Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Experimenting with Relocation and Migration in Distributed Systems

Abstract

This document is a collection of papers written during Phase II of the Advanced Networked Systems Architecture (ANSA) project.

The work described in the above papers had two major affects:

1. The structure and content of Interface References both architecturally and in its example implementation in ANSAware were greatly influenced by the conclusions reached by this work.

2. The mechanisms which deal with relocation and migration were added as an integral part of ANSAware.

For further queries please contact Yigal Hoffner, yh@ansa.co.uk

APM.1654.01 **Approved** 6th December 1995
Technical Report

Distribution:
Supersedes:
Superseded by:

Experimenting with Relocation and Migration in Distributed Systems

Experimenting with Relocation and Migration in Distributed Systems

APM.1654.01

6th December 1995

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

1.1 Overview

The following chapters in this document are a collection of papers written during Phase II of the Advanced Networked Systems Architecture (ANSA) project. The documents were written by Michael Olsen from HP while seconded to the ANSA project:

- RC.215 User Requirements for Object Migration
- RC.280 Handcrafting snapshot operations for an object
- RC.288 Automated generation of snapshot operations for an object
- RC.301 Design and Implementation of a SnapshotBase
- RC.307 Design and Implementation of a StorageLocator
- RC.321 Handcrafting passivation and activation
- RC.326 Automated Generation of Passivation and Activation
- RC.329 Handcrafting Object Migration
- RC.334 Automated Generation of a Migration Infrastructure.

The work described in the above papers had two major affects:

1. The structure and content of Interface References both architecturally and in its example implementation in ANSAware were greatly influenced by the conclusions from the work.
2. The mechanism which deal with relocation and migration were added as an integral part of ANSAware. The reader can refer to the “ANSAware 4.1 Reference Manual”, APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD, UK, (1994), for a description of the mechanism which deal with relocation and migration. The ANSAware 4.1 Reference Manual also contains information about the implementation of Interface References.

2 User Requirements for Object Migration

2.1 Introduction

The purpose of this note is to stimulate discussion on user requirements for object migration.

2.2 Object migration

The effect of object migration is that the location of an object is changed within a distributed system, without affecting the semantics of the services the object provides.

2.3 Why migrate objects?

The are several possible motivations for being able to migrate objects:

- *Resource utilisation*: objects are migrated in order to balance the load over the resources in a distributed system.
- *Local communication*: objects are migrated to avoid inter-machine communication between them; this can improve performance if they communicate frequently.
- *Fault-tolerance*: objects are migrated to locations that suit reliability requirements.
- *Dynamic reconfiguration*: objects are migrated from locations that will become (temporarily) unavailable.
- *Objects follow users*: objects are migrated as users move from machine to machine.
- *Garbage collection*: objects are migrated to locations where bindings to their interfaces are being held, thus allowing a local garbage collection scheme.

It is instructive to note that none of these are motivated by the ANSA computational model. This is because the ANSA computational model is location transparent and assumes infinite resources. Migration is motivated by the restrictions available resources put on the engineering of applications.

2.4 Who may want to migrate objects?

There are several possible initiators for object migration:

- *Resource managers*: managers administrating available resources migrate objects.
- *System administrators*: administrators of system hardware migrate objects before machines are being made unavailable.

- *Transparent programming semantics*: semantics of language constructs effects that objects are being migrated; for example objects providing interfaces given as parameters to operation invocations.
- *Application programmer*: programmers wishing to exploit the benefits of locality (for example maximum application speedup) will need to be aware of and take advantage of the distribution of objects.
- *Garbage collectors*: A garbage collector migrates objects to the domain of another garbage collector if this is where the only bindings to the interfaces of the objects are being held.

2.5 Computational view

Application programmers must ensure that the services that they want to migrate can be migrated. That is, any default co-location of computational objects performed during the compilation of programs must be switched off. This can be achieved by requiring that migration attributes decorate the object construction form or the interfaces of the specification of an object that should be migratable.

