A model for evolution of services in distributed systems

T. Senivongse and I.A. Utting
University of Kent at Canterbury
Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK. Tel: +44 (0)1227 764000. Fax: +44 (0)1227 762811.
email: {tsna,iau}@ukc.ac.uk

Abstract

Large software systems are never static. They exist in an environment that is subject to constant changes in both functionality and technology. This is particularly a problem for large-scale distributed systems, where different components may be subject to different, divergent, pressures. This paper describes work carried out on an RM-ODP-based model for evolution of distributed application services under such conditions. It examines problems that may occur when services evolve to their new versions without corresponding evolution in their clients and presents a mechanism to preserve such old services as are still needed. The model uses the idea of 'mapping' to provide the impression of old services using their new versions. A prototype of the model has been implemented on the ANSAware platform, giving an example of how the model can be applied in a working distributed environment.

Keywords

Evolution, transparency, RM-ODP, interoperability, distributed systems

1 INTRODUCTION

One of the causes of the current trend towards distribution of information processing system components is the need to exchange information between interconnected systems both within one organisation and between cooperating organisations. A major requirement for the construction of such distributed systems is to mask from clients details and differences in the mechanisms used to provide a particular service. This leads to the goals of 'transparency', typically hiding details of location, failure, and access mechanisms. To these familiar transparencies, we add 'evolution', hiding from clients (where necessary) the details of the changes occurring to a service over time.

1.1 Problems in evolution of distributed services

Lehman and Belady (1985) remarked that software systems are never completed; they just continue to evolve. Requirements for new functionality or for improvements in the system are likely to result in modification in the design and implementation of a service. Such modification often has effects on the environment and poses the following problems:

- As client/server systems are loosely-coupled and autonomous, but share resources, there is a conflict between the needs of servers and clients. If a particular system, providing a shared service that can be accessed both locally (from the server's own domain) and remotely, needs to modify the service it provides, clients will be affected by the change unless the new service is strictly compatible with the old one.
  Evolution usually preserves the functionality of old services but possibly results in strictly incompatible new services. Incompatibility means the new version cannot directly substitute for the old counterpart, as clients are not able to interact with it in the same way. Some provision is consequently required for clients to continue using the service.
  Normally, all should be fine for local clients because they and the server follow the same local policy, and any changes in the service should be propagated to them. Remote clients may be tied to a different policy and may prefer not to change anything; the old service may serve them well enough for their requirements. It may even be that clients are not aware of changes in remote services at all. In large-scale distributed systems, informing all clients of changes to all the services they use may impose an intolerable load on the maintainers of client systems.
  This problem is a result of a limited view of the compatibility relationship in distributed object models which is usually restricted to incremental subtyping. A more general meaning for compatibility (i.e. compatibility of functionality) needs to be assumed rather than what interface subtyping normally provides.
- Although evolution of services is a useful activity as it leads a system to an appropriate current state, it is a complicated and inconvenient process. Apart from introducing a new service, the evolver who arranges the evolution may have to decide what should be done to the old service and whether clients need any notification of the changes.
  In the case of completely incompatible new versions, changes must be reported to all clients so that any necessary conversions can be made to them to upgrade their behaviour appropriately. The evolver will also need to consider whether and how evolution of a service has an impact on other services. Subtype services are usually affected by evolution of their supertypes as they inherit behaviour from them.

In practice, the problem of incompatibility is common among different areas of interest. W. Brookes and Yang (1995) studied differences in the type systems of various distributed platforms that inhibit interoperability between distributed domains, comparing type systems that differ in both syntax and semantics, and including RM-ODP-based system, ANSAware, DCE, CORBA as well as Liskov and Wing's work (Liskov and Wing, 1993) and America's work (America, 1991) on type systems in object-oriented applications. The difficulty addressed arises when applications from different domains with different type models must interwork when selection and matching of resource types can cross platform boundaries. The proposed solution assumes one-to-one mappings between type models with a type manager in each type domain querying other federated type managers about the differences among type systems and maintaining information for automatic inter-domain type matching.

