A Framework for Dynamic Evolution of Distributed Systems Specifications*

Mohammed Erradi, Gregor v. Bochmann and Rachida Dssouli

Université de Montréal, Département d'Informatique et
de Recherche Opérationnelle, CP. 6128, Succ. "A"
Montréal, (Québec) Canada H3C-3J7

Email : {erradi, dssouli, bochmann} @iro.umontreal.ca
Tel : (514) 343-7484
Fax : (514) 343-5834

RÉSUMÉ. *Récemment, les spécifications orientées objets des systèmes distribués ont suscité beaucoup d'interêt. L'approche orientée objet offre beaucoup de flexibilité pour la construction des systèmes. Cependant, un des problèmes culminant est d'effectuer des modifications de façon dynamique durant le processus de développement d'exploitation et de maintenance.*

Nous nous intéressons dans cet article, aux techniques de description formelle qui permettent le développement et la modification dynamique des spécifications exécutables. Nous introduisons un modèle à deux niveaux pour l'évolution des spécifications orientées objets. Le premier niveau concerne la modification dynamique des types (classes), alors que le second niveau est consacré à la modification dynamique des modules. Pour chacun des niveaux, nous définissons un ensemble de contraintes, structurelles et comportementales, qui assurent la consistence de la spécification après sa modification. Pour effectuer les modifications de façon dynamique, nous avons développé un langage réflexif de spécification orientée objet. Ce langage permet de définir les opérations de modifications à un meta-niveau en utilisant des meta-objets. Dans ce langage, les types et les modules sont des objets.

MOTS-CLES. *Spécification orientée objet, évolution de logiciel, modification des types, compatibilité des modules, réflection, modification dynamique.*

ABSTRACT. *Recently, object-oriented specifications of distributed systems has gained more attention. The object-oriented approach is known by its flexibility for system construction. However, one of the major challenges is to provide facilities for the dynamic modifications of such specifications during the development and maintenance process. Yet, current work has not addressed the dynamic modifications of specifications of distributed systems.*

In this paper, we are concerned with formal description techniques that allow for the development and the dynamic modification of executable specifications. A two-level model for the evolution of large object-oriented specifications is introduced. The first level deals with the dynamic modification of types (classes), while the second level deals with the modification of modules. We have defined a set of structural and behavioral constraints to ensure the specification consistency after its modification at both levels. To allow for dynamic modification of types and modules, we have developed a reflective object-oriented specification language which uses meta-objects to support the modification operations. In this language, types and modules are objects.

KEY-WORDS: *Object-oriented specifications, software evolution, type modification, modules compatibility, reflection, dynamic modifications.*

---

* Revue Réseaux et Informatique Répartie, Vol.4, No.2

1. Introduction and motivations

In any software system, especially distributed systems, that exists for long period of time, it becomes necessary to periodically change various components of the system. While it has long been understood that support for software evolution and maintenance is an important part of programming methodology, the proper techniques and tools for performing that evolution and maintenance easily and correctly are often not available. There are a number of reasons that might necessitate evolutionary change. One possible reason is the users' requirements change over time. Another is to improve efficiency. Finally, systems need to evolve as the application environment changes. All these cases are examples of change and evolution problems faced with large software systems.

In large distributed software systems, it may not be possible to stop the entire system to allow modification to part of it. A major challenge is to provide facilities for the dynamic modifications of an evolving system without interrupting the processing of those parts of the system which are not directly affected. In general, evolutionary changes are difficult to accommodate because they cannot be predicted at the time the system is designed. Systems should be sufficiently flexible to permit arbitrary, incremental changes. In addition, when systems are large, their manipulation, understanding, and maintenance become difficult. The importance of decomposing large systems into modules is widely recognized within the software engineering community [Brin 89], [Webe 86], [Parn 72]. Therefore, the availability of modules is of great practical use for the production of structured systems that are easier to manipulate, understand, analyze and maintain. System modularity is essential, and permits the use of composition to form a system from reusable and independent modules. The use of well defined module interfaces allows for the validation of module interconnections.

We believe that software system modifications are most of the time incremental. Therefore, 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 dynamic extensions of distributed systems specifications in the context of the object-oriented specification language Mondel [Boch 90]. Mondel has important concepts as a 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, Mondel has been used for the specification of problems related to network management [Boch 91] and OSI directory system [Boch 92]. The object-oriented approach is known by its flexibility for system construction. This is partly due to the inheritance property. However, there has been little suggestions to provide facilities for the dynamic modification of object-oriented systems.

In this paper, a two-level generic model for managing large specifications evolution is introduced. This model consists of the *in-the-small* and the *in-the-large* levels. The *in-the-small* level deals with the dynamic modification of classes within an object-oriented specification. The *in-the-large* level deals with the dynamic modification of modules within large object-oriented specifications. The distinction between the two levels is made because modules are different from classes used by the object-oriented approach. The construction of modules is different from the composition of objects. Modules may be constructed in many different ways and do not always follow the inheritance relationship which exists between classes and subclasses. In addition, classes are primarily aimed at supporting "programming in the small", while modules are used for "programming in the large".

