A Framework for Dynamic Evolution of Object-Oriented Specifications

Mohammed Erradi, Gregor v. Bochmann, and Rachida Dssouli

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

Email: {erradi, bochmann, dssouli} @iro.umontreal.ca

Abstract

The evolution of specifications is necessary to accommodate the evolution of requirements and design decisions during the software development and maintenance process. We are concerned here with formal description techniques that allow the development of executable specifications, especially executable object-oriented specifications of distributed systems. In this paper, we propose a two-level model for the evolution of large object-oriented specifications. The first level deals with the dynamic modification of types (classes), while the second level deals with the modification of modules. To allow for dynamic modification of types and modules, we have developed a reflection based technique using meta-objects where the modification operations are defined. In our approach we have defined a set of structural and behavioral constraints to ensure the specification consistency after its modification at both levels.

Key words: Distributed systems evolution, type modification, object-oriented specifications, modules compatibility, reflection, dynamic change.

1. Introduction and motivations

Most of the software development cost is spent in the maintenance phase [1]. Specification modifications are most costly because they imply modifications in the other phases. Therefore, software design practice should include criteria for maintainability [2], such as: design software with maintainability in mind, develop design criteria for achieving maintainability, and provide a change management strategy. We believe that maintenance costs can be reduced when formal methods are used at the specification level, and that facilities for modification are provided at that level. This can provide a systematic way for change propagation from the specification level to the implementation level, and make it easier to check system consistency.

The object oriented approach is known by its flexibility for system construction, and allows to cope with the problems related to software development. 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 [3], that 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, our language Mondel has been used for the specification of problems related to network management [4] and other distributed applications [5].

When specifications are large, their manipulation, understanding, and maintenance become difficult. However, the availability of modules is of great practical use for the production of structured specifications that are easier to manipulate, understand, analyze and maintain. Specification modularity is essential, and permits the use of composition to form a specification from reusable and independent modules. Therefore, the use of well defined module interfaces allows for the validation of module interconnections.

To achieve our goal that is the construction of dynamically modifiable specifications, and having the above criteria in mind, in this paper we propose a two-level generic model for managing large specifications evolution. This model consists of the *in-the-small* and the *in-the-large* levels. We will present each of these levels and describe various uses that are made of these levels to aid the evolution process. 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. Some properties of distributed systems such as blocking should be considered for preserving consistency. In our model, the consistency requirements are addressed at the *in-the-small* and at the *in-the-large* levels.

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

0-8186-2980-0/92 $03.00 © 1992 IEEE

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 needed to maintain consistency at the module boundaries. In Section 3, after an overview of Mondel and RMondel (a reflective version of Mondel) we will show how the two levels of modification, introduced by the generic model, are supported in the RMondel language. In addition, we will show how the structural and the behavioral consistencies are supported at both levels. In Section 4 we discuss related works. Conclusions are drawn in Section 5.

2. A generic model at two levels

Specifications may be too large, and therefore very complex and difficult to understand. Therefore, we assume in our approach that we have modular specifications. This notion of modularity provides the module concept as a structuring concept for large specifications. Based on that concept, modules will be interconnected to form new modules. The entire specification is also a module. Fig.2.1 gives a global view of the two levels of specification evolution. 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.

2.1. *In-the-small* level

For object-oriented systems 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. At the *in-the-small* level we consider that 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; that is the lower level node is a specialization of the higher level node. The inheritance structure of object classes (found in most object-oriented languages) is a useful concept for the structuring of complex specifications and programs. It also plays an important role for the issues of software reusability and extendibility. In this section we will enumerate the allowed modifications at the *in-the-small* level, and we discuss the consistency requirements at this level.

2.1.1. Class evolution: Software developers or database designers working with an object oriented system are frequently led to modify existing class definitions so that they suit their needs. In the area of object oriented databases, schema modifications have been extensively studied in the recent literature [6], [7], [8], and [9]. 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. A major concern in designing a methodology for class modification is how to bring existing objects in line with a modified class definition.

