計算機アーキテクチャ 104−12
システムソフトウェアと
オペレーティング・システム 62−12
（１９９４．１．２８）

Formal Framework for Dynamic Extension of Distributed Systems Behaviors

Issam A. Hamid
Tohoku University of Art & Design
Department of Information Design
200 Kamisakurada, Yamagata, JAPAN

abstract:

The evolution of specifications is necessary to accommodate the evolution of requirements and design decisions during the software development and maintenance process. Among the possible modifications, the addition of new features is an important issue. The effort for adding features to telecommunication system, like adding new functionalities to any large distributed software system might be tremendous. Each new features 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-lived 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 of extensions dynamically, without interrupting the processing of those parts of the system which are not affected. we describe a formal approach for extending specification behaviors, and a methodology for dynamic evolution of specifications in the context of a reflective object-oriented specification language *(RMondel)*.

分散システム行動のダイナミックな拡張のための形式上の骨組み

アイサム. A. ハミド
東北芸術工科大学
情報デザイン学科
山形市上桜田２００番地

仕様書の発展は、必要条件の発展とソフトウェア開発とメンテナンスプロセスの間の設計決定を調節するために必要である。可能な修正の間で新しいフィーチャーの追加は、重要な問題である。テレコミュニケーションシステムにフィーチャーを加えることのための努力は、つまり大きい分散ソフトウェアシステムに新しい関数を追加するような努力は、難しいかもしれない。各々の新しいフィーチャー、または加えられた関数は多くの存在しているフィーチャーと共に相互に作用するかもしれない。そのような相互作用は、状況（例、デッドロック）システム故障を妨げることの結果となるかもしれない。それに加えて大規模な長生きの分散システムにとって、その拡張を可能とするために、全体のシステムを止めることは、難しい。それゆえ、重要で難しい問題は、影響を及ぼされないシステムの他の部分の処理を中断することなしに、動的に拡張の修正をすることである。我々は反射するオブジェクト指向仕様書言語（ RMondel ）において、仕様行動を拡大するための形式上の方法と、仕様書ダイナミックな発展のための方法論を記述する。

1- Introduction[1]

Distributed system may evolve through the addition of new functionalities according to new requirements, or through the modification of existing functionalities. New functionalities added to given system may interfere with the existing ones. 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}. For large long-lived 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.

We have developed a new object-oriented specification language, called *Mondel* [Boch 90], which has important concepts as a specification language, for application in the area of distributed systems. It has a formal semantics, expressed by means of a translation into a labeled 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 object and transaction facilities, and (d) supporting the object concept. In order to allow for the construction of dynamically modifiable specifications, we need to have access to, and be able to modify, specifications during execution-time. We developed *RMondel* , a reflective version of *Mondel* , which provides facilities for the dynamic modification of specifications. 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.

2. General framework

2.1. The object model

In this section we briefly recall the fundamental concepts of the object model which are relevant for our discussion. An object oriented specification is described as a collection of objects. An 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 (called *class*) that specifies the properties that are satisfied by all its instances. These properties include the **interface,** that is, the visible attributes and operations and their results and parameter types, and the object **behavior,** which defines the possible order of operation executions, related internal state changes, and the results returned by operation calls, if any. An important aspects of the object model is the inheritance relationship. The inheritance relation between type definitions of a specification leads to a type lattice. A node in the lattice represents a type and an edge between a pair of nodes represents the inheritance relationship.

A very important principle in distributed system specifications is communications. We assume that objects interact by operation calls. An operation call is a request for the called object to execute the appropriate statements or procedures. The caller object explicitly mentions the identifier of the called object and the operation name with the appropriate parameters, if any. The caller object blocks until the callee has returned an answer. In our model, objects are running in parallel, except during communication.

2.2. Constraints for modifications

We consider in this section several kinds of constraints that may be imposed on modifications of the system specification. These constraints are introduced in order to ensure a kind of "property conservation", namely that the modified system conserves certain important properties that are already verified for the original specification. We therefore assume in the following that the modified system specification is a kind of "specialization" of the original one. We say that an object type **C' specializes** a type C if an object instance of type C can be replaced, within the overall system, by an instance of the type C' without invalidating the important system properties. In the following, we distinguish two kinds of constraints.

