Vol. 35 No. 7 Transactions of Information Processing Society of Japan July 1994

Regular Paper

Type Evolution in a Reflective Object-Oriented Language

ISSAM A. HAMID[†] and SETSUO OHSUGA[††]

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

1. Introduction and Motivations

The object oriented approach is known by its flexibility for system construction. This is partly due to the inheritance property which permits class reuse and incremental construction of systems. We have developed a new object-oriented specification language, called *Mondel*[6] that has important concepts as an executable specification language to be applied in the area of distributed systems. The motivations behind Mondel are: (a) writing system descriptions at the specification and design level, (b) supporting concurrency as required for distributed systems, (c) supporting persistent objects and transaction facilities, and (d) supporting the object concept. Presently, our language Mondel has been used for the development of executable specifications of problems related to network management[6] and OSI directory system.[5]

In a wide spectrum of distributed applications, software systems require modifications to accommodate evolutionary changes, particularly for systems with a long expected lifetime (like database systems). In general, evolutionary changes are difficult to accommodate because they cannot be predicated at the time the system is designed. Therefore, systems should be sufficiently flexible to permit arbitrary, incremental changes. The evolution of such systems is necessary to accommodate the evolution of requirements and design decisions during the software development and maintenance process. We believe that software system modification are most of the time incremental.[27] They consist of adding new functionalities or extending some existing ones. In this paper, we consider that an executable specification of a system is an implementation model of such system. Therefore, we examine a method of supporting extensions of distributed system specifications in the context of the object-oriented specification language Mondel. A difficult and important issue is that of making modification dynamically, without interrupting the processing of those parts of the specification which are not directly affected. There has been little suggestion as to how such dynamic modification should be specified managed and controlled. For instance, Database systems are mostly real world applications, and they are distributed system in nature. Some part of those systems cannot stop running because they are related to real world application.[15] The model can be changed or evolved. This change should be implemented as well. So we cannot stop the system to allow the new modification to be inserted. The new added specification on the system should be checked

---

[†] Department of Information Design, Tohoku University of Art & Design

[††] Research Center for Advanced Science & Technology, University of Tokyo

while the system is running. Therefore the executable specification should be run continuously to allow the system to be evolved with its environment. Using reflection in Mondel language, we can have a mechanism to allow the dynamic modification of the executable specification while the system is running. Therefore the executable specification should be run continuously to allow the system to be evolved with its environment. Using reflection in Mondel language, we can have a mechanism to allow the dynamic modification of the executable specification while the system is running. It is necessary to provide facilities for controlling changes in order to preserve the specification consistency. The specification consistency concerns both, behavior and structure. We use a transaction based mechanism and a locking protocol to ensure that the specification remains consistent after its modification. We describe formal and systematic approach for extending distributed system behavior specifications. The semantics of the behavior specifications are modeled by Labeled Transition Systems. The approach consists of building a new behavior specification S_{new} by adding a *new* behavior described by S_{added} to a behavior specification S_{old} and avoiding the feature interaction problem. Providing certain sufficient conditions, the newly derived behavior specification S_{new} extends S_{old} and S_{added}. To achieve our goal that is the construction of dynamically modifiable specifications we define a reflective object oriented language called RMondel (Reflective Mondel) which is based on the Mondel language. Recently, reflection[31)] has gained wider attention as indicated by the first and second workshops on reflection and meta-level architectures in object-oriented programming[30)] held in conjunction with OOPSLAs '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 Ref. 20) following Smith's earlier work,[25)] where a meta-object is associated with each object in the system to represent information about the implementation and the interpretation of the object. Meta objects for object-based concurrent systems must represent not only an object's methods and state but also, object communication procedures.[6)] 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 look up are described at the meta-level.

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

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

The paper is organized as follows: In section 2 we introduce RMondel, a reflective version of Mondel, and show how dynamic type modifications are supported. In Sec. 3 we give the type definitions and their relationships used in our RMondel. To preserve types consistency a set of invariant is defined in Sec. 4. In Sec. 5 we define a set of primitives which are used to modify the structure and behavior of type definitions. A transaction mechanism and a locking protocol which ensure the consistency of the whole specification are given in Sec. 6. Sec. 7 discusses related works. Conclusions are given in Sec. 8.

2. Configuration of Reflection in RMondel

So far, we have introduced primitives for structural and behavioral modifications and invariant for preserving types consistency. To allow for the construction of dynamically modifiable specifications, we need to access and to modify types during execution-time.[9] In this section, we give an overview of Mondel and its characteristics. Then we discuss reflection as supported in *RMondel* and show how dynamic type modifications and dynamic checking of type consistency are implemented using RMondel facilities.

2.1 Mondel overview

We have developed Mondel: An object-oriented specification language[6] with certain particular features, such as multiple inheritance, type checking, rendezvous communication between objects, the possibility of concurrent activities performed by a single object, object persistence and the concept of transaction. Mondel is particularly, suitable for modeling and specifying applications in distributed systems.

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

-Attributes; An object type includes a certain number of named attributes. Such that, each object instance of that type have fixed references to other object instances, one for each attribute. An attribute can be declared non-visible or internal.

-Operations; they define the functions and procedures that the object can accept during execution. The operations are externally visible and represent actions that can be invoked by other objects. An object have internal procedures which can be called by other objects on a rendezvous basis. Each operation declaration (signature) is defined by its name, a list of parameters and a result, if any. The types of input and output objects required by an operation are often called the signature of an operation.

-Typing; *Mondel* supports strong type checking based on the declared object types. The type consistency of the effective parameters of operation invocations and object instantiation can be checked by a compiler. The language also, includes an operator to dynamically determine the type conformance relation between type parameters.

-Behavior, which provides certain details as constraints on the order of execution of operations by the object, and also determines properties of the possible returned results of these operations by the object. Users observe object "behavior" in terms of the operations that may be applied to objects and the results of such operations. These operations comprise an object's *interface* with its users. Changing the state of an object means changing the collection of object types that apply to that object. Therefore, to change an object's state, the applicable object type must be identified. Object *behavior*, therefore, is defined by its state-changing process with respect to its structure. We call "behavior of an object type" the information which known about the execution of operations and the initialization of a newly create object instance.

-Inheritance, Types can be related to each other by means of the inheritance relation. *Mondel* allows multiple inheritance where a given type may inherit from several supertypes as long as the inherited properties are without contradictions. Class inheritance implements the generalization hierarchy by making it possible for one class to share the data structure and operations of another class.

A Mondel specification corresponds to a type lattice. In such a lattice, nodes represent types, and edges represent the inheritance relation. The execution of a specification consists of a set of objects that run in parallel. Each object has its individual behavior which provides certain details as constraints on the order of the execution of operations by the object, and determines properties of the possible returned results of these operations. Among the actions related to the execution of an operation, the object may also, invoke operations on other objects. Basically, communication between objects is synchronous, based on rendezvous mechanism. The basic statement of Mondel is the operation call, which is syntactically represented by the "!" operator. For instance, in the statement *m! InsertCoin* (see line 27 of Fig. 2), "m" designates the called objects, and *InsertCoin* is an operation defined within the type of "m" (i. e., the type *Machine*).

Mondel has a formal semantics which associates a meaning to the valid language sentences. Such a semantics was defined based on the operational approach. In this approach an abstract machine simulates the real computer role. The meaning of a specification is expressed in terms of actions made by the abstract machine. We have particularly, applied the technique of Plotkin[24] where state/transition systems are taken as machine models. the Mondel formal semantics was the basis for the verification of Mondel specifications,[6] and has been used for the construction of an interpreter.[32]

(a) Object structure and its semantics

In *Mondel*, the structure of an object is considered as a triple ⟨Id, Bind, Stat⟩. Where Id is a unique identifier generated for the object, Bind is a binding which associates actual values to formal attributes, and Stat represents the statement being executed by the object. As an example, we consider the following object type definition :

type A = object **with**
    x: B;
behavior
    new C(x);
endtype A

Object instances of type "A" have an attribute "x" of type "B". Their behavior consists of the single statement "new" which creates an instance of type "C". A particular instance of type "A" could be represented as the following triple: ⟨idl, {id2/x}, new C(x)⟩, where idl is a unique identifier generated for this object, {id2/x} is a binding which associates the object reference id2 to the formal attribute "x", and new(x) represents the statement being executed by the object. When "new C(x)" is executed the system state is extended with a new object instance of type "C".

The execution of a Mondel specification is modelled by a set of states with transitions between them, starting from a specific initial state. The transitions are given by a transition relation denoted "→". The transition relation is defined inductively by a set of inference rules. The dynamic semantics of Mondel has therefore, the two aspects of representation of states and definition of inference rules. A state is a set of active objects (as shown in the above example). In general the semantics of an active object is obtained by combining together several inference rules. As an example we show the rules related to object creation. At the object level we have the following rule :-

if
obj = ⟨Id, T, Bind, new TypeName ($\mathrm{Id}_1, \cdots \mathrm{Id}_n$)⟩ (1)
fattr (TypeName) = $\mathrm{AttrName}_1, \cdots, \mathrm{AttrName}_n$ (2)
Bind = {$\mathrm{Id}_1/\mathrm{AttrName}_1, \cdots, \mathrm{Id}_n/\mathrm{AttrName}_n$} (3)
NewId = newsym (4)
Obj' = ⟨Id, T, Bind, NewId⟩ (5)
then
→(*Obj*, new (TypeName, Bind, NewId), *Obj'*) (6)

This rule says, on line 6, that the object *Obj* can execute the action new (TypeName, Bind, NewId) and transforme into *Obj'* if the preconditions of lines 1 to 5 are fulfilled. Line 1 defines the structure, i. e., the state, that *Obj* must have. On line 2 the function fattr yields for every type name its ordered list of formal attributes. This list is used on line 3 to define the binding of the created object for which a new identifier is generated on line 4 by the newsym function. The *newsym* function is not in pure first order logic and is introduced for the sake of simplicity. It can be easily be translated into pure logic by keeping track, from one state to another, of identifiers in use. At last, line 5 defines the successor state *Obj'* of the creator.

The full definition of object creation also, requires a higher level inference rule defining the transition from a set objects to a new set of

objects which will contain the new instance :-
if

$$S = A + \{Obj\} \qquad (1)$$

$$\rightarrow (\mathrm{Obj},\ \mathrm{new}\ (\mathrm{TypeName},\ \mathrm{Bind},\ \mathrm{NewId}),\ Obj' \qquad (2)$$

$$S' = A + \{Obj',\ \langle \mathrm{NewId},\ \mathrm{TypeName},\ \mathrm{Bind},\ getinitbeh\ (\mathrm{TypeName}) \rangle \qquad (3)$$

then

$$\rightarrow (S,\ \mathrm{new}\ (\mathrm{TypeName},\ \mathrm{Bind},\ \mathrm{NewId}),\ S') \qquad (4)$$

This rule defines on line 4, a state change of the set of objects S on action new (TypeName, Bind, NewId) with successor state S′. Line 1 describes the structure of S which is a set of objects A plus an object *Obj*. On line 2 we say that *Obj* is the creator of the new object. Here we apply the object level inference rule defined above. Line 3 defines the structure of the successor object set. The function *getinitbeh* yields for every type name the corresponding initial behavior.