There have also been efforts to accommodate schema evolution in object-oriented databases by class modification (J. Banerjee and Korth, 1987; Penney and Stein, 1987) and class versioning (Skarra and Zdonik, 1988; Björnerstedt and Hultén, 1989; Monk and Sommerville, 1993). Unlike class modification which takes an existing class definition and converts its instances into those of the modified class definition, class versioning allows multiple versions of a class to coexist by retaining the old class definition as well as creating a new version.

The ENCORE database (Skarra and Zdonik, 1988) supports class versioning by using exception handling to cope with mismatches between the version of the object expected by the query and the actual version of the instance. All incarnations of a class definition are collected in a version set whose version set interface contains every attribute, with every value declared valid for it, and every operation ever defined by a version of the class. Each class version is associated with handlers that correspond to the behaviour that is not defined by itself but by others. The AVANCE project (Björnerstedt and Hultén, 1989) adopts a similar approach to ENCORE, using exception handlers to service the query with values appropriate to the version expected by the query. Monk and Sommerville (1993) introduced the CLOSQL system that emphasises changes in the underlying semantics of the data as well as the structure of the database. The system uses update and backdate functions that are defined between each pair of consecutive versions to dynamically convert a particular instance of a version to that of the version implied in the query.

In the context of databases, it may be possible to allow multiple versions of a class definition and their instances to truly coexist as the nature of the object data is generally static. In distributed environment, it may be the case that only one version (the new version) can be allowed if the old and new versions rely upon the same changing data (e.g. a clock) and the semantics of the object type requires that there is one instance of such data. To avoid the possible inconsistency problem, this paper aims to achieve the virtual coexistence of service versions by giving only the appearance of the availability of the old services.

The work described above resembles that described here in trying to overcome incompatibility by using some form of mapping to support interoperability and to provide *evolution transparency* to clients of distributed services. The model presented in this paper provides client objects with transparency of location, access, and persistence over time. The impression of old services will persist after evolution and clients can still access those services in the same way although the services have migrated (in time) to their new versions. The model aims to resolve the possible conflict between a server and its clients as well as to ease the evolver's job. It focuses on changes of service interfaces that would influence the behaviour of the objects interacting with them, evolution not requiring such changes being handled by the type system of the supporting distributed object model.

1.2 Evolution and distributed processing standards

Even though the idea of service evolution has not been a focus of distributed systems development, standardisation efforts have acknowledged its importance and declared it a feature which distributed systems should support.

RM-ODP

RM-ODP describes evolution as a characteristic of an open distributed system. It supports the idea that a system generally has to face many changes during its working life, which are motivated by technical progress enabling better performance at a better price or by changing goals (ISO/IEC, 1995a). However, it does not impose any requirements on how the system should respond to evolutionary pressures.

RM-ODP defines the concept of *behavioural compatibility* between two objects if the first object can replace the second in some environment without the environment being able to notice the difference in the object's behaviour with respect to a set of criteria (ISO/IEC, 1995b). The environment should be capable of fully exercising the original behaviour without any unexpected results. The criteria may permit *coerced behavioural compatibility* by allowing modification of an otherwise incompatible object so that it behaves as an acceptable replacement.

In practice, a service undergoing evolution is unlikely to maintain behavioural compatibility with its old version because evolution tends to stem from changes in behaviour. To mask the incompatibility from clients who are parts of the environment, some modification or concealment may be necessary to force the new service to imitate the behaviour of its original counterpart.

OMA

The Object Management Architecture (OMA) of Object Management Group (OMG, 1990) defines an extensible and dynamic nature as one of the objectives that technology supporting applications based on distributed interoperating objects should accomplish. That is, it should be possible to dynamically change the implementation of objects without affecting other objects. For the time being, however, OMA emphasises the technology to support adding new implementations (e.g. new classes) and replacing implementations without changing object interfaces. The issue of replacing implementations where the interface is changed is a secondary, and largely unaddressed, goal.

Changes in the interface would affect the environment that interacts with the object and OMA requires that a mechanism be provided so that existing objects can continue to use older implementation, suggesting a versioning mechanism or a dynamic upgrade facility that allows existing objects to use the new interface correctly.