2.2.1. Consistency requirements

Any specification should satisfy certain consistency requirements which depends on the specification language and the nature of the specified system. We distinguish the following two aspects:

i) **Static requirements**: They correspond to the syntax and semantic constraints imposed by the

---

※1 This reseach is supported by a grand from Ministry of Education of Japan.

specification language. They usually are verified by the compiler and may include type checking rules.

ii) **dynamic requirements**: They relate to the dynamic properties of the system and define certain general properties which should be satisfied by any system, such as the absence of the deadlock.

2.2.2. Conformity constraints

Here we consider those constraints which the modified specification must satisfied in order to conserve the properties of the original specification. We consider for this purpose the subtyping (inheritance) relation defined for object-oriented specification, and impose that the modified version of a type definition should be a subtype of the original one. Since the object properties include both, the interface and the object behavior, we distinguish the following two aspects of conformity:

i) structural conformance: These constraints maintain interface compatibility between a type definition and its modified version. For instance, the type T1 of an attribute A1 of a type definition T may be specialized to a type T1'. In this case, the interface of type T1' should be a subtype of type T1.

ii) Behavioral conformance: It is important that certain semantic constraints between a specification and its extended version are preserved. These constraints ensure that the behavior of the modified behavior is an specialization of the original one [Boch 92]. In the following we consider in particular the case that " specialization" means "extension". For instance, the modified behavior B' must exhibit at least the behavior of the original behavior B without introducing new non-determinism, nor deadlocks.

2.3 Classification of Type changes

For objects-oriented specifications to fulfill their promise for fast prototyping, and easy modifications, a well defined and consistent methodology for type modification must be developed. We consider in the following only changes that satisfy the conformity constraint described above, that means, the modified version T' of the original type specification T should be a subtype of T. Modifications of type definitions are typically achieved by adding or removing attributes and/or operations, modifying behaviors, rearranging inheritance links within the type lattice, etc. We distinguish the following two kinds of modifications:

2.3.1. Structural changes

It is important to note that the existing approuches deal mainly with sequential systems and do not address behavior modifications. An acute problem in designing a methodology for type modification is how to bring existing objects in line with a modified type. Structural type updates may be classified into several categories. In the following, we enumerate the most important update operations:

(1) Modifications to the contents of a node (given type definition) in the type lattice

(i) Modifications to an attribute of a type (1) *Add on attribute A to a type T* , (2) *Drop an existing attribute A from a type T*, (3)*Change the type T of an attribute A*.

(ii) Modification to an operation of a type: (1) *Add the operation O to the type T*, (2) *Drop the existing operation O from the type T*, (3) *Change the signature S of the operation O*.

(2) Modifications to an edge of the lattice

(i) make a type S a supertype of type T

(ii) Delete a parent S (supertype) of the type T

(3) Adding (or removing) a type definition to (from) the type lattice

2.3.2 Behavior changes

The behavior changes may be related to the structural changes described above. For instance, an extension in the form of a new operation in the interface of a type definition requires the definition of its semantics, which implies an update of the behavior definition of the type.

In general, the modified type definition must be obtained in some way during the software maintenance process, usually through the intervention of the designer. We consider in Sec.3 the particular case, where the modified specification S' is obtained through the automatic composition of the original specification S with another additional specification S1, such that S' is the extension of both, the specifications S and S1. This means that the modified system provides the services of both, S and S1.

In the case that the approach of Sec. 3 is used for the construction of the modified specification, the conformity constraints are satisfied automatically, due to the nature of the algorithm used for the construction of the modified specification.

3. Extending distributed system specification behavior

3.1 Communication model

In this section we are concerned with the comparison of the extended behavior of the modified system with a behavior of the original specification. We may consider the following aspect of behaviors:

(a) Depending on the state of the object, which are the

operations that are acceptable, and for which input parameters values?

(b) what is the result returned by an acceptable operation call, depending on the state of the object and the input parameters value?

(c) What is the new state of the object after the execution of an operation, depending on the state of the object and the input parameter values?

Concerning point (a), most object-oriented programming languages assume implicitly that each operation is acceptable in each state of the object. In real-time applications, however, it is often important to restrict the order in which operations may be called. This kind of object communication can be modeled by so-called rendezvous interactions which required not only the readiness of the caller, but also of the callee, for the execution of an interaction.

