Dynamic extension of object-oriented distributed system specifcations*

Mohammed Erradi, Ferhat Khendek, Rachida Dssouli and Gregor v. Bochmann

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

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

Tel : (514) 343-7484
Fax : (514)343-5834

Abstract

The effort for adding features to a telecommunication system, like adding new functionalities to any large software system might be tremendous. Each new feature or added functionality may interact with many existing features. such interactions may lead to blocking situations (e.g., deadlock) or system breakdown. In addition, for large long-life distributed systems, it may be not possible to stop the entire system to allow its extension. Therefore, an important and difficult problem is that of making modifications or extensions dynamically, without interrupting the processing of those parts of the system which are not affected.

In the context of an object-oriented executable specification language *Mondel*, we study the dynamic extension of *Mondel* specifications. A *Mondel* specification consists of a set of interacting objects. The behavior of an object is formally specified by a translation to labeled transition systems (LTS). Using LTS formalism, we describe how objects behaviors and consequently *Mondel* specifications can be extended without interaction problem. To allow for the dynamic extension of specifications, we define *RMondel* a reflective version of *Mondel*, and we introduce a transaction mechanism in order to preserve the consistency of the whole specification.

* This research was supported by a grant from the Canadian Institute for Telecommunications Research (CITR) under the NCE program of the Government of Canada.

1. Introduction

Distributed systems evolve by adding new functionalities according to new requirements, or modifying some existing functionalities. The addition of new functionalities to a given system may interfere with the existing and verified functionalities. The interference may be caused by the overlapping of these new functionalities with the existing ones. This is known as "features interaction" problem [Bowe 89].

In this paper, we will consider the feature interaction problem at the specification level. We are interested in system extension by adding functionalities and studying the feature interaction problem from the formal point of view. Formal specifications allow formal reasoning about systems, analysis of correctness, systems equivalence, and transformations.

Recently, the object-oriented approach to programming and designing complex software systems has received tremendous attention in several disciplines of computer science. The objet-oriented approach is known by its flexibility for system construction. We have developed a new object-oriented specification language, called *Mondel* [Boch 90] that has important concepts as a specification language to be applied in the area of distributed systems. It has a formal semantics, expressed by means of a translation into a state transition system. The motivations behind *Mondel* are: (a) writing system descriptions at the specification and design level, (b) supporting concurrency as required for distributed systems, (c) supporting persistent objects and transaction facilities, and (d) supporting the object concept. Presently, *Mondel* has been used for the specification of problems related to network management [Boch 91] and other distributed applications [Boch 92].

Informally, given a specification S_{old}, we want to add a new behavior, corresponding to a new functionality, described by S_{added}. This new behavior may interact with the behavior described by S_{old}. In our approach, the features interactions problem between S_{old} and S_{added} may have the following observable effects:

- the new specification S_{new} is not able to behave as S_{old} or S_{added}, or
- S_{new} may block in some situations, where S_{old} or S_{added} can not block.

In order to prevent such problems, the new specification S_{new} has to be an extension of S_{old} and S_{added}. The extension is a formal relation between behavior specifications [Brin 86].

In this paper, we present a systematic approach for the extension of individual objects, based on a formal technique. It consists of building a new object behavior S_3 by adding a new behavior described by S_2 to an object behavior S_1 without features interactions problem between the original behavior and the added one. The newly derived object behavior S_3 extends S_1 and also S_2.

In order to allow for the construction of dynamically modifiable specifications, we need to have access, and to be able to modify specifications during execution-time. Therefore, we developed *RMondel*, a reflective version of *Mondel*, that uses meta-objects to provide facilities for the dynamic modifications of specifications [Erra 92a]. The extension of individual objects of the specification do not ensure that the obtained specification is an extension of the original one. Therefore, it is necessary to provide facilities for controlling change 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.

The remainder of this paper is structured as follows. Section 2 gives an overview of Mondel language, the labeled transition system notations, and the definition of the extension relation. Section 3 is mainly devoted to the algorithm for objects behaviors extension. In Section 4, we introduce the facilities for the dynamic extension of Mondel specifications. Section 5 describes the locking protocol and the transaction mechanism necessary for maintaining the whole specification consistency. Before concluding, we review the related works.