(b) Conformance relation and inheritance

Mondel allows for a form of multiple inheritance where a given type may inherit from several supertypes, as long as the inherited properties are without conflict. The intention is that an instance of a subtype can be used in any specification context where an instance of one of its supertypes can be used. Henceforth we call this relation "conformance" : a subtype conforms to the more general supertype. There are different aspects of object behavior that are relevant to the conformance relation such as the following.[5] (1) Set of Values : The set of objects belonging to a subtype is included in the set of objects belonging to its supertype. (2) Attributes : A conforming object has (at least) all the attributes defined for the more general object type. The attributes may be more specialized (conforming). (3) Operation signatures : A conforming object has (at least) all the operations defined for the more general object type, where the operation result must be conforming and the input parameters must be inversely conforming. (4) Behavior of operations : The effect of the operations of the refined type satisfy the requirements specified for the more general object type.

(c) Object persistence and transactions

A *Mondel* specification has certain aspects related to databases ; in particular, persistent objects can be accessed through the equivalent of database queries. The concept of transactions (atomic operations) is also supported to provide distributed, fail-safe implementations of *Mondel* specifications by using standard fault recovery procedures developed for distributed databases. Certain operations of objects may be declared as "atomic". If several atomic operations are executed in parallel, they are assumed to be executed in a serializable manner. The concept of invariants, associated with the atomic operations concept, allow the specification of conditions, such as conventional database integrity rules.

2.1.1 Example of a *Mondel* specification

The following example will be used throughout the paper. Let us consider a vending machine which receives a coin and delivers candies to its user, as shown in **Fig. 1**. We distinguish two types of objects : the type *Machine* and the type *User*, as shown in the Mondel specification of **Fig. 2**. The relation between the *Machine* and the *User* is expressed by the fact that the user knows the machine. Such a relation is modeled by the attribute "m" defined in the User type.

The behavior of the *User* type is specified within the behavior clause as shown in lines 23 to 33 of Fig. 2. The user is initially in a *Thinking* state, and when he decides to buy a candy he inserts a coin. After the coin has been accepted, the user enters the *GetCandy* state. Then the user pushes the machine's button to get a candy. Once the candy is delivered, the user

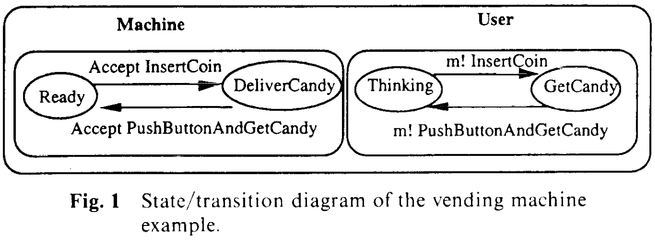


Fig. 1 State/transition diagram of the vending machine example.

```
0 unit spec =

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

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

20 endtype Machine
```

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

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

34 endtype User

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

39 endunit spec
```

Fig. 2 Mondel Specification of the vending machine example.

enters the *Thinking* state again. The behavior of the *Machine* type is specified as shown in lines 5 to 19 of Fig. 2. The machine is initially in the *Ready* state, ready to accept a coin. Once a coin is inserted, the machine accepts the coin and then enters the *DeliverCandy* state. After the user has pushed the button of the machine, the latter delivers a candy and becomes Ready to accept another coin.

Note that object operations model the occurrences of events. The behavior of the vending machine system is defined as the composition of interacting objects (i. e., *Machine* and *User* objects, see lines 35 to 38 of Fig. 2). The object's behaviors are specified using a state oriented style.[28] The internal state of an object is modeled as a Mondel procedure.

2.2 Reflection in RMondel

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

RMondel (*Reflective Mondel*), is a reflective object-oriented concurrent language, where each object is an instance of a type, and the type of an object can be considered as its meta-object from the structural reflection (SR) point of viev From the computational reflection (CR) poir of view, each object has an associated interprete object defined as an instance of the INTERPRETER type or of one of its subtypes. To define a reflective architecture, one has to defin the nature of meta-objects and their structur and behavior. In addition, one has to show hov the handling of object communications and operations look up are described at the meta-level.[14] In RMondel, types are used for structural description (i. e., for the definition of the structure of objects and of applicable operations), and interpreters are used for the behavioral description (i. e., how the rendezvous communication is interpreted and the operations are applied) of their associated objects, called *referents*. Types are considered to be structural meta-objects, while interpreters are behavioral meta-objects. Types and interpreters are instances of the kernel types TYPE and INTERPRETER respectively. This approach shows many advantages.

-Types are objects, instances of the type TYPE which is defined at a meta-level.

-Operations for type modifications can be defined at the meta-level (i. e., within *TYPE*).

-An object behavior may be modified according to the modifications of its type.

-An object behavior can be monitored by its interpreter.

- -New communication strategies can be defined by creating subtypes or different versions of INTERPRETER.
- -Communication between the baselevel and the meta-level is possible.
- -The definitions of the structure and the behavior of objects are dynamically accessible.

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

2.2.1 Structural reflection

Like in databases, to integrate data schema and meta-schema,[20] we need to design a meta-schema which describes the class of acceptable schemas for the data model in use. In *RMondel* we have to consider also the behavior of objects. In *RMondel*, the design of a meta-schema corresponds to the design of a meta-model based on *Mondel* type definitions.

In Mondel objects with the same properties are grouped within the same type. In *RMondel*, a type and its components such as attributes, operations, and behavior, are considered as objects which are instances of specific types, called *kernel types*, as shown in **Fig. 3**. This allows for the access of the different components of a type, and give more flexibility in order to dynamically modify types. The structure of RMondel is supported by instantiation and inheritance graphs. The instantiation graph represents the *instance-of* relationship, and the inheritance graph represents the *conforms-to* relationship. The objects TYPE (called CLASS in other language) and OBJECT are the respective roots of these two graphs.[7]

The *structural reflection* (*SR*) is supported in a similar manner as in *ObjVlisp*.[11] The *ObjVLisp* model addresses mainly the structural aspects of reflection, whereas BR is not adequately supported. In contrast to the *ObjVlisp*, we consider not only the structural aspect (i. e., SR) of reflection, concerning the language structure of object classes and instances, but we also, address interactively the reflectivity for object attributes, operations (methods) and behaviors. This will make the implementation related to dynamic adaptation more flexible and tractable in comparison to *ObjVlisp*. The implementation of dynamic behavior is possible when we consider both the SR and BR in the language. For example, a user may want to tune language system to adapt to a specific application to improve its efficiency. This case is difficult in *ObjVlisp*, because the behavior of the run-time kernel cannot be changed by the user. Tying the behavioral aspects of modeling to the structural aspects (as in our language) is very important. Without this modeling integration, object-oriented implementation would not be possible. Also, for the BR, a meta object called *interpreter* object, is associated with each object at creation time. A meta-object deals with the computational aspect of its associated objects. The SR concerns static structure of objects in the system, while BR is directed toward dynamic computational process. SR involves the semantic domain elements such as objects and message environments which constitute structural aspects. BR concerns with semantic domain

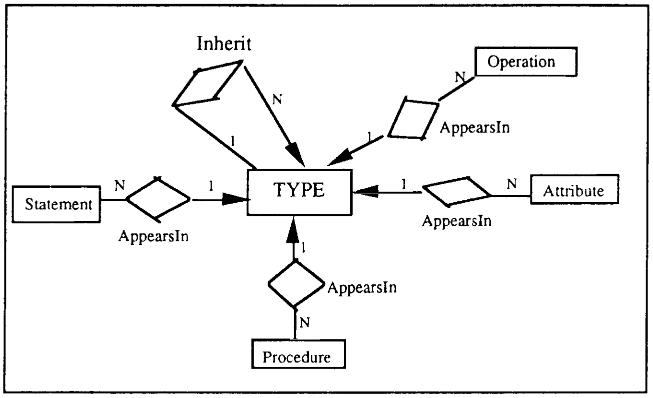


Fig. 3 A meta-model of Mondel type definition.

elements such as environment, continuation, and object table which appear as the arguments to the valuation functions. The most important aspect of SR in RMondel, is that each object is an instance of a type, and types are objects. For each object we introduce the attribute *MyType* that links the object to its sype, as shown in **Fig. 4**. Another aspect of SR is that the RMondel statements and expressions are objects. For instance, one can specify the operation call, and accept statements as instances of the *Opcall* and *Accept* types, respectively, as shown in **Fig. 5**. Each statement object accepts the *Eval* operation, that implements the semantics rule associated with such a statement.

(a) The structure of RMondel objects

In RMondel, the structure of an object is considered as a finite set of attributes represented

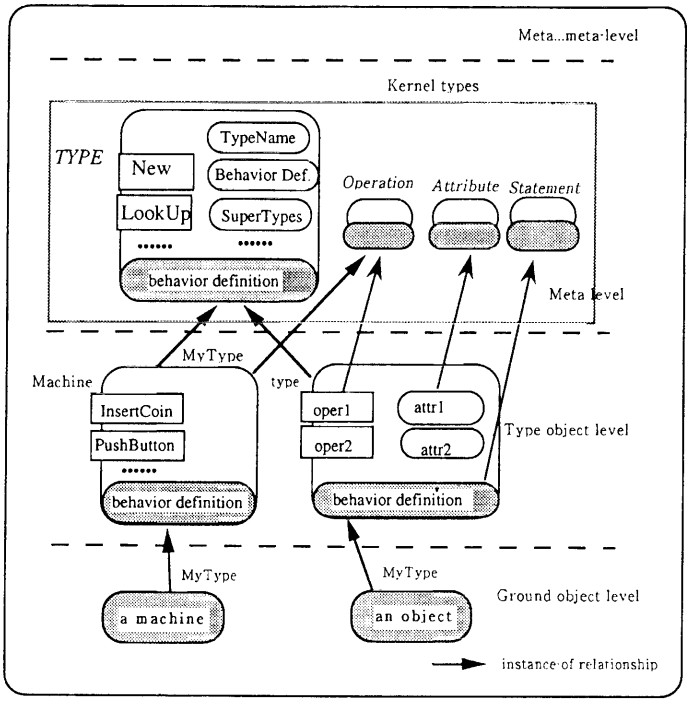


Fig. 4 Structural reflection basis.