An object that has not been specified with appropriate migration attributes before it is compiled cannot be expected to be a migratable unit. The compiler may have organised that several computational objects specified without migration attributes have been compiled into a single engineering object which is the unit of migration for reasons of optimisation.

There may be other restrictions as to which objects are migratable. For example it does not make sense to migrate factories because they are taylored for producing machine specific code. RC.210.01 mentions that a so-called location server for associating socket addresses of interfaces in migrated objects with interface identifiers can be migrated itself if another location server registers its migration; but this must bottom out at a location server that is itself not migratable.

Application programmers migrate objects by invoking a migrate operation in a migration interface. This will effect that all interfaces related to the object providing the migration interface will change location (i.e. new socket addresses).

It is the object that provides a migration interface that alone determines whether to accommodate a migration request. This allows for object specific migration policies. An object may deside not to migrate for reasons of security (the invoker did not have the rights for the migration to be performed) or decide to delay migration until it has completed current activity. Migration invocations should be considered a request if it is motivated by an expected increase in performance and should be considered a demand if it is motivated by an expected increase in reliability.

The semantics an object gives to a migration interface can involve that other objects are requested to migrate. Although, the application programmer may have organised services together in computational objects, there may be conflicting aims of being migratable and communicating frequently. So if an object communicate frequently with other objects all being separately migratable it may be useful to migrate an object as a consequence of migrating another. The extreme is that the semantics of migrating an object will invoke migrate in all interfaces that the object has bindings to. Such "bulk" migration

can be very useful when having to move a large collection of related objects away from a location that will become unavailable.

2.6 Programming level concepts

To enable the appliation programmer to migrate objects it may be necessary to introduce some new programming level concepts:

- *Location*: There need to be an abstraction of physical locations that the programmer can use when programming if specification of absolute location is required. For example, a programmer may want to specify that an object is migrated to a specific machine or a specific capsule. Alternatively, location is relative to other objects. For example, a programmer may want to specify that an object is migrated to a capsule where another object is located without knowing the absolute location. This latter relative location also avoids the problem of how programmers get to know about locations in a distributed system. However, application programmers that will need to fully exploit a distributed system will most likely require an absolute location abstraction.
- *Invocation constructs*: To improve performance three new invocation constructs may be introduced; call-by-move, call-by-visit and call-by-move-return. These mechanisms can be obtained by manually migrating objects supporting interfaces given as parameters to invocations, but the new mechanisms enables packaging the parameter objects in the same network package as the invocation request. A call-by-move invocation moves the parameter objects to the same location as the invoked object. A call-by-visit in addition to call-by-move moves the parameter objects back to their original location after the invocation has completed. A call-by-move-return moves the result objects to the location of the invoker.
- *Operations in migration interface*: The migration interface may be required to have operations locationOf, moveTo, fixAt, unFix and reFix. The present location of the object supporting a migration interface can be obtained by invoking locationOf. The object can be migrated by invoking moveTo with a location as parameter. An object can be fixed at a location by invoking fixAt. Only an unFix or a reFix can allow further migration of the object. Invoking unFix after fixAt has been invoked will discard the fixAt. Invoking reFix will atomically first unFix an object and then it will fix it at a new location. The fix and unfix operations are useful for avoiding that objects oscillates between locations (due to object competition between users) but will probably require that only few objects will be able to fix and unfix objects.
- *Terminations*: Object migration will require terminations for informing that failure of invocation is due to migration being carried out and migration having been carried out. This is discussed in RC.210.01.
- *Location servers*: programmers implementing the operation that migrates an object must ensure that it is possible to find its interfaces when objects that were holding bindings to the interfaces before the migration perform invocations after the migration. RC.210.01 shows how interfaces of migrated objects must be registered in location servers so that objects can lookup addresses of interfaces if presently used addresses fail to work.