2. The language and formalisms

2.1. Mondel Overview

We have developed *Mondel*: An object-oriented specification language [Boch 90] with certain particular features, such as multiple inheritance, type checking, rendezvous communication between objects, the possibility of concurrent activities performed by a single object, object persistence and the concept of transaction. *Mondel* is particularly suitable for modeling and specifying applications in distributed systems. Each *Mondel* object has an identity, a certain number of named attributes (i.e., each object instance 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 object is an instance of a type definition (i.e., called class in most object-oriented languages) that specifies the properties that are satisfied by all its instances. 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.

Mondel has a formal semantics which associates a meaning to the valid language sentences. Such a semantics was defined based on the operational approach. In this approach an abstract machine simulates the real computer role. The meaning of a specification is expressed in terms of actions made by the abstract machine. We have particularly applied the technique of Plotkin [Plot 81] where state/transition systems are taken as machine models. The *Mondel* formal semantics was the basis for the verification of *Mondel* specifications [Barb 91b], and has been used for the construction of an interpreter [Will 90].

Example of a Mondel specification

In the following we show an example using *Mondel* language. This example will be used throughout the paper. Let us consider a vending machine which receives a coin and delivers candies to its user. We distinguish two types of objects: the type *Machine* and the type *User*, as shown in *Mondel* specification of Fig.2.1. The relation between the *Machine* and the *User* is expressed by the fact that the user knows the machine. Such a relation is modeled by the attribute "*m*" defined in the *User* type.

The user is initially in a *Thinking* state, and when he decides to buy a candy he inserts a coin. After the coin has been accepted, the user enters the *GetCandy* state. Then the user pushes the machine's button to get a candy. Once the candy is delivered, the user enters the *Thinking* state again. The 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.2.1.). The object types are specified using a state oriented style [Viss 88]. Each object internal state is modelled as one *Mondel* procedure.

```
0 unit spec =

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

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

20 endtype Machine

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

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

34 endtype User

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

39 endunit spec
```

Fig.2.1. *Mondel* specification of the vending machine.

2.2. Labelled Transition Systems

Labelled Transition Systems (LTSs) [Kell 76] are used as a model for a number of specification languages, e. g. LOTOS [Bolo 87, ISO 8807], CCS [Miln 80], CSP [Hoar 85], Mondel [Boch 90]. For the remainder of this paper, an LTS is considered as a formalization of a Mondel object behavior. We may refer to a Mondel object behavior by its corresponding LTS.

Definition [Kell 76]

An LTS TS is a quadruple <S, L, T, S_o>, where

S is a (countable) non-empty set of states.

L is a (countable) set of observable actions.

T: S x L " {τ} ∅ S, is a transition relation, where a transition from a state S_i to state S_j by an action μ (μ □ L"{τ}) is denoted by S_i–μ∅S_j.

S_o is the initial state of TS.

τ represents the internal, non-observable action.

The LTS representing the behavior of an object of type machine is given in Fig. 2.2. The state "Ready" is the initial state.

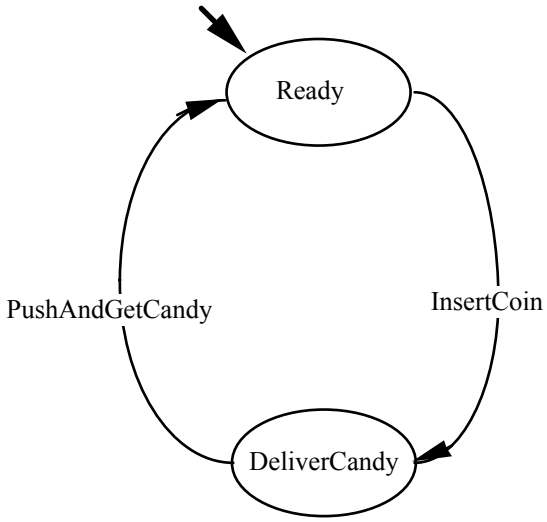

Fig.2.2. LTS S1 representing the behavior of a vending machine object

Intuitively, different LTSs may describe the same observable behavior. Therefore different equivalence relations have been defined based on the notion of observable behavior. They range from some coarse trace equivalence [Hoar 85] to a much finer bisimulation equivalence [Miln 80]. However, for our considerations, one does not need equivalence relations, but rather ordering relationships. The extension relation [Brin 86] is most appropriate for our purpose of specifications enrichment without feature interaction problem.