Distributed processing standards have clearly expressed concerns about the evolution of services. Their consideration strengthens the need for a facility to support the evolution activity. Section 2 of this paper presents a model that facilitates this process; section 3 illustrates the model by describing a prototype system that has been implemented on the ANSAware platform; section 4 draws conclusions from the present work and discusses future research directions.

2 THE MODEL

The model proposed to accommodate distributed service evolution is founded on the object-based model described in RM-ODP. The model presents a dynamic upgrade mechanism that coerces compatibility between versions of evolving services. Similarly to the related work discussed in section 1.1, it adopts the idea of using version mappings to allow existing client objects to correctly use new service interfaces without notice of the changes. The model also automates the process of evolution as much as possible as an aid to those responsible for the modification of services.

2.1 The scope and assumptions

The scope of the proposed model is as follows:

- Only evolution in the interface definition of a service type can be managed automatically, since it is only the interface part which is visible to the outside world and by which object compatibilty is determined.
- Only operational interfaces are considered.
- The evolution is restricted to a version chain; that is there is only one new version evolving from an old service type at any one time. However, the new version is not restricted to being expressed in only one new interface type. That is to say that evolution may result in more than one new interface type but there must be only one set of corresponding interfaces that is meant to replace an old interface.
- The model only aims to bridge incompatibility between versions of a service and facilitate the evolution process. It does not guarantee that the incompatibility will be completely transparent, as it is possible that mappings cannot be generated. In such a case, it is the responsibility of the evolver to suggest an appropriate course of action.

The model makes the following assumptions:

- The old and new service versions do not truly coexist. This is to avoid the issue of maintaining consistency between their underlying state. The new version alone is responsible for providing the underlying service after evolution.
- The definition of evolution implies that there are no drastic changes to the service. The new version still preserves, in some sense, the original functionality.
- All newly created object instances are bound to the current version at creation time. Client objects will always refer to the versions of services that are current at the time they are created.
- The evolver must bear in mind when changing to a new version that they need to prepare to cope with the incompatibility that may result from the changes.

2.2 Evolution function

In RM-ODP, a trading function is provided by a third party object to mediate the advertisement of a server object's interfaces and the discovery of those interfaces by client objects. A client obtains a reference to a server's interface that matches its requirements from a trading object and uses the interface reference to interact with the server. Such a request to the trading object would fail if the server's interface type has migrated to a new version incompatible with the original one and the trading object cannot find another compatible service that can fulfil the client's requirements.

The model proposes an *evolution function* that manages the evolution of services so that evolution transparency is accomplished. Its strategy is to conceal the fact of changes in services from their clients by the means of version mapping. It controls the use and the generation of mappings that are the key components in dealing with version incompatibility.

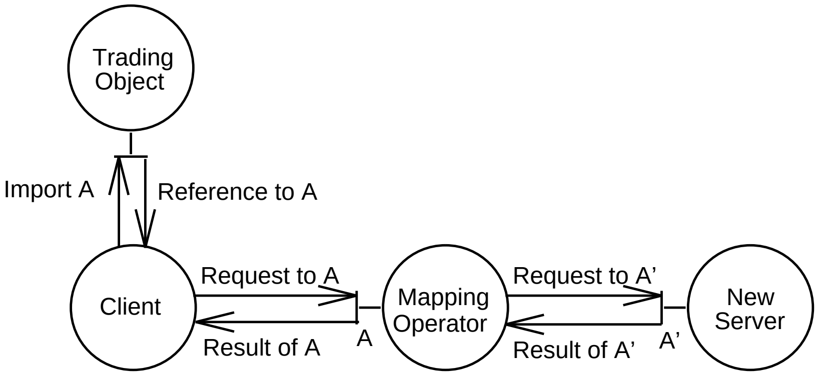

Figure 1 The model for a mapping operator.

The mapping operator

The model uses the existing new server to serve clients on behalf of the old server. Due to the incompatibility between the two versions, the clients access the new service interface indirectly, and conversion is required to transform messages of one version into those of the other version. The evolution function defines such conversion as the responsibility of a *mapping operator*.

The mapping operator is a special service that enforces compatibility in a pair of incompatible service versions. The evolution process requires that the mapping operator object be created at the time a service evolves to a new version and allows the creation to be as automatic as possible.