In our model, 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. Modifications may be performed on the structure as well as on the behavior of the specification. Therefore, we consider both structural and behavioral consistencies requirements. Structural consistency deals with preserving the consistency of the structure of the specification after its modification. This 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 of the specification after its modification. This concerns mainly some properties of distributed systems such as blocking. The consistency requirements are addressed at the *in-the-small* and at the *in-the-large* levels.

In the context of Mondel, large specifications are constructed from *units* (called modules in other languages) which are composed of *types* (called classes in most object-oriented languages). In order to allow for the construction of dynamically modifiable specifications, we need to have access, and to be able to modify units and types during execution-time. Therefore, we developed RMondel, a new version of Mondel, where types are objects, and meta-objects

are used to provide facilities for the dynamic modifications of types [Erra 92c]. In a similar way, meta-objects are also used to allow for the dynamic modifications of units.

The paper is structured as follows: Section 2 introduces the two-level generic model for large specifications evolution. The first level describes the evolution of object-oriented specifications by considering classes as the basic units of specification construction. The needed requirements to maintain the consistency at this level are also addressed. Then, we describe the second level that presents the module concept as the unit of large specifications composition, and the needed requirements to maintain consistency at the module boundaries. In Section 3, after an overview of the Mondel specification language and RMondel, we show how the two levels of modification, introduced by the generic model, are supported in the RMondel language. Thereafter, we introduce the constraints which preserve the structural and the behavioral consistencies for both levels, and we discuss how modifications may be performed dynamically in these two levels. Conclusions are drawn in Section 4.

2. A generic model at two levels

In this section we discuss a generic model for the evolution of object-oriented specifications. This model consists of two levels of specification evolution which are the class level and the module level, called *in-the-small* and *in-the-large*, respectively. Figure 1 gives a global view of these two levels. The *in-the-large* level deals with the specification modules and their interconnections while the *in-the-small* level deals with classes and their relationships.

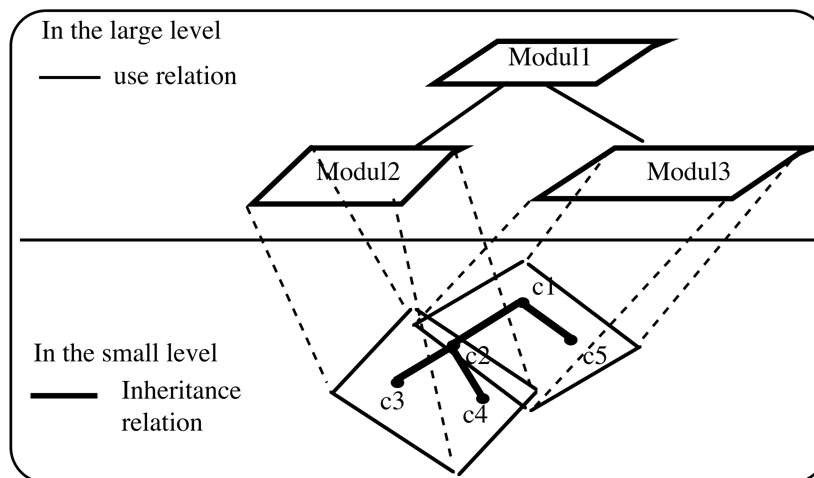

Figure 1. Two-level view

2.1. The *in-the-small* level

For object-oriented specifications to fulfill their promise as vehicles for fast prototyping, ease of maintenance, and ease of modification, a well defined and consistent methodology for class modification must be developed. A class is a set of objects called its instances. A class definition specifies the features (a feature can be an instance variable or a method) and the allowable behavior of its instances. At the *in-the-small* level a specification consists of a class lattice. A node in the lattice represents a class and an edge between a pair of nodes represents the inheritance relationship. Inheritance allows a new class to be derived from an existing class. The new class, called a subclass, inherits all the features of the existing class. One may also specify additional features for the subclass. In the following we enumerate the allowed modifications at the *in-the-small* level, and we discuss the consistency requirements at this level.

2.1.1. Class modifications

Class modifications are typically achieved by adding or removing instance variables, reimplementing methods, rearranging inheritance links, etc. Such modifications indicate that the existing classes are not entirely satisfactory. In the area of object oriented databases, these 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. It is important to note that the existing approaches deal mainly with sequential systems and do not address behavior modifications. An acute problem in designing a methodology for class modification is how to bring existing objects in line with a modified class. This problem becomes more acute in the case of distributed systems such as our environment where object behaviors impose additional constraints.

Class updates may be classified into three categories [Bane 87]: (1) updates to the contents of a node in the class lattice, (2) updates to an edge in the class lattice, and (3) updates to a node in the class lattice. In the following we enumerate the most important update operations on classes.

(1) Modifications to the contents of a node in the class lattice.

(i) Modifications to an instance variable of a class.

- *Add an instance variable V to a class C.*
- *Drop an existing instance variable V from a class C.*
- *Change the class C of an instance variable V.*

(ii) Modifications to an operation of a class.