Definition [Brin 86]

S_1 extends S_2, (written S_1 ext S_2), if

- S_1 may perform any trace of actions that S_2 may perform, and
- S_1 will not block where S_2 do not block.

3. The extention of objects behaviors

In this section we describe our approach for the extension of object behaviors. We assume that an object behavior is transformed to its corresponding LTS. We also assume that these behaviors are finite state, and therefore they are represented by finite state LTS (FLTS). Our approach for behaviors extension consists of an algorithm for merging behaviors.

Given two FLTS S1 and S2, the merging algorithm deduces systematically a new FLTS S3. S3 is an extension of both S1 and S2. Moreover, the cyclic behaviors in S1 and S2 remain cyclic in S3. The intial state of S3 is represented by a tuple composed by the intial states of S1 and S2. Therefore, S3 offers behaviors of S1 (or S2) after behaving like S2 (or S1), alternatively.

In the following, we summarize the FLTSs merging algorithm. For reason of simplicity, we will only consider FLTSs without the internal action τ. More formal and complete version of this algorithm has been developed in [Khen 92a] as well as its correctness proof. Note that the notation $S_j \not\rightarrow a\varnothing$ is used to denote that there is no transition labelled by action "a" from state S_j.

FLTSs merging algorithm

Input: FLTSs S1 = <St1, L1, T1, $S1_o$> and S2 = <St2, L2, T2, $S2_o$>
Output: FLTS S3 = <St3, L1 " L2, T3, <$S1_o$, $S2_o$>>
The initial state of S3 is represented by the tuple <$S1_o$, $S2_o$> composed by the initial states of S1 and S2.

A state St in St3 may be a tuple <$S1_i$, $S2_j$> consisting of state $S1_i$ from St1 and $S2_j$ from St2 as for the initial state <$S1_o$, $S2_o$> or *simple* state $S1_i$ from St1 or $S2_j$ from St2. These states and the transitions which reach them are added step by step into St3 and T3, respectively. Initially, St3 contains only the initial state <$S1_o$, $S2_o$>.

The definition of the transitions from a state <$S1_i$, $S2_j$> in S3 depends on the transitions from $S1_i$ in S1 and from $S2_j$ in S2. We want to avoid any new non-determinism by construction. For instance, for a given state <$S1_i$, $S2_j$>, if there exist a transition $S1_i - a\varnothing S1_k$ in T1 and a transition $S2_j - a\varnothing S2_m$ in T2, then the state <$S1_k$, $S2_m$> is added into St3 and the two transitions are combined into one transition <$S1_i$, $S2_j$>$-a\varnothing$<$S1_k$, $S2_m$> in T3. This is the case when S1 and S2 have a common trace from their initial state to $S1_k$ and $S2_m$, respectively. Another illustration of this construction, if for a given state <$S1_i$, $S2_j$>, there exists a transition $S1_i - a\varnothing S1_k$ in T1, but there is no transition labelled by "a" from $S2_j$ in T2, then the state $S1_k$ is added into St3 and the transition $S1_i - a\varnothing S1_k$ in T1 yields the transition <$S1_i$, $S2_j$>$-a\varnothing S1_k$ in T3. The transitions from a *simple* state in St3, like state $S1_k$ for instance, remain the same as defined in S1. The states reached by these transitions are

added into St_3, except for the intial state which is replaced by the initial state $<S1_o, S2_o>$ of S_3.

