the same.

tl' <: tl for l=1,..,n parameter types are inversly conforming.

4. Primitive operations for type modifications

In the following we give a classification of type modifications that are supported in our language, and we provide the description of their semantics. In comparison with the classification of the class modifications in ORION [Bane 87] which considers structural modifications only, our approach considers those type modifications, both structure and behavior, that lead to new types which conform to old ones.

4.1. Structure modifications

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. Name conflict is not addressed here, but may be avoided in a similar way as in [Delc 91].

Change the type T of an attribute A by the type T1: This update is allowed only if T1 *conforms-to* T.

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.

Change the signature S of the operation O:
(i) 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 is allowed only if T conforms to T'.
(ii) 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' conforms to T.

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

Add a new type T: If no supertype of T is specified, then the type *OBJECT* (i.e. the root of the type hierarchy) is the default supertype of T. If a supertype is specified, then all attributes and operations from the supertype are inherited by T. The name of the added type T must not

be used by an already defined type. The specified supertype of T must have been previously defined.

4.2. Behavior modifications

For the modification of the behavior of types, we consider those modifications which extend the existing behavior. This is similar to the notion of incremental specifications proposed for a subset of basic LOTOS language [Ichi 90]. The modifications of behaviors are based on the following language constructs: sequential, choice, and parallel composition.

Sequential composition: An existing behavior may evolve into a new behavior by appending an other behavior to the existing one. The consistency of this modification is guaranteed by the *conforms-to* relation given in Definition 4.

Choice composition: It has been shown that the choice operator does not guarantee subtyping [Rudk 91], because non-determinism may be introduced. For instance, the combination of recursion and choice may lead to a violation of the second property of Definition 3. 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. Examples of two possible cases, deterministic and non-deterministic, are given in [Erra 92b].

Parallel composition: Two behaviors may be composed, using the parallel composition operation, to obtain a new behavior. The new behavior, whose construction is based on the pure interleaving semantics (i.e., independent parallelism), preserves the ordering of constraints of actions of the two initial behaviors. This kind of modification is guaranteed by the *conforms-to* relation as well.

5. Configuration of reflection in RMondel

So far, we have introduced primitives for structural and behavioral modifications and invariants for preserving types consistency. To allow for the construction of dynamically modifiable specifications, we need to access and to modify types during execution-time [Erra 92c]. In this section, we give an overview of *Mondel* and its characteristics. Then we discuss reflection as supported in *RMondel* and show how dynamic type modifications and dynamic checking of type consistency are implemented using RMondel facilities.

5.1. 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. It has a formal semantics, expressed by means of a translation into a state transition system. Each Mondel object has an identity, a certain number of named attributes (i.e., each object will have fixed references to other objects, one for each attribute), and acceptable operations which are externally visible and represent actions that can be invoked by other objects. An object is an instance of a type that specifies the properties that are satisfied by all its instances.

A Mondel specification corresponds to a type lattice. In such a lattice, nodes represent types, and edges represent the inheritance relation. The execution of a specification consists of a set of objects 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 rendezvous mechanism. The basic statement of Mondel is the operation call, which is syntactically represented by the "**!**" operator. For instance in the statement *m! InsertCoin* (see line 27 of Fig.5.1.), "*m*" designates the called object, and *InsertCoin* is an operation defined within the type of "*m*" (i.e., the type *Machine*).

Mondel has a formal semantics which associates a meaning to the valid language sentences. Such a semantics 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 have particularly applied the technique of Plotkin [Plot 81] where state/transition systems are taken as machine models. The Mondel formal semantics was the basis for the verification of Mondel specifications [Barb 91], and has been used for the construction of an interpreter [Will 90].

5.1.1. Example of a Mondel specification

The following example will be used throughout the paper. Let us consider a vending machine which receives a coin and delivers candies to its user. We distinguish two types of

objects: the type *Machine* and the type *User*, as shown in the Mondel specification of Fig.5.1. 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 behavior of the *User* type is specified within the behavior clause as shown in lines 23 to 33 of Fig.5.1. 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 behavior of the *Machine* type is specified as shown in lines 5 to 19 of Fig.5.1. 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.

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, see lines 35 to 38 of Fig.5.1). The types are specified using a state oriented style [Viss 88]. Each internal state of an object is modeled as a Mondel procedure.

```
0 unit spec =

1 type Machine = object with
2 operation
3   InsertCoin;
4   PushAndGetCandy;
5 behavior
6   Ready
7 where
8   procedure Ready =
9       accept InsertCoin do
10            return;
11        end;
12      DeliverCandy;
13  endproc Ready

14 procedure  DeliverCandy =
15   accept PushAndGetCandy do
16     return;
17   end;
18   Ready;
19 endproc DeliverCandy

20 endtype Machine

21 type User = object with
22   m: Machine;
23 behavior
24   Thinking
25 where
26  procedure Thinking =
27       m! InsertCoin;
28       GetCandy;
29  endproc Thinking

30  procedure  GetCandy =
31   m! PushAndGetCandy;
32   Thinking;
33   endproc GetCandy

34 endtype User

{the vending machine system behavior}
35 behavior
36    define Amachine = new (Machine) in
37          eval  new(User (Amachine));
38    end;

39 endunit spec
```

Fig.5.1. Mondel specification of the vending machine.

5.2. Reflection in RMondel

In the formalism used to define the semantics of Mondel, types are static and used as templates for object creation. Only the instances of a type are considered as objects. In order to modify types dynamically, types must be objects. Therefore, types will be accessible and may be modified during execution time. For this purpose, reflection is a promising choice.

To define a reflective architecture, one has to define the nature of meta-objects and their structure and behavior. In addition, one has to show how the handling of object communications and operations lookup are described at the meta-level [Ferb 89]. In RMondel, types are used for structural description (i.e., for the definition of the structure of objects and of applicable operations), and interpreters are used for the behavioral description (i.e., how the rendezvous communication is interpreted and the operations are applied) of their associated objects, called referents. Types are considered to be structural meta-objects, while interpreters are behavioral meta-objects. Types and interpreters are instances of the kernel types *TYPE* and *INTERPRETER* respectively. This approach shows many advantages:

- Types are objects, instances of the type *TYPE* which is defined at a meta-level.
- Operations for type modifications can be defined at the meta-level (i.e., within *TYPE*).
- An object behavior may be modified according to the modifications of its type.
- An object behavior can be monitored by its interpreter.
- New communication strategies can be defined by creating subtypes or different versions of *INTERPRETER.*
- Communication between the baselevel and the meta-level is possible.
- The definitions of the structure and the behavior of objects are dynamically accessible.