- *Add the operation O to the class C.*
- *Drop the existing operation O from the class C.*
- *Change the signature S of the operation O.*

(2) Modifications to an edge of the lattice.

(i) Make a class S a superclass of class C.

(ii) Delete a parent S (superclass) of the class C.

(3) Modifications to a node of the lattice structure.

(i) Add a new class C.

(ii) Delete an existing class C.

2.1.2. Kinds of consistency

At the *in-the-small* level three kinds of consistencies must be addressed w.r.t. class evolution: the structural consistency, the semantical consistency, and the instance-of relationship consistency.

The structural consistency : It ensures that the structure of the specification (class lattice) is maintained according to the inheritance relation. This kind of consistency is widely investigated in object-oriented databases area where some invariants are used to define the consistency requirements of the class lattice [Bane 87], [Penn 87] (e.g., the distinct name invariant, ensures that all instance variable and method names of a class whether explicitly defined or inherited, are distinct).

The semantical consistency: While most existing approaches [Bane 87], [Penn 87], and [Delc 91] have focused on preserving structural consistency, we believe that the semantical consistency which deals with object behaviors must be addressed. The methodology of Skarra and Zdonik [Skar 87] goes a long way toward preserving behavior in sequential systems. Their methodology implements class modification by the use of versions. However, we are exploring solutions to class modification that do not require versioning. Our definition of the semantical consistency, that we call behavior extension, will be given in Section 3.3.1.

The instance-of relationship consistency: While classes evolve, their existing instances must be changed in order to remain in line with their classes. This kind of consistency can be defined according to the allowed class modifications which, in a distributed environment, deals with both structural and behavioral aspects of objects. For instance the addition of an instance variable within a class involves the restructuration of the existing objects of this class.

2.2. The *in-the-large* level

Before addressing the problem of specification modifications and the consistency requirements at the in-the-large level, we need to understand what are the components of a specification and their relationships. A large specification consists of a lattice of interconnected modules. A node in the lattice represents a module, and an edge between a pair of nodes means that the upper level module uses the module of the level below. The hierarchic organization of modular specifications does not predetermine the way they are developed i.e., top-down, bottom-up, or in an iterative manner.

2.2.1. Constituent parts of modules

A module consist of three parts: an export interface, an import interface, and a module body.
The export interface is the visible part which must be known for using this module in connection with other modules. It allows different aspects of information hiding such as:
- It prevents a user from looking into the internal structure of a module.
- It protects some of the resources that exist internally from their use from outside the module.

The import interface contains references to one or more other modules. Modules may not import each other cyclically.

The module body is intended to define the construction of the export interface using the import interface, and may contain auxiliary hidden resources such as classes and objects, which do not belong to any other part of the module.

2.2.2. Module interconnections

For large specifications, the development process consists of a sequence of alternating incremental completions of incompletely developed modules and refinements through successive decompositions and compositions for the top-down or bottom-up continuation of the development process. For this, a set of fundamental operations on module specifications has been provided [Blum 87]. These include horizontal structuring operations such as composition and union, and vertical development steps, such as refinement.

- The *composition* of two modules M1 and M2 connects the import interface of M2 with the export interface of M1. The composite module (M1 comp M2) will have the same import interface as M1, the same export interface as M2, while the body of (M1 comp M2) is given by the union of the body parts in M1 and M2 (see Figure 2. (a)).
- The *union* M1 U M2 of two modules M1 and M2 is the disjoint union of M1 and M2. The constituent parts of the resulting module (M1 U M2) are the union of the corresponding parts of the original modules (see Figure 2. (b)).
- The *extension* $ext_E(M)$ of a module M is the result of extending some or all constituent parts of the module M by additional items, where E denotes the collection of all extended items. The extension construction is used to augment a given module by adding items in the export, import or body part of a module (see Figure 2.(c)). This construction is important to build up modules step by step.

2.2.3. Module modifications

Since each module forms a small and independent piece of the whole specification, then modules can be developed, implemented, and modified individually. As specifications evolve, designers can be led to modify modules so that they suit their needs. This is typically achieved by modifying module constituent parts (e.g., adding or removing a resource to/from an import or export interface). We classify module modifications into the following categories:

(1) modification of the export interface: Adding and/or removing a named object or a class to/from the export interface of the module.

(2) modification of the import interface: Adding and/or removing a named object or a class to/from the import interface of the module.

(3) modification of the body part of a module: As a consequence of the import and/or the export interface modifications, the body of the module may be changed. Sometimes, one needs to modify the body of a module without modifying the interface (e.g., performance enhancement).

(4) Addition and/or deletion of a module.

A particular aspect of module modification in object-oriented systems is to specialize objects or classes by means of inheritance. This aspect will be discussed further in relation with our environment in Section 3.4.