In this section, we consider a simplified rendezvous communication model which ignores interaction parameters. We assume that the behavior of an object is described by a labeled transition system. Each execution of an operation call is modeled by transition of the system, which, in general, leads to a new system state. The returned result may either be included in the same transition or it may be modeled by a separate subsequent transition.

3.2 Labeled Transition systems

From an abstract point of view, the behavior of a distributed system specification and the behavior of its subsystem specifications can be seen as processes, which are expressed by labeled transition systems(LTS for short). An LTS is a graph in which nodes represent internal state, and transitions represent action occurring during state changes.

Definition 3.1

An LTS TS is a quadrable $<S, L, T, S_o>$, where S; is a (countable) nonempty set of states. L; is a (countable) set of observable actions. $T: S \times L \cup \{\tau\} \rightarrow S$ is a transition relation, where a transition from a state S_i to state S_j by an action $\mu(\mu \in L \cup \{\tau\})$ is denoted by $S_i\text{-}\mu\rightarrow S_j$. τ represents the internal, nonabservable action. S_o is the initial state of TS. #

A finite LTS (FLTS for short) is an LTS in which S and L are finite. We denote by $Tr(S_i)$ for "traces", the set of all sequences of observable actions that can be performed starting in state S_i. The point (".") is used to represent the concatenation of traces. We may also write act (TS), instead of L, to denote the set of observable actions of TS. The behavior specification of a distributed system may be considered as a composition of its subsystem specification behaviors. A mong the possible compositions, the parallel composition operator and the action hiding operator are of special interest in this paper. The parallel composition operator $(B1 \mid \{\mu_1,\ldots,\mu_n\} B2)$ allows to express the parallel execution of the behaviors B1 and B2. B1 and B2 synchronize on actions in $\{\mu_1, \ldots, \mu_n\}$ and interleave with respect to other actions. The hiding operator allows hiding of actions, which will occur as internal actions. We write B/A to denote the hiding of the actions in A in the behavior B. The environment of B will not be able to synchronize with B on these actions.

Intuitively, different LTSs may describe the same observable behavior. Different equivalence relations have been defined based on the notion of observable behavior. The failure equivalence relation is finer than the trace equivalence relation, but coarser than the strong bisimulation equivalence. However, for our considerations in respect to behavioral conformance (see Section 2.2.2), we do not need equivalence relations, but rather ordering relationships. Among them, we have the redaction and extension relations. These relations may serve different purposes during the specification life cycle. The extension relation is most appropriate for extending specification behaviors. Informally, S1 extends S2, if and only if S1 may perform any trace that S2 may perform, and S1 can not refuse what S2 can not refuse after a given trace of S2.

3.3. An approach for merging distributed system specification behaviors

We consider distributed system specification behaviors, which consist of a parallel composition of subsystem specification behaviors. Such specifications have the following from: $S = (S1 \mid_A S2) \backslash B$, where A and B represent sets of actions. The subsystem specifications S1 and S2 may also have the same form as S and so on, until a level where the specifications have no structure and are defined directly in terms of some allowed ordering of actions. These specifications are called basic components, they may be nondeterministic, but are assumed to be finite state. Given a distributed system specification behavior S_{old}, which consists of a parallel composition of subsystem specification behaviors and so on until the basic components, and a new behavior S_{added} to be added to S_{old}. We want to deduce a specification behavior S_{new}, such that S_{new} extends S_{old}, S_{new} extends S_{added}, and S_{new} preserves the structure of S_{old}.

We assume that S_{old} and S_{added} have an identical

structure. In other words, the form of the expression S_{old} is identical to the form of the expression S_{added}. To every subsystem specification in S_{old} corresponds a subsystem specification in S_{added} and vice et versa. To every basic component Ci_{old} in S_{old}, corresponds to a basic component Ci_{added} in S_{added} and vice et versa. If S_{old} and S_{added} consist of parallel composition of subsystem specifications, but their structure are not identical, the structure of S_{added} can be transformed[Khen92b]. If S_{added} is given in a high level form, without an internal structure, it may be transformed into a structure identical to the S_{old} structure using the transformation algorithms described in [Lang90].

Before introducing the algorithm for merging system behaviors, which consists of parallel combination of subsystem behaviors, we describe the basic algorithm for merging behaviors described by simple FLTSs.