```
type Statement = OBJECT endtype Statement
type Expression = OBJECT endtype Expression
{Details on the definitions of the statement and expression objects are given in [Erra92]
 type Accept = Statement with
      OpName        :string;
       AcceptBody    :Statement;
  operation
       Eval;
  behavior
        { semantics rule of the accept statement }
 endtype  Accept
 type OpCall = Statement with
        Callee      : Expression; {restricted to object identifier, for simplicity}.
         OpName   : string;
   operation
        Eval;
  behavior
        {semantics rule of the operation call statement }
  endtype OpCall
```

Fig. 5 Example of the specification of a subset of RMondel statements.

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

(1) *The initial attributes are* :

(i) the unique object identifier, named *ObjecId*, is commonly known as *self*. Such identifier is generated automatically. For the sake of readability we will consider that object identifiers, for types are constructed by means of the type name prefixed by "Id" (e. g., the type *Machine* of Fig. 2 is identified by IdMachine).

(ii) the identifier if the type the object, named *MyType* which is the type of the created object.

(iii) the identifier of the object behavior, named *Behavior*, which represents the initial behavior of the created object. The value of the Behavior attribute can change as the execution of the object's behavior evolves. It is important to mention that an object's behavior is also an object.

(2) The *effective attributes* are separately created by the *NewAttr* operation defined in the *OBJECT* type which defines the common behavior of each object in the system. These two kinds of attributes, initial and effective attributes, constitute the explicit definition of an object in the following form :

$$O=\langle(\mathrm{ObjectId},\ \mathrm{Ido}),\ (\mathrm{MyType},\ \mathrm{Id}_{type}),\ (\mathrm{Behavior},\ \mathrm{Id}_{beh}),\ \{\cdots(\mathrm{Name}_{attri},\ \mathrm{Id}_{attri}),\ \cdots\}\rangle$$

where Ido, Id_{type} and Id_{beh} designate the initial attributes of the object O. The set $\{\cdots, (\mathrm{Name}_{attri}, \mathrm{Id}_{attri}), \cdots\}$ contains the effective attributes of O.

(b) The kernel type specifications

In the following subsections we will describe the components of *RMondel* structure. These components consist of the kernel types shown in Fig. 4.

(b. 1) The type TYPE

The type TYPE initially exists in the system as an instance of itself. It defines the behavior for types, e. g., the type Machine of Fig. 2 is created as an instance of *TYPE*. It holds the effective attributes *TypeName, BehaviorDef, SuperType* etc··· which refer to the name of a type, the behavior defined in such a type, its parent, etc. **Figure 6** gives a definition of the type *TYPE*.

```
type TYPE = OBJECT with
    TypeName :string;
    BehaviorDef :var[Statement];
    SuperType :TYPE; { for simplicity, we consider here single inheritance only }
    Attributes :set[AttributeDef];
    Operations :set[Operation];
    Procedures :set[Procedure];
•••
operation
    { the operation New creates an object according to RMondel sructure }
    New :OBJECT;
    <: (t: TYPE) : Boolean; { checks if a type t conforms-to with self }
    { the operation LookUp checks if the operation "OpName" is defined for an
     object's type or for one of its supertypes; then returns the associated statements.
     The <: relation is the clousre of the inheritance relation.}
    LookUp (OpName : string) : Statement;
behavior
    LookUpProc; •••
where
    Procedure LookUpProc =
      Accept LookUp do
        ifexist  Op:Operation suchthat
                 Operations.contains(Op) and Op.OpName = OpName
        then { let AcceptBody be the object of types Statement that is associated
               with the operation defined by Op. }
               return (AcceptBody);
        else { recurse on supertypes }
             return (SuperType! LookUp(OpName));
      end;
    endproc LookUpProc
  •••
  endtype TYPE
```

Fig. 6 The definition of *TYPE*.

The *LookUp* and *New* operations are defined within *TYPE* as shown in Fig. 6. The *LookUp* operation is used to find an operation in the called object's type or in its supertypes. The *New* operation allows for object (i. e., types or instances) creation.

We assume that an instance of the *TYPE* type object exist initially, it has as its type itself. The structure of the *TYPE* object is:

```
<(ObjectId, IdTYPE),
 (MyType, IdTYPE),
 (Interpreter, nil), (Stat, δ),
 {(TypeName, "TYPE"), (Stat, IdS1)}> ;
```

Where IdS1 is an object reference to the specified behavior within the *TYPE* type definition, among others, we find the ***New*** operation definition. δ corresponds to the initial behavior of the *TYPE* object. The object Stat refers to the behavior definition within the type *TYPE* (see Fig. 6). The *TYPE* object is useful for the creation of type definitions as well as their instances. Moreover, user meta-types can be defined by inheriting from the *TYPE* object, and adding other features as shown in the following simple example:

Consider the following type definition:

```
type PART=OBJECT with
    Color : string ;
behavior
    COLOR! print ;
endtype Part
```

This type definition is represented by an object of type *TYPE*, an object type *Attribute*, and an object type *Statement* as follows:

```
<(ObjectId, IdPART),
 (MyType, IdTYPE),
 (Interpreter, nil), (Stat, δ),
 {(TypeName, "PART"), (Stat, IdS1)}> ;
<(ObjectId, IdATTR),
 (MyType, IdATTRIBUTE),
 (Interpreter, nil), (State, nil),
 {(AttrName, "COLOR"),
 (AttrTYPE, Idstring),
 (AppearsIn, IdPART)}> ;
Stat=<(ObjectId, IdStat),
     (MyType, IdOpCall),
     (Interpreter, nil), (Stat, nil),
     {(Callee, "COLOR"),
      (OpName, "print"),
      (Binding, nil)}> ;
```

The object Stat ; refers to the behavior definition within the type PART. If we assume an object P1 which is an instance of type "PART" with a "red" color, P1 can be represented as:

```
<(ObjectId, IdP1),
 (MyType, IdPART),
 (Interpreter, nil),
 {(COLOR, Idred), (Stat, IdStat)}> ;
```

Idred refers to the string "red". When IdP1 starts its execution, the name "COLOR" in the behavior will be substituted by the actual value Idred.

(b.2) The type *OBJECT*

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

```
type OBJECT=with
        ObjectId         : integer unique ;
        MyType           : TYPE ;
        Behavior         : var[Statement];
        operation New : OBJECT ;
                    NewAttr (A : Attribute) ;
                    {A is the added attribute}
        invariant
          {the constraints which must hold to
           maintain the system in a consistent
           state. These constraints define the
           consistency requirements of the
           type lattice corresponds to the static
           semantics rules checked by the
           Mondel compiler.}
        behavior
          {specification of the semantics rule
           of NewAttr}
endtype OBJECT
```

2.2.2 *Behavioral reflection*

Beside the *structural reflection* of our model, the *behavioral reflection* (BR) must be represented. There are two main aspects in the *BR* model. First, it must describe the behavior of objects using other objects. Second, it must provide a method invocation protocol that will allow the user to intervene on the current execution in order to modify the course of events, i. e., reflect. Therefore, we have associated an interpreter object (i. e., behavioral meta-object) to each object as shown in **Fig. 7**. An interpreter object deals with the computational aspect of its associated object called *referent*. Interpreter objects are defined as instances of the type *INTERPRETER*. An interpreter object may have its own interpreter object ; thus the number of interpreter objects is virtually infinite. Specialized interpreters can be defined for monitoring the behavior of objects or for dynamically modifying their behaviors.

A possible specification of the type *INTERPRETER*, where the incoming calls of its referent can be recorded, is given in **Fig. 8**. Such specification shows how the rendezvous communication between objects is interpreted. One can define new ways of handling the object communication by specifying subtypes or different versions of the *INTERPRETER* type. We can see from the *INTERPRETER* definition that the accept statement is an object. To avoid an infinite loop of operation calls, the basic operation call ("!") is used to define the semantics of

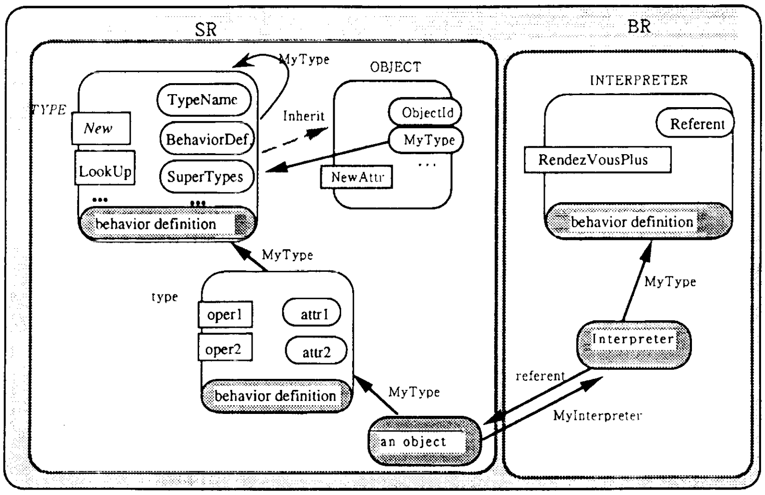


Fig. 7 BR and SR in RMondel.