Begin of the algorithm

The set of states St_3 and the transition relation T_3 are determined recursively as follows:

For every state St in St_3,

- if $St = <S1_i, S2_j>$ then

for every transition $S1_i - a\varnothing S1_k \in T_1$, with $S1_k \in St_1$

if $S2_j - a\varnothing S2_m \in T_2$, then add $<S1_k, S2_m>$ to St_3 and add $<S1_i, S2_j> - a\varnothing <S1_k, S2_m>$ to T_3

else if $S1_i - a\varnothing S1_o \in T_1$ and $S2_j$

Error!

else if $S1_i - a\varnothing S1_k \in T_1$, with $S1_k \neq S1_o$ and $S2_j \neq a\varnothing$ in T_2, then add $S1_k$ to St_3 and add $<S1_i, S2_j> - a\varnothing S1_k$ to T_3

else if $S1_i$

Error!

and same for every transition $S2_j - a\varnothing S2_m \in T_2$, with $S2_m \in St_2$.

- if $St = S1_i \in St_1 - \{S1_o\}$

for every transition $S1_i - a\varnothing S1_o \in T_1$, add the transition $S1_i - a\varnothing <S1_o, S2_o>$ to T_3,

for every transition $S1_i - a\varnothing S1_k \in T_1$, , with $S1_k \neq S1_o$, add state $S1_k$ to St_3 and add transition $S1_i - a\varnothing S1_k$ to T_3.

- if $St = S2_j \in St_2 - \{S2_o\}$

for every transition $S2_j - a\varnothing S2_o \in T_2$, add transition $S2_j - a\varnothing <S1_o, S2_o>$ to T_3,

for every transition $S2_j - a\varnothing S2_m \in T_2$, , with $S2_m \neq S2_o$, add state $S2_m$ to St_3 and add transition $S2_j - a\varnothing S2_m$ to T_3.

- The construction of St_3 and T_3 ends, when these sets remain the same for two successive steps and all the transitions for all states in St_3 have been considered.

End of the algorithm

To illustrate the FLTSs merging algorithm, we consider again the example S1 of Fig.2.2., which represents the behavior of a vending machine object offering a Candy to the user after the Coin insertion. We want to add a new functionality to our vending machine. The

FLTS S2 of Fig. 3.1.(a) describes this new functionality. Applying the FLTS merging algorithm described above leads to the new FLTS S3 of Fig.3.1. (b). S3 is an extension of both S1 and S2. It has both functionalities, without feature interaction problem.