In the following we introduce the enhancements of the Mondel original language in order to define the *structural reflection* (*SR*), and the *behavioral reflection* (*BR*) that are the fundamental features of reflection in *RMondel.*

5.2.1. Structural reflection

In Mondel objects with the same properties are grouped within the same type. In RMondel, a type and its components such as attributes, operations, and behavior, are considered as objects which are instances of specific types, called *kernel types*, as shown in Fig.5.2. This allows for the access of the different components of a type, and gives more flexibility in order to dynamically modify types. The structure of RMondel is supported by

instantiation and inheritance graphs. The instantiation graph represents the *instance-of* relationship, and the inheritance graph represents the *conforms-to* relationship. The objects *TYPE* (called *CLASS* in other languages) and *OBJECT* are the respective roots of these two graphs [Erra 90].

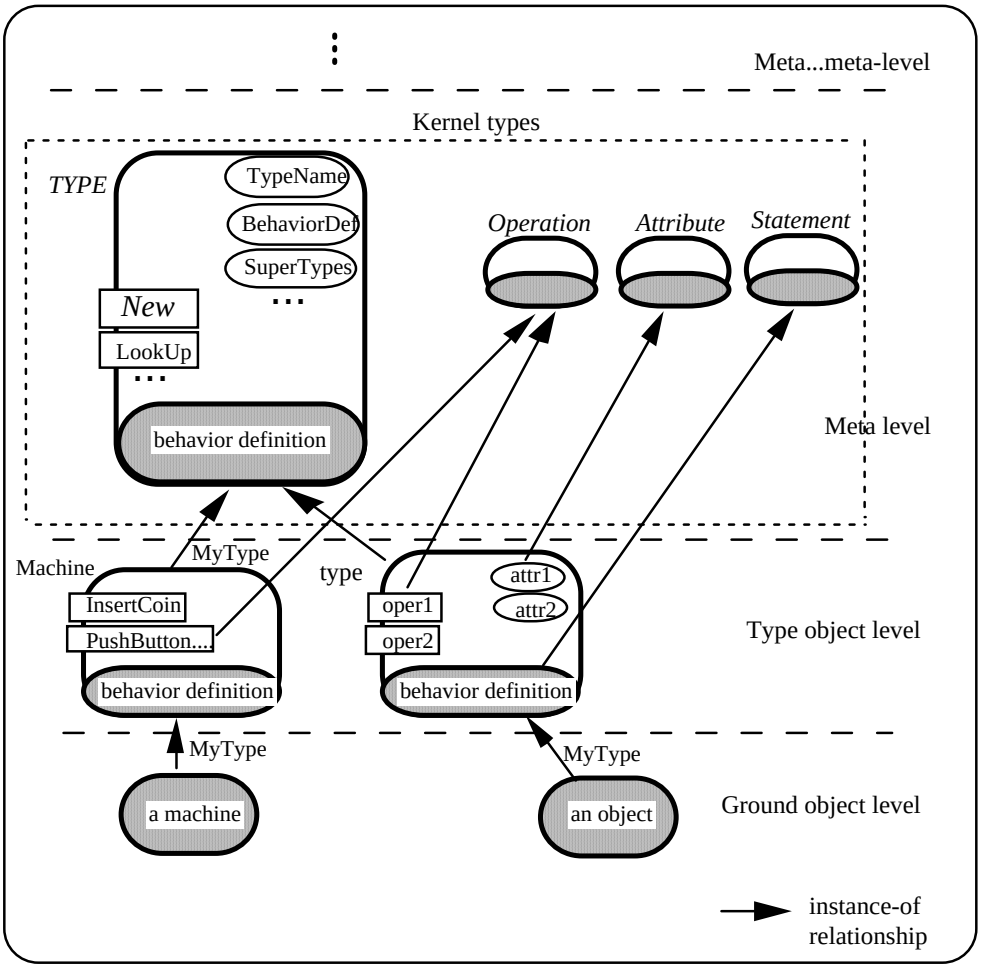

Fig.5.2. *Structural reflection* basis

The *structural reflection* is supported in a similar manner as in *ObjVlisp* [Coin 87]. The most important aspect of *SR* in RMondel, is that each object is an instance of a type, and types are objects. For each object we introduce the attribute *MyType* that links the object to its type, as shown in Fig.5.2. Another aspect of *SR* is that the RMondel statements and expressions are objects. For instance, one can specify the operation call, and accept statements as instances of the *Opcall* and *Accept* types, respectively, as shown in Fig.5.3. Each statement object accepts the *Eval* operation, that implements the semantics rule

associated with such a statement. For the sake of simplicity, we do not consider here the operation parameters and results.

```
1   type Statement = OBJECT endtype Statement
2   type Expression = OBJECT endtype Expression
    {Details on the definitions of the statement and expression objects are given in[Erra 92]}
3   type Accept  = Statement with
4           OpName          : string;
5           AcceptBody      : Statement;
6     operation
7           Eval;
8     behavior
9           { semantics rule of the accept statement }
10  endtype Accept
11  type  OpCall = Statement with
12          Callee          : Expression;{restricted to object identifier, for simplicity}.
13          OpName          : string;
14    operation
15          Eval;
16     behavior
17          { semantics rule of the operation call statement }
18  endtype OpCall
```

Fig.5.3. Example of the specification of a subset of *RMondel* statements.

5.2.1.1 The structure of RMondel objects

In RMondel, the structure of an object is considered as a finite set of attributes represented by pairs. Each attribute is represented by a pair ($Name_{attri}$, Id_{attri}) which is a substitution (i.e., binding) assigning an object identifier (Id_{attri}) to an attribute name ($Name_{attri}$). In the following, we will use the term *attribute* to designate such a pair. We have two types of attributes: *initial attributes* and *effective attributes.*

(1) The *initial attributes* are: (i) the unique object identifier, named ObjectId, is commonly known as *self*. Such identifier is generated automatically. For the sake of readability we will consider that object identifiers, for types, are constructed by means of the type name prefixed by "Id" (e.g., the type *Machine* of Fig.5.1 is identified by *IdMachine*). (ii) the identifier of the type of the object, named *MyType* which is the type of the created object. (iii) the identifier of the object behavior, named *Behavior,* which represents the initial behavior of the created object. The value of the *Behavior* attribute can change as the execution of the object's behavior evolves. It is important to mention that an object's behavior is also an object.