```
type INTERPRETER = OBJECT with
  referent :OBJECT;
  NbCall :var[integer]; •••
 operation { the operation RendezVousPlus interprets object communication }
  RendezVousPlus (OpC: OpCall); •••
 behavior
  RendezVousProc; •••
 where
  Procedure RendezVousProc =
     Accept RendezVousPlus do { We can record the number of incoming calls of
                                the referent object }
           IncrementNbCall;
           { let AcceptBody be the object of type Statement that corresponds to
             the called operation; then evaluate such a statement. }
           define AcceptBody = OpC.Callee.MyType ! LookUp(Op.C.OpName) in
              { create a Context object which contains the callee attributes and
                parameters binding }
              AcceptBody ! Eval (Context);
            end;
          end;
        RendezVousProc;
    endproc RendezVousProc
    Procedure IncrementNbCall =
          { Increment NbCall }
    endproc IncrementNbCall
endtype INTERPRETER
```

Fig. 8 The definition of INTERPRETER.

the accept statement.

To deal with interpreter objects, we add a specific attribute, called *MyInterpreter*, to each object structure. This leads to the modification of the type *OBJECT* as shown in **Fig. 9**. The added attribute is optional because not all objects need to have a specific interpreter. If the value of the attribute is nil then a default interpreter is invoked. The *OBJECT* specification becomes:

type OBJECT = **with**

......

MyInterpreter: INTERPRETER **opt**; {**opt** stands for optional}

......

endtype OBJECT

Then the structure of an *RMondel* object; o becomes:

$$o = \langle (\mathrm{ObjectId}, \mathrm{Ido}), (\mathit{MyType}, \mathrm{Id}_{\mathrm{type}}), (\mathit{MyInterpreter}, \mathrm{Id}_{\mathrm{interpreter}}), (\mathrm{Behavior}, \mathrm{Id}_{\mathrm{beh}}), \{\cdots, (\mathrm{Name}_{\mathrm{attri}}, \mathrm{Id}_{\mathrm{attri}}), \cdots\}\rangle$$

where the attribute $(\mathit{MyInterpreter}, \mathrm{Id}_{\mathrm{interpreter}})$ refers to the interpreter $\mathrm{Id}_{\mathrm{interpreter}}$ of the object o.

2.2.3 A simple RMondel interpreter

The definition of reflection in RMondel allows the access to the definition of an object's structure (i. e., its type) and to the language statements which are objects. To access the context of the execution of an object's behavior, we use the context objects (instances of the context type) which contain the binding of attribute names and local variables with values. A context object is created to bind the actual arguments of the operation call with the operation parameters. Local variables and attribute are specified in the context object. The current context is passed as an argument to the *Eval* operation of a statement object of the body of called operation (e. g., see lines 13 to 15 of Fig. 8). For instance, if the statement object is an attribute reference, the identifier of the referred attribute is retrieved from the context. Context objects are managed based on the conventional stack approach used for the processing environments of procedural programming languages.

Let us describe a simple **R**Mondel **I**nterpreter (RMI) which coordinates the execution of the objects of a given RMondel specification. The RMI has a global view of the existing objects, i. e., the kernel objects and the objects of the specification. According to *RMondel* semantics,

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

Fig. 9 The OBJECT definition with the attribute *MyInterPreter*.

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

Fig. 10 A simple *RMondel* interpreter.

which is based on state/transition systems, objects are executed in parallel. Therefore, the RMI selects an object and tries to fire a transition within the object's behavior. The most important transitions are operation calls. If the called object has an associated interpreter, (i. e., the value of its attribute *MyInterpreter* is not nil) then the evaluation of the operation call is delegated to this interpreter (see **Fig. 10**). A search for the called operation is performed, within the type of the called object, by mean of the *LookUp* operation. The *LookUp* operation is defined at the meta-level within the type *TYPE*. For *LookUp* operation definition see lines 10 to 22 of Fig. 6.

3. The Specification Components and Their Relationships

Before addressing the problem of specification modifications, an understanding of the specification model, its components and their relationships is required. A specification is defined as a type lattice system where nodes represent types and edges represent inheritance relation. Our interpretation of inheritance considers both the structure and the behavior aspects. In object oriented approaches; Subtype is a specialized object type. All the properties that apply to an object type apply to its subtypes. A subtype has additional properties. For example, all the properties of Person apply to Man and Woman. The extension of a subtype is a subset of a given object type's set. (A more general object type is called a supertype.) Supertype is a generalized object type. An object type with properties more general than its subtypes. All the instances of an object type are instances of its supertype, but not the other way around.

In the following, we give the definitions of types and the inheritance relationship as supported in our model, assuming all types are of type lattice system.

Definition 1:

A type t consists of an interface I_t and a behavior B_t, $t=<I_t, B_t>$.

Then $I_t=<A_t, OP_t>$ 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. □

Users observe object behavior in terms of the operations that may be applied to objects types and the results of such operations. These operations comprise an object's interface with its users. A class specifies the data structure of types for each of its types and the operations that are used when accessing the types. The structure of types, which corresponds to types' interface, are used as a basis for the traditional inheritance scheme of object-oriented languages. Thus, a type has a 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[4]). Based on this aspect of inheritance we give a recursive definition of the *structural consistency* relation as follows.

Definition 2:

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

1. $A_{t'} \supseteq A_t$. t′ has at least all attributes of t.
2. For each operation o in OP_t there is a corresponding operation o′ in Opt′ 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 (see part a of subsec. 2.1), the behavior of objects is formally specified by a translation to labeled transition systems. 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 corresponding to the *extension* relation defined for Lotos specification.[10]

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 is 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). □

The allowance defined in definition 3, means that if $B_{t'}$ is acceptable only if it does not introduce a cycle in the inheritance hierarchy. (This acceptance is checked by the invarients given in Sec. 4.) 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 Emerland[4] and Eiffel.[21] However, there are other important aspects to inheritance related to the dynamic behavior of objects,[1] including constraints on the results of operations, the ordering of operations execution, and the possibilities of blocking.[5] 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** (i. e., <:) a type $t = <I_t, B_t>$ if: t′ is *structurally consistent* with t and $B_{t'}$ *extends* B_t. □

We denote by "<:" the *conforms-to* relation introduced in Definition 4.

Corollary: The "<:" relation is a partial order, i. e., reflexive, transitive, and antisymmetric.

Proof: Evident. (this is because the "<:" relation is the closure of the inheritance relation.)

$\forall t \in T$, where t is a type belongs to set of all types T, then $t = t$ (i. e., reflexive),

$\forall t_1, t_2, t_3$ if $t_1 = t_2$ and $t_2 = t_3$ then $t_1 = t_3$ (i. e., transitive),

$\forall t_1, t_2$ if $t_1 = t_2$ then $t_2 \neq t_1$ (i. e. antisymmetric).

The proof of this corollary becomes more evident when we understand Sec. 4, and the explanation given after Definition 5.

Definition 5:

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

If type t′ conforms-to type t then we say that t′ is a subtype of t and t is a supertype of t′. Types can be related to each other by means of the conformance relation. Currently, we restrict the conformance relation to inheritance, that is a type t1 conforms to a type t2 (noted t1<: t2) if and only if there exist an instance of type INHERT such that: i. Sub=t1 and i. Sup=t2. The INHERT type models the inheritance relation as follows:

```
  type INHERT=OBJECT with
        Sub, Sup: TYPE;
  endtype INHERT
```

The fact that a type t1 inherit from a type t2 is represented by an instance of the INHERT type where Sub=t1 and Sup=t2. The semantics of the operation "<:", defined within the *TYPE* type, is given below based on the procedure InheritFrom (t1, t2) which checks whether t1 inherits from t2. Such a procedure may be defined recursively as follows:

```
procedure InheritFrom (t1, t2: TYPE):
boolean =
  ifexist I: INHERT suchthat
          I. Sub=t1 and I. Sup=t2
  then return true
  else ifexist t3: TYPE suchthat
          InhertFrom (t1, t3) and InheritFrom
          (t3, t2)
       then return true
       else return false
endproc InheritFrom
```

This works as validation rules for subtyping types. As well as every type object is inherited from other types, then the definitions 2-4 are all terminated after they find for certain types what is their inherited types in the lattice.

4. Preserving Consistency

To maintain the *conforms-to* relation, we deduce from the definitions of Sec. 3, a set of invariant which must be satisfied by each type and its related types in the lattice. These invariant which will be used for dynamic type checking after type updates. This set of invariants define mainly the consistency requirements of the type lattice, which corresponds to the static semantic rules of *Mondel*. For instance, we identify the following rules:

- The type lattice is seen a directed acyclic graph, where the root is the OBJECT type, and each node (a type) is reachable from the root. Each type in the lattice has a unique name.
- All attribute and operation names of a type, whether defined or inherited are distinct.
- A type inherits all attributes and operations from each of its supertypes.

In order to keep a system in a consistent state, these invariants must be satisfied by each type and its related types in the inheritance graph.

The invariants on type definitions are specified by assertions within the specification of the *Type* object (as shown in Fig. 13).

Using the definition of a type, as given in Definition 1, we introduce a formal definition of the invariants as follows:

(1) *Type hierarchy invariant*:: the type hierarchy (i. e. lattice) is a directed acyclic graph, where the root is a system-defined type called *OBJECT*, and each node (i. e, a type) is reachable form the root. Each type in the hierarchy has a unique name.

$\forall$ t1, t2$\in$T with $t1 <: t2$, then
$\exists$t3$\in$T such that $t2 <: t3$ and $t3 <: t1$.

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

$\forall$ attr1, attr2$\in A_t$ such that attr1 = (a1 : t1) and attr2 = (a2 : t2)
{attri = (ai : ti) means: ai is the attribute name and ti is the attribute type}
then $a1 = a2 \Rightarrow attr1 = attr2$

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

$\forall$ op1 = <opname1, [p1 : t1, ⋯, pi : ti, ⋯ pn : tn], [r1]>,
op2 = <opname2, [p1′ : t1′, ⋯, pi′ : ti′, ⋯ pn′ : tn′], [r2]> $\in$ Opt, [] means optional
then $opname1 = opname2 \Rightarrow op1 = op2$

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

$\forall i \in O$, $\exists t \in T$ such that i is an instance of t.

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

(5.1) For attributes:
$\forall$t1, t2$\in$T with $t1 <: t2$,
$\forall$attri = (ai : ti) $\in A_{t1}$,
$\exists$attrj = (aj : tj) $\in A_{t2}$ such that ai = aj and $ti <: tj$

(5.2) For operations:
$\forall$ opi = <opname$_i$, [p1 : t1, ⋯, pl : tl, ⋯pn : tn], [r1]> $\in$Opt1
$\exists$ opj = <opname$_j$, [p1′ : t1′, ⋯pl′ : tl′, ⋯pn′ : tn′], [r2]> $\in$Opt2
such that opname$_i$ = opname$_j$
op$_i$ and op$_j$ have the same name.
(the co-variant rule holds)
$r1 <: r2$ the result of opi conforms to the result of opj.
and (the contravariant rule holds)
$pl = pl'$ for $l = 1, \cdots, n$ parameter names are the same.
$tl' <: tl$ for $l = 1, \cdots, n$ parameter types are inversely conforming.

These invariants must be preserved by each type and its related types in the lattice. They are checked when an object is created and after type updates. This check works as validation rules for subtyping types updates. Also, because each type is inherited types in the lattice, this conforms that the definitions 2-4 are terminated because, each type is inherited from other subtypes.

5. Primitive Operations for Type Modifications

In the following we give a classification of type modifications that are supported in our language, and we provide the description of their semantics. In comparison with the classification of the class modifications in ORION[2] which considers structural modifications only. Our approach considers those type modifications, both structure and behavior, that lead to new types which conform to old ones.

5.1 Structure modifications

We have three categories of updates: (1) updates to the contents of a node in type lattice (e. g., addition of an attribute/operation), (2) updates to an edge in the type lattice (e. g., addition of an edge), and (3) other updates of the type lattice (e. g., add a node). These updates may be performed on a type T as long as the obtained type T' is structurally consistent with T (according to Definition 2). The semantics of these update operations are as follows:

(a) *Add an attribute* A to a type T: This update allows the user to append an attribute definition to a given type definition. We suppose that the added attribute A causes no name conflicts in the type T or any of its subtypes. Name conflict is not addressed here, but may be avoided in a similar way as in Ref. 12).

(b) *Change the type T1 of an attribute A by the type T*: We assume that the type *T* of an attribute A can only be specialized to a type *T1*. This update is allowed only if *T1 conforms-to* T.

(c) *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.

(d) *Change the signature S of the operation O*:

(i) Change the type *T* of the parameter p in S: This update allows the change of the type *T* of the parameter p in S, to become *T′*. This update is allowed only if *T* conforms to *T′*.

(ii) Change the type *T* of the result, if any, of the operation O: This update allows the change of the type *T* of the result to become of type *T′*. This update is allowed only if *T′* conforms to *T*.

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

(f) *Add a new type T*: If no supertype of *T* is specified, then the type *OBJECT* (i. e., the root of the type hierarchy) is the default supertype of *T*. If a supertype is specified, then all attribute and operations from the supertype are inherited by *T*. The name of the added type *T* must not be used by an already defined type. The specified supertype of *T* must have been previously defined.

5.1.1 Repercussions of type changes on existing instances

Transforming all instances whose type has been modified seems like the most natural approach for dealing with change propagation. In this subsection we analyze the impact of each type modification on existing instances. In some cases instances need to be converted so that their structure matches the description of the type they belong to.

(a) *Add an attribute to a type*: This leads to the logical addition of the attribute to all instances of the type and to those of the subtypes inheriting the attribute. A nil value is given by default to the added attribute. The nil value is an instance of the most specialized type *NONE* which is compatible with any type.

(b) *Change the type T1 of an attribute A defined within a type T*: We have seen that the type *T1* of an attribute A can only be changed into *T2* which conforms to *T1*. Therefore, instances of the type *T* are not affected by this change because the operations accepted by the instances of *T1* remain accepted by those of *T2*.

(c) *Add the operation O to the type T and/or change of the operation signature*: There is no impact on the existing instances of the type T. Operations appear only in the type.

(d) *Make a type S a supertype of type T*: This update involves the addition of the inherited attributes to T′s instances and to T′s subtypes' instances. In this case, the impact on the instances of T is the same as all changes above.

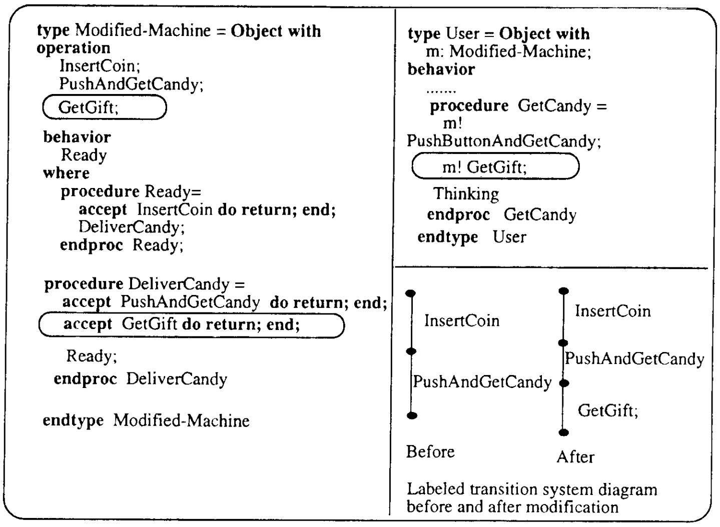
(e) *Add a new type T*: There is no impact because T is new and has no existing instances.

5.2 Behavior modifications

For the modification of the behavior of types, we consider those modifications which extend the existing behavior to meet new requirements according to the constraints of Definition 3. This is similar to the notion of incremental specifications proposed for a subset of basic LOTOS language.[13] However, Lotos was not concerned with an object-oriented approach. In comparison with Mondel, Lotos processes are anonymous while Mondel objects are uniquely identified. Lotos process are statically linked using the parallel composition operators. However, a Mondel object obtain dynamically the identifier of another object, by means of an attribute or an operation parameter, in order to communicate with it.

The behavior of objects is 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. Therefore, changes to the type interface may lead, in most cases to changes in the behavior. We distinguish two kinds of behavior modifications: those related to the interface changes, and those not affecting the interface.

Behavior changes related to interface