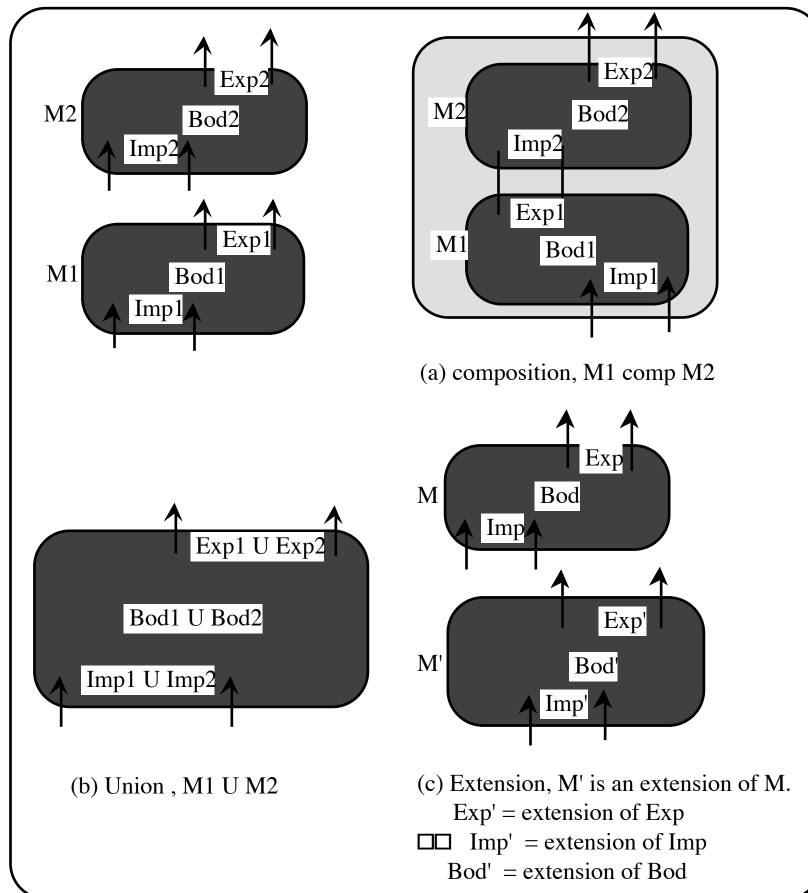

Figure 2. Operations on modules

2.2.4. Kinds of consistency in the in-the-large level

According to the allowed modifications of modules, there are two kinds of consistencies to be considered. First, the structural consistency deals mainly with type checking at the module boundaries. For example, the extension of a module should ensure type compatibility, at the import and export interfaces, between the original module and its extended version. Second,

the semantical consistency deals with the behavior aspect of modules. That is, an extended module should provide the behavior required by the whole specification. The extended module should provide at least what the original module provides.

These two kinds of consistency must be addressed at the module level and at the whole specification level. At the module level we should ensure that the allowed modifications, of the import and the export interfaces of a module, will not violate the static semantics rules, e.g., an object or a class name must not be imported and exported by the same module. Moreover, the modifications of the body part of a module must be done without resulting in run-time errors, blocking, or any uncontrollable situation. At the whole specification level, one needs to check the impact of the module modification on the other modules. This should preserve the specification in a consistent state after the modification of one or more modules. According to our goal which is the dynamic modification of a running specification, it should not be necessary to stop the execution of the specification to modify parts of it. One should define a mechanism to determine the part of the specification which are affected by the change. The rest of the specification should be able to continue its execution normally. In Section 3.4, we address these problems in the context of RMondel.

3. Specification Evolution in RMondel

According to the generic model presented in Section 2, we will show how the features of such a model are supported by RMondel. RMondel is an object-oriented specification language, suitable for the specification and modeling of distributed systems. It provides facilities for building dynamically modifiable specifications. After an overview of the original language Mondel, we introduce the main characteristics of RMondel language. Then we describe evolution at the *in-the-small* level and at the *in-the-large* level as supported in this language.

3.1. 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. An object is an instance of a type (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 which may be fixed references to other objects, and operations which are externally visible.

A Mondel specification corresponds to a type lattice. In such a lattice, types are linked by mean of 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 the rendezvous mechanism. Mondel has a formal semantics which associates a meaning to the valid language sentences. The behavior of objects is formally specified by a translation to labeled transition systems. 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].

3.1.1. Example of a Mondel specification

In the following we show an example using the *Mondel* language. Let us consider a vending machine which receives a coin and delivers candies to its user. The specification of the vending machine system consists of one module composed of two types: the type *Machine* and the type *User*, as shown in Figure 3. The root of the type lattice in Mondel is the most general type *object* (called *top* in other languages). Therefore, the types *Machine* and *User* inherit from *object* as shown in lines 1 and 21 of Figure 3. The relation between the type *User* and the type *Machine* is represented by the attribute *m*, of type *Machine* (see line 22 of Figure 3), defined within the *User* type. The operations, InsertCoin and PushAndGetCandy, are specified within the operation clause as shown in lines 2 to 4 of Figure 3. Note that these operations are without

parameters and result. 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. 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. The behavior of the vending machine system is defined as the composition of two interacting objects (i.e., *Machine* and *User* objects) (see lines 35 to 38 of Figure 3.) . The types are specified using a state oriented style [Viss 88] where internal states are modeled as Mondel procedures.

```
0 unit VMsystem =

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 VMsystem
```

Figure 3. Mondel specification of the vending machine system

3.2. RMondel facilities

In the formalism used to define the semantics of Mondel, types are static and used as templates for instance creation. Only the instances of a type are considered as objects. To support the construction of dynamically modifiable specifications we need to have access to types during execution-time. For this purpose, reflection is a promising choice.