(2) The *effective attributes* are separately created by the *NewAttr* operation defined in the *OBJECT* type which defines the common behavior of each object in the system. These two

kinds of attributes, initial and effective attributes, constitute the explicit definition of an object in the following form:

O = <(ObjectId,Ido),(MyType,Id_{type}), (Behavior,Id_{beh}), {...,($Name_{attri}$,Id_{attri}),..}> where Ido, Id_{type}, and Id_{beh} designate the initial attributes of the object O. The set {...,($Name_{attri}$,Id_{attri}),..} contains the effective attributes of O.

5.2.1.2. The type TYPE

The type *TYPE* initially exists in the system as an instance of itself. It defines the behavior for types, e.g. the type *Machine* of Fig.5.1 is created as an instance of *TYPE*. It holds the effective attributes *TypeName*, *BehaviorDef*, *SuperType etc*... which refer to the name of a type, the behavior defined in such a type, its parent, etc.... Fig.5.4 gives a definition of the type *TYPE* . The *LookUp* and *New* operations are defined within *TYPE* as shown in Fig.5.4. The *LookUp* operation is used to find an operation in the called object's type or in its supertypes. The *New* operation allows for object creation.

```
1  type TYPE = OBJECT with
2        TypeName        : string;
3        BehaviorDef     : var[Statement];
4        SuperType       : TYPE; { for simplicity, we consider here single inheritance only}
5        Attributes      : set [AttributeDef];
6        Operations      : set [Operation];
7        Procedures      : set[Procedure];
         . . .
8   operation
         {the operation New creates an object according to RMondel object structure}
9        New  : OBJECT;
         <: ( t: TYPE) : Boolean; { checks if a type t conforms-to with self }
         {the operation LookUp checks if the operation "OpName" is defined for an object's type or for
          one of its supertypes; then returns the associated statements}
10       LookUp (OpName : string) : Statement;
11   behavior
12        LookUpProc; . . .
    where
13       Procedure LookUpProc =
14           Accept LookUp do
15                ifexist  Op:Operation  suchthat
16                         Operations. contains(Op)  and Op.OpName = OpName
17                then  {let AcceptBody be the object of type Statement that is associated with the
                          operation defined by Op.}
18                      return (AcceptBody);
19                else {recurse on supertypes}
20                      return (SuperType ! LookUp(OpName));
21           end;
22       endproc LookUpProc
         ....
    endtype TYPE
```

Fig.5.4. The definition of *TYPE*

5.2.1.3. The type OBJECT

OBJECT is the most general type. It describes the common characteristics of all objects. Each object is characterized by its unique identifier, its type, its effective attributes (i.e, binding) and its behavior. The type *OBJECT* provides the *NewAttr* operation for effective attributes creation. *OBJECT* is the root of the inheritance graph. It is defined, using Mondel, as follows:

```
type OBJECT= with
   ObjectId: integer unique;
   MyType: TYPE;
   Behavior          : var[Statement];
  operation
      NewAttr (A:Attribute); {A is the added attribute }
  behavior
      {specification of the semantics rule of NewAttr}
endtype OBJECT
```

5.2.2. Behavioral reflection

Beside the *structural reflection* of our model, the *behavioral reflection* (*BR*) must be represented. Therefore, we have associated an interpreter object (i.e., behavioral meta-object) to each object as shown in Fig.5.5. An interpreter object deals with the computational aspect of its associated object called *referent*. Interpreter objects are defined as instances of the type *INTERPRETER*. An interpreter object may have its own interpreter object; thus the number of interpreter objects is virtually infinite. Specialized interpreters can be defined for monitoring the behavior of objects.

A possible specification of the type *INTERPRETER*, where the incoming calls of its referent can be recorded, is given in Fig.5.6. Such specification shows how the rendezvous communication between objects is interpreted. One can define new ways of handling the object communication by specifying subtypes or different versions of the *INTERPRETER* type. We can see from the *INTERPRETER* definition that the accept statement is an object. To avoid an infinite loop of operation calls, the basic operation call ("!") is used to define the semantics of the accept statement.