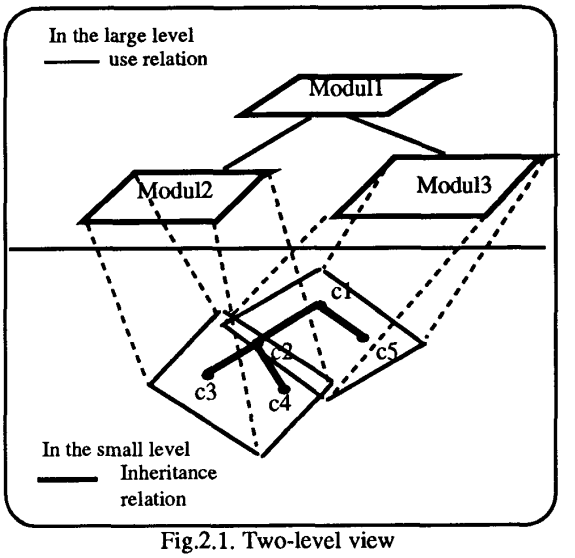

Fig.2.1. Two-level view

The class updates are classified into three categories [6]: (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 lattice.
 (i) Modifications to an instance variable of a class.
  - Add/Drop an instance variable A to/from a class T.
  - Change the class T of an instance variable A.
 (ii) Modifications to a method of a class.
  - Add/Drop the method m to/from the class T.
  -Change the signature S of the method m.
(2) Modifications to an edge of the lattice.
 (i) Make a class T a superclass of class S.
 (ii) Delete a parent S (superclass) of the class T.
(3) Modifications to a node of the lattice structure.
 (i) Add a new class T.
 (ii) Delete an existing class T.

2.1.2. Kinds of consistency: At the *in-the-small* level three kinds of consistencies must be addressed: structural consistency, semantical consistency, and instance-of relationship consistency.

-Structural consistency: It ensures that the structure of the specification (class hierarchy) is maintained according to the inheritance relation. This is widely investigated in object-oriented databases where some invariants are used to define the consistency requirements of the class hierarchy [6].

-Semantical consistency: While most existing approaches [6], [7], and [9] 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 [8] goes a long way toward preserving behavior. In effect, their methodology implements class modification by the use of versions and exception handling mechanisms. However, we are exploring solutions to class modification that do not require versioning. As we are interested in distributed systems, we believe that additional constraints such as blocking must be addressed.

-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. For instance the addition of an attribute within a class involves changes to the existing instances of this class. In the case of behavior change, object behaviors must conform to their behavior before the change.

2.2. *In-the-large* level

The importance of decomposing large specifications into modules is widely recognized within the software engineering community [10], [11], [12]. Modular systems are interconnections of modules with matching import and export interfaces [13], [14], e.g., the imported modules' export interfaces must match with the importing module's import interface. However, the precise manner in which modules should be combined, refined, modified, and organized during the evolutionary development of a system is not well understood.

2.2.1. Modules and their interconnections: At the *in-the-large* level we consider that a large specification consists of a hierarchy of interconnected modules. A node in the hierarchy represents a module, and an edge between a pair of nodes means that the upper level module employs the module of the level below. A module consist of three parts: an export interface, an import interface, and a module body. For the sake of simplicity, we do not consider parameterized modules.

(1) 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 protect some of the resources that exists internally from their use from outside the module.

(2) The *import interface* contains reference to one or a number of other modules. However, modules may not import each other cyclically.

(3) 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 class and object definitions, which do not belong to any other part of the module.

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. Therefore, a set of fundamental operations on module specifications is developed in [13].

-The *composition* of two modules M1 and M2 connects the import interface of M1 with the export interface of M2. The composite module (M1 *comp* M2) will have the same import interface as M2, the same export interface as M1, while the body of (M1 *comp* M2) is given by the union of the body parts in M1 and M2.

-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. This construction is important to build up modules step by step, adding more and more operations.

-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. For instance the export interface of (M1 U M2) is the disjoint union of the export interface of M1 and the export interface of M2.

2.2.2. Module modifications and consistency: Since each module forms a small, rather 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 modules constituent parts. We classify module modifications into the following categories:

(1) Modification of the export interface: Adding or removing a named object or a class to/from the export interface of the module.
(2) Modification of the import interface: Adding 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 can 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.

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, e.g., 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 version of a module should provide the behavior that is required by the system (i.e., the extended version should provide at least what the original module provides).

These two kinds of consistency must be addressed at two levels, i.e., 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., a resource 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.

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 a reflective object-oriented specification language, suitable for the specification and modeling of distributed systems. It provides facilities for building dynamically modifiable specifications [15]. 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 the *in-the-large* level as supported in RMondel.