The mapping operator is the middleman between clients and the new server. Figure 1 shows how it is placed in the general invocation model to support evolution transparency. To provide a client with indirect access to the new interface type (A'), it impersonates a server of the old interface (A) by exporting an offer of the old type to the trading object and, at the same time, acting as a client of the new server. The mapping operator intercepts the client's request to the old interface and communicates with the new server before returning the result of interaction back to the client. It uses mapping functions to generate the new version request out of the old version and to convert the new version result back to the form required by the old version. Thus, the client is given the impression of the existence of the old service.

The evolution function will be more transparent if the mapping operator makes use of the relocation function defined in RM-ODP in such a way that the original service is migrated to the mapping operator object. This helps prevent clients who already possess the service interface reference from noticing the invocation failure that results from the absence of the original interface after the evolution.

The generation of mapping functions

The evolution process requires that mapping functions for a pair of service versions be generated as automatically as possible. Basically, to bridge incompatibility, knowledge of the behaviour of the services is necessary. Since this mapping involves the semantics of services, evolvers must cooperate and supply such information.

Figure 2 depicts two sources of mapping functions. The generation involves the comparison of

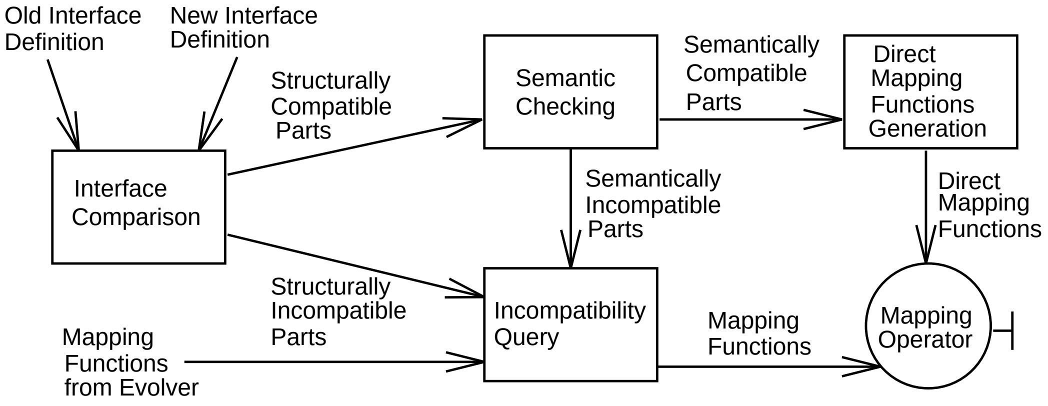

Figure 2 The generation of mapping functions.

the two interface definitions. The new interface type is structurally checked to determine whether it is a subtype of the old interface type using the subtyping rules defined in ISO/IEC (1995c). Structurally compatible definitions need further examination for semantic compatibility. Any incompatibility found in the comparison needs the evolver's involvement to specify appropriate mapping functions in the form of procedural code. The evolution function can often define mapping functions automatically if changes are trivial e.g. semantically compatible. These mapping functions are then used by the mapping operator.

2.3 Supported changes in evolution

A change of type structure (Figure 3) involves changes in the signatures of the operations embedded in the interface, whereas a change in type hierarchy implies changes in behaviour inherited from related interfaces. The node 'Change of Semantics' of a parameter indicates that although a parameter of the operation may look the same as it did before the evolution, its meaning may have changed.

Note that it is possible that changes to the interface definition have an impact on the hidden state of the interface. In this case, the evolver must intervene to maintain cross-version consistency of the state.

2.4 Mapping chain

A service type may evolve several times during its lifetime. Simplistically, as shown in Figure 4(a), the evolver has to go through the evolution process $n - 1$ times, once for each older version when introducing version n of the type. The evolver thus needs knowledge of all versions in the history to resolve all incompatibility effectively. This is particularly difficult if the history line is long or more than one evolver is involved in the evolution history. In addition, all previous mapping operators and mapping functions must be discarded when a new version is created.