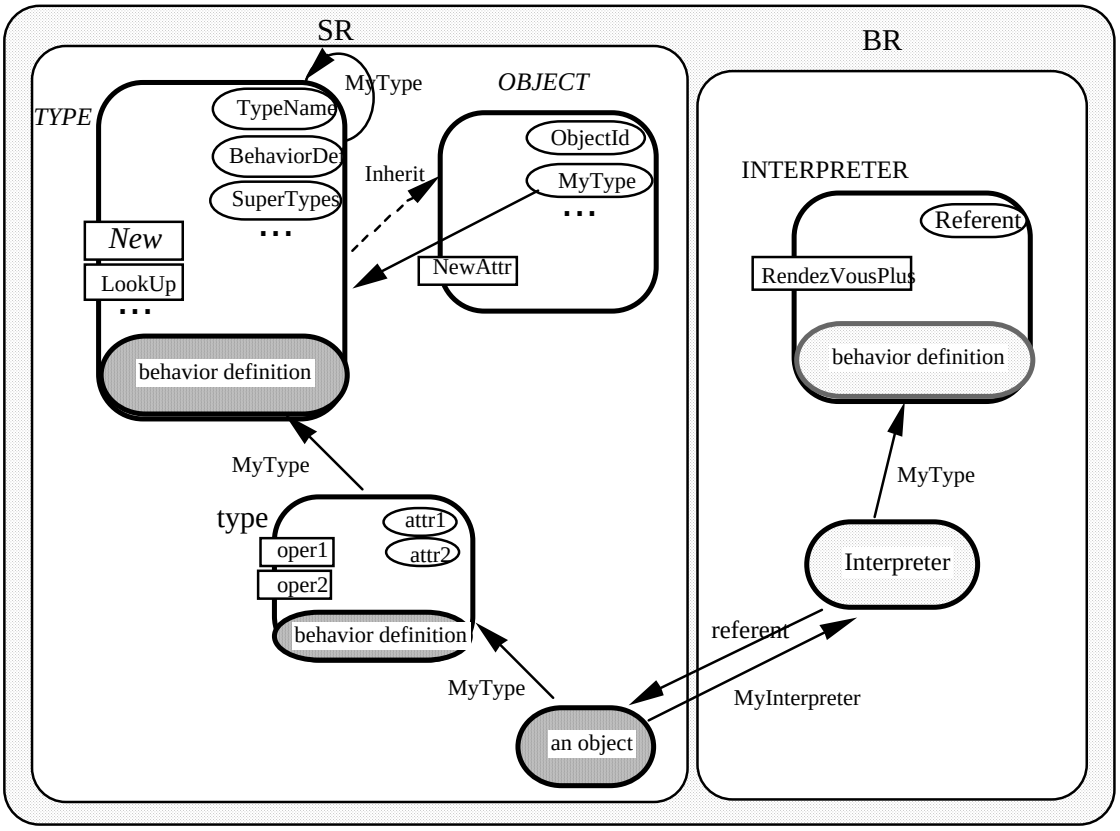

Fig.5.5. BR and SR in *RMondel*.

```
1   type INTERPRETER = OBJECT with
2         referent  : OBJECT;
3         NbCall   : var[integer];. . .
4   operation  {the operation RendezVousPlus interprets object communication}
5         RendezVousPlus (OpC: OpCall); . . .
6   behavior
7         RendezVousProc; . . .
8   where
9         Procedure RendezVousProc =
10            Accept RendezVousPlus do {We can record the number of incoming calls of the referent object}
11                  IncrementNbCall;
12                  {let AcceptBody be the object of type Statement that corresponds to the called operation; then
                     evaluate such a statement.}
13                  define AcceptBody = OpC.Callee.MyType ! LookUp(OpC.OpName) in
14                        { create a Context object which contains the callee attributes and parameters binding }
15                      AcceptBody ! Eval (Context);
16                  end;
17            end;
18            RendezVousProc;
19        endproc RendezVousProc
20        Procedure IncrementNbCall =
21                  { Increment NbCall }
22        endproc IncrementNbCall
    endtype INTERPRETER
```

Fig.5.6. The definition of *INTERPRETER*

To deal with interpreter objects, we add a specific attribute, called *MyInterpreter*, to each object. This leads to the modification of the type *OBJECT* as shown in Fig.5.7. The added attribute is optional because not all objects need to have a specific interpreter. If the value of the attribute *MyInterpreter* is nil then a default interpreter is invoked.

```
type OBJECT= with
   ObjectId: integer unique;
   MyType: TYPE;
   MyInterpreter      : INTERPRETER opt; {opt stands for optional}
   Behavior           : var[Statement];
 operation
     NewAttr (A:Attribute); {A is the added attribute }
 behavior
     {specification of the semantics rule of NewAttr}
endtype OBJECT
```

Fig.5.7. The OBJECT definition with the attribute *MyInterpreter*

5.2.3. A simple RMondel interpreter

The definition of reflection in RMondel allows the access to the definition of an object's structure (i.e., its type) and to the language statements which are objects. To access the context of the execution of an object's behavior, we use the *context* objects (instances of the *context* type) which contain the binding of attribute names and local variables with values. A context object is created to bind the actual arguments of the operation call with the operation parameters. Local variables and attributes are specified in the context object. The current context is passed as an argument to the *Eval* operation of a statement object (e.g., see lines 13 to 15 of Fig.5.6). Context objects are managed based on the conventional stack approach used for the processing environments of procedural programming languages.

Let us describe a simple RMondel interpreter (RMI) which coordinates the execution of the objects of a given RMondel specification . The RMI has a global view of the existing objects, i.e., the kernel objects and the objects of the specification. According to RMondel semantics, which is based on state/transition systems, objects are executed in parallel. Therefore, the RMI selects an object and tries to fire a transition within the object's behavior. The most important transitions are operation calls. If the called object has an associated interpreter, (i.e., the value of its attribute *MyInterpreter* is not nil) then the evaluation of the operation call is delegated to this interpreter (see Fig.5.8). A search for the called operation is performed, within the type of the called object, by mean of the *LookUp* operation. The *LookUp* operation is defined at the meta-level within the type *TYPE* .

```
type RMondel-Interpreter = OBJECT
with
        ...
behavior
        ...
        { the RMondel interpreter selects an object O}
        ifexist O: OBJECT suchthat
                { the behavior of O is an operation call }
                O.Behavior.Mytype ≤ OpCall
        then
                if O.Myinterpreter <> nil
                then O.Myinterpreter ! RendezVousPlus (O.Behavior)
                else {the default interpreter is invoked}
        ...

endtype RMondel-Interpreter
```

Fig.5.8. A simple *RMondel* interpreter

5.3. Dynamic type modifications and consistency checking

In the following we show how the *RMondel* facilities are used for dynamic type modifications, and dynamic checking of type consistency.

5.3.1. Primitives for dynamic type modifications

Fig.5.4. shows the type *TYPE* used to implement two important aspects of reflection, which are instantiation and operation lookup. To support dynamic type modifications in RMondel, we modified the type *TYPE* by adding a set of primitive operations such as AddAttr, AddOper, etc... The resulting *TYPE* is given in Fig.5.9. Note that one can define a subtype, let say Modifiable-Type, of *TYPE* in order to hold the primitives operations.

Let us give a simple example to illustrate the notion of dynamic type modifications. Let T be a type. In RMondel, T is a type as well as an instance of *TYPE*. Consequently, T accepts the operations defined in *TYPE*. For example, it accepts the operations *AddAttr* to add an attribute to its own attributes. Existing instances of type T are modified according to the *AddAttr* operation semantics as defined in Section 4.

5.3.2. Dynamic checking for type consistency

In Section 3, we defined a set of invariants which are used to ensure the consistency of the type structure after modifications. To perform dynamic checking of type consistency, we

incorporate these invariants, into RMondel, within the *invariant* clause of the type *TYPE*. An example of such invariant definition is shown in Fig. 5.9.