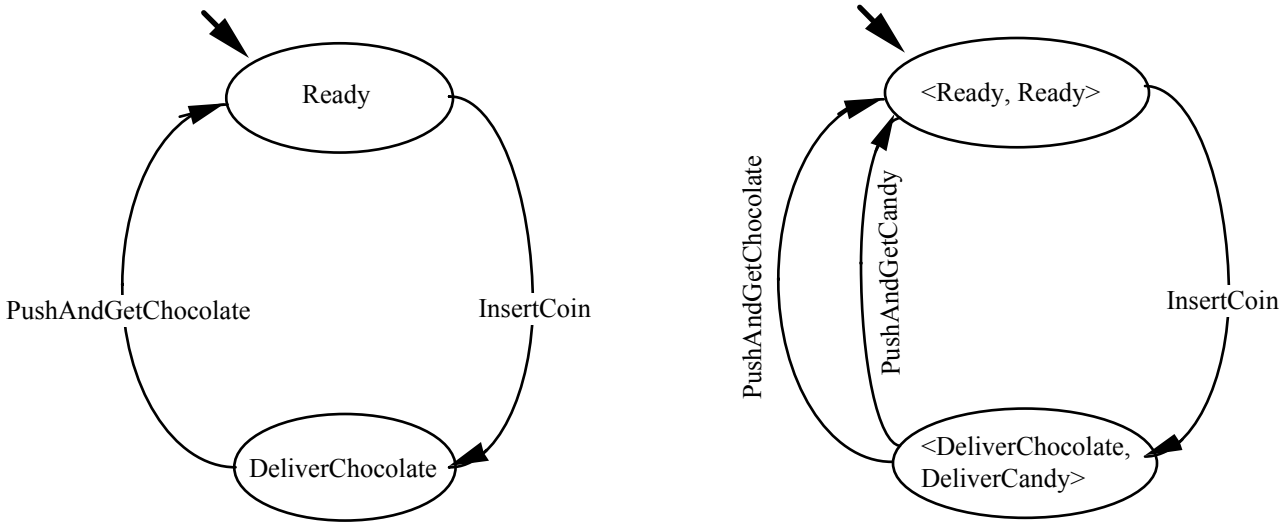

Fig. 3.1. **(a)** FLTS S2 **(b)** FLTS S3 an extension of S1 and S2

4. Dynamic extension of object-oriented specifications

4.1. RMondel facilities

In the formalism used to define the semantics of *Mondel*, the type definitions are static and used as templates for instance creation. Only the instances of a type are considered as objects. To support the construction of dynamically modifiable specifications we need to have access to type definitions of the specification during execution-time. For this purpose, reflection is a promising choice [Ibrahim, 90 #963; Ibrahim, 91 #981].

To define a reflective architecture, one has to define the nature of meta-objects and their structure and behavior. In addition, one has to show how the handling of objects communications and operations lookup are described at the meta-level. Therefore, we developped RMondel [Erra 90], a reflective version of Mondel. In *RMondel*, types are used for structural description (i.e., for the definition of the structure of object instances and of applicable operations), and interpreters are used for the behavioral description of their associated objects called *referents*. This approach shows many advantages:
- types are objects.

- operations for type modifications can be defined at the meta-level.
- an object behavior can be monitored and/or modified by its interpreter.
- new communication strategies can be defined.

The most important aspect of reflection in *RMondel*, is that each object is an instance of a type, and types are object instances of a meta-type called *Modifiable-Type* which is a subtype of the meta-type*TYPE.* Some aspects of the *TYPE* and *Modifiable-Type* definitions are given in Figure 4.1. Another aspect is that the *RMondel* statements and expressions are objects. More details on the *RMondel* definition and its semantics are given in [Erra 92b].

```
type TYPE = OBJECT with
        TypeName          : string;
        BehaviorDef       : var[Statement];
        DirectSuperTypes  : set [TYPE];
        Attributes        : set [AttributeDef];
        Operations        : set [Operation];
        Procedures        : set[Procedure];
        . . .
 operation
        . . .
        {the operation New creates an object according to RMondel object structure}
        New  : OBJECT;
        {the operation LookUp checks if the operation "OpName" is defined for an object's type or for
         one of its supertypes; then returns the associated statements}
        LookUp (OpName : string) : Statement;
Behavior
        { the semantics definitions of the above operations}
endtype TYPE

{ the class of modifiable types is defined as a subclass of TYPE as follows }
type Modifiable-Type = TYPE with
        FLTS-Merge(S: Statement);
        AddAttr (A:Attribute);
        AddOper(O:Operation);
        AddProc(P:Procedure);
        ...
 invariant
        { We define here, the structural consistency invariants which correspond to the static semantics rules
          checked by the Mondel compiler. }
 behavior
        { The semantics definitions of the modification operations above . }
endtype Modifiable-Type
```

Figure 4. 1. Some aspects of the *TYPE* object specification.

Since the type and behaviors are objects, a given behavior can be extended by providing the additional behavior as a parameter of the FLTS-Merge operation as shown in Fig.4.1. The FLTS-Merge operation is the RMondel specification of the merging algorithm described is Section 3. When a type t accepts the operation FLTS-Merge, then its behavior defined by the attribute *BehaviorDef* (see the definition of *TYPE* in Fig.4.1) will be merged with the behavior object given as a parameter of the FLTS-Merge operation. The result will be the

update of the *BehaviorDef* of t according the the extension accomplished by the FLTS-Merge algorithm.

4.2. Maintaining the specification consistency

The modification of the structure and behavior of types, must be done without resulting in type checking errors, run-time errors, new deadlocks, or any other uncontrollable situation. So the semantics of type changes should ensure that a modified type t leads to a type t' which conforms to t. The conforms to relation is defined in terms of both structure and behavior. An object type t' conforms to an object type t if the interface (i.e., attributes and operations signatures) of t' is compatible with the interface of t and the behavior of t' extends the behavior of t. Therefore, we distinguish the structural consistency (i.e., interface compatibility) and the behavioral conformance (i.e., behavior extension) [Erra 92a].