It is important that the introduction of new programming level constructs do not affect existing constructs. This ensures that only if the new constructs are being used will there be some incurred costs.

2.7 The cost of migration

Calculating the cost of migration can be a complicated affair. It involves a measure for what the performance would have been if the object was not migrated in order to find out whether there is any performance gain:

performance gain = performance at new location
- performance at previous location
- cost of performing the migration

The performance gain depends of the future activity of the object. If the object later is moved back to its original location it might be that it would have been better not to migrate in the first place. The performance gain also depends on the size of the object being moved, the number of invoctions performed by remote objects on the migrated object compared to invocations by objects not being remote, and the relative cost of mobility and local and remote invocation.

Another factor that influences overall performance gain is whether there are several migration policies in effect simultaneously. For example programmer controlled object migration may conflict with load-balancing such that an expected performance gain turns into a decreased performance because a machine is being overloaded with objects that communicate frequently. Migration for reasons of resource management is likely to be an infrequent event whereas invocation based migration is likely to occur much more frequent.

Facilities for object migration must be used with great care in a distributed system where objects are being shared across machines to obtain global optimal performance. What appears to be a performance gain for one client of an object may turn out to be a perfomance loss to another client.

2.8 Engineering view

The realisation of object migration has already been touched upon when describing object migration in the computational view: an application programmer who have chosen to exploit object distribution will program invocations in the migration interfaces of migratable objects and will (to some extent) implement the operations that migrates objects. The engineering view of object migration is concerned with how to implement object migration and the possible trade-offs available.

The two major problems in engineering object migration are heterogeneity and on-the-fly migration.

Objects are manufactured by factories for a specific machine. Therefore, when migrating an object in a heterogeneous system, it is likely that it will be represented in terms of a different instruction format, instruction set and memory layout on the remote machine. In addition, migrating an object from a machine that allows true concurrency (multi-processor) to a machine that can

only simulate concurrency on a single processor will require considerable non-trivial transformation of the representation of the object. ANSA is an architecture for distributed heterogeneous systems. Therefore it is a requirement that objects can be migrated between heterogeneous machines.

One way of being able to migrate objects between heterogeneous machines is to keep a high level representation of the object inside itself and having an interpreter of the high level representation manipulating it.

Migration can be performed on-the-fly if migration can be performed at any given time - even while one or more activities are being performed by the object. The problem with on-the-fly migration is to stop ongoing (possibly concurrent) activity in an object and extract all the necessary information in stacks and registers so that it can be setup on a remote machine without affecting the semantics of the activity. It has been identified as a requirement to ANSA that object migration is on-the-fly.

Some information may have to be translated into relative addresses. For example addresses in processor registers such as the program counter and the stack pointer, and procedure return addresses on the stack. (But keeping a high level representation as described earlier can avoid this).

It may be that it is possible to achieve performance gain by making the service of an object being migrated incrementally unavailable. For example if some state can only be manipulated by certain interfaces an interface becomes unavailable as the state it can manipulate is being moved, while other interfaces may continue to be available until their associated state will be moved.

An important requirement to the engineering of object migration is that objects are not lost or duplicated. If a failure occurs during object migration one and only one object must be left to perform the service that was to be migrated. For example, object migration may fail because there are not sufficient resources at the intended new location for the object.

A requirement for object migration to be possible is that an appropriate factory exists at the remote location.

2.9 Related work

Related work on object migration that can serve as input to the work on object migration in ANSA are [Short characterisation of each to be provided]:

- DEMOS/MP
- EDEN
- EMERALD
- DISTRIBUTED V
- SOS
- AMOEBA
- CHORUS
- COMANDOS

3 Handcrafting snapshot operations for an object

This paper is the first in a series of papers describing the development of a prototype of the ANSA Storage Model [Olsen 91a], with support for activation, passivation and migration of interfaces and the objects which provide them. The prototype is being built using the ANSA Testbench 3.0 developed in the ISA project [AIM 91], and the plan for developing it is described in [Olsen 91b].