3.1. Mondel overview

We have developed Mondel an object-oriented specification language [3] 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 distributed applications. 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 (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.

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, and modify the specification during run time. For this purpose, reflection is a promising choice. Recently, in object-oriented languages, reflection has gained wider attention. A language is called reflective if it uses the same structures to represent data and programs. The original model of reflection was proposed in [16] following Smith's earlier work [17], where a meta-object is associated with each object in the system to represent information about the implementation and the interpretation of the object.

3.2. RMondel facilities

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

In RMondel we distinguish two main features: Structural reflection (SR) and behavioral reflection (BR). The most important aspect of *SR*, is that each object is an instance of a type, and types are objects of a meta-type called *TYPE*. Another aspect of *SR* is that the RMondel statements and expressions are objects. The structure of RMondel is supported by an instantiation and an inheritance graphs [18]. The instantiation graph represents the instance-of relationship, and the inheritance graph represents the subtype-of relationship. The objects *TYPE* and *OBJECT* are the respective roots of these two graphs.

Beside the structural reflection, the behavioral reflection (*BR*) must be represented. Therefore, an interpreter object (i.e., behavioral meta-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 which in its turn may have its own interpreter etc.. leading to an infinite tower of meta-interpreters. Specialized interpreters can be defined for monitoring the behavior of objects, or for dynamically modifying their behaviors. More details on the RMondel definition, and the specification of *OBJECT, TYPE, INTERPRETER*, and other kernel objects, are given in [19].

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* level 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 latter conforms to the former. The modification of S can be done through the modification of its types.

In RMondel specifications, which mainly describe distributed applications, objects' dynamic behaviors are of extreme importance. Therefore, our interpretation of type modifications takes into account the dynamic behavior of objects. Then the type modifications involve not only the type structures but the dynamic behavior of objects as well. 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 at the type level and at the specification level. 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. Consistency at the type level: 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 the set of attributes and Op_t is the set of operations. B_t 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 [20]). 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 [21]. 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 [22].

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. $B_{t'}$ may perform any trace of actions that B_t may perform ($B_{t'}$ 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* [20] and *Eiffel* [23]. However, there are other important aspects to inheritance related to the dynamic behavior of objects [24], including constraints on the results of operations, the ordering of operation execution, and the possibilities of blocking [25]. 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'. In order to provide the facilities for the dynamic modification of object types, and to ensure type consistency, we deduce from the previous definitions a set of invariants [26]. These invariants check the type interfaces compatibility and the behaviors extension, respectively.

The constraints introduced in Definition 4 allow to ensure that an object type can be extended to get an object type which inherits from the former. Recall that types are objects in RMondel. Therefore, our strategy for type modification allows the modification of types without changing the type object identity. This implies that the whole specification remain structurally consistent i.e., we do not need to recompile the whole specification. This assertion can be proved according to two situations, which are assignment and parameter passing. However, the *conforms-to* relation do not ensure that the whole obtained

specification remains behaviorally consistent (i.e., the specification after modifications is not necessarily an extension of the initial specification). An example which illustrate this situation is given in [26].

3.3.2. Structure and behavior modifications

In the following we give a classification of type modifications that are supported in RMondel. As we are concerned by the incremental approach for specification evolution, and in comparison with the classification of the class modifications in ORION [6], we will consider only those type modifications that lead to new types which conform to old ones. For structure modifications we distinguish the following:

- Add an attribute A to a type T
- Change the type T of an attribute A
- Add the operation O to the type T
- Change the signature S of the operation O
- Make a type S a supertype of type T
- Add a new type T

In RMondel, types are objects (i.e., types are instances of the *TYPE* object which is defined at a meta-level). The *TYPE* object provides the primitive operations for type modifications, and holds invariants which define the static semantic rules of the language (e.g., all attribute and operation names of a type, whether explicitly defined or inherited, must be distinct). These invariants must be satisfied by each type and its related types in the type lattice.

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 definition, accordingly. The possibilities of behavior definition 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 original behavior. An algorithm for systematic construction of object behavior extensions is given in [27].