3.3.1 The algorithm FLTS-merge

The algorithm FLTS-merge uses an intermediary representation, the Acceptance Graphs (AG for short). The AGs can be manipulated more easily than the LTSs, since the nondeterminism is modeled in the labels of the states and not in the labels of the transitions as for LATSs.

Definition 3.2

An AG G is 5-tuple <Sg, L, Ac, Tg, Sg_o>, where Sg is a(countable) nonempty set of states. L is a (countable) nonempty set of events. Ac: Sg→*P* (*P* (L)) is a mapping from Sg to a set of subsets of L.

Ac:(Sg_i) is called the acceptance set of Sg_i. Tg: Sg x L → Sg is a transition function, where a transition from state Sg_i to state Sg_j by an action a (a∈L) is denoted by Sg_i -a→Sg_j. Sg_o is the initial state of G. #

Given two FLTSs TS1 = <S1, L1, $S1_o$> and TS2= <S2, L2, T2, $S2_o$>, the algorithm FLTS-merge consists, first, to transform the FLTSs TS1 and TS2 into the failure equivalent FAGs G1 = <Sg1, L1, Ac1, Tg1, Sg1o> and G2=<Sg2, L2, Ac2, Tg2, Sg2o>, respectively. The transformation algorithm is very similar to the usual algorithms for the transformation of a nondeterministic automata to a deterministic one.

The FAGs G1 and G2 are then merged into the FAG G3 =<Sg3, L1" L2, Ac3, Tg3, <$Sg1_o$, $Sg2_o$>>, such that a state Sg_i in Sg3 can be a tuple <$Sg1_i$, $Sg2_j$> consisting of state $Sg1_i$ from Sg1 and Sg2j from Sg2 (as for the initial state <Sg1o, Sg2o>) or simple state Sg1i from Sg1 or Sg2j from Sg2. These states and the transitions which reach them are added step by into Sg3 and Tg3, respectively. Initially, Sg3 contains only the initial state <$Sg1_o$, $Sg2_o$>.

The definitions of the transitions from state <$Sg1_j$, $Sg2_j$> in Sg3 depends on the transitions from Sg1i in Sg1 and from Sg2j in Sg2. For instance, for a given state <$Sg1_i$, $Sg2_j$>, if there is a transition $Sg1_i$ - a→$Sg1_k$ in Tg1 and a transition $Sg2_j$ - a→ $Sg2_m$ in Tg_2, then the state <$Sg1_k$, $Sg2_m$> is added into Sg_3 and the two transitions are combined into one transition <Sg1i, Sg2j>-a→<$Sg1_k$, $Sg2_m$> in Tg_3. This is the situation when G1 and G2 have a common trace from their initial state to $Sg1_k$ and $Sg2_m$, respectively. Another illustration of this construction, if for a given state <$Sg1_i$, $Sg2_j$>, there exists a transition $Sg1_i$-a→$Sg1_k$ in Tg1, but there is no transition labelled by a from $Sg2_j$ in Tg2, then the state $Sg1_k$ is added into Sg3 and the transition $Sg1_i$-a→ $S1_k$ in Tg1 yields the transition <$Sg1_i$, $Sg2_j$>-a→$S1_k$ in Tg_3. The transitions from a simple state in Sg_3, like state $Sg1_k$, for instance, remain the same as defined in G1. The states reached by these transitions are added into Sg3, except for the initial state, which is replaced by the initial state <Sg1o, Sg2o> of G3.

The mapping Ac3 is defined as follows: For every state Sgi in Sg3, if Sg_i=<$Sg1_i$, $Sg2_j$>, then $Ac_3(Sg_i)$={X1 ∪ X2 ¦ X1∈ Ac1($Sg1_i$) and X2∈ Ac2($Sg2_j$)}, if Sg_i=$Sg1_i$, with $Sg1_i$∈ Sg1, then Ac3(Sgi)=Ac1($Sg1_i$), if Sg_i = $Sg2_j$, with $Sg2_j$∈ Sg2, then Ac3(Sg_i) = Ac2($Sg2_j$).