The behavior of objects is to some degree dependent upon preserving *structural consistency*. For instance, when an operation is called on an object, the associated code to be executed is determined by the object's type or supertypes. Additionally, once the operation code is located, its implementation is dependent on the called object' structure. This structure has to be present in all objects that are instances of the type where the operation is defined. So, changes to the type interface may lead, in most cases, the user to change the behavior definition accordingly. In the following we address the issue of dynamic checking of structural and behavioral consistencies of the new specification.

Structural aspect: The main question here is: if we replace a type definition t by t' in some specification S, where t' is *structurally consistent* with t, does the resulting specification S' remain consistent w.r.t. S? The specification S' is consistent with S if the modification does not introduce compiling errors (type checking). Therefore we assert that the obtained specification S' remains consistent. This assertion can be proved according to assignment and parameter passing where type checking is important. In addition we have defined a set of invariants which ensure the structural consistency of a created or modified type [Erra 92b].

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 no, and this is proved by the following counter example: Consider the behavior of S, which is defined by the

parallel composition of two behaviors Bt1 and Bt2, where Bt1 is defined by a type t1 and Bt2 is defined by a type t2. Suppose we extend Bt2 to obtain Bt2', see Fig.4.2. Now if we compose Bt1 with Bt2', to obtain S', the resulting behavior may refuse to perform action b after action a, whereas the original specification never refuses to perform action b after a.

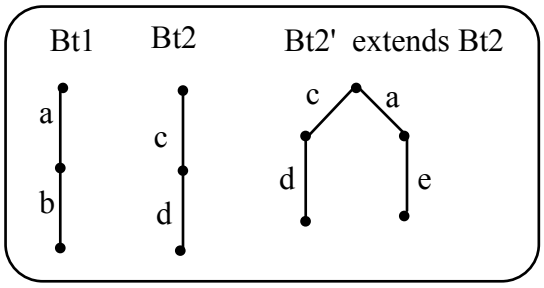

Fig.4.2. labelled transition systems of Bt1, Bt2, and Bt2'.

In order to ensure the consistency of the whole specification after its modification, we use the concept of transaction to provide fail-safe specifications. The user formulates his requirements within a transaction which consists of type update operations. In the following, we introduce a transaction mechanism and explain how the consistency of the whole specification can be dynamically checked.

5. The transaction mechanism and the locking protocol

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

According to the updates of a type T, its existing instances must be converted accordingly. When a type has to be updated, its instances must be locked until the type modifications are accomplished. If the updates do not succeed, e.g., because of invariant violation, then the type will be rolled back to its state before the updates, and the instances will be released to pursue their behavior progress. In the case where the type updates succeed, the instances will be converted accordingly, and released to behave normally. Each object can be active, passive or locked. The object state/transitions are shown in Fig.5.1. Object instances can be ready for convertion, only when they enter their locked state.

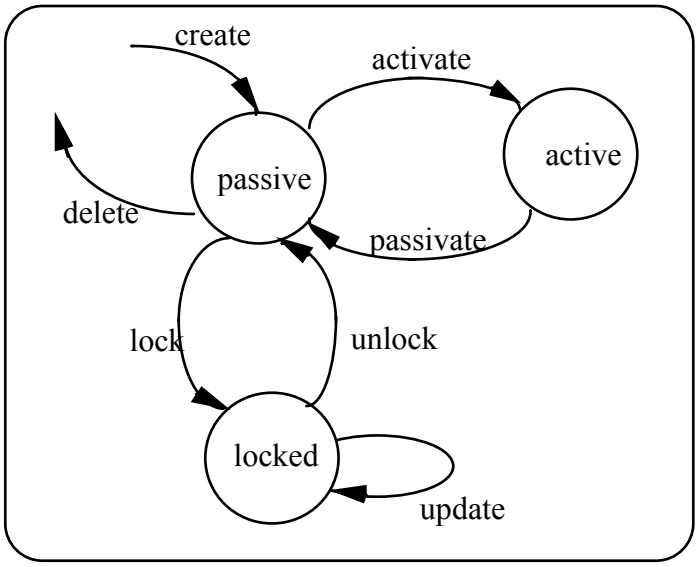

Fig.5.1. Object state/transitions

The fact that a specification is organized as a type hierarchy has a major impact on the locking protocol. The modification operations can 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. The instances of a locked type will be locked until their type becomes unlocked. Fig.5.1 shows the possible states and transitions of an object w.r.t. modifications. Note that objects (i.e., either types or their instances) can be modified only when they are locked.