3.1 Context

The prototype is to consist of a number of infrastructural objects which support the interfaces of the ANSA Storage Model and some protocols to form a persistent object system. A *persistent object system* is an infrastructure which enables application systems which are built on top of it to treat objects stored on secondary storage as in-memory objects. Such a system eliminates the *impedance mismatch* between the type system of application systems and the type system of secondary storage abstractions like databases and file systems [Copeland 84]. A persistent object system automatically moves objects, on demand, between secondary storage and main memory. The fact that an object can refer to the interfaces of objects which reside on secondary storage, means that invocations can be extremely slow due to the overhead of turning an object on secondary storage into an in-memory object. Therefore, the challenge is to make persistent object systems run fast.

In order to move an in-memory object to secondary storage, a persistent object system must change the representation of the object so it complies with the type system of a secondary storage abstraction. To facilitate the migration of objects, the chosen secondary storage representation should also comply with the type systems of networks. A non-interpreted sequence of bytes is an example of such a representation.

A sequence of bytes which represents a single object is termed a *snapshot* of that particular object. The snapshots considered in this paper consists of two parts: (i) a state component which denotes the current state of an object in terms of name to value bindings and references to interfaces, and (ii) a schema component which denotes the set of interface types and instances currently supported by an object.

A snapshot of an object is different from a *checkpoint*. Whereas a snapshot only is a representation of a single object, a checkpoint is recursively defined as being composed by a snapshot of an object and a checkpoint of each object which provides an interface referred to by a reference in the initial object, so all the snapshots composing a checkpoint are mutually consistent. If an object has no references to interfaces of other objects, a snapshot of that object is also a checkpoint.

By representing an object which currently performs no activities as a snapshot stored on secondary storage, the in-memory resources occupied by the object can be freed; the in-memory representation need not be available until the object must perform activities again.

3.2 Problem statement

Passivation and migration requires an object to be represented as a snapshot.

A snapshot and the operations which produce and install a snapshot must be generic to all servers. If snapshot operations are not generic, and a client must invoke snapshot operations on many servers, the client must know the snapshot type for each server. Also, there would be a naming problem because the name of the snapshot type for a server is required to be different from all other snapshot type names which are to be known by a client.

The type of a snapshot should conform to the low level types supported by networks and file systems. If the type of a snapshot does not conform to the low level types supported by networks and file systems, then the value of a snapshot must be mapped between these types whenever the snapshot is to be sent over the network or stored in a file.

Only objects of the same type as the object from which a snapshot originates should be able to interpret the snapshot. If a client which obtains a snapshot of an object can interpret the snapshot, the client is capable of manipulating the snapshot thus violating one of the principles of object-orientation: encapsulation of state.

3.3 Purpose

The purpose of this paper is to describes how an application programmer can handcraft extensions to an application in order to support an operation which returns a snapshot of an object's state, and an operation which takes a snapshot and installs it as the state of an object.

The findings of this paper is to serve as requirements for generating snapshot operations automatically.

3.4 Main ideas

A generic snapshot type is defined to be a sequence of bytes. Thus, a client will only have to know a single snapshot type, and it is straight forward to store a snapshot on disk. However, it is possible for a client to manipulate (or rather corrupt) a snapshot, but the client cannot interpret a snapshot. At the server end, it is possible to detect corruption of snapshots, for example if a check sum is included in a snapshot.

By producing a snapshot as a sequence of bytes, there is no conceptual difference between writing a snapshot to a network and writing it to a disk. The conversion from application values to a sequence of bytes and vice versa can be done automatically by marshalling and unmarshalling stubs used for remote procedure calls.

3.5 Limitations and related work

In general, a snapshot of an object consists of an object type component, a state component, and a schema component.

In this paper a snapshot is restricted to be of idle objects only. This simplifies the contents of a snapshot as the state component merely consists of that object state which is globally accessible from operations in the object's interfaces.

