Dynamic Modifications of Object-Oriented Specifications

M.Erradi, G.v. Bochmann, I. Hamid

Département d'Informatique et de Recherche
Opérationnelle
Université de Montréal
C.P. 6128, Succ. "A", Montréal,
P.Q., Canada, H3C-3J7
E-mail: {erradi bochmann hamid} @iro.umontreal.ca

Abstract:

RMondel (Reflective Mondel) is a reflective object-oriented specification language developed for the description of distributed systems. The objective of *RMondel* is to allow the development of dynamically modifiable specifications. We will show how the features of the language are useful for the modification and construction of valid specifications. Therefore, the user of this language can modify certain specification by adding or modifying objects and types to get a new adapted specification. A predefined set of constraints, allow the construction of valid specification. *RMondel* gives an interesting framework, based on formal semantics, to develop user friendly interfaces and CASE tools to construct and eventually modify specifications. We have illustrated our approach using a switch system example.

1. Introduction and Motivations

Recently, an object-oriented approach to programming and designing complex software systems has received tremendous attention in several disciplines of computer science such as programming languages, databases, distributed systems, and operating systems. Object-oriented programming offers several important advantages over control-oriented programming [Meye 88]. Objects are collection of operations that share a state. The operations determine the calls (messages) to which the object can respond, while the shared state is hidden from the outside and is accessible only to the object's operations. Another advantage of object-oriented programming is the notion of class (type) and inheritance. Classes serve as templates for objects creation. Inheritance allows the reuse of behavior of a class in the definition of new classes. Subclasses of a class inherit the operations of their parent class and may add new operations and new attributes.

We have developed a new object-oriented specification language, called *Mondel*[1] [Boch 90] that has important concepts as a specification language to be applied in the area of distributed systems. The motivations behind *Mondel* are: (a) writing system descriptions at the specification and design level, (b) supporting concurrency as required for distributed systems, (c) supporting persistent objects and transaction facilities, and (d) supporting the object concept. Presently, our language *Mondel* has been used for the specification of problems related to network management [Boch 91a] and other distributed applications [Boch 91b].

In a wide spectrum of applications, system specifications require modifications to accommodate evolutionary change, particularly for those systems with long expected lifetime. They need to evolve along with changes of human needs, technology and/or the application environment. The changes may require modifications of certain functions already provided by the system, or some extension introducing new functions. In general, evolutionary changes are difficult to accommodate because they cannot be predicted at the time the system is designed. So, systems should be sufficiently flexible to permit arbitrary, incremental changes. To support the construction of dynamically modifiable systems, written in *Mondel*, we need to have access to, and modify the specification and the implementation of the system during run time.

---

1 *Mondel* stands for **Mon**treal **de**scription **l**anguage which was developed within a CRIM (Centre de Recherche Informatique de Montréal) / BNR (Bell Northern Research) project.

The object oriented approach is known by its flexibility for system construction. This is partly due to the inheritance property that permit class reuse and incremental construction of systems. However, it is not possible to introduce arbitrary changes in a given system specification. Recently, in object-oriented languages, a new concept called reflection, has gained wider attention as confirmed by the first and second workshops on reflection and metalevel architectures in object-oriented programming [Work 90, Work 91] held in conjunction with OOPSLA'90 and 91. Reflection is the capability of a system to reason and act upon itself. A language is called reflective if it uses the same structures to represent data and programs. In conventional systems, computation is performed only on data that represent entities of an application domain. In contrast, a reflective system contains another type of data that represent the structural and computational aspects of itself. The original model of reflection was proposed in [Maes 87] following Smith's earlier work [Smit 82], where a meta-object is associated with each object in the system to represent information about the implementation and the interpretation of the object.

To achieve our goal that is the construction of dynamically modifiable specifications and implementations, we define a reflective object oriented language called *RMondel*, directly based on the *Mondel* language. Reflection in *RMondel* is supported by two fundamental features of reflection related to object oriented languages which are: Structural Reflection (SR) and Computational Reflection (CR). For the SR we consider that a type (i.e., class) is an object and types are instances of other types, also called metatypes. Also, we address the reflectivity for object attributes, operations (methods) and behaviors. For CR, 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 interpreters can be defined for monitoring the behavior of objects, or for dynamically modifying their behavior.