5.2. The transaction steps

A transaction is constructed based on the modification primitive operations that consist of several successive modifications of one or more object types. The following steps show how the different actions (i.e., those involved in a type updates) work and lead to a consistent specification.

Step 1: *Transaction construction*: through an interface object, the user formulate a transaction (called an atomic operation in *Mondel*) as an operation call, specifying his requirements. In the case of behavior extension, the transaction mainly consists of a call to the operation FLTS-Merge defined within the meta-type Modifiable-Type as shown in Fig. 4.1. The FLTS-Merge operation takes the new behavior which will be merged with an existing one. The new behavior is specified as an object of type Statement. In addition, the

modification of the type structure if any, must be specified in terms of the type update primitive operations (AddAttr, AddOper...). For instance, in the example of Fig.3.1.(b) the extension of the machine behavior requires a change to the machine interface. Therefore, the interface of the type object Machine of Fig.2.1 will be updated by adding the operation PushAndGetChocolate. This update is accomplished by the following operation call:
Machine ! AddOper(PushAndGetChocolate), where Machine is a reference to the type object Machine.

Step 2: *Checkpoint*: This step consists of saving the state of the type subhierarchy and the object instances of those types in the subhierarchy. Then, apply the locking protocol to prevent inconsistent use of the type to be modified and of its instances.

Step 3: *Structural consistency checking:* the checking process consists of maintaining the structural consistency, after the type modifications, according to the set of predefined invariants (See Fig.4.1). Such invariants correspond mainly to the static semantic rules of the language. The structure of a specification (i.e., modified or newly constructed specification) mustt comply with those invariants. Otherwise the user is informed of which part of his transaction does not satisfy the invariants. Then the user has to modify his transaction restarting from step 1 to make the specification comply with the invariants.

Step 4: *Behavioral conformance checking*: The behavioral conformance deals with the dynamic behavior of objects. The behavioral conformance relation is dynamically checked using a dependency graph and the reachability analysis techniques. 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 a unexpected deadlock situation is detected, then the system reflects the inconsistencies and the modified type must be revised again through step 1.

Step 5: *Instances conversion:* when the type modification transaction succeeds, (i.e. the structural consistency and the behavioral conformance relations hold) then the instances (locked previously), must be converted to remain conform with their modified type. The conversion of the instances according to the semantics of each type evolution primitive operation, is described in [Erra 92b]. Then, the type subhierarchy and the instances are unlocked, after their modifications, and enter their passive state (see Fig.5.1).

6. Related Work

In [Ichi 90], the problem of incremental specification in LOTOS has been approached. They have considered only a certain kind of non determinism. The internal action is not taken into consideration. They introduced a new LOTOS operator "&", called specification merging operator. Moreover, B_1 & B_2 describes a behavior where the environment has to choose behavior B_1 or behavior B_2 once and for all. B_1 & B_2 may behave only as B_1 once the environment has chosen B_1. It will not offer B_2 after behaving as B_1.

In [Rudk 91] the notion of inheritance is defined for LOTOS. It is seen as an incremental modification technique. A corresponding operator is introduced and denoted by "&". This operator is defined such that if s = t & m , then s extends t and any recursive call in t or m is redirected to s. However strong restrictions are imposed on t and m, such that m should be stable (no internal action as first event), the initial events of m should be unique and distinct from initial events of t, and so on. There is no requirement such that s should also extend m, and no considerations to the structure of t or how this modification m is propagated to the processes in t.