The object type component is needed for activation which must instantiate an object in which a given snapshot can be installed. Activation is covered by a separate paper, see [Olsen 91b], and therefore the object type component is left out of the snapshots considered here.

When a snapshot is installed in an object, the schema component will not be used for establishing a set of interface instances in the object. This is because a location service is needed to enable clients to replace references to interface instances of objects from which a snapshot has been produced, with references to interface instances of objects in which the snapshot has been installed. A separate paper describes a prototype of a location service, see [Olsen91b].

The restricted snapshot is similar to the support in Arjuna for storing object state [Shrivastava 89]. However, an Arjuna object has one access point only (a single interface) and therefore Arjuna does not have to deal with the problems of a dynamic set of access points (multiple interfaces).

The approach of converting application values into sequences of bytes by using routines similar to those found in stubs for remote procedure calls was suggested in [Birrell 87] where the mechanism is termed *pickles*.

3.6 Audience and prerequisites

This paper targets system developers and programmers who are designing and implementing persistent object systems. It also addresses application programmers who choose to take control over passivation, activation and migration.

Readers should be acquainted with ANSA [Warne 91], The ANSA Testbench3.0 [AIM 91], and The ANSA Storage Model [Olsen 91a].

3.7 Organisation

An example application is used throughout this document to demonstrate how an application programmer can extend an application so it supports snapshot operations. The complete code for the example application and its extensions are given in appendix A and B.

The example application is briefly summarized in section 2, and section 3 describes how an application programmer can extend the server to support snapshot operations. Section 4 concludes the paper.

3.8 The example application

The example application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone

numbers. It exports a reference to a PhoneBookOp interface instance to a trader, which enables a client to insert, remove, lookup and list names and phone numbers. See appendix A for the complete code for the application.

3.9 Extensions to the application

The server is to be extended so it has an operation for producing a snapshot of its encapsulated state and an operation for installing a snapshot as its encapsulated state, both on request from the client.

Steps which an application programmer must take to extend the server with snapshot operations include:

- describe a generic snapshot type and snapshot operations in IDL
- decide which interface provides the snapshot operations
- record the interface instances created in the server
- decide what application state is to be subject to the snapshot operations
- implement operations for producing and installing the application state and the recorded interface instances of the server
- map the object representation type to the generic snapshot type
- implement the snapshot operations in the server
- provide concurrency control which ensures mutual exclusion between a snapshot operation and any other operation in the server
- modify the client so it can invoke the snapshot operations on the server
- modify the Imakefile(s) affected by the modifications

Sections 3.1 to 3.11 describe one possible way an application programmer can extend the server and the client.

3.9.1 A generic snapshot type and snapshot operations

A generic snapshot type and snapshot operations are defined by the SnapshotOp interface:

```
SnapshotOp: INTERFACE =
BEGIN
  Snapshot : TYPE = SEQUENCE OF OCTET;
  produceSnapshot : OPERATION []
                    RETURNS [snapshot: Snapshot];
  installSnapshot : OPERATION [snapshot: Snapshot]
                    RETURNS [];
END.
```

The produceSnapshot operation returns a result of type Snapshot, and the installSnapshot operation takes an argument of type Snapshot.

3.9.2 Which interface?

The server must provide the snapshot operations in some interface to enable the client to request the production and installation of snapshots. There are several possible ways of providing the produceSnapshot and installSnapshot operations. Some of them are:

- include the snapshot operations in all interfaces provided by the server

- include the snapshot operations in a separate interface which only contains these operations
- include the snapshot operations in one of the interfaces provided by the server

The last option was chosen to avoid the complication of having to deal with an additional interface (the second option), and because the last option easily can be specialized to the first option via the IS COMPATIBLE WITH feature in IDL.