3.3.2 The algorithm for merging distributed system specification behaviors

The algorithm for merging distributed system specification behaviors (merge) is recursive over the structure of Sold and Sadded. It is based on the algorithm FLTS_merge, which serves for the merging of the basic components.

```
Begin
merge(S1, S2) =
if S1 = (S11 |A S12)/B, S2 = (S21 |C S22)\D,
        then (merge(S11, S21) |(AUC) merge
    (S12, S22))\ (BUD)
        else FLTS_merge(S1, S2) (* S1 and S2
                are basic components *)
End
```

S_{new}, obtained by merge(S_{old}, S_{added}), has a structure identical to the structure of S_{old} and S_{added}. As basic component, instead of Ciold, it has Ci_{new} which results from the merging of Ciold and Ci_{added} by the algorithm FLTS_merge. Unfortunately, S_{new} does not always

extend Sold and Sadded. Consider the counter example in Fig.1, where S_{old} = ($C1_{old}$ |{g1} C2old)/{g1}, S_{added} = ($S1_{added}$ |{g2} $S2_{added}$)/{g2}. The structure of the specification S_{new} is identical to the structure of S_{old} and S_{added}, but Snew does not extend S_{old} neither S_{added}.

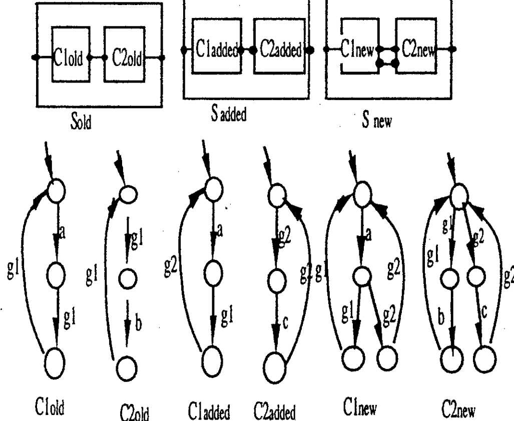

Fig.1 Counter Example

Indeed, S_{old} never refuses the action b after trace a, whereas Snew may refuse action b after trace a. The same observation holds for action c after trace a.

The trace a is common for S_{old} and S_{added}. It is followed by a hidden action g1 in $C1_{old}$ and g2 in $C1_{added}$. The merging of $C1_{old}$ and C1added leads to a choice between the two hidden actions g1 and g2 after the trace a, in C1new. The components $C1_{new}$ and $C2_{new}$ may, internally, choose to synchronize on action g1 or g2, after a trace a, and offer only action b or only action c.

In Theorem 1 , we have stated below the sufficient conditions for S_{old} and S_{added} such that Snew extends S_{old} and also S_{added}. We denote by HG_{old} the set of hidden action names in S_{old}, and by HG_{added} the set of hidden action names in S_{added} .

Condition (a) says that the names of hidden actions in Sadded should not conflict with the names of observable or hidden actions in S_{old}. Reciprocally, the names of hidden action in S_{old} should not conflict with the names of observable or hidden actions in S_{added}. These actions may be renamed without any observable effect, in order to satisfy this condition.

Condition (b) says that there is no observable action of S_{old} and S_{added} shared by two (or ore) basic components of S_{old} (respectively S_{added}). A basic component Ci_{old} in S_{old} may have common observable actions only with the corresponding basic component Ci_{added} in S_{added}, and reciprocally. Conditions (c) and (d) state that S_{old} should not be able to perform an action from HG_{old} before interacting with the environment and S_{added} also should not be able to perform an action from HG_{added} before interacting with the environment, respectively.