Let us consider the type T again. Since T is an instance of *TYPE*, the invariants defined in *TYPE* must always hold for T, especially when T is first created and after any possible modification. For example, if we attempt to add to T an attribute definition which has the same attribute name as an inherited one, the invariant *Inv1* in Fig.5.9, prevents the completion of this modification.

```
type TYPE = OBJECT with
        TypeName        : string;
        BehaviorDef     : var[Statement];
        SuperType       :  TYPE;
        Attributes      : set [AttributeDef];
        Operations      : set [Operation];
        Procedures      : set[Procedure];
        . . .
 operation

        AddAttr (A:AttributeDef);
        AddOper(O:Operation);
        AddProc(P:Procedure);
        AddStat(S:Statement);
        ...
 invariant
        { We define here an example of invariant.}
      "Inv1"   { attributes must have distinct names }
                 [ Forall a1,a2 : AttributeDef suchthat
                        Attributes.contains(a1) and SuperType.Attributes.contains(a2) ]
                 ( a1.AttrName <> a2.AttrName )

        ...
 behavior
        { The semantics definitions of the modification operations. }
        ...
endtype TYPE
```

Fig.5.9. TYPE specification with invariants and modification operations.

5.4. Consistency at the specification level

We have discussed dynamic type modifications and dynamic checking of type consistency at the type level only. We now address the issue of dynamic checking of structural and behavioral consistencies at the specification level as a whole.

Structural aspect: The main question here is: if we replace a type definition t by t' in some specification S, where t' is *structurally consistent* with t, does the resulting specification S' remain consistent w.r.t. S? The specification S' is consistent with S if the modification does not introduce compiling errors (type checking). Therefore we assert that the obtained

specification S' remains consistent. This assertion can be proved according to assignment and parameter passing where type checking is important.

Behavioral aspect: Similarly, if we replace t by t' such that t' *extends* t, does S' *extends* S? The answer is in general no, as shown by the following counter example: Consider the behavior of S, which is defined by the parallel composition of two behaviors Bt1 and Bt2, where Bt1 is defined by a type t1 and Bt2 is defined by a type t2. Suppose we extend Bt2 to obtain Bt2', see Fig.5.10. Now if we compose Bt1 with Bt2', to obtain S', the resulting behavior blocks with respect to action a, whereas the original specification S is free of deadlock.

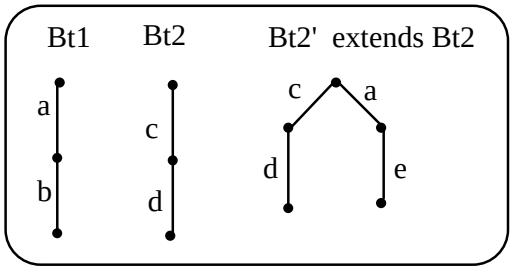

Fig.5.10. labelled transition systems of Bt1, Bt2, and Bt2'.

We conclude that the extension of a part of a specification does not imply the extension of the whole specification. Before incorporating the change to the specification, we have to check dynamically for the specification consistency. Therefore, in the following we use a transaction mechanism and a locking protocol, which are well known for database systems, to ensure the whole specification consistency.

6. The transaction mechanism

In this section, we define a transaction mechanism which is used to realize the dynamic checking of structural and behavioral consistencies at the specification level.

6.1. Locking protocol

To make dynamic specification modifications without interrupting the processing of those parts of the specification which are not directly affected by the change, we define a locking protocol to isolate the parts of the specification which are affected by the modifications. Such a protocol is incorporated within the transaction mechanism described in the next subsection.

According to the updates of a type T, its existing instances must be converted accordingly. When a type has to be updated, its instances must be locked until the type modifications are accomplished. If the updates do not succeed, e.g., because of invariant violation, then the type will be rolled back to its state before the updates, and the instances will be released to pursue their behavior progress. In the case where the type updates succeed, the instances will be converted accordingly, and released to behave normally. Each object can be active, passive or locked as shown in Fig.6.1. Initially, an object is in a passive state if it is not involved in a current transaction (e.g., an object is in a passive state after its creation). When the object is involved in a transaction, its state becomes active (e.g., the object is asking, by means of operation calls, for other objects' services). An object in a passive state, may be locked for the purpose of an update (e.g., object conversion after its type modification).

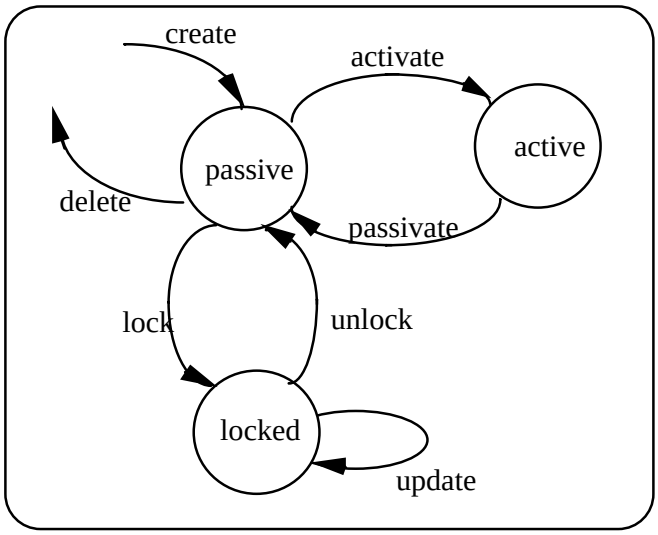

Fig.6.1. Object state/transitions

The fact that a specification is organized as a type lattice has a major impact on the locking protocol. The modification operations may involve a type and all its subtypes (e.g., if we have to add an attribute to a type, then the structure of its instances and of the instances of its subtypes has to be modified). Thus not only the instances of the modified type must be locked, but the instances of its subtypes as well. Therefore, we define a type sublattice to be a type and all its direct and indirect subtypes in the type lattice. To update a type, we adapt the *x lock* mechanism [Gray 78] to be applied for a type sublattice. That is when a type has to be modified, a lock is set not only on the type itself, but also on each of its descendant types on the type sublattice. The instances of a locked type will be locked until their type becomes

unlocked. Fig.6.1 shows the possible states and transitions of an object w.r.t. modifications. Objects (i.e., either types or their instances) can be modified only when they are locked.