3.3.3. Instance-of relationship consistency: While types evolve, their existing instances must be changed to remain in line with their types. Therefore, two problems should be addressed: when and how objects can be converted, accordingly. We consider that the modifications of type objects are performed within a transaction. This ensures that no conversion is done until the whole modifications have been completed. A transaction is constructed based on the modification operations which consist of several successive modification of an object type. We assume that each object can be active, passive, or locked, and that all the objects involved in a type modification transaction must be in their passive state. Within a transaction, the type (and its subtypes) to be modified and its existing objects will be locked if they reach their passive state. A locking protocol [19] is used to ensure that the objects behave according to their types, and to maintain the specification in a consistent state.

3.4. *In-the-large* level in RMondel

A modular language has to suffice 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: In RMondel, a unit consist 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 visible. This implies that exported objects and 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: Like a flat RMondel specification, a unit contains 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.

3.4.2. Structural consistency: An important issue in large specification developments is interface control, to establish and maintain consistent interfaces between the numerous components. The unit construct is defined in accordance with a set of constraints that must hold in order to have a structurally correct specification configuration. A configuration is a combination of two or more units by means of the composition and/or the union operations

which are defined in Section 2.2. A specification configuration corresponds to an internal node in the hierarchy.

We write a specification configuration as:
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 C={C1,C2,...,Cn} is well-formed if it satisfies the following conditions:
(1) Every type and every object that are exported by C are exported by some Ci.
$(EO(C) \cup ET(C)) \subseteq \cup_i (EO(Ci) \cup ET(Ci))$ for i=1,.., n.
(2) C imports those types and objects imported by all Ci except for types and objects already exported by some other component of C.
$IO(C) \cup IT(C) \supseteq (\cup_i (IO(Ci) \cup IT(Ci))) - (\cup_i (EO(Ci) \cup ET(Ci)))$ for i=1,.., n.
(3) C does not export and import the same types and objects
$(IO(C) \cup IT(C)) \cap (EO(C) \cup ET(C)) = \emptyset$
(4) No type or object is exported by more than one component. $(EO(Ci) \cup ET(Ci)) \cap (EO(Cj) \cup ET(Cj)) = \emptyset$ for all Ci, Cj $\in$ C, i $\neq$ j.
(5) All C_i (for i =1,.., n) are well-formed ❑

3.4.3. The behavioral consistency: In the following, we define the constraints that must hold to maintain the specification behavioral consistency after the unit modifications. This must preserve some behavioral constraints as the *extends* relation of Definition 4.

Definition 6 [28]: 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:
$(EO(U2) \cup ET(U2)) \supseteq (EO(U1) \cup ET(U1))$ and
$(IO(U2) \cup IT(U2)) \subseteq (IO(U1) \cup IT(U1))$ ❑

This definition is based only on imported and exported object 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 as defined among object types in Definition 4, and consider the dynamic behavior of the units. Therefore, we define the *UpWardConform* relation as follows:
Notation: t1 <: t2 means that the object type t1 *conforms-to* the object type t2.

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 exported object by U1.
$\forall$ O1 $\in$ EO(U1), $\exists$ O2 $\in$ EO(U2) such that
type (O2) <: type (O1))
(3) The type of an imported object in U1, *conforms-to* the type of an imported object in U2.
$\forall$ O2 $\in$ IO(U2), $\exists$ O1 $\in$ IO(U1) such that
type (O1) <: type (O2)
(4) Every exported type by U2, *conforms-to* a type exported by U1.
$\forall$ t1 $\in$ ET(U1), $\exists$ t2 $\in$ ET(U2) such that (t2 <: t1)
(5) Every imported type in U1, *conforms-to* a type imported by U2.
$\forall$ t2 $\in$ IT(U2), $\exists$ t1 $\in$ IT(U1) such that (t1 <: 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 allow only those unit updates that lead to an extension of the original unit, according to Definition 7. We distinguish the following categories of modifications:
(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 type 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) Modification of types and unit bodies is performed by using those modification operations defined for the modification of types at the "in the small level".
(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.

Note that we allow only those modifications which preserve the constraints of Definition 7. 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. We believe that modifications should be supported dynamically, without interrupting the processing of those part of the specification which are not

directly affected. Therefore, in a similar way as our reflection based model used to support dynamic type modifications [26], we consider that 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.