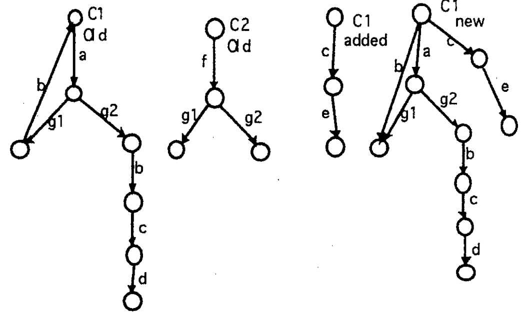

Fig. 2. Illustration for Conditions e-3 and e-4

Condition (e-3) and condition (e-4) are introduced in order to avoid the situations similar to the one shown in Fig. 2. Assume that S_{old}= ($C1_{old}$ |{g1, g2} $C2_{old}$)\{g1, g2} and S_{added} = ($C1_{added}$ | ϕ stop)\ ϕ . The merging algorithm for structured specifications leads to S_{new} = (C1new |{g1, g2} C2new)\{g1, g2}, where C1new is shown in Fig .3 and $C2_{new}$ = $C2_{old}$. We have $C1_{new}$ ext $C1_{old}$ and $C1_{new}$ ext $C1_{added}$ as well as $C2_{new}$ ext $C2_{old}$ and $C2_{new}$ ext $C2_{added}$. However, S_{new} does not extend S_{old}. For instance , after the trace f.a.b.c, S_{new} refuses to perform action d, whereas S_{old} never refuses to perform action d after trace f.a.b.c. S_{new} refuses to perform action d, whereas S_{old} never refuses to perform action d after trace f.a.b.c. This is due to the fact what we have two traces $\sigma 1$ = ag1.b and σ_2 = ag2.b in $C1_{old}$, such that $\sigma 1 \neq \sigma 2$, $\sigma 1 \backslash HG_{old} = \sigma 2 \backslash HG_{old}$, $\sigma 1$ is cyclic, $\sigma 2$ is not cyclic, $\sigma 2$.c is a trace in $C1_{old}$, and c is a trace in $C1_{added}$. It is possible to characterize these situations with weaker conditions than condition (e-3) and condition (e-4) as explained in this example. However the verification of such conditions may be complex, whereas condition (e-3) and condition (e-4) can be checked very easily in the case of FLTSs.

Theorem 1

Given S_{old} in the required hierarchical structure with the basic components $C1_{old}$, $C2_{old}$,..., Cn_{old}, and S_{added} with an identical structure and the basic components $C1_{added}$, $C2_{added}$,...Cn_{added}, S_{new}=merge(S_{old}, S_{added}), and for i=1,...,n, Ci_{new}=FLTS_merge(Ci_{old},Ci_{added}),

We have that S_{new} extend S_{old} and S_{new} extends S_{added}, **if** the following conditions are satisfied:

(a) $\forall i$, i=1,.....n, $act(Ci_{old})$ ($HG_{added}=\phi$), and $act(Ci_{added})$ ($HG_{old}=\phi$),

(b) $\forall i, j, i \neq j$, $(act(Ci_{old}) \cup act(Ci_{added})) \cap (act(Cj_{old}) \cup act(Cj_{added})) \cap (act(S_{old}) \cup act(S_{added}))\ \phi$,

(c) $\exists\ Ci_{old}$ and Cj_{old}, such for some $g \in HG_{old}$, $g \in Tr(Ci_{old})$ and $g \in Tr(Cj_{old})$,

(d) $\exists\ Ci_{added}$ and Cj_{added}, such for some $g \in HG_{added}$, $g \in Tr(Ci_{added})$ and $g \in Tr(Cj_{added})$,

(e) $\forall i$, i=1,...,n,

(1) $\forall \sigma \in Tr(Ci_{old})-\{\varepsilon\}$, $\exists \sigma.x$ $Tr(Ci_{old})$ with $x \in HG_{added}$, (2) $\forall \sigma \in Tr(Ci_{added})-\{\varepsilon\}$, $\exists \sigma.x$ $Tr(Ci_{old})$ with $x \in HG_{old}$, (3) $\forall a \in act(S_{old})$, if $a \in Tr(Ci_{old})$, then $\exists \sigma.a \in Tr(Ci_{added})$, unless σ is cyclic in Ci_{added}, (4) $\forall a \in act(S_{old})$, if $a \in Tr(Ci_{added})$, then $\exists \sigma.a \in Tr(Ci_{old})$, unless σ is cyclic in Ci_{old}. #

4. Dynamic modification in an object-oriented environment

To make dynamic modifications to an executable specification without interrupting the processing of those parts of the specification which are not directly affected by the change, we use the concept of transaction to provide fail-safe specifications. Modifications are performed within a transaction. The transaction concept is well known for database systems. Transactions serve three distinct purposes: i) they are logical units that group together operations comprising a complete task; ii) they are atomic units whose execution preserves the consistency of the system; iii) they are recovery units that ensure that either all the steps enclosed within them are executed or none. The principle of transactions is that if the system is in a consistent state before a transaction starts execution, it will be in a consistent state when the transaction terminates. In order to ensure the specification consistency, we have defined a set of structural and behavioral invariants. A transaction commits when the invariants are satisfied after the modifications, or aborts whenever these invariants are violated.