6.2. The transaction steps

The user formulates his requirements within a transaction which consist of type update operations. We use the concept of transaction to provide fail-safe implementations of specifications by using standard fault recovery procedures developed for database systems [Gray 81]. A transaction consists of several successive modifications of one or more types. The following steps show how the different actions (i.e., those involved in a type updates) work and lead to a consistent specification. These steps are represented, through the different levels, by the heavy dotted lines in Fig.6.2.

Step 1: *Transaction construction*: through the interface object, the user formulates a transaction (called an atomic operation in *Mondel*) as an operation call, specifying his requirements (i.e., in terms of operations for type modifications). The transaction is composed of a set of primitive operation calls (i.e., predefined primitives for type modifications as shown in Fig.5.9) which are defined at the meta level within *TYPE* .

Step 2: *Checkpoint*: This step consists of saving the state of the type sublattice and all objects of those types in the sublattice. Then, apply the locking protocol to prevent inconsistent use of the type to be modified and of its instances. The locking protocol is also applied recursively to the subtypes of the modified type, and to their instances.

Step 3: *Modifications performed:* This step consists of performing the changes as specified by the transaction. The old definitions of the types involved in the change are saved within the previous step. The modification are performed on these types without changing their identities. Therefore, we do not need to recompile the specification.

Step 4: *Structural consistency checking and SR:* the checking process consists of maintaining the structural consistency, after the type modifications, according to the invariants defined within the type *TYPE*. Such invariants correspond mainly to the static semantic rules of the language. If the structure of a specification (i.e., modified or newly constructed specification) does not comply with those invariants, then the *SR* is used to reflect the anomalies to the previous level (i.e., meta-level) in order to inform the user of which part about his transaction does not satisfy the invariants. Then the user has to modify his transaction through the meta-level (from step 1), in order to make the specification comply with the invariants.

Step 5: *Behavioral conformance checking*: The behavioral conformance deals with the dynamic behavior of objects as introduced in Definition 3. According to the behavior modifications, if any, we need to check dynamically that the modification of the behavior of an object does not introduce new deadlocks in the overall specification. Among the existing approaches for deadlock detection (e.g., program transformation, simulation, reachability analysis) we use a dependency graph and the reachability analysis techniques widely used for the validation (e.g., deadlock detection) of communication protocols [Zafi 78], [Zhao 86]. A dependency graph is constructed based on the relation of dependency between types. A type t1 depends on a type t2 if the former uses one or more operations of the later. If the *extension* relation is violated, e.g., a deadlock is detected, then the system reports the inconsistencies and the type must be revised again.

Step 6: *Istances conversion:* when the type modification transaction succeeds, (i.e. the structural consistency and the behavioral conformance relations hold) then the instances (locked previously), at the ground object level, must be converted to remain conform with their modified type. The conversion of the instances according to the semantics of each type evolution primitive operation, is described in [Erra 92b].

Step 7: *Transaction commit:* In this step, the transaction commits and the type sublattice and the instances are unlocked, after their modifications, and enter their passive state.

6.3. Example

Let us consider the vending machine example for which a specification was given in Fig.5.1. Suppose now that we want to modify the initial machine to deliver candies or chocolates, instead of candies only. This imply that we have to modify the type *Machine* and the type *User*, accordingly. For this purpose, we have to modify both interface and behavior defined of the initial *Machine*. In the following we will show how the modifications are performed upon the type *Machine*, according to the different steps of the mechanism described earlier. For the type *User* the modification can be done in a similar way. To the *Machine*'s interface, we add the operation "*PushAndGetChocolate*", and for the behavior we modify the procedure *Ready* by modifying the procedure body as shown in Fig.6.3.

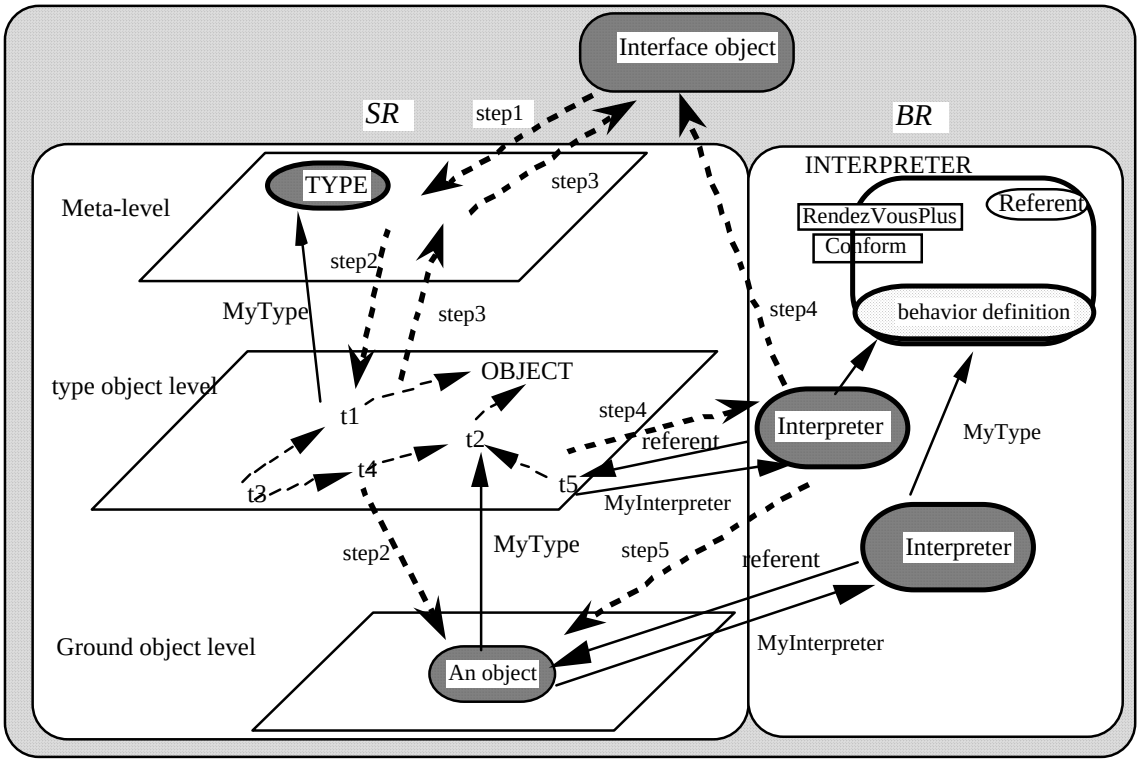

Fig.6.2. Reflection-based mechanism.

```
type Machine= object with
operation
   (same as in Fig.5.1.)
   PushAndGetChocolate;
behavior
   Ready
where
 procedure Ready =
   accept InsertCoin do return; end;
   choice
            DeliverCandy;
   or       DeliverChocolate;
   end;
  endproc Ready

procedure  DeliverCandy =
   (same as in Fig.5.1.)
 endproc DeliverCandy

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

endtype Machine
```