The server's PhoneBookOp interface is made compatible with the SnapshotOp interface by changing the specification of PhoneBookOp to:

```
PhoneBookOp: INTERFACE =
NEEDS PhoneBookTypes;
IS COMPATIBLE WITH SnapshotOp;
BEGIN
  insert: OPERATION [entry: PBEntry]
          RETURNS [];
  remove: OPERATION [name: Name]
          RETURNS [];
  lookup: OPERATION [name: Name]
          RETURNS [entry: PBEntry];
  list  : OPERATION []
          RETURNS [entryList: PBEntryList];
END.
```

3.9.3 Recording the interface instances of an object

The set of interface instances of an object can change dynamically during the lifetime of the object. Therefore, it is necessary that an object maintains a table of its interface instances, and makes this information part of a snapshot's schema component. This enables an object, when it is requested to install a snapshot, to ensure that it has a set of interface instances which conforms to the set of interface instances contained in the snapshot.

The table which records the interface instances can be implemented as a global variable, myInterfaces, in the server and it is of type InterfaceSet which is defined in IDL by:

```
InterfaceSet: INTERFACE =
BEGIN
   InterfaceTypeName: TYPE = ARRAY 32 OF CHAR;
   InterfaceInfo: TYPE = RECORD
                         [ ir: InterfaceRef,
                           type: InterfaceTypeName,
                           concurrency: INTEGER
                         ];
   InterfaceSet: TYPE = SEQUENCE OF InterfaceInfo;
END.
```

An InterfaceInfo must be added to myInterfaces whenever an interface instance is created. The InterfaceInfo which contains a given interface reference must be removed when the interface instance referred to by the interface reference is destroyed.

InterfaceSet is implemented as an abstract datatype which has an initialization operation and operations for adding and removing InterfaceInfo's.

3.9.4 Which application state?

The application state which is to be subject to the snapshot operations should be the state which persists between invocations of the PhoneBookOp interface of the server. The server encapsulates a table containing entries consisting of a name and a phone number. The table is called entryList and its type is a user defined IDL type called PBEntryList. Therefore, the state component of a snapshot of the server must contain the state which can be accessed from entryList.

3.9.5 Producing and installing an object representation

To ensure the snapshot operations are generic, the application specific representation of an object must be produced and installed by operations which are implemented separately from the snapshot operations.

The type specific representation of the server object and the operations which operate on the representation are defined by:

```
ObjRepOp: INTERFACE =
NEEDS PhoneBookOp;
NEEDS InterfaceSet;
BEGIN
ObjRep: TYPE = RECORD
               [entryList: PBEntryList,
                interfaces: InterfaceSet
               ];
produceObjRep: OPERATION []
               RETURNS [objRep: ObjRep];
installObjRep: OPERATION [objRep: ObjRep]
               RETURNS [];
END.
```

An ObjRep is defined as an IDL record with a field for each variable which holds part of the application state which is subject to a snapshot, and a field for the set of interface instances which are recorded in myInterfaces.

The produceObjRep operation returns a result of type ObjRep, and the installObjRep operation takes an argument of type ObjRep. The implementation of the produceObjRep operation simply assigns the fields of an ObjRep the appropriate values of the server state. The implementation of the installObjRep operation first frees memory which previously has been allocated for the server state and its interface instances, then it allocates memory into which the values of the fields of ObjRep are copied.

3.9.6 Mapping an object representation to a snapshot

An operation which takes an ObjRep value and returns a Snapshot value, and an operation which takes a Snapshot value and returns an ObjRep value are needed for mapping between application types and the generic snapshot type. Such operations correspond to marshal and unmarshal operations for ObjRep.

Rather than implementing a marshal and an unmarshal operation for ObjRep by hand, the application programmer can get them for free from the server stub which is generated for ObjRepOp.idl by stubc. The server stub is contained in the file sObjRep.h and it includes the procedure

```
S_ObjRepOp_produceObjRep(_attr, _buf, _bp)
```

This procedure calls `ObjRepOp_produceObjRep(_attr, &objRep )` to obtain an ObjRep value, and it converts the ObjRep value into a Snapshot value which is returned in _buf.