Recently, in object-oriented languages, reflection has gained wider attention as confirmed 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.

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 [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*. One can say that types are structural meta-objects, while interpreters are behavioral meta-objects.

More details on the definition of RMondel and of other kernel objects, are given in [Erra 92d]. Let us now introduce the fundamental features of reflection as supported in RMondel. We distinguish two main features: Structural reflection (SR) and behavioral reflection (BR). 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. Another aspect of *SR* is that the RMondel statements and expressions are objects. 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 *subtype-of* relationship. *TYPE* and *OBJECT* are the respective roots of these two graphs [Erra 90]. Figure 4 gives a subset of the definitions of the structures used to create new types and to detect the inconsistencies in the type lattice induced by modifications.

The *behavioral reflection* (*BR*) deals with object behaviors. Therefore, an interpreter object is associated to each object. An interpreter object deals with the computational aspect of its associated object called *referent*. Interpreter objects are defined as instances of the type

INTERPRETER. Also 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, or for dynamically modifying their behaviors. A possible specification of the type *INTERPRETER* is given in [Erra 92d].

Types and interpreters are instances of the kernel types *Modifiable-Type*, a subtype of *TYPE*, and *INTERPRETER*, respectively. This approach shows many advantages:
- types are objects, instances of the *Modifiable-Type*, which is defined at a meta-level.
- operations for type modifications may be defined at the meta-level (i.e., within *Modifiable-Type* as shown in Figure 4).
- an object behavior can be monitored and/or modified by its interpreter.
- new communication strategies can be defined by creating subtypes of *INTERPRETER*.

```
  type TYPE = OBJECT with
        TypeName          : string;
        BehaviorDef       : var[Statement];
        DirectSuperTypes  : set [TYPE];
        Attributes        : set [AttributeDef];
        Operations        : set [Operation];
        Procedures        : set[Procedure];
        . . .
  operation
        . . .
        {the operation New creates an object according to RMondel object structure}
        New : OBJECT;
        {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}
        LookUp (OpName : string) : Statement;
 Behavior
        {□the semantics definitions of the above operations}
endtype TYPE

{ the type Modifiable-Type is defined as a subtype of TYPE as follows }

type Modifiable-Type = TYPE with
operation
        AddStat (S: Statement);
        AddAttr (A:Attribute);
        AddOper(O:Operation);
        AddProc(P:Procedure);
        . . .
behavior
        { The semantics definitions of the modification operations above . }
endtype Modifiable-Type
```

Figure 4. TYPE and Modifiable-Type specifications with the modification operators

The reflection facilities of RMondel together with the principles introduced by the generic model in Section 2, form the basis of the dynamic evolution of large specifications written in RMondel.

3.3. *In-the-small* modifications in RMondel

We are mainly interested in the modifications of a specification S which lead to a consistent specification S' using an incremental approach. The incremental approach consists of dynamically extending the specification S to get a consistent specification S' such that the later conforms to the former. The type modifications in RMondel may be seen as a specialization of the *in-the-small* level introduced in the generic model of Section 2.

In RMondel specifications, which mainly describe distributed applications, the dynamic behavior of objects is of extreme importance. Therefore, our interpretation of type modifications takes into account the dynamic behavior of objects. According to the generic model, the *in-the-small* level in RMondel is concerned about type modifications and the consistency requirements which ensure both structural and behavioral consistencies. The structural consistency deals with the compiling constraints (e.g., type checking), while behavioral consistency deals with the dynamic behavior of objects (e.g., possibility of blocking).

3.3.1. Definition of consistency constraints

Before addressing the in-the-small modifications of RMondel specifications, an understanding of types and their relationships is required.

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. ❑

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 recursive 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'. ❑

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 92a]. 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). ❑

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 90], 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. ❑

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

3.3.2. Structure modifications

In the following we give a classification of type interfaces modifications that are supported in *RMondel*, and we provide the description of their semantics. As we are concerned by the incremental approach for specification evolution, we will consider those type modifications that lead to new types which are structurally consistent with the old ones.

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.

Change the type T of an attribute A by the type T1: This update is allowed only if T1 inherits from 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 inherits from 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' inherits from 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 lattice. The attributes and operations provided by S, are inherited by T and by the subtypes of T.

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 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.

In *RMondel*, types are objects (e.g., instances of Modifiable-Type which is defined at a meta-level). The Modifiable-Type provides the primitive operations for type modifications defined above (see Figure 4.).