Fig.6.3. Modified vending machine system.

Step 1: The user formulates the atomic operation (i.e., a transaction) using RMondel statements. In Fig.6.4 we give a possible specification of an atomic operation (see line 2 of Fig.6.4). The user formulates his atomic operation using the predefined kernel types (e.g., *Procedure*, *Statement*, etc..) to create the necessary objects (see lines 4 to 12 of Fig.6.4). The type *Machine* which is an instance of *TYPE*, accepts the modification primitives defined in the type *TYPE* . Among the actions of the *updateMachine* atomic

operation, we have the *AddOper* (i.e., to add an operation) call on the type *Machine*. The *AddOper* call takes the operation to be added as a parameter (see lines 12,13 of Fig.6.4).

Another change, is to modify the body of the procedure *Ready* of the initial specification, accordingly. The procedure defined in the initial specification, is a sequential composition where the first statement is "accept InsertCoin do return; end;" and the second statement is the procedure call "DeliverCandy". Now we change the second statement by the new added statement of type *Choice* (i.e., "choice DeliverCandy or DeliverChocolate" as shown in lines 6 to 10 of Fig.6.4). Then an instance of the predefined kernel type *Procedure*, is created to hold the "DeliverChocolate" procedure as shown in lines 14,15 of Fig.6.4.

Step 2: This step consists of applying the locking protocol to prevent inconsistent access to the type (and its instances) under modification. Then the states of the type *Machine* and its existing instances are saved. This is done implicitly according to the atomic operation semantics, to allow a roll back in the case where the atomic operation aborts.

Step 3: The changes are performed on the type *Machine* as specified in lines 6 to 17 of Fig.6.4.

Step 4 and **step 5**: *Structural consistency* and *Behavioral conformance checking*: At the end of the transaction, just before the return of the atomic operation (see line 18 of Fig.6.4), the predefined invariant must hold for the type *Machine* after its modification. The invariants are triggered automatically to ensure the consistency of the *Machine* structure. In this stage, the *SR* will have a role because if the user adds certain information which does not produce the specification's needed structure, then there will be a reflection from the object type level to the meta-level represented as *SR*. This type of reflection informs the user at the meta-level which part of his/her specification should be re-modified/updated such as to make it in line with the structure needed by the specification. It is obvious from the resulting *Machine*'s specification shown in Fig.6.3, that the structural consistency as defined in Section 2 is preserved. The addition of the operation "*PushAndGetChocolate*" in a choice composition as shown in Fig.6.3, preserves the *behavioral conformance* requirements according to Definition 3.

Step 6: At the end of the transaction, and after both structure and behavior are checked, the existing instance of the type *Machine* has to be converted accordingly. In this example, the modification (i.e., addition of operation) has no impact on the structure of

the existing instances if any. Because the added operation appears only within the type *Machine* , the instance behaviors evolve dynamically when they become unlocked.

```
1  type TransExample = Object
     ...
   operation
2 updateMachine : atomic;
                ...
    behavior
3       accept updateMachine do

4     { let Machine be the object type Machine of Fig.2.1, which is an instance of TYPE }
5     { let ProcReady be the object of type Procedure where the procedure body is a statement object of
        type Sequence. ProCall and Choice are predefined kernel types for procedure call and choice
        statements, respectively. More details on Statement object can be found in [Erra 92] }

6             define   DelivCand = new ProCall ("DeliverCandy") ;
7                      DelivChoc = new ProCall ("DeliverChocolate") in
8                      define  CanOrChoc = new Choice (DelivCan, DelivChoc) in
9                              ProcReady.ProcBody.Stat2 := CanorChoc;
10                     end;
11            end;
12            define op = new Operation ("PushAndGetChocolate") in
13                     Machine ! AddOper(op);
14                     define ProcChoc = new Procedure ( ...) in
15                             Machine ! AddProc(ProcChoc);
16            end;
17            end;
18            return;
19       end;
20    end;
21 endtype TransExample
```

Fig.6.4. An example of a transaction for the type *Machine* updates.

7. Related works

In the area of object oriented databases, class modifications have been extensively studied in the recent literature [Bane 87], [Penn 87], [Skar 87], and [Delc 91]. The available methods determine the consequences of class changes on other classes and on the existing instances, so that possible violations of the integrity constraints can be avoided. These approaches deal mainly with sequential systems and have focused on preserving only structural consistency. In our approach, we address both the structural and behavioral consistencies. For the behavioral consistency we deal mainly with the behavior of objects and we consider some properties of distributed systems such as blocking. Moreover, we use reflection which provides a flexible and uniform environment for dynamic type modifications using specific meta-operations and meta-objects. Another work on class modification using meta-operation

is that of [Tan 89], where a lazy evaluation method of schema evolution which minimize the amount of object manipulation is proposed. However, [Tan 89] does not address the issues of behavioral conformance and the transaction mechanism which are central to our work.

Kramer and Magee have addressed the problem of dynamic change management for distributed systems [Kram 90, Kram 89]. Their approach focuses mainly on changes specified in terms of the system structure and provides a separate language for changes specification. Our approach deals with type modifications and uses one language to specify types and their changes. Unlike their approach, which concentrate on the logical structure of a system, we consider the dynamic behavior of a specification and we take into account the inheritance property which is inherent to the object-oriented aspect of our language. The unit of change in our model is a type (class), instead of a module.