```
type Modified-Machine = Object with
operation
   InsertCoin;
   PushAndGetCandy;
 ( GetGift; )
 behavior
   Ready
 where
   procedure Ready=
      accept InsertCoin do return; end;
      DeliverCandy;
   endproc Ready;

 procedure DeliverCandy =
    accept PushAndGetCandy do return; end;
 ( accept GetGift do return; end; )
    Ready;
  endproc DeliverCandy

 endtype Modified-Machine
```

```
type User = Object with
  m: Modified-Machine;
behavior
   .......
   procedure GetCandy =
     m!
PushButtonAndGetCandy;
 ( m! GetGift; )
    Thinking
  endproc GetCandy
 endtype User
```

Fig. 11 Behavior modification by sequential composition.

changes:
The possibilities of behavior modifications, presented here, are based on some basic language constructs which are the sequential, choice, and parallel composition operators. Note that, we consider only finite behaviors for the behavior modifications presented in this section. This restriction simplifies the checking of the *extension* relation. We will describe the allowed behavior modifications using the vending machine example.

(a) *Sequential composition*: Suppose that we want to modify the vending machine specification to deliver a gift to its user after each purchase. We modify the type interface of the type *Machine* by adding the *GetGift* operation. The code associated to the *GetGift* operation is added in the Machine's behavior definition in sequence with the existing behavior (as shown in **Fig. 11**). This modification is allowed according to the *extension* relation of Definition 3. Therefore, any object of the modified type *Machine* accepts the *PushAndGetCandy* operation as any object of the initial type *Machine*. An object of the modified type *Machine* does not block where an object of the initial type *Machine* does not.

An existing behavior may involve into a new behavior by appending another behavior to the existing one. The consistency of this modification is guaranteed by the *conforms-to* relation given in Definition 4.

(b) *Choice composition*: It has been shown that the choice operator does not guarantee subtyping,[22] because non-determinism may be introduced. For instance, the combination of recursion and choice may lead to a violation of the second property of Definition 3. Also, if two behaviors are combined by the choice operator, and these two behaviors have non-empty insertion of their initial actions, then non-determinism is introduced. In the following we distinguish two cases:

-Deterministic case: We can introduce the behavior associated with an added operation using the choice composition operator. Suppose that we want to modify the vending machine of Fig. 2, in order to allow its user to buy either a candy or chocolate. The *PushAndGetChocolate* operation is introduced by mean of the choice operator as shown in (a) of **Fig. 12**. For the *extension* relation, both properties of Definition 3 are satisfied. Therefore, an object of the modified type *Machine*, accepts the same operations in the same order as any object of the initial type *Machine*. Also, the behavior of an object of the modified type *Machine*, does not block where an object of the initial type *Machine*, does not. We conclude that the behavior, defined in the modified type *Machine*, *extends* the behavior defined within the initial type *Machine*.

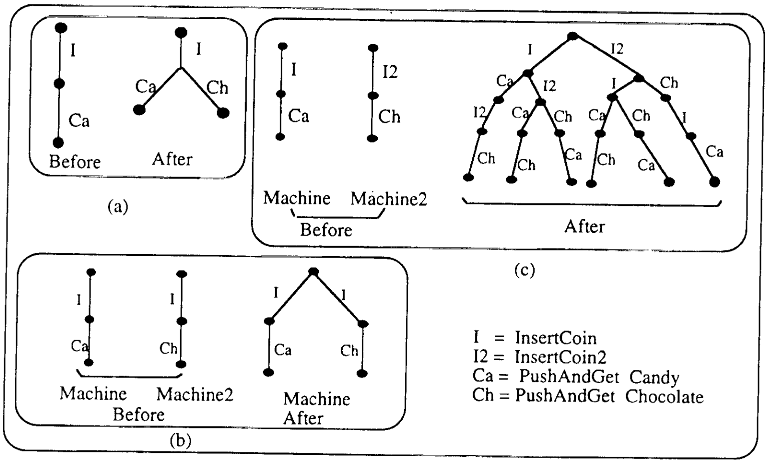


Fig. 12 Labeled transition system for behavior change.

-*Non-deterministic case*: Let us consider the type *Machine2*, defined in a similar way as the initial type *Machine*, that delivers chocolate. We will modify the initial *Machine* in order to also, provide the behavior defined by the *Machine2*. The behavior obtained as the combination of the initial *Machine* and the *Machine2* behaviors, does not satisfy the second property of the *extension* relation. This is because the behavior of the modified *Machine* introduces non-determinism as shown in (b) of Fig. 12. This non-determinism is illustrated by the existence of two branches with the same initial action (i. e., *InsertCoin* operation). The introduced non-determinism can be removed by combining the initial common actions, this lead to the same specifications as in (a) of Fig. 12.

(c) *Parallel composition*: Two types of behavior may be composed, using the parallel composition operation, to obtain a new behavior. The new behavior, whose construction is base on the pure interleaving semantics (i. e., independent parallelism), preserves the ordering of constraints of actions of the two initial types of behavior. This kind of modification is guaranteed by the *conforms-to* relation as well. In this case, we want to modify the initial machine by combining its behavior with the behavior of the second machine, using the parallel operator. The obtained machine should behave like both machines, it should deliver both candies and Chocolate. The labeled transition system diagram of the resulting machine, shown in (c) of Fig. 12, is constructed based on the pure interleaving semantics (i. e., independent parallelism) where the ordering constraints of actions must be preserved. We conclude that the behavior obtained by the parallel composition presented above (i. e., using sequential, choice and parallel operators), can be easily inferred from those given[13] for Lotos specifications. An additional operator may be provided to construct only the allowed behaviors described above. An algorithm for behaviors decomposition is given in Ref. 7)

Behavior changes not affecting the type interface:

Another aspect of type modification is performance enhancement. These modifications have no impact on the type interface of the modified type. In this case only the implementations of the operations, are modified. Therefore, the modified behavior provides the same services as the old behavior. These modifications should not lead to behaviors which block more than the old behavior, and naturally, both structural consistency and behavior extension are preserved.

5.3 Dynamic type modification and consistency checking

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

5.3.1 Primitives for dynamic type modifications

Figure 6, shows the type *TYPE* used to imple-