The server stub also includes the procedure

```
S_ObjRepOp_installObjRep(_attr, _buf, _bp)
```

which converts a Snapshot value into an ObjRep value and assigns it to the application variable objRep by calling `ObjRepOp_installObjRep(_attr, objRep)`.

3.9.7 Implementation of the snapshot operations

The implementation of produceSnapshot simply calls

```
S_SnapshotOperations_produceSnapshot(_attr, _buf, _bp)
```

after it has initialized _buf and _bp. The value returned in _buf is then copied into a variable of type Snapshot which is returned as the result.

The implementation of installSnapshot simply copies the argument of type Snapshot into a variable _buf and calls

```
S_SnapshotOperations_installSnapshot(_attr, _buf, _bp)
```

after it has initialized _bp.

3.9.8 Concurrency control

In order to ensure that a snapshot operation is executed mutually exclusive with all other operations of an object, it is necessary to provide some concurrency control which ensures this. The concurrency control can either control the access to the state which is subject to the snapshot operations, or it can control the execution of all the operations provided in the interfaces of an object.

The server already provides concurrency control on the operations rather than the state, allowing only a single invocation of the insert or remove operations, or multiple executions of the list and lookup operations (similar to single writer and multiple readers). It is therefore straight forward to extend the "single write" lock to cover the snapshot operations.

3.9.9 Modifications to the server code

Most of the code which is needed for supporting snapshot operations is separated out from the server; it can be compiled separately and must subsequently be linked to the server code. The modifications required in the server to support the snapshot operations are:

- include the function headers for the operations which are needed for recording the interface instances created by the server:

  ```
  #include "RecordMyInterfaces.h"
  ```

- include the function headers for the operations which implement PhoneBookOp_installSnapshot and PhoneBookOp_produceSnapshot:

  ```
  #include "SnapshotOp.h"
  ```

- add a line which calls addMyInterfaces after each line in the server which creates an interface instance, and add a line which calls remMyInterfaces after each line which destroys an interface instance.

See appendix B for the code which is separated out from the server.

3.9.10 Modifications to the client code

The modifications to be made to the client to enable it to test the servers's snapshot operations are:

- add the parameter number for a snapshot test operation
- add "snapshottest" to the list of commands which the client accepts as arguments
- add the code which implements the snapshot test at the end of the client's argument dispatcher. This code first invokes produceSnapshot to obtain a snapshot of the server. It then performs an invocation which alters the state of the server followed by an invocation of list. The result of invoking list is displayed on the screen. Then the previously obtained snapshot is installed in the server followed by another list invocation. Again the result of invoking list is displayed on the screen thus showing that the server state prior to the state alteration has been installed.

3.9.11 Modifications to the Imakefile

The modifications required to be made to the Imakefile in order to link the server code with the code which implements most of the snapshot support are:

- Add InterfaceSet.idl and ObjRepOp.idl to IDLFILES, and InterfaceSet.sif and ObjRepOp.sif to SIFFILES
- Add RecordMyInterfaces.c, SnapshotOp.c, and ObjRepOp.c to SRCS
- Add the line IDLDepend(InterfaceSet) followed by the line IDLDepend(ObjRepOp)
- Add RecordMyInterfaces.o, SnapshotOp.o, sObjRepOp.o, and ObjRepOp.o to SingleProgramTarget for server

3.10 Conclusion

It has been show how an application programmer can extend an example application so it supports an operation which produces a snapshot of an object's state, and an operation which installs a snapshot as the state of an object.

The snapshot operations are generic for all objects and they take and return a snapshot which is defined to be a sequence of bytes. The low level type of a snapshot enables a snapshot to be sent out on networks and to be stored in files without the need for any application specific type information. A client which invokes snapshot operations