3.3.3. Behavior modifications

For the modification of the behavior, 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 behavior of objects is to some degree dependent upon preserving *structural consistency*. For instance, when an operation is called on an object, the associated code 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's 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, to changes in the behavior, accordingly.

The possibilities of behavior modifications are based on the language constructs which can be involved in such modifications. The behavior obtained after modification, should be an extension of the old behavior. A modified behavior consists of the composition of an old behavior with a new one. This composition is based on the sequential, the choice, or the parallel composition operators of RMondel. Further details on the behavior modifications are given in [Erra 92b]. An algorithm for behaviors composition is given in [Khen 92].

3.3.4. Invariant definitions

In this section we define a set of invariants which are deduced from the definitions of Section 3.3.1. Such invariants must be satisfied by each type and its related types in the type lattice. The invariants are checked when an object or a type is created and after type updates. In RMondel, the invariants are checked dynamically. Therefore, they are defined at the meta-level within the *invariant* clause of the Modifiable-Type as shown in Figure 5.

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, whether explicitly defined or inherited, are distinct.

Object Representation Invariant: 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 conflicts are not addressed here, but may be avoided using the name conflict detection algorithms in a similar way as in [Delc 91].

Type Compatibility Invariant: If a type T defines an attribute with the same name as an attribute it would otherwise inherit from a supertype S, the type of T's attribute must be the same or a subtype of S's attribute.

```
{ the class of modifiable types is defined as a subclass of TYPE as follows }
type Modifiable-Type = TYPE with
            AddStat (S: Statement);
            AddAttr (A:Attribute);
            AddOper(O:Operation);
            AddProc(P:Procedure);
            ...
  invariant
            { We define here, the invariants described above, which correspond to the static semantics rules
              of the language.  A formal definition, in Mondel, of these invariants is given in [Erra 92a] }
   behavior
            { The semantics definitions of the modification operations above . }
endtype Modifiable-Type
```

Figure 5. Modifiable-Type specification with the invariants

3.3.5. Consistency checking of a specification

In this section we discuss the consistency checking of a specification after type modifications. On the one hand, if we replace a type t by t' in some specification S, where t' is *structurally consistent* with t, does the resulting specification S' remain structurally consistent w.r.t. S ? Recall that types are objects in *RMondel*. Our strategy for type modification allows the modification of types without changing their identities. This implies that the whole specification remains consistent from the structural point of view, i.e., we do not need to

recompile the whole specification. This may be proved according to two situations, which are assignment and parameter passing, where type checking is important.

On the other hand, if we replace a type t by t' in some specification S, where t' *extends* t, does the resulting specification S' *extends* S ? The answer is in general no, an illustrative example is given in [Erra 92b]. Therefore, 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.

In order to ensure the consistency of the whole specification after its modification, we use the concept of transaction which is well-known for database systems. The user formulates his requirements within a transaction which consists of one or several type update operations. In order to allow for dynamic modifications of a given specification 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 those parts of the specification which are affected by the modifications. This protocol also ensures the mutual exclusion of concurrent transactions. The other parts of the specification continue to behave normally. This protocol is incorporated within our transaction mechanism. A detailed description of the transaction mechanism and the locking protocol, as supported in RMondel, is given in [Erra 92d].

3.4. In-the-large modifications in RMondel

For large specifications, the availability of modules is of great practical use for the production of structured specifications that are easier to manipulate, understand, analyze and maintain. Modules can be developed independently, and they can be analyzed or even compiled separately. Moreover, modules of general purpose can be reused within several specifications. In order to realize these features, a modular specification language has to fulfill several requirements. First, to enhance the independent development, analysis, and compilation of

modules, they should be represented as syntactical components in the language. Second, the composition of modules to build a complete specification should be simple, e.g., this aspect can be realized by means of the import/export mechanism.

In the following, we will show how these features are supported in RMondel using *units*. Then we introduce the structural and behavioral consistency requirements which allow for the construction of valid specifications. Afterwards, we introduce the unit modifications and their semantics as defined in RMondel.

3.4.1. The unit concept in RMondel

A unit consists of the following constituent parts: an import interface, an export interface, types, and a unit body. There are two forms of the import interface:

(1) Use U1, U2, ..., Un

(2) From Uj Use N1, N2, ..., Nm

Where the Ui are unit identifiers and the Ni are named objects or type names defined within the Ui. The first form makes the names of the units Ui (i=1,...,n) visible. This implies that the exported objects and the types of Ui are visible. The second form makes only the names N1, N2, ..., Nm visible from the unit Uj. This assumes that the names Ni are available in Uj.

The export interface has the form:

Export N1, N2, ..., Nm

where the Ni are named objects or type names. The export interface is intended to be the visible part of the module. The types of a unit constitute a type lattice where types are linked by means of the inheritance relation. The body part of a unit must include the definition of the exported objects . It can includes also a collection of types that can be used only within the unit.

There are certain semantical requirements that have to be fulfilled before several units may be considered to form a correct specification.

- A specification consists of a number of *units*. One unit may depend on objects and types given in other units. A given specification is executed by executing sequentially the behaviors in the different units. The units should be executed in such an order that if a unit A uses a unit B then the unit B is executed before the unit A.