Practically, only one new mapping operator and one new set of mapping functions between the last version and the new one are needed at each evolution point (Figure 4(b)). To compute a mapping between version i and j ($i < j$), a composition of all pairwise mappings between

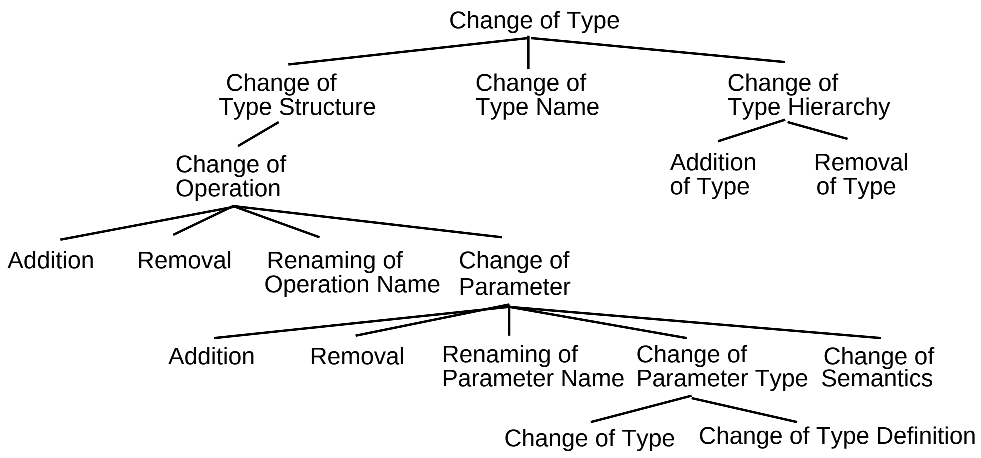

Figure 3 Supported changes to an interface type.

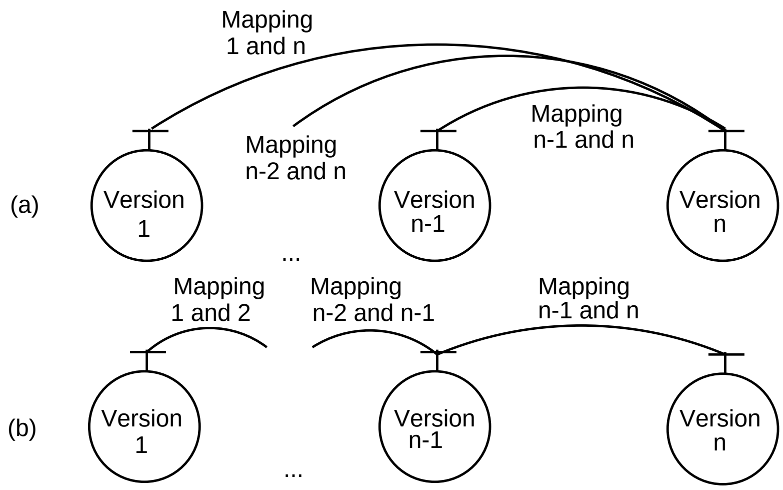

Figure 4 Application of mapping: (a) one-to-one mapping (b) mapping chain.

version i and j is necessary. The mapping chain yields no extra work for the evolver in going back through the history of the service and all mappings are always kept for subsequent use.

2.5 Evolution and inheritance

If an interface type that has a subtype evolves, the subtype also evolves at the same time, as it inherits the supertype's behaviour. Such propagation of changes results in a new version of the subtype that is itself a subtype of the new version of the original type. However, defining a subtype for a type usually requires a deep understanding of the semantics of the two related types. In some cases, it may not be appropriate to propagate changes to the subtype as the type may evolve in such a way that its new version should no longer have such a subtype. A subtype may belong to a particular version of the type and not to all versions.

The model does not require automatic propagation of changes in a type to its subtypes as it may not be correct to automatically assume a relationship between evolving types. In the case that any subtype needs to change accordingly, the evolver has to evolve it separately, ensuring that all mapping functions, created when evolving the supertype, are available for reuse to reflect a mapping for the inherited behaviour that has changed.

3 THE PROTOTYPE

A prototype of the model has been developed on the ANSAware 4.1 platform running on a SUN workstation. This section shows how the model can be put to use in this environment and identifies some limitations of the current system.