In this paper, we focus mainly on structural reflection. With respect to computational reflection we consider that the objects in the system share one interpreter. The main issue is to show how structural reflection can be useful to change dynamically a specification. The need for validation of the changes to maintain system consistency is also discussed. The paper is organized as follows. Section 2 gives an overview of the original language *Mondel* and its important characteristics. Section 3 explains the architecture, semantics and the interpreter of *RMondel*. Section 54 show, through an example, how a specification written in *RMondel* can be dynamically modified to satisfy new requirements.

2. *Mondel* Overview

The *Mondel* language [Boch 90] is object-oriented 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* has also a formal semantics, expressed by means of a translation into a state transition system. An object is an instance of a type definition that specifies the properties that are satisfied by all its instances. Each *Mondel* object has an identity, a certain number of named attributes (this means that each object instance of that type will have fixed references to other object instances, one for each attribute), and acceptable operations which are externally visible and represent actions that can be invoked by other objects.

An executable system specification in *Mondel*, 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 remote procedure call or rendezvous mechanism. An operation call is syntactically represented by the "!" operator. For instance in the statement "c!failure" of Figure 1, "c" designates the called object, and "failure" is an operation defined within the type (*Controller*) of "c".

In Figure 1 we give an 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. This example will be used trough out the paper to illustrate our approach for the dynamic modification of specifications.

```
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
```

Figure 1: A *Mondel* specification example

3. *RMondel* Architecture

To support the dynamic modification of objects structure and their behavior, we developed *RMondel*, a reflective version of *Mondel*, to provide a framework for the construction of flexible systems specifications. In this section, we will show the architecture of *RMondel*, and we describe its components. The *RMondel* system consists in a User Interface, a translator, a set of constraints, the kernel types, and an *RMondel* interpreter as shown in Figure 2. The user interface allows the user to compose his new specification and eventually introduce changes to such specification. The translator takes an *RMondel* specification and produces a set of *RMondel* objects according to *RMondel* semantics [Erra 90].

To maintain the system in a consistent state, the *RMondel* interpreter uses a set of predefined static constraints that define the consistency requirements of the type lattice and those which maintain the type-instance relationship. Also, the interpreter uses a set of predefined kernel objects such as *TYPE* and *OBJECT* described in Section 3.2. Because in *RMondel* the attributes, the operations, and the statements (behavior) of an object are also objects, then the predefined kernel objects are initially existent to avoid a circular definition. It is important to mention that the *RMondel* system can be used for the construction of specifications as well as for the modification of an existing specification.