```
type TYPE = OBJECT with
     TypeName :string;
      BehaviorDef :var[Statement];
      SuperType :TYPE;
      Attributes :set[AttributeDef];
      Operations :set[Operation];
      Procedures :set[Procedure];
      •••
operation
      AddAttr (A: AttributeDef);
      AddOper (O: Operation);
      AddProc (P: Procedure);
      AddStat (S: Statement);
      •••
Invariant

     "Invl" {attributes must have distinct names }
            [ Forall a1, a2 : AtrributeDefinition such that;
               Attributes.contains(a1) and SuperType.Attributes.contains(a2)]
            (a1.AttrName <> a2.AttrName)
      •••
behavior
      LookUpProc; •••
       where
{ The semantics definition of the modification operations. }
      •••
endtype TYPE
```

Fig. 13 TYPE specification with invariants and modification operations.

ment two important aspects of reflection, which are instantiation and operation look up. To support dynamic type modifications in *RMondel*, we modified the type *TYPE* by adding a set of primitive operations such as AddAttr, AddOper, etc. The resulting *TYPE* is given in **Fig. 13**. Note that one can define a subtype, let say Modifiable-Type, of Type in order to hold the primitive operations. For example, let T be a type. In *RMondel*, T is a type as well as an instance of *TYPE*. Consequently, T accepts the operations defined in *TYPE*. For example, it accepts the operations *AddAttr* to add an attribute to its own attributes. Existing instances of type T are modified according to the *AddAttr* operation semantics as defined in subsection 5.1.

5.3.2 Dynamic checking for type consistency

In Section 4, we defined a set of invariant which are used to ensure the consistency of the type structure after modifications. To perform dynamic checking of type consistency, we incorporate these invariant, into RMondel, within the invariant clause of the type *TYPE*. An example of such invariant definition is shown in Fig. 13.

Let us consider the type T again. Since T is an instance of *TYPE*, the invariants defined in *TYPE* must always hold for T, especially when T is first created and after any possible modification. For example, if we attempt to add to T an attribute definition which has the same attribute name as an inherited one, the invariant *Invl* in Fig. 13, prevents the completion of this modification.

5.4 Consistency at the specification level

The modification of the structure and behavior of types must be done in such a way that no type checking errors, run-time errors, blocking, or any other uncontrollable situation may occur. Therefore, the semantics of type changes should, ensure that a type t to be modified leads to a type t′ which *conforms to* t. However, does the *conforms-to* relation guarantee the consistency of the whole specification? In the following we address the issues of the whole specification consistency, and of the dynamic checking of structural consistency and behavior extension.

Structural Aspect : The main question here is : if we replace a type definition t by t′ in some specification S, where t′ is *structurally consistent* with t, does the resulting specification S′ remain consistent with respect to S? The specification S′ is consistent with respect to S, the structural point of view, if the modification of a type interface in S does not lead to a run-time error. Therefore, the obtained specification S′ remains

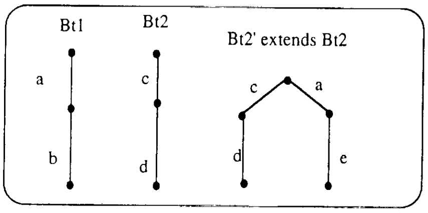


Fig. 14 Labeled transition systems of Bt1, Bt2, and Bt2′.

consistent because of the invariants discussed before, which must always hold. This assertion can be proved according to assignment and parameter passing where type checking is important.

Behavioral aspect : Similarly, if we replace t by t′ such that the behavior defined by t′ extends the behavior defined by t, does S′ extends S? The answer is in general no, as shown by the following counter example : consider the behavior of S, which is defined by the parallel composition of two behaviors Bt1 and Bt2, where Bt1 is defined by a type t1 and Bt2 is defined by a type t2. Suppose we extend Bt2 to obtain Bt2′ as shown in **Fig. 14**. Now if we compose Bt1 with Bt2′, to obtain S′, the resulting behavior blocks with respect to action a, whereas the original specification S is free of deadlock.

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

6. The Transaction Mechanism

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

6.1 Locking protocol

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 the parts of the specification, which are affected by the modifications. This protocol also ensures the mutual exclusion of concurrent transactions.

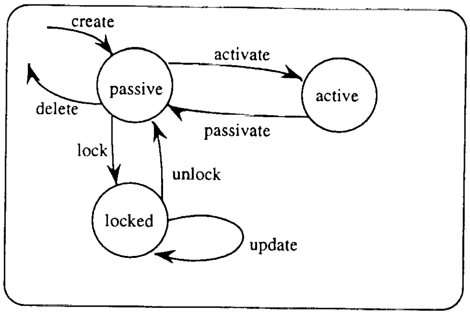


Fig. 15 Object state/transitions.

This protocol is incorporated within the transaction mechanism described in the next subsection.

According to the updates of a type T, its existing instances must be converted accordingly. When a type has to be updated, its instances must be locked until the type modifications are accomplished. If the updates do not succeed, e. g., because of invariant violation, then the type will be rolled back to its state before the updates, and the instances will be released to pursue their behavior progress. In the case where the type updates succeed, the instances will be converted accordingly, and released to continue their normal progress. Each object can be active, passive or locked as shown in **Fig. 15**. Initially, an object is in a passive state if it is not involved in a current transaction (e. g., an objects in a passive state after its creation). When the object is involved in a transaction its state becomes active (e. g., the object is asking, by means of operation calls, for other objects' services). An object in a passive state may be locked for the purpose of an update (e. g., object conversion after its type modification).

The fact that a specification is organized as a type lattice has a major impact on the locking protocol. The modification operations may involve a type and all its subtypes (e. g., if we have to add an attribute to a type, then the structure of its instances and of the instances of its subtypes has to be modified). Thus not only the instances of the modified type must be locked, but the instances of its subtypes as well. Therefore, we define a type sublattice to be a type and all its direct and indirect subtypes in the type lattice.[19] To update a type, we adapt the *x lock* mechanism[15] to be applied for a type sublattice. That is when a type has to be modified, a lock is set not only on the type itself, but also, on each of its descendant types on the type sublattice. The instances of a locked type will be locked until their type becomes unlocked. Figure 15, shows the possible states and transitions of an object with respect to modifications. Objects (i. e., either types or their instances) can be modified only when they are locked.

6.2 The transaction steps

The user formulates his requirements within a transaction which consists of type update operations. We use the concept of transaction to provide fail-safe implementations of specifications by using standard fault recovery procedures developed for database systems.[16] A transaction consists of several successive modifications of one or more types. The following steps show how the different actions (i. e., those involved in a type updates) work and lead to a consistent specification. These steps are represented through the different levels, by the heavy dotted lines in **Fig. 16**.

Step 1: *Transaction construction*: Through the interface object, the user formulates a transaction (called an atomic operation in Mondel) as an operation call, specifying his requirements (i. e., in terms of operations for type modifications). The transaction is composed of a set of primitive operation calls (i. e., redefined primitives for type modifications as shown in Fig. 13) which are defined at the meta level within *TYPE*.

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

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

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

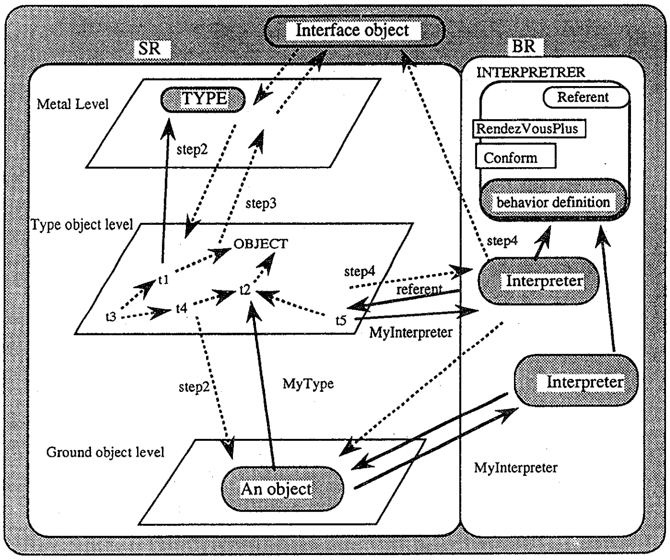
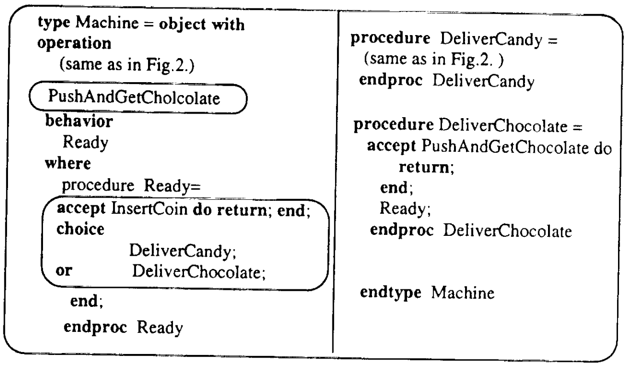


Fig. 16 Reflection-based mechanism.



```
type Machine = object with
operation
   (same as in Fig.2.)
 PushAndGetCholcolate
behavior
   Ready
where
   procedure Ready=
 accept InsertCoin do return; end;
 choice
           DeliverCandy;
 or        DeliverChocolate;
   end;
   endproc Ready
```

```
procedure DeliverCandy =
  (same as in Fig.2. )
 endproc DeliverCandy

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


endtype Machine
```

Fig. 17 Modified vending machine system of Example 1.

step 1), in order to make the specification comply with the invariant.

Step 5: *Behavioral Conformance checking*: The behavioral conformance deals with the dynamic behavior of objects as introduced in Definition 3. According to the behavior modifications, if any we need to check dynamically that the modification of the behavior of an object does not introduce new deadlocks in the overall specification. Among the existing approaches for deadlock detection (e. g., program transformation, simulation, reachability analysis) we use a dependency graph and the reachability analysis techniques widely used for the validation (e. g., deadlock detection) of communication protocols.[33),34)] A dependency graph is constructed based on the relation of dependency between types. A type t1 depends on a type t2 if the former uses one or more operations of the later. If the *extension* relation is violated, e. g., a deadlock is detected, then the system reports the inconsistencies and the type must be revised again.

Step 6: *Instances conversion*: When the type modification transaction succeeds, (i. e., the structural consistency and the behavioral conformance relations hold) then the instances (locked previously), at the ground object level, must be converted to remain conform with their modified type. The conversion of the instances according to the semantics of each type evolution primitive operation, is described in Ref. 8).

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

6.3.1 Example-1

Let us consider the vending machine example for which a specification was given in Fig. 1. Suppose now that we want to modify the initial machine to deliver candies or chocolates, instead of candies only. This imply that we have to modify the type *Machine* and the type *User*, accordingly. For this purpose we have to modify both interface and behavior defined of the initial *Machine*. In the following we will show how the modifications are performed upon the type *Machine*, according to the different steps of the mechanism described earlier. For the type *User* the modification can be done in a similar way. To the *Machine*'s interface, we add the operation *"PushAnd GetChocolate"*, and for the behavior we modify the procedure *Ready* by modifying the procedure body as shown in **Fig. 17**.

Step 1: The user formulates the atomic operation (i. e., a transaction) using RMondel statements. In **Fig. 18** we give a possible specification of an atomic operation (see line 2 of Fig. 18). The user formulates his atomic operation using the predefined kernel types (e. g., *Procedure, Statement*, etc.) to create the necessary objects (see lines 4 to 12 of Fig. 18). The type *Machine* which is an instance of *TYPE*. Among the actions of the *updatesMachine* atomic operation, we have the *AddOper* (i. e., to add an operation) call on the type *Machine*. The *AddOper* call takes the operation to be added as a parameter (see lines 12, 13 of Fig. 18).

Another change, is to modify the body of the procedure *Ready* of the initial specification, accordingly. The procedure defined in the ini-