3.1 The ANSAware system

ANSAware is a software framework developed to demonstrate and validate the ANSA architecture for open distributed processing (APM, 1992). It provides a basic platform and software development support in the form of system management applications and tools for the design and construction of distributed applications.

A trader in ANSAware provides the trading function referred to above. It organises exported offers of services by type space and context space. The hierarchical type space keeps track of all interface types that the trader manages together with their compatibility relationships. Subtyping in ANSAware is limited to incremental subtyping such that the subtype must have at least the operations that the supertype provides, but at present ANSAware does not provide any mechanism to examine such type conformance. The context space is the hierarchical administrative name space where offers are placed when they are exported. When an offer is published to the trader, its interface type, context and optionally, some instance properties are specified.

The specification of an interface is written in the Interface Definition Language (IDL) which provides primitive data types, constructors for building more complex data types, and a method for specifying operation signatures. IDL supports inclusion of interface types through an [IMPLEMENTATION] IS COMPATIBLE WITH clause which permits inheritance of the specification of included types.

3.2 The implementation

The *evolution manager* is the main component that supports evolution function. It controls the interaction between evolvers and the prototype system, as well as the creation of mapping functions and mapping operators. An evolver invokes the evolution manager to carry out an evolution process.

Different versions of a service must have different interface names under ANSAware, as it distinguishes interface types by their names and not by their structures. This is to ensure that a request to an old server after its evolution is directed to its mapping operator and not to a new server with similar interface name but incompatible interface specification.

The evolver specifies evolution information including the interface pair and instances of the two versions involved. The evolution manager parses the interface definitions and structurally compares them according to the RM-ODP subtyping rules. The evolver is asked to confirm the semantic compatibility and to supply mapping functions for any incompatibilities found. The evolution manager then automatically produces program code for the mapping operator.

Although we are considering evolution of interface types, the actual activity occurs at instance level. As there may be several instances of an interface type that are exported with different properties and contexts, evolving an interface type eventually involves evolving its instances. The new version interface is instantiated in such a way that there is at least one new interface instance that corresponds to each old interface instance and is ready to act as its replacement.

The number of old interface instances in the mapping operator object is determined by the number of old interface instances being evolved. The evolver can either choose a fixed instance selection to specify which old interface instance maps to which new interface instance or specify a selection condition that the mapping operator can use to match an old interface instance with a new one dynamically.

The prototype is able to handle all the changes to an interface type illustrated in Figure 3. It can also deal with changes in user-defined data types, as this kind of change simply reflects changes in parameter types. Changes in supertype, defined by [IMPLEMENTATION] IS COMPATIBLE WITH, are handled by expanding the clause and considering changes in the inherited behaviour. With respect to the old version, a mapping function is defined for each operation and each argument/result parameter of the operation. The evolver supplies all nontrivial mapping functions whereas the evolution manager is responsible for creating mapping functions for direct matching only.

The prototype maintains a *history database* whose function is to keep track of the evolution of service types. This helps the evolution manager when generating program code for mapping operators as well as when creating direct mappings for interface reference data types. It has a checkpoint mechanism to keep the database stable and a log mechanism to facilitate recovery from failure.

There remain some limitations in this implementation. Though clients are provided with evolution transparency through the prototype, evolvers are assisted in the evolution process only to a certain extent as they still have to be aware of the existence of mapping operators. In the current implementation, mapping operators are not automatically instantiated. Evolvers are provided with their program code but need to instantiate them manually. Additionally, the evolution manager does not incorporate ANSAware's relocation function into the prototype to prevent clients who already possess a reference to an old interface from invocation failure. In this instance, re-trading for a new reference always solves the problem.

3.3 An example

The following example illustrates how the evolution process is accomplished in the prototype.