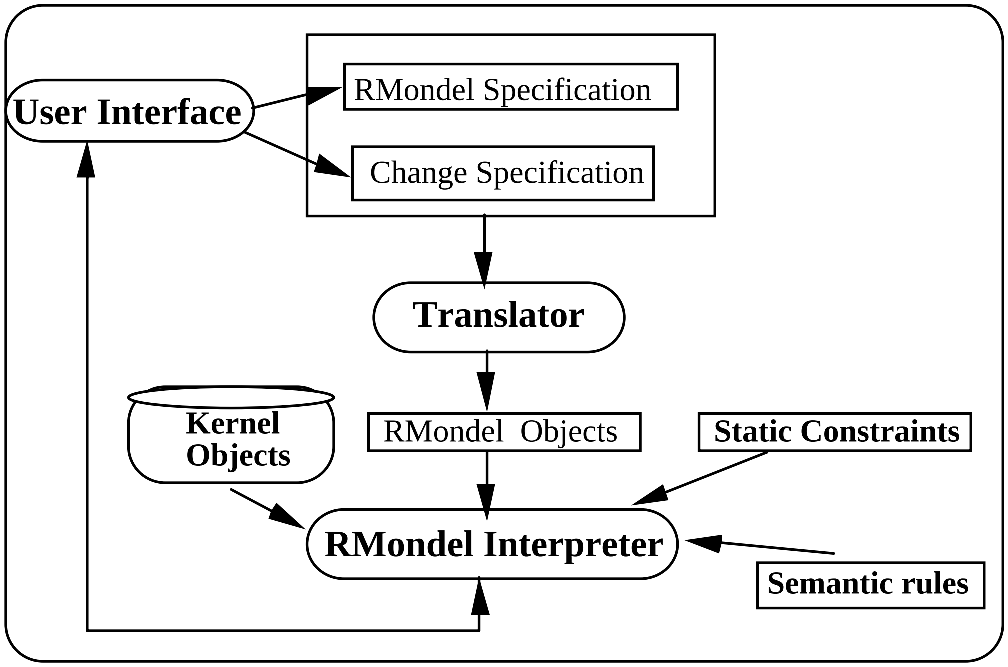

Figure 2: *RMondel* Architecture

The *RMondel* interpreter is based on a formal dynamic semantics definition. The dynamic semantics associates a meaning to the valid language sentences. To define the formal semantics of *RMondel* we adopted the operational approach [Plot 81] based on transition systems. An important use of formal semantics lies in the verification of the correctness of a specification [Barb 90]. Formal semantics is necessary for system refinement or implementation, and development of test cases. Also, it can be used for the generation of a language interpreter from the rules that constitute the operational definition of the language. Details on *RMondel* semantic rules are given in [Erra 90]. In the following sections we describe the components of *RMondel* system shown in Figure 2. Let us first show the structure of *RMondel* objects, to help the understanding of the other sections.

3.1. Object Structure

In *RMondel*, the structure of an object is considered as a finite set of attributes represented by pairs. Each attribute is represented by a pair ($Name_{attri}$, Id_{attri}) which is a substitution (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 couple. We have two types of attributes: initial attributes and effective attributes. The "initial attributes" are:

- The unique object identifier, named ObjectId, which is commonly known as "self", is generated automatically. For the sake of readability we will consider that object identifiers, for type objects, are constructed by means of the type name prefixed by "Id" (i.e., the type object equip of Figure 1 is identified by Idequip).
- The identifier of the object type, named *MyType*, which is the type of the created object, and
- The identifier of the object behavior, named *State*, which represents the initial behavior of the created object. The value of the *State* 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.

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

$o = $ <(ObjectId,Ido),(MyType,Id_{type}), (State,Id_{beh}), {...,($Name_{attri}$,Id_{attri}),..}>

where Ido, Id_{type}, and Id_{beh} designate the initial attributes of the object o. The set {...,($Name_{attri}$,Id_{attri}),..} designates the set of the effective attributes of o.

3.2. The Kernel Type Specifications

The kernel objects constitute a database of *RMondel* predefined objects that are the basis of *RMondel* architecture. The structure of *RMondel* is supported by an instantiation and an inheritance graphs. The instantiation graph represents the "instance of" relationship, and the inheritance graph represents the "subtype of" relationship. The objects *TYPE* and *OBJECT* are the respective roots of these two graphs. In the following we give the structure of these objects. Note that other kernel objects, such as those representing the different statements of the language, are part of the kernel objects database. For the lack of space, we describe here only the *TYPE* and *OBJECT* objects.

3.2.1. The *TYPE* Object

TYPE initially exists in the system, it defines the behavior for types object, as for instance type equip of Figure 1. The *TYPE* object holds the attributes *TypeName* and *Statdef* which refer to the name of a type, and the statements defined in such a type respectively, as shown in Figure 3. The *TYPE* object definition contains also the ***New*** operation which creates object instances. We assume that the *TYPE* object exists initially as an instance of itself. The structure of the *TYPE* object is:

<(ObjectId,IdTYPE),(MyType,IdTYPE), (State,S), {(TypeName,"TYPE"), (Statdef,IdS1)}>;