- The types and those objects that are exported can be used by other units if the later explicitly indicate that they use the former.
- Units may not import each other cyclically.
- The exported objects and types should be uniquely identified to avoid name conflicts.

3.4.2. Notion of configuration

A large specification is a directed acyclic graph where the leaf nodes are units, and the internal nodes are subspecifications. Each subspecification is realized by a configuration of its successor nodes which may, in turn, be other subspecifications or modules. A configuration may be a combination of two or more units by means of the *composition* and/or the *union* operations introduced in Section 2.2.2. The characteristics of meaningful configurations are embodied in the concept of well-formed configurations introduced by Tichy [Tich 82]. However, Tichy's approach is purely syntactic. He does not consider the dynamic behavior aspect of modules. Our approach considers the dynamic behavior aspect and the inheritance relation as supported by the RMondel language.

The unit construct is defined in accordance with a set of constraints in order to have a structurally correct configuration. A structurally correct configuration must ensure that the same object and/or type cannot be imported and exported within such configuration. It is also important to ensure that the types and those objects that are exported by a unit are not exported by other units within the same configuration.

Let a specification configuration C = < C1,C2, ..., Cn > where each Ci may be a unit or another configuration. According to our definition of the unit we introduce the following notations:

EO(U) is the set of objects exported by the unit U,
ET(U) is the set of types exported by the unit U,
IO(U) is the set of objects imported by the unit U, and
IT(U) is the set of types imported by the unit U.

Definition 5: A specification configuration is well-formed if it satisfies the following conditions:

(1) Every type and object that is exported by C is exported by some Ci.

$$(EO(C) \cup ET(C)) \subseteq \cup_i (EO(Ci) \cup ET(Ci)) \text{ for } i=1,.., n.$$

(2) C imports those types and objects imported by all the Ci except for the types and objects already exported by some other component in the configuration.

$$IO(C) \cup IT(C) \supseteq (\cup_i (IO(Ci) \cup IT(Ci))) - (\cup_i (EO(Ci) \cup ET(Ci)) \text{ for } i=1,.., n.$$

(3) C does not export and import the same types and objects

$$IO(C) \cap EO(C) = \emptyset \quad \text{and} \quad IT(C) \cap ET(C)) = \emptyset$$

(4) No type or object is exported by more than one component.

$$EO(Ci) \cap EO(Cj) = \emptyset \quad \text{and} \quad ET(Ci) \cap ET(Cj) = \emptyset \quad \text{for all Ci, Cj of C, } i \neq j.$$

(5) The conditions (1) to (4) are satisfied by all Ci (for i=1,.., n). ❑

3.4.3. Consistency constraints for modifications

As modifications can be carried out on components of a configuration, it is important to ensure a continued validity of the specification as it evolves. In the following, we define the constraints that must hold to maintain the structural and behavioral consistencies of a specification after its modification.

Definition 6 [Tich 82]: A unit U2 is *UpWardCompatible* to the unit U1 if and only if: U2 exports at least what U1 exports, and imports not more than what U1 does. That means that U2 can be used instead of U1, but not vice versa:

$$(EO(U2) \cup ET(U2)) \supseteq (EO(U1) \cup ET(U1))$$

$$\text{and} \quad (IO(U2) \cup IT(U2)) \subseteq (IO(U1) \cup IT(U1))$$ ❑

This definition is based only on imported and exported objects and types. However our interpretation of the upward compatibility relation is not satisfied by this definition. We need to take into account the *conforms-to* relation defined for types in Section 3.3.1, and consider the dynamic behavior of the units. Therefore, we define the *UpWardConform* relation as follows:

Definition 7: A unit U2 is *UpWardConform* to U1 if the following conditions are satisfied:
(1) U2 is *UpWardCompatible* with U1.
(2) The type of an object exported by U2 *conforms-to* the type of an object exported by U1.

$\forall$ O1 $\in$ EO(U1) , $\exists$ O2 $\in$ EO(U2) such that (type (O2) *conforms-to* type (O1))

(3) The type of an object imported in U1 *conforms-to* the type of an object imported in U2.

$\forall$ O2 $\in$ IO(U2), $\exists$ O1 $\in$ IO(U1) such that: (type (O1) *conforms-to* type (O2))

(4) Every type exported by U2 *conforms-to* a type exported by U1

$\forall$ t1 $\in$ ET(U1) , $\exists$ t2 $\in$ ET(U2) such that : (t2 *conforms-to* t1)

(5) Every type imported in U1 *conforms-to* a type imported by U2

$\forall$ t2 $\in$ IT(U2), $\exists$ t1 $\in$ IT(U1) such that : (t1 *conforms-to* t2)

(6) The behavior of U2 (specified by its body part) *conforms-to* the behavior of U1. ❑

3.4.4. Semantics of the unit modifications

We classify the unit modifications that we support in RMondel and define their semantics. We distinguish the following categories: (1) modification of the export interface: Adding a named object or a type to the export interface of the unit. (2) modification of the import interface: Removing a named object or a type from the import interface of the unit. (3) modification of the types: these are the same as those allowed by the in-the-small level, as has been shown in Section 3.3. (4) modification of the body part of a unit: similar to the behavior modifications of types.