An interface type `Emp` provides a service to access the employee database of a company that comprises two branches, north and south. It provides an operation `Fetch` that takes an employee's name (`name`) and branch name (`branch`) and returns the employee's department name (`dept`) and annual salary (`salary`) (Figure 5(a)). For some reason, the company decides to split the database into two so that each database contains only the data of the employees in the branch and the recording of salary is changed to a monthly basis. The company also decides to evolve the interface `Emp` to `BrEmp` to track the database reorganisation. `BrEmp` takes only the employee's name (`empname`) and returns department name (`dept`) and monthly salary (`salary`) (Figure 5(b)).

```
(a)   Emp : INTERFACE =
      BEGIN
          Branch : TYPE = {north,south};
          Fetch : OPERATION [name : ansa_String; branch : Branch]
                  RETURNS [dept : ansa_String; salary : ansa_Real];
      END.
(b)   BrEmp : INTERFACE =
      BEGIN
          Fetch : OPERATION [empname : ansa_String]
                  RETURNS [dept : ansa_String; salary: ansa_Real];
      END.
```

Figure 5 Versions of a service interface: (a) old (b) new.

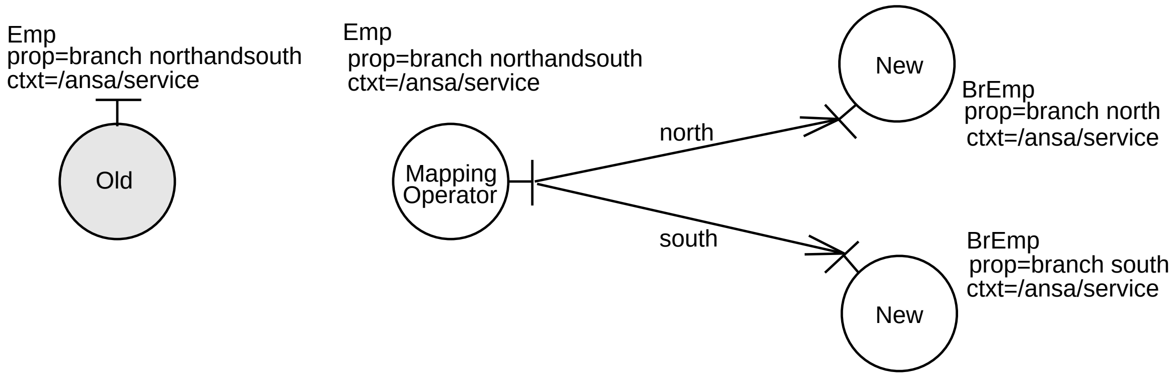

Figure 6 The relationship between old server, mapping operator, and new server.

The single instance of the interface `Emp` is replaced by two instances of the interface `BrEmp`. Their exported properties correspond to the branch they are for. The evolution manager will create a mapping operator that has only one interface instance of type `Emp` with its exported property and context similar to those of the old server (Figure 6).

As also shown in Figure 6, the evolver can specify the condition for the mapping operator to perform dynamic instance selection whenever a `Fetch` request is made. The condition is: if `branch` is `north`, forward the request to the new server whose property is `branch north` otherwise choose the new server with property `branch south`.

On comparing the two interfaces, an incompatibility is found for `name` and `empname` together with a semantic incompatibility for `salary`. The evolver has to specify a forward mapping to obtain `empname` from the arguments of the old version request and a backward mapping to get the annual `salary` from the new version results (Figure 7). A direct mapping function for `dept`, which is semantically consistent in both versions, is created by the evolution manager.

```
Forward Mapping:
int maparg_Fetch_empname(ansa_String Emp_name, Branch Emp_branch, ansa_String *BrEmp_empname)
{   strcpy(*BrEmp_empname, Emp_name);
    return Ok;
}

Backward Mapping:
int mapres_Fetch_salary(ansa_String BrEmp_dept, ansa_Real BrEmp_salary, ansa_Real *Emp_salary)
{   *Emp_salary = BrEmp_salary*12;
    return Ok;
}
```

Figure 7 User-defined mapping functions.

4 CONCLUSION AND FUTURE WORK

The proposed model provides evolution transparency to clients when services evolve. It supports forward compatibility such that clients can continue using the services without having to change their existing programs or track any changes. In addition, it is able to cope with evolution in semantics as well as in structure. The model extends the concept of interface type compatibility, defined in distributed object models, from structural compatibility to functional compatibility.

The model alleviates evolvers' responsibility at evolution time by putting as much automation as possible in the process. It aids cross-version transformation by automatically generating direct mapping functions and all program code necessary for providing evolution transparency.