4.1. Locking Protocol

To isolate the parts of the specification which are affected by the modifications, we define a locking protocol. According to the updates of a type, 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 then released. Each object can be active, passive or locked. The object state/transitions are shown in Fig. 3. Object instances are ready for conversion only when they enter their locked state. Thus not only the instances of the modified

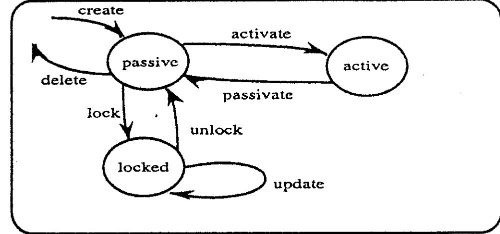

Fig. 3 Objects state/transitions

types 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.3 shows the possible states and transitions of an object w.r.t. modifications.

4.2. Transaction Mechanism

The user formulates his requirements within a transaction which consists of type update operations. The following steps show how the different actions, involved in a type updates can lead to a consistent specification.

Step 1: *Transaction construction* : the user formulates a transaction specifying his requirements (i.e., in terms of operations for type modifications).

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.

Step 3: *Modifications performed* : This step consists of 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 modification are performed on these types without changing their identities.

Step 4: *Consistency Checking* :

-Structural checking : the checking process consists of maintaining the structural consistency, after the type modifications, according to the invariants which correspond mainly to the static semantic rules of the language. If the structure of a specification does not comply with those invariants, then the anomalies are reported in order to inform the user which part of his transaction does not satisfy the invariants. Then the user has to modify his transaction (from step 1), in order to make the specification comply with the invariants.

-Behavioral checking : This check deals with the

behavior specification of the system. The sufficient conditions of theorem 1 introduced in Sec.3 are modeled as invariants which must hold along the transaction. In the case where an invariants are violated, one cannot deduce that the new specification does not extend the original one.

Therefore, these two specifications may be compared using an approach such as the reachability analysis technique.

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.

Step 6: *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.

5. Reflective Framework

5.1. An overview of Mondel

We have developed *Mondel* , an object-oriented specification language with certain particular features, such as multiple inheritance, type checking, rendezvous communication between objects. *Mondel* is particularly suitable for modeling and specifying applications in distributed systems. *Mondel* has a formal semantics which associates a meaning to the valid language sentences. The *Mondel* formal semantics was the bases for the verification of *Mondel* specifications, and has been used for the construction of an interpreter.

Each *Mondel* object has an identity, a certain number of named attributes 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 (called *class* in most object-oriented languages) that specifies the properties that are satisfied by all its instances.

5.2. RMondel facilities

To define a reflective architecture, one has to define the nature of meta-objects and their structure and behavior. In addition, one has to show how the handling of inter-object communication and operation lookup are described at the meta-level. Therefore, we developed RMondel, a reflective version of Mondel . In *RMondel* , types are used for structural description and interpreters are used for the behavioral description of their associated objects called referents. This approach shows many advantages:

The most important spect 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 Fig.4. Another aspect is that the *RMondel* statements and expressions are objects.

Since the type and behaviors are objects, a given behavior may be extended by providing the additional behavior as a parameter for the FLTS-merge operation as shown in Fig.4. The FLTS-merge operation is the RMondel representation of the FLTSs merging algorithm described in Sec. 3.3.1. When a type t accepts the operation FLTS-merge, then the behavior of t defined by the attribute *Behaviore Def* (see the definition of *TYPE* in Fig.4) will be merged with the behavior object given as a parameter of the FLTS-merge operation. The result will be update of the *Behavior Def* of t according to the extension accomplished by the FLTS-merge algorithm.

6. Conclusions

We have developed a formal approach and mechanisms for the dynamic extension of distributed system specifications, especially, object oriented system specifications in the context of Mondel language. In this paper, we have used the extension relation as a formalization of the addition of new functionnalities to a given specification. However, other relations may be considered for behavior and extensibility. Mondel has been implemented on a sun workstation, and used for simulating the specifications of the OSI directory system, and personal communication services.

```
type TYPE = OBJECTwith
     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 distin
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. 4 Type Object Speification