```
1  type  TransExample = Object
     .........
   operation
2        updateMachine: atomic;
     ...........
    behavior
3     accept  updateMachine do
4  {Let Machine be the object type Machine of Fig.2,  which is an instance of TYPE }
5  { Let ProcReady be the object of type Procedure where the procedure body is a
    a statement object of type Sequence.  ProCall and Choice are predefined kernel
    types for procedure call and choice statements, repectively.  }
6      define  DeliverCand = new ProCall ("DeliverCandy");
7               DeliverChoc = new ProCall ("DeliverChocolate") in
8                define  CanOrChoc = new Choice (DelivCan, DelivChoc) in
9                     ProcReady.ProcBody.Stat2:= CanorCho;
10               end;
11      end;
12     define op = new Operation ("PushAndGetChocolate") in
13              Machine! AddOper(op);
14               define ProcChoc =  new Procedure ( .......) in
15                    Machine! AddProc (ProcChoc);
16               end;
17      end;
18      return;
19    end;
20  end;
21 endtype  TransExample
```

Fig. 18 A transaction for the type *Machine* updates for Example 1.

tial specification, is a sequential composition where the first statement is "accept InsertCoin do return ; end ;" and the second statement is the procedure call "DeliverCandy". Now we change the second statement by the new added statement of type *Choice* (i. e., "choice DeliverCandy or DeliverChocolate" as shown in lines 6 to 10 of Fig. 18). Then an instance of the predefined kernel type *Procedure*, is created to hold the "DeliverChocolate" procedure as shown in lines 14, 15 of Fig. 18.

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

Step 3: The changes are performed on the type *Machine* as specified in lines 6 to 17 of Fig. 18.

Step 4 and **Step 5**: *Structural consistency* and *Behavioral conformance checking*: At the end of the transaction, just before the return of the atomic operation (see line 18 of Fig. 18), the predefined invariant must hold for the type *Machine* after its modification. The invariant are triggered automatically to ensure the consistency of the *Machine* structure. In this stage, the SR will have a role because if the user adds certain information which does not produce the specification's needed structure, then there will be a reflection from the object type level to the meta level represented as SR. This type of reflection informs the user at the meta level which part of his/her specification should be re-modified/updated such as to make it in line with the structure needed by the specification. It is obvious from the resulting *Machine*'s specification shown in Fig. 17, that the structural consistency as defined in Sect. 4 is preserved. The addition of the operation *"PushAndGet-Chocolate"* in a choice composition as shown in Fig. 17, preserves the *behavioral conformance* requirements according to Definition 3.

Step 6: At the end of the transaction, and after both structure and behavior are checked, the existing instance of the type *Machine* has to be converted accordingly. In this example, the modification (i. e., addition of operation) has no impact on the structure of the existing instances if any. Because the added operation appears only within the type *Machine*, the instance behavior evolve dynamically when they become unlocked. Another example is given below, to show the dynamic modification.

6.3.2 Example-2

In order to allow for the construction of dynamically modifiable specifications, we need to have access, and to be able to modify type definitions during run-time. For the dynamic modification of type definitions, we need to define some primitive operations within the object *TYPE*, which allow the modification of types. To add a new operation to a type definition T, we have to call the *addOper* operation with the specification of the added operation given as parameter value. This can be written as: *T!AddOper*(*O1*), where *O1* is an object reference to the added operation. Recall that T was created as instance of *TYPE*. The invariants defined in the invariant clause, ensures that the semantics of such added operation is specified within the behavior clause. We remark that the invariants define within *TYPE* play an important role to maintain consistency between all the component of a type definition. Now after the addition of the operation *O1*, each newly created instance of T, can accept such an operation. We give below an example to support this explanation.

In **Fig. 19**, we give another supporting example of a *Mondel* specification. The described example consists of a system switch composed of unreliable pieces of equipment and a controller. Initially the system is in a working state. When a failure occurs, the system status changes to the failed state. The system remains in the failed state until the failed equipment is repaired. Initially an equipment is in a working state. When a failure occurs, a signal (operation call) is sent to the controller and the equipment enters a failed state as shown in Fig. 19. This example will be used to illustrate the dynamic modification of specifications using *RMondel*. An equipment is either in a working state or in a failed state. The RMondel specification consists of the definition of two object types (as shown in Fig. 19)

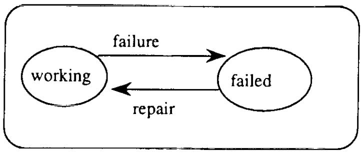


Fig. 19 Initial system specification of Example 2.

Practically, the specification of the switch system is not complete. Such system is vulnerable, because if a failure occurs in one equipment the system will be down until the equipment is repaired. To increase its system reliability, we introduce a standby equipment that will substitute the failed equipment. With such modification to the original system specification, the system can be in a protected state when a standby is available.

The introduction of this standby equipment will involve some modification to the system behavior as well as to the piece of equipment. When a failure occurs, a switching phase ensures the replacement of the failed equipment by the standby equipment. Two alternatives are possible: if the standby detects no problem, the original piece of equipment enters a failed state and the switching phase is complete. The system then moves to the unprotected state. If the standby also, detects a failure, the conclusion is that the malfunction origin is not the piece of

```
type controller = object with
 operation
  failure; repair
 behavior
  working
 where
   Procedure working =
     accept failure do
       failed;
     end;
     endproc working

 Procedure failed =
     accept repair do
       working;
     end;
   endproc failed
 endtype controller

type equip = object with
  c: controller;
 behavior
   working;
  where
    Procedure working=
      c! failure; failed;
    endproc working

    Procedure failed =
      c! repair; working;
     endproc failed
   endtype equip
```

Fig. 20 A Mondel specification of Example 2.

equipment. Then, the system moves to the breakdown state. The system requires service and may be restarted in the protected state. The system status may change from unprotected to failed if either another piece of equipment fails or the standby fails. The system remains in the failed state until either a piece of equipment or the standby is repaired. **Figure 20**, shows the state transition diagram of the new system configuration.

Let us show how the user can construct a new specification based on the existing one (shown in **Fig. 21**). The construction of the new system specification involves the addition of many objects and the renaming of other objects. For instance, the states *protected, switching*, and *breakdown*, shown in the state/transition diagram are specified as procedures within *RMondel* specification. Such procedures must be created as new objects of the *Procedure* type. The *Procedure* type is a predefined kernel object. The *Procedure* type modelizes the definition of procedures which consists of a procedure name, a list of optional parameters, and a procedure body. The procedure *working* in the initial specification (see line 7 in Fig. 21) is renamed to become the procedure unprotected as shown in line 32 of **Fig. 22**. Also, the body of the procedure *working* is replaced by a new object of the *Choice* type as shown in line 33 of Fig. 22 (*choice* is a kernel object type that represents the choice construct of *RMondel*). This new object is built out of a set of other objects that represents the statements of the different alternatives of the choice as shown in Fig. 22.

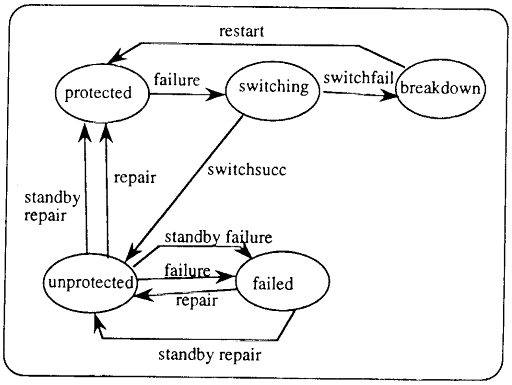


Fig. 21 New updated system specification of Example 2.

```
 1 type controller = object with
 2 s: standby;
 3 operation
 4   restart; failure; standbyfail;
 5   switchfail; switchsucc;
 6   repair; standbyrepair;
 7 behavior
 8   (*initialisation *)
 9    protected;
10    where

11    procedure breakdown =
12      accept restart do
13        return; protected;
14      end;
15    endproc breakdown

16    procedure protected =
17     accept failure do
18       s! failure; return; switching
19      end;
20    endproc protected

21   procedure switching =
22     choice
23      accept switchsucc do
24        s! switchsucc, return; unprotected;
25      end;
26    or
27  accept switcfail do
28    s! switch; return; breakdown;
29  end;
30 end;
31 endproc switching

32 procedure unprotected =
33   choice
34     accept failure do
35       return; failed;
36     end;
37   or
38     accept standbyfail do
39       return; failed;
40     end;
41   or
42     accept repair do
43       s! repair; return;
44       protected;
45     end;
46   or
47     accetp standbyrepair do
48       return; protected
49     end;
50   end;
51 endproc unprotected