The prototype in the ANSAware environment demonstrates the application of the model. A better user interface and inclusion of the relocation service in the prototype are required to improve its use.

The model should be extended to cope with nonlinear versioning. There is a possibility that more than one evolver, working in parallel, will propose different new versions for the same service. The model must be enhanced to cover version hierarchies and the criteria to select an appropriate new version when a view of an old version is needed.

A further investigation of evolution in stream interfaces is envisaged. It is likely that the use of mappings can also be applied to manage evolution of this kind of interfaces. However, evolving stream interfaces may be achieved in different ways depending on their characteristics and how they evolve. In the case that the evolution involves changes in hardware or protocol, the use of interceptors may be helpful to provide appropriate conversion.

REFERENCES

America, P. (1991) Designing an Object-Oriented Programming Language with Behavioural Subtyping, in *Foundations of Object-Oriented Languages*, number 489 in Lecture Notes in Computer Science, REX School/Workshop, Noordwijkerhout, The Netherlands. Springer-Verlag.

APM Ltd. (1992) *ANSAware 4.1 Application Programmer's Manual*. APM Ltd., Cambridge, UK.

Banerjee, J., Kim, W., Kim, H.J. and Korth, H.F. (1987) Semantics and Implementation of Schema Evolution in Object-Oriented Databases, in *ACM SIGMOD Proceedings*, San Francisco, 311–322.

Björnerstedt, A. and Hultén, C. (1989) Version Control in an Object-Oriented Architecture, in *Object-Oriented Concepts, Databases, and Applications* (ed. W. Kim and F.H. Lochovsky), ACM Press, 451–485.

Brookes, W., Indulska, J., Bond, A. and Yang, Z. (1995) Interoperability of Distributed Platforms: a Compatibility Perspective, in *Proceedings of ICODP'95*, Brisbane, Australia, 53–64.

ISO/IEC (1995a) *ISO/IEC 10746-1 ODP Reference Model Part 1: Overview*.

ISO/IEC (1995b) *ISO/IEC 10746-2 Open Distributed Processing - Reference Model - Part 2: Foundations*.

ISO/IEC (1995c) *ISO/IEC 10746-2 Open Distributed Processing - Reference Model - Part 3: Architecture*.

Lehman, M.M. and Belady, L.A. (1985) *Program Evolution: Processes of Software Change*. Academic Press.

Liskov, B.H. and Wing, J.M. (1993) A New Definition of the Subtype Relation, in *Proceedings of ECOOP'93* (ed. O. Nierstrasz), number 707 in Lecture Notes in Computer Science, Kaiserslautern, Germany. Springer-Verlag, 1811–1841.

Monk, S.R. and Sommerville, I. (1993) Schema Evolution in OODBs Using Class Versioning. *SIGMOD Record*, **22(3)**, 16–22.

OMG (1990) *Object Management Architecture Guide 1.0*, OMG TC Document 90.9.1. Object Management Group.

Penney, D.J and Stein, J. (1987) Class Modification in the GemStone Object-Oriented DBMS, in *Proceedings of OOPSLA'87*, Orlando, Florida, 111–117.

Skarra, A.H. and Zdonik, S.B. (1988) Type Evolution in an Object-Oriented Database, in *Research Directions in Object-Oriented Programming* (ed. B. Shriver and P. Wegner), MIT Press, 393–415.

BIOGRAPHY

Twittie Senivongse is currently a Ph.D. student in Computer Science at the University of Kent at Canterbury. She received a B.Sc. degree in Statistics from Chulalongkorn University, Thailand, in 1989, and an M.Sc. degree in Computer Science (Conversion) from Imperial College, UK, in 1992. Her research interests in distributed systems include Open Distributed Processing, system interoperability, and distributed object models.

Ian Utting gained a B.Sc. degree in Computers and Cybernetics from the University of Kent at Canterbury in 1978. After spending four years in commercial office systems research and development, he returned to Kent to research into electronic publishing and distributed systems. He has been a lecturer in Computer Science since 1986, and his current research is focused on issues in the engineering of large-scale distributed systems.