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

In the area of object oriented databases, class modifications have been extensively studied in the recent literature [Bane 87], [Penn 87], [Skar 87], and [Delc 91]. The available methods determine the consequences of class changes on other classes and on the existing instances, so that possible violations of the integrity constraints can be avoided. These approaches deal mainly with sequential systems and have focused on preserving only structural consistency. In our approach, we address both the structural and behavioral consistencies. For the behavioral consistency we deal mainly with object behaviors and we consider some properties of distributed systems such as blocking. The methodology of Skarra and Zdonik [Skar 87] implements class modification by the use of versions and goes

a long way toward preserving behavior for sequential systems. However, we are exploring solutions to type modification, in a distributed environment, that do not require versioning. Moreover, we use reflection which provides a flexible and uniform environment for dynamic type specifications as well as their modifications using specific meta-operations and meta-objects.

7. Conclusions and discussions

Dynamic type extension is an interesting and challenging research problem. Object oriented systems in conjunction with reflection and formal methods, allow us to approach this problem that conventional systems have not been able to adress.

In this paper, we have approached the features interactions problem from the formal point of view. We have developed an approach and mechanisms for the dynamic extension of distributed system specification, especially, object oriented system specifications in the context of Mondel language. We used the Labelled Transition Systems to model the objects behaviors and we have designed an algorithm for the extension of finite state Labelled Transition Systems. We introduced reflective version of Mondel and the transaction mecanism to allow dynamic extension and to preserve specification consistency.

We are working toward applying this approach for adding new features dynamically to a telephone system specification.

References

[Bane 87] J. Banerjee, W. Kim, H. J. Kim and H. F. Korth, *Semantics and implementation of schema evolution in object oriented databases*, in Proceedings, ACM SIGMOD Int. Conf. On Management of Data, San Fransisco, CA, May 1987, pp. 311-322.

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

[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.,

[Bolo 87] T. Bolognesi and E. Brinksma, Introduction to the ISO specification Language LOTOS, Computer Networks and ISDN Systems, Vol. 14, No. 1 pp. 25-59, 1987.

[Bowe 89] T. F. Bowen et al., The feature interaction problem in telecommunication systems, Proceedings of the Software Engineering for Telecommunication Switching Systems, 1989, pp. 59-62.

[Brin 86] E. Brinksma, G. Scollo and S. Steenbergen, LOTOS specifications, their implementations and their tests, Protocol Specification, testing and verification, VI, Montréal, Canada, 1986, Sarikaya and Bochmann (eds.).

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

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

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

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

[Hoar 85] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[Ichi 90] H. Ichikawa, K. Yamanaka and J. Kato, Incremental Specification in LOTOS, Protocol Specification, Testing and Verification X (1990), Ottawa, Canada, Logrippo, Probert and Ural (ed.).

[ISO 8807] ISO - Information Processing Systems - Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.

[Kell 76] R. Keller, Formal verification of parallel programs, Comm. of the ACM 19 July 1976, pp. 371-384.

[Khen 92a] F. Khendek and G. v. Bochmann, Extending finite state system specifications, Technical report, UdeM.

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

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

[Lang 90] R. Langerak, Decomposition of functionality : a correctness-preserving LOTOS transformation, Protocol Specification, Testing and Verification X (1990), Ottawa, Canada, Logrippo, Probert and Ural (ed.).

[Miln 80] R. Milner, A calculus of communicating systems, Lecture Notes in Computer Science, No. 92, Springer Verlag, 1980.

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

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

[Rudk 91] S. Rudkin, Inheritance in Lotos, Formal description technique (FORTE), Sydney, Australia, 1991, pp. 415-430.

[Skar 87] A. H. Skarra and S. B. Zdonik, *Type evolution in an Object-Oriented Databases*, Research directions in object-oriented programming, Eds. Peter Wegner and Bruce Shriver, MIT press, pp.393-415.

[Viss 88] C. Vissers, G. Scollo and M. v. Sinderen, *Architecture and Specification Style in Formal Descriptions of Distributed Systems*, Proc. IFIP Symposium on Prot. Spec., Verif. and Testing, Atlantic City, 1988.

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