Where IdS1 is an object reference to the specified behavior within the *TYPE* type definition, among others, we find the ***New*** operation definition. (State,S) corresponds to the initial behavior of the *TYPE* object. The *TYPE* object is useful for the creation of type definitions as well as their instances.

type TYPE = OBJECT **with**
    TypeName : string;
    Stat : Statement;
  operation
    New : OBJECT; {The type *OBJECT* is defined below }
    <*:* (t : TYPE): boolean; { the conformance relation: it checks if self conforms to t. The "<:" relation is the closure of the inheritance relation. }
  invariant
    { We define here, the constraints which must hold to maintain the system in a consistent state.
    These constraints define the consistency requirements of the type lattice which corresponds to the
    static semantics rules checked by the *Mondel* compiler [Erra 90].}
  behavior
    { We specify here, in which order the operations, provided by an object of type *TYPE*, can be
    executed and what are the possible returned results. }
endtype TYPE

Figure 3. The definition of *TYPE*

3.2.2. The OBJECT object

OBJECT is the most general type. It describes the common characteristics of all objects (types and instances). Each object is characterized by its unique identifier, its type, its effective attributes (binding) and its behavior. Also it provides the ***NewAttr*** operation for attribute instances creation. *OBJECT* is the root of the inheritance graph, and it is initially defined in the form:

<(ObjectId,IdOBJECT),(MyType,IdTYPE), (State,St), {(TypeName,"OBJECT"), (Statdef,IdS2)}>;

Where IdS2 is a reference to the specified behavior within the *OBJECT* object. It corresponds to ***NewAttr*** operation definition. (State,St) corresponds initially to the *OBJECT* object behavior, which is the same as for *TYPE* because *OBJECT* is an instance of *TYPE*.

3.3. *RMondel* Objects

The attributes, operations and the statements of an object (called master object) are also objects (called fine grain objects) according to *RMondel* semantics definition. In the remaining of the paper we make the distinction between master objects and fine grain objects only if necessary, otherwise we use ‘object’ to designate both. The fine grain objects are linked to their master object by a reference called “Appears-In”.

Let us consider the example of Figure 1, the type equip, as a master object, is represented by the following structure that corresponds to *RMondel* objects internal representation. Such a representation is generated by the *Translator* of Figure 2.

(1) < (ObjectId,Idequip), (MyType, IdTYPE), (State,IdTYPEBehavior), {(TypeName,"equip"), (Statdef, IdProCallWorking)}>;

This object (1) corresponds to the type specification “equip”, it is an instance of the *TYPE* object (MyType, IdTYPE), its state is the same as the *TYPE* object (State,IdTYPEBehavior), its name is “equip” (TypeName,"equip"), and the behavior definition within the type “equip” is an object referred to by IdProCallWorking.

The fine grain objects associated to the object “equip” are:

(1) <(ObjectId,Idc), (MyType, IdAttribute),.., {(AttrName,"c"), (AttrType, IdController),(AppearsIn,Idequip)}> ;

(2) <(ObjectId,IdWorking),(MyType,IdProcedure),...,{ (ProcName,”Working”),(AppearsIn,Idequip) } >;

(3) <(ObjectId,IdFailed),(MyType,IdProcedure),...,{ (ProcName,”failed”),(AppearsIn,Idequip) } >;

The object in line (1) corresponds to the attribute definition named “c” of type “Controller”. We remark that this object is linked to its master object by the link “AppearsIn” (AppearsIn,Idequip) . In line (2) and (3) we find two objects that correspond to the procedures *working* and *failed* respectively. This gives a powerful flexibility to *RMondel* to allow dynamic modification of a specification. A change to a specification, will be introduced by adding and/or deleting objects.

In the previous sections we have described the components of *RMondel* system. In the following sections we show how the *RMondel* interpreter works, and how it facilitates the dynamic modification of specifications.