(1) Add a named object or a type to the export interface of a unit: this update should not cause a name conflict, and the added object or type must be defined within the unit. This modification has no impact on the existing modules.
(2) Delete a named object or a type from the import interface of a unit: This is the case where a unit can produce the same service with less resources. This implies that the unit behavior and/or the types, where the removed object or type is used, must be changed accordingly.
(3) the modifications of types and of the unit body are performed by using those modification operations defined for the modification of types.
(4) Add a unit: The added unit must be previously created, and can import the existing units. It can also, exports named objects or types which should be eventually used by other added units.

The deletion of an exported object or type, the addition of an object or type to the import interface of a given unit, and the deletion of a unit may be useful for changing the configuration of the specification.

3.4.5. Dynamic modification of a modular specification

In order to allow the construction of dynamically modifiable large specifications, we need to have access, and to be able to modify units during the specification execution. Modifications should be supported dynamically, without interrupting the processing of those parts of the specification which are not directly affected. In a similar way as our reflection based model used to support dynamic type modifications, a unit is an object of type UNIT which is defined at the meta-level. The type UNIT provides some primitive operations for unit modifications as shown in Figure 6.

The unit components are defined as variable attributes (UnitName, Import, Export, Types, and Body) within the UNIT type (see Figure 6.). The constraints defined by Definition 7, are introduced as invariants which must hold for each unit. The invariants are checked at creation time and after the unit modifications. The allowed modification primitives, are defined as operations which can be accepted by the units. The semantics of these primitive operations is specified in RMondel within the *Behavior* clause of the UNIT type.

```
type UNIT = OBJECT with
   UnitName          :  string;
   Import            : set [UsedUnit];
   Export            : set [NamedObject];
   Types             : set [TypeDef];
   Body              : var [statement];
Invariant
   { the constraints (1) to (6) of Definition 7 above, are specified here as invariants that must
      hold after a unit modifications}
Operation
   DelImp(Unit);              {drop a unit from the import list}
   AddExp(NamedObj);          { add the "NamedObj" object to the export interface}
*  Type update: for type modifications see Figure 4.
   AddStat(statement);        {add a statement to the unit body}
Behavior
   {we specify here the semantics, in terms of RMondel statements, of the above operations}
Endtype Unit
```

Figure 6. The type UNIT specification.

4. Conclusions

We have studied dynamic modifications within an object-oriented language that is particularly suitable for distributed systems modeling and specification. 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 presented a generic model with two levels of specification modification. This model allows for the evolution of large specifications at both the type and the module levels. We have shown how such a model is supported by the RMondel reflective language. We have gained more flexibility for the modification of large specifications by considering that both types (classes) and units (modules) are objects, and by defining the modification operations at a meta-level. In order to maintain the consistency of a specification after its modification, we have introduced a set of constraints at both levels. The allowed modifications proposed in our approach are restricted to those which extend a given specification. However, performing any kind of modifications remain a difficult problem. It is interesting to note that a set of modifications may be consistent when performed together, whereas each single update realized independently may yield an inconsistent specification. Therefore the use of the transaction mechanism is very important.

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. A prototype of the RMondel interpreter is implemented in Mondel. Our approach gives an interesting framework based on a formal approach, for the development of corresponding CASE tools, especially for specifications prototyping and maintenance. 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
This research was supported by a grant from the Canadian Institute for Telecommunications Research (CITR) under the NCE program of the Government of Canada. The authors gratefully acknowledge the insightful comments of the referee.

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.

[Blum 87] E. K. Blum, H. Ehrig and F. Parisi-Presicce, *Algebraic specification of modules and their basic interconnections*, Journal of Computing and System Sciences, V.34, pp.293-339, 1987.

[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.

[Brin 89] E. Brinksma, *Specification Modules in LOTOS*, FORTE'89, pp.137-156.

[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 92a] 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.

[Erra 92d] M. Erradi, G. v. Bochmann and I. A. Hamid, *Type Evolution in a Reflective Object-Oriented Language*, submitted.

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

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

[Khen 92] F. Khendek and G. v. Bochmann, *Incremental Construction of LOTOS Specifications with Internal Structure*, submitted for publication.

[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.

[Parn 72] D. L. Parnas, *A technique for software module specification*, CACM, Vol. 15, No.5, pp.330-336, 1972.

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

[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, 1987.

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

[Tich 82] W. F. Tichy, *A data model for programming support environments and its application*, In Automated Tools for Information System Design and development, Eds. H. J. Schneider and A. I. Wasserman, Amsterdam: North-Holland, 1982, pp. 31-48.

[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.

[Webe 86] H. Weber and H. Ehrig, *Specification of modular systems*, IEEE Trans. on Soft. Eng. V. SE-12, N. 7, July 1986, pp.784-798.

[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.