52 procedure failed =
53   choice
54     accept repair do
55       return; unprotected;
56     end;
57   or
58     accept standbyrepair do
59       return; unprotected;
60     end;
61   end;
62 endproc failed
63 endtype controller
```

Fig. 22 New updated Modified specification of Fig. 21.

To maintain the consistency of the specification construction, a set of constraints defined as invariants within *TYPE* object specification must be satisfied. We distinguish three categories of invariants (as given in Sec. 4) : general invariants, type definitions invariants, and the inheritance invariants. For instance, the "accept" switchsucc "statement in line 23 of Fig. 22, cannot be validated by *RMondel* system while the *switchsucc* operation is not defined within the controller type as shown in line 5 of Fig. 22. For this purpose, the user has to add the *switchsucc* operation definition by using the *addoper* defined within the *TYPE* object. Because the controller type is an instance of *TYPE*, then it can accept the *ADDOper* to add the *switchsucc* to the set of defined operations. To complete the construction of the new specification of the switch system, the user must create and add other objects that represent states (procedures) and transitions (operation calls and acceptance). Such objects are added using the same mechanism described above. It is important to note that all the modification must be realized as an atomic operation (transaction) to ensure a valid construction of the new specification. This validity is governed by the set of predefined invariants as mentioned before. After the construction of the new specification, the user can invoke a verifier to check the correctness of the added object behavior. This concerns the verification of certain properties such as termination, the absence of deadlocks, and the specific properties of the specified problem. We have a verifier developed for the verification of Mondel specification.[3] This verfier has been considered to be adapted for *RMondel* specifications.

7. Related Works

In the area of object oriented databases, class modifications have been extensively, studied in the recent literature.[2],[12],[23],[26] 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 consistency and behavioral extension. For the behavioral extension we deal mainly with the behavior of objects and we consider some properties of distributed systems such as blocking. Moreover, we use reflection which provides a flexible and uniform environment for dynamic type specification as well as their modifications using specific meta-operation and meta-object. Another work on class modification using meta-operation is that of Ref. 18), where a lazy evaluation method of schema evolution which minimize the amount of object manipulation is proposed. However, Ref. 18) does not address the issues of behavioral conformance and the transaction mechanism which are central to our work.

Kramer and Magee have addressed the problem of dynamic change management for distributed systems.[17] Their approach focuses mainly on changes specified in terms of the system structure and provides a separate language for changes specification. Our approach deals with type modifications and uses one language to specify types and their changes. Unlike their approach, which concentrate on the logical structure of a system, we consider the dynamic behavior of a specification and we take into account the inheritance property which is inherent to the object-oriented aspect of our language. The unit of change in our model is a type (class), instead of a module.

8. Conclusions

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

RMondel provides an elegant manner for dynamic type modifications. It also, gives an interesting framework based on formal semantics, to develop adaptable CASE tools for executable specifications development. *RMondel* framework may be easily adapted for other object-oriented languages. The *Mondel* language has already been implemented on Sun workstation using prolog language, and used for writing and simulating the specifications of the OSI directory system, and the personal communication services. The choice of Prolog was made because it was easy to translate the formal semantics rules of *Mondel* to Prolog predicates. A verifier based on a Petri net approach is also implemented at the University of Montreal, and a prototype of *RMondel* is under development. 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.

Acknowledgment: This research is joint research project sponsored by the Ministry of Education, Science and Culture of Japan, International Scientific Research Program.

References

1) America, P.: A Behavioral Approach to Subtyping in Object-Oriented Programming Languages, *Phlips Journal of Research*, Vol. 44, pp. 365-383 (1990).
2) Banerjee, J., Kim, W. and Korth, H. F.: Semantics and Implementation of Schema Evolution in Object-Oriented Databases, *Proceedings, ACM SIGMOD Int'l Conf. on Management of data*, SanFransisco., CA, pp. 311-322 (May 1987).
3) Barbeau, M. and v. Bochmann, G.: Formal Semantics and Formal Verification of Object Oriented Specification Based on the Colored Petri Net Model, submitted to IEEE Trans. on Software Eng.
4) Black, A., Hutchinson, N., Jul, E., Levey, H. and Carter, L.: Distribution and Abstract Types in Emerland, *IEEE Trans. on Soft. Eng.*, Vol. SE-13, No. 1, pp. 65-76 (1987).
5) v. Bochmann, G.: Inheritance for Objects with Concurrency, Publication Departmentale # 678, Departement IRO, Université de Montréal (Sept. 1989).
6) v. Bochmann, G., Barbeau, M., Erradi, M., Lecomte, L., Mondain-Moval, P. and Wiliams, N.: Mondel: An Object-Oriented Specification Language, Publication Departmentale # 748, Departement IRO, Université de Montréal (Nov. 1990).
7) Erradi, M., v. Bochmann, G. and Hamid, I. A.: RMondel: A Reflective Object-Oriented Specification Language, The ECOOP/OOPSLA '90 First Workshop on: Reflection and Metalevel Architectures in Object-Oriented Programming, Ottawa (1990).
8) Erradi, M., v. Bochmann, G. and Hamid, I. A.: Dynamic Modifications of Object-Oriented Specifications, CompEurop '92, IEEE Int'l Conf. on Computer Systems and Software Engineering (May 1992).
9) Erradi, M., v. Bochmann, G. and Dssouli, R.: Framework for Dynamic Evolution of Object-Oriented Specifications, Networking and Distributed Computing, (La revue Réseaux et Informatique Répartie), Vol. 4, No. 2, pp. 360-378 (1992).
10) Brinksma, E. and Scollo, G.: Lotos Specification, Their Implementations and Their Tests, Protocol Specification, Testing and Verification VI (IFIP Workshop, Montréal, 1986), North-Holland Publ., pp. 349-360.
11) Cointe, P.: Metaclasses Are First Class: The objVLisp Model, OOPSLA '87, ACM, Sigplan Notices 22, 12, pp. 156-167 (1987).
12) Delcourt, C. and Zicari, R.: The Design on an Integrity Consistency Checker (ICC) for an Object Oriented Database System, ECOOP '91 (1991).
13) Ichikawa, H., Yamanaka, K. and Kato, J.: Incremental Construction of LOTOS Specification, PSTV '90, pp. 185-200 (1990).
14) Ferber, J.: Computational Reflection in Class Based Object-Oriented Languages, *Proc. of OOPSLA '89*, Oct. 1-6, 1989, pp. 317-326 (1989).
15) Gary, J.: Notes on Database Operating Systems, IBM Research Report: RJ2188, IBM Research San Josee, California (1978).
16) Gary, J.: The Transaction Concept: Virtues and Limitations, *IEEE Proc. Conference on Verification Languages and DataBases*, Cannes, pp. 144-154 (Sept. 1981).
17) Kramer, J. and Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management, *IEEE Trans. on Software Engineering*, Vol. 16, No. 11, pp. 1293-1306 (1990).
18) Tan, L. and Katayama, T.: Meta Operations

for Type Management in Object-Oriented Databases—A Lazy Mechanism for Schema Evolution—, *Proc. of the 1st Int'l Conf. on Deductive and Object-Oriented Databases* (Oct. 1989).

19) Milner, R.: *A Calculus of Communication Systems*, Lecture Notes in Computer Science, No. 92, Springer-Verlag (1980).

20) Maes, P.: Concepts and Experiments in Computational Reflection, OOPSLA '87, ACM Sigplan Notices 22, pp. 147-155 (1987).

21) Meyer, B.: *Object Oriented Software Construction*, Hoare, C. A. R. Series Editor, Prentice Hall (1988).

22) Rudkin, S.: Inheritance in LOTOS, *4th Int'l Conf. of Description Techniques, FORTE '91*, pp. 415-430 (1991).

23) Penney, D. J. and Stein, J.: Class Modification in the GemStone Object-Oriented Databases, OOPSLA '87, pp. 111-117 (1987).

24) Plotkin, G. D.: A Structural Approach to Operational Semantics, Aarhus University, Report DAIMI FN-19 (1981).

25) Smith, B. C.: Reflection and Semantics in a Protocol Programming Language, Ph. D., Thesis, MIT, MIT/LCS/TR-272 (1982).

26) Skarra, A. H. and Zdonik, S. B.: Type Evolution in an Object-Oriented Databases, *Research Directions in Object-Oriented Programming* (Wegner, P. and Shriver, B. (eds.)), MIT Press, pp. 393-415 (1987).

27) Stefik, M. and Bell, D. G.: *Object-Oriented Programming: Themes and Variations*, MIT Press, pp. 40-62 (1985).

28) Vissers, C., Scollo, G. and v. Sinderen, M.: Architecture and Specification Style in Formal Descriptions of Distributed Systems, *Proc. IFIP Symposium on Protocol Specification, Verification, and Testing*, Atlantic City (1988).

29) Ibrahim, M. H.: ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architecture in Object-Oriented Programming, Ottawa (Oct. 1990).

30) Ibrahim, M. H.: ECOOP/OOPSLA '91 Workshop on Reflection and Metalevel Architecture in Object-Oriented Programming (Oct. 1990).

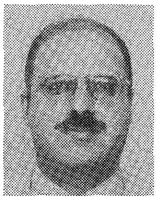
31) Wegner, P.: Concepts and Paradigms of Object-Oriented Programming, OOPS Messenger, Quarterly Publication of the ACM SIGPLAN, Vol. 1, No. 1 (1990).

32) Williams, N.: Un Simulateur Pour un Language de Spécification Orienté-Object, Msc. thesis, Université de Montréal (1990).

33) Zafiropulo, P.: Protocolvalidation by Duologue-Matrix Analysis, *IEEE Trans. on Comm.*, Vol. COM-26, No. 8, pp. 1187-1194 (1978).

34) Zaho, J. R. and v. Bochmann, G.: Reduced Reachability Analysis of Communication Protocols: A New Approach, *Proc. IFIP Workshop on Protocol Spec. Testing and Verification*, North-Holland Publ., pp. 234-254 (1986).

(Received September 27, 1993)
(Accepted January 13, 1994)



Issam A. Hamid was born on May 1955. He has graduated from the Faculty of Engineering of the University of Manchester, England, on 1979. He received his Master, and Doctor degree in Information Engineering from Tohoku University, on 1985 and 1988, respectively. He then joined the Large Computer Center of Tohoku University as research associate. He moved to the University of Tokyo, Research Center for Advanced Science and Technology (RCAST) on April 1989 as Assistant Professor. On December 1989, he moved to join the Department of Information and Operational Research (IRO) of the University of Montreal, Canada as visiting Professor. Currently, he is an Associate Professor in the Department of Information Design of Tohoku University of Art & Design, Yamagata, Japan. He is also, research consultant at the CRIM (Computer Research Institute of Montreal) Canada. He contributed in several research projects in Montreal for Dynamic Information Modeling using Objet-oriented programming. His research interest is on Artificial Intelligence, dynamic modification of information system, and highly parallel computation. He is a member of IEEE and IPSJ. He is a committee member of Tohoku-branch of IPSJ.

Setsuo Ohsuga is currently a professor in the Research Center for Advanced Science and Technology (RCAST) at the University of Tokyo. He has been director of RCAST. He has also been president of the Japanese Society for Artifical Intelligence. He graduated at the University of Tokyo in 1957. From 1957 to 1961 he worked in Fuji Precision Machinery (the present Nissan Motors). In 1961 he moved to the University of Tokyo and received his Ph. D. in 1966. He became associate professor in 1967 and professor in 1981. His research interests are artifiicial intelligence, knowledge information processing, databases and CAD. He has received awards for his researches twice from the Academic Society in Japan. He in a member of the editorial boards of 9 scietific journals.