4. Dynamic modification of *RMondel* specification

4.1. Support for dynamic modification of specifications

In order to allow for the construction of dynamically modifiable type specifications, we need to have access, and to be able to modify type specifications during run-time. As has been shown in the previous sections, types are instances of *TYPE*, so they are accessible like any other object in the system. For the dynamic modification of type specifications, we need to define some primitive operations within the object *TYPE*, which allow the modification of a type specification. Since these operations are defined within the *TYPE* object, each instance (i.e. a type specification) of such a type can accept such operations. Therefore, we enhance the *TYPE* object specification as follows to include the type specification modification operations:

type *TYPE* = OBJECT **with**

TypeName : **string**;

Stat : *Statement*;

operation

New : *OBJECT*;

<: (t : TYPE): boolean; {the conformance relation: it checks if self conforms to t (see Figure 3).}

AddAttr (A:*Attribute*);

DelAttr(A: *AttrName*);

AddOper(O:*Operation*);

DelOper(O:*Operation*);

AddProc(P:*Procedure*);

DelProc(P:*Procedure*);

AddStat(S:Statement);

DelStat(S:Statement);

. . .

invariant

{ We define here, the constraints which must hold to maintain the system in a consistent state. These constraints define the consistency requirements of the type lattice which corresponds to the static semantics rules checked by the *Mondel* compiler.}

behavior

{ We specify here, in which order the operations, provided by an object of type *TYPE*, can be executed and what the possible returned results are. }

endtype TYPE

Figure 4: Revised definition of *TYPE* object

To add a new operation to a type specification 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 an 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 defined within *TYPE* play an important role to maintain consistency between all the component of a type specification. Now, after the addition of the operation O1, each newly created instance of T, can accept such an operation. We will give in the next section a simple example to illustrate our approach

4.2. Example of dynamic modifications.

To illustrate the dynamic modification of *RMondel* specifications, we consider here the *RMondel* specification of the switch system of Figure 1. The system consists 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 as shown in Figure 5. The system remains in the failed state until the failed equipment is repaired. 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 previously shown in Figure 1.

From a practical point of view, the specification of the switch system given above is not complete. Such a system is vulnerable, because if a failure occurs in one equipment the system will be down until the equipment is repaired. Let us consider that we need a more reliable system. In this case we introduce a standby equipment that will be substituted for the failed piece of equipment; the standby then does the work of the original piece of equipment. With this 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 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. However, if the standby also detects a failure, the conclusion is that the malfunction origin is not the piece of 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 6 show the state transition diagram of the new system configuration.

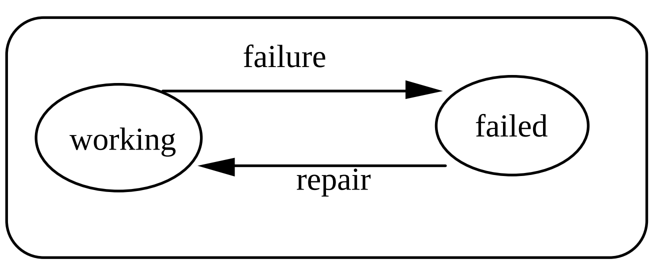

Figure 5: Initial system specification.