8. Conclusions

Dynamic type modifications is an interesting and challenging research problem. Object-oriented systems in conjunction with reflection, allow us to approach this problem that conventional systems have not been able to address. We have developed RMondel, a reflective concurrent object-oriented specification language, based on the Mondel language designed for distributed systems modeling and specification. The objective of RMondel is to allow the development of dynamically modifiable specifications. We have shown the fundamental features of RMondel, mainly the structural reflection and the behavioral reflection. Then we have explained how the features of the language are useful for the dynamic modification and construction of valid specifications. We have illustrated through an example how the language allows dynamic modifications. Therefore the user of this language can modify his/her specification by adding new objects and types to get a new adapted specification. A predefined set of constraints allow to maintain the structural consistency and behavioral conformance of the modified specification. The allowed modifications lead to new types which conform to the old ones.

RMondel provides an elegant manner for dynamic type modifications. It also gives an interesting framework based on formal semantics, to develop adaptable CASE tools for executable specifications development. RMondel framework may be easily adapted for other object-oriented languages. Mondel has been implemented on a Sun workstation, and used for writing and simulating the specifications of the OSI directory system, and the personal communication services. Currently, *RMondel* is in the implementation phase. Our future research focuses on how and under which conditions the modifications

to a given specification may be performed upon an implementation within the same transaction. The modification must be done in a way to preserve the conformance relation between the implementation and its specification. We are also considering the development of a version control mechanism in order to keep track of the evolution history of an evolving specification.

Acknowledgement: The authors are grateful to M. Faci for his useful discussions and his revisions to the paper. This research was supported by a grant from the Canadian Institute for Telecommunications Research (CITR) under the NCE program of the Government of Canada.

References

[Amer 90] 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 91] M. Barbeau, *Vérification de spécifications en langage de haut niveau par une approche basée sur les réseaux de Pétri*, Ph.D. Thesis, Université de Montréal, 1991.

[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.,

[Boch 91] G. v. Bochmann, L. Lecomte and P. Mondain-Monval, *Formal Description of Network Management Issues*, Proc. Int. Symp. on Integrated Network Management (IFIP), Arlington, US, April 1991, North Holland Publ., pp. 77-94.

[Boch 92] G. v. Bochmann, S. Poirier and P. Mondain-Monval, *Object-oriented design for distributed systems and OSI standards*, to be published in Proc. of IFIP Int. Conf. on Upper Layer Protocols, Architectures and Applications, Vancouver, May 1992.

[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.

[Coin 87] P. Cointe, *Metaclasses are first class: The ObjVLisp Model*, OOPSLA'87, ACM Sigplan Notices 22, 12, pp.156-167.

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

[Erra 90] M. Erradi and G. v. Bochmann, *RMondel: A Reflective Object-Oriented Specification Language*, The ECOOP/OOPSLA'90 First Workshop on: Reflection and Metalevel Architectures in Object-Oriented Programming, Ottawa 1990.,

[Erra 92] M. Erradi and G. v. Bochmann, *Semantics and definition of RMondel : A Reflective Object-Oriented Language*, Internal report, departement IRO, University of Montreal, 1992.

[Erra 92b] M. Erradi, G. v. Bochmann and R. Dssouli, *Semantics and implementation of type dynamic modifications*, Publication #813, Department IRO, University of Montreal, March 1992.

[Erra 92c] M. Erradi, G. v. Bochmann and I. Hamid, *Dynamic Modifications of Object-Oriented Specifications*, CompEurop'92, IEEE Int. Conf. on Computer Systems and software Engineering, May 1992.

[Ferb 89] J. Ferber, *Computational Reflection in Class based Object Oriented Languages*, Proceedings of OOPSLA'89 , October 1-6, 1989, pp. 317-326.

[Gray 78] J. Gray, *Notes on data base operating systems*, IBM research report: RJ2188, IBM Research, San Josee, California, 1978.

[Gray 81] J. Gray, *The transaction concept: virtues and limitations*, Proc. Conf. on VLDB, Cannes, Sept. 1981 (IEEE), p. 144-154.

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

[Kram 89] J. Kramer, J. Magee and M. Sloman, *Configuration support for system description, construction and evolution*, IEEE Proc. of the Fifth Int. Work. on Soft. Spec. and Design, May 1989, pp.28-33.

[Kram 90] J. Kramer and J. Magee, *The evolving philosophers problem: Dynamic change management*, IEEE, trans. on Soft. Eng. Vol.16, No.11, November 1990.

[Maes 87] P. Maes, *Concepts and Experiments in computational reflection*, OOPSLA'87, ACM Sigplan Notices 22, 12, pp.147-155.

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

[Penn 87] D. J. Penney and J. Stein, *Class Modification in the GemStone object-oriented DBMS*, OOPSLA'87, 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, Eds. Peter Wegner and Bruce Shriver, MIT press, pp.393-415.

[Smit 82] B. C. Smith, *Reflection and Semantics in a Procedural Programming Language*, Ph.D. Thesis, MIT, MIT/LCS/TR-272, 1982.

[Tan 89] L. Tan and T. Katayama, *Meta operations for type management in object-oriented databases - A lazy mechanism for schema evolution-*, Proceedings of the First Int. Conf. on Deductive and Object-oriented Dtatbases, Octobre 1989.

[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.

[Wata 88] T. Watanabe A. Yonezawa, *Reflection in an Object-Oriented Concurrent Language,* Proceedings of OOPSLA'88, pp. 306-315.

[Will 90] N. Williams, *Un simulateur pour un langage de spécification orienté-objet*, MSc thesis, Université de Montréal, 1990.

[Work 91] M. H. Ibrahim, *ECOOP/OOPSLA' 90/91 Workshops on Reflection and Metalevel Architectures in Object-Oriented Programming.*

[Zafi 78] P. Zafiropulo, *Protocol validation by duologue-matrix analysis*, IEEE Trans. on Comm. Vol. COM-26, No.8,1978, pp. 1187-1194.

[Zhao 86] J. R. Zhao and G. v. Bochmann, *Reduced reachability analysis of communication protocols: a new approach*, Proc. IFIP Workshop on Protocl. Spec. Testing and Verification, North-Holland Publ., 1986, pp. 234-254.