Fig.3.1. shows a possible RMondel specification of the UNIT type. The unit components are defined as variable attributes (UnitName, Import, Export, Types, and Body). 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 (units are objects). 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 of Definition 7 are specified  as invariants}
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 [26].
    AddStat(statement);   {add a statement to the unit body}
Behavior
  {we specify here the semantics of the above operations}
Endtype Unit
```

Fig.3.1. The type UNIT specification.

4. Related works

Our work adapts and extends some of the concepts introduced by different researchers. The problem of maintaining the integrity of an evolving configuration has been addressed recently using module interconnection languages [29]. The approach reported in [29], allows that one component of a modular software system can be substituted for another provided that the specification of the new component is an upward compatible extension of the specification for the original component. Such approach introduces a separate language, the module interconnection language, for configuration descriptions. In our approach we use one language for both writing the system components as well as their interconnections, rather than introducing a special language for recording module dependencies. The work reported in [29], is limited to sequential systems and does not provide any facility for dynamic evolution of distributed systems.

Kramer and Magee have addressed the problem of dynamic change management for distributed systems [30]. Their approach focuses mainly on changes specified in terms of the system structure and provides a separate language for changing specifications. Unlike their approach, which concentrates 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. Their approach deals with configuration change, however, our approach deals also with the way the component can be changed

In the area of object oriented databases, class modifications have been extensively studied in the recent literature [6], [7], [8], and [9]. 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 object behaviors and we consider some properties of distributed systems such as blocking.

5. Conclusions

We have studied dynamic modifications within an object-oriented language that is particularly suitable for distributed systems modeling and specification. Dynamic module and type modification is a difficult problem. Using module hierarchies and performing associated structural consistency checks is well researched and practiced for non object-oriented systems. Its application for dynamic modification of object-oriented systems is equally fruitful. We believe that object-oriented systems in conjunction with reflection, allow us to approach problems that conventional systems have not been able to address in a uniform way. A generic model with two levels for dynamic modification of distributed systems specifications is presented. 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. Therefore, we gain 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. In order to validate the actual effectiveness of our approach, we are implementing an interpreter of RMondel using Mondel environment. Our approach gives an interesting framework based on a formal approach, for the development of corresponding CASE tools.

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. We wish to thank M. Faci for his stylistic comments. We gratefully acknowledge the comments of the reviewers.

References

[1] Schneidewind, N. F., *The state of software maintenance*, IEEE Trans. on Soft. Eng. Vol. SE-13, No.3, pp. 303-310, March 1987.

[2] McClure, C. L., *Managing software development and maintenance.*, New York: Van Nostrand, 1981.

[3] Bochmann, G. v., Barbeau, M., Erradi, M., Lecomte, L., Mondain-Monval, P. and Williams, N., *Mondel: An Object-Oriented Specification Language*, 90.

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

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

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

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

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

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

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

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

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

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

[14] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B. and Nelson, G., *Modula 3 Report*, DEC 1988.

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

[16] Maes, P., *Issues in Computational Reflection*, Meta-Level Architectures and Reflection, eds. P.Maes and D. Nardi, 1986.

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

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

[19] Erradi, M., Bochmann, G. v. and Hamid, I. A., *Type Evolution in a Reflective Object-Oriented Language*, Publication departementale #827 DIRO, University of Montreal, July 1992.

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

[21] Erradi, M. and Bochmann, G. v., *Semantics and definition of RMondel A Reflective Object-Oriented Language*, Internal report DIRO, University of Montreal 92.

[22] Brinksma, E. and Scollo, G., *Lotos specifications, their implementations and their tests*, PSTV VI (IFIP Workshop, Montreal, 1986), North Holland Publ.

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

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

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

[26] Erradi, M., Bochmann, G. v. and Dssouli, R., *Dynamic Modification of Types for Evolving Specifications*, submitted.

[27] Erradi, M., Khendek, F., Dssouli, R. and Bochmann, G. v., *Dynamic Extension of Object-Oriented Distributed System Specifications*, To appear in Proceeding of the International Workshop on Feature Interactions in Telecommunications Software Systems, Florida, Dec. 3-4, 1992.

[28] Tichy, W. F., *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, the Netherlands: North-Holland, 1982, pp. 31-48.

[29] Narayanaswamy, K. and Scacchi, W., *Maintaining Configurations of Evolving Software Systems*, IEEE Transactions on Software Engineering, Vol. SE-13, No.3, March 1987, pp.324-334.

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