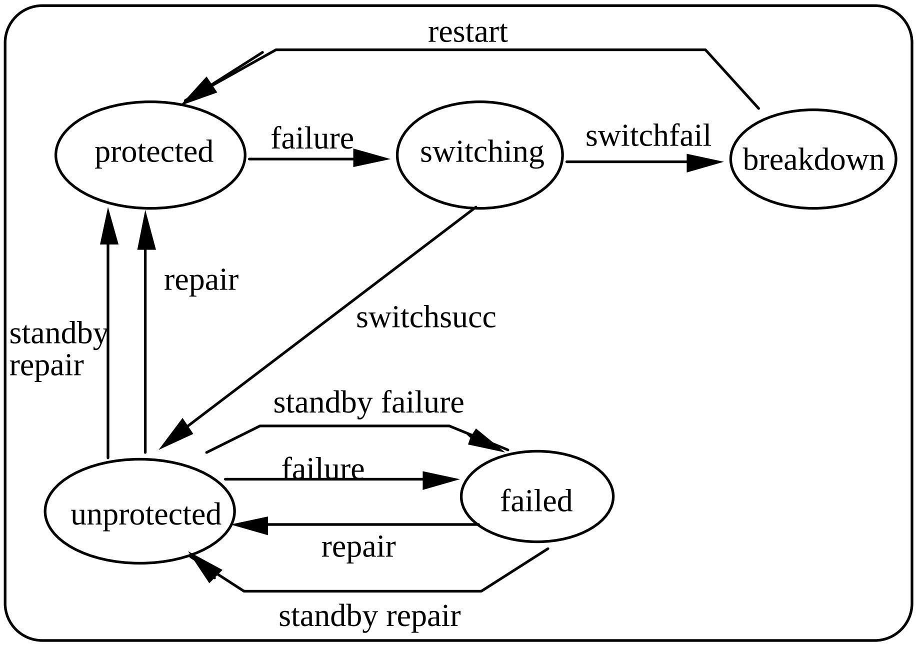

Figure 6: New system specification.

Let us show how a user can construct a new specification based on the existing one. 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 of Figure 6, 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, not shown here for the lack of space, for more details interested readers are refered to [Erra 90]. 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 Figure 1) is renamed to become the procedure *unprotected* as shown in line 32 of Figure 7. Also the body of the procedure *working* is replaced by a new object of the *Choice* type as shown in line 33 of Figure 7(*Choice* is a kernel type that represents the choice construct of *RMondel* [Erra 90]).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 Figure 7.

To maintain the consistency of the specification construction, a set of constraints defined as invariants within the *TYPE* object specification must be satisfied. We distinguish three categories of invariants: general invariants, type definitions invariants, and inheritance invariants. Details on these invariants are given in [Erra 90]. For instance the “accept switchsucc “ statement in line 23 of Figure 7, cannot be validated by *RMondel* system while the *switchsucc* operation is not defined within the *controller* type as shown in line 5 of Figure 7. For this purpose, the user has to add the *switchsucc* operation definition by using the *AddOper* operation defined within the *TYPE* object. Because the controller type is an instance of *TYPE* , then it can accept the *AddOper* to add the *switchsucc* operation to the set of defined operations.

```
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 switchfail do
28          s!switchfail; 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       accept 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
```

Figure 7: modified specification

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 modifications 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 objects 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 [Barb 90]. This verifier has been considered to be adapted for *RMondel* specifications.

The *Mondel* language has already been implemented on a Sun workstation using prolog language. The choice of prolog was made because it was easy to translate the formal semantic rules of *Mondel* to prolog predicates. A verifier based on a petri net approach is also implemented at the University of Monreal, and a prototype of *RMondel* is under development.

5. Conclusion

We have developed *RMondel*, a reflective concurrent object oriented specification language, based on *Mondel* language designed to support the description of distributed systems. The objective of *RMondel* is to allow the development of dynamically modifiable specifications. We have shown the architecture of *RMondel*, and how the features of the language are useful for the modification and construction of valid specifications.We have illustrated through an example how the language can adapt dynamic modifications. Therefore the user of this language can modify his/her specification by adding new objects and types to get a new adapted specification. A predefined set of constraints, allow the construction of valid specifications. *RMondel* gives an interesting framework based on formal semantics, to develop user friendly interfaces and adaptable CASE tools.

Acknowledgments
The authors would like to thank the *Mondel* group involved in the CRIM-BNR project, in particular M. Barbeau for helpful discussions. Financial support from the IDACOM-NSERC-CWARC industrial research chair is gratefully acknowledged.

References

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

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

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

[Boch 91b] G. v. Bochmann, S. Poirier and P. Mondain-Monval, *Object-oriented design for distributed systems: The OSI Directory example*, submitted for publication.

[Erra 90] M. Erradi, *Dynamically modifiable object-oriented specifications and implementations*, Ph.D. thesis in progress, département IRO, University of Montreal.

[Work 90] M. H. Ibrahim, *ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, Ottawa, October 1990.

[Work 91] M. H. Ibrahim, *ECOOP/OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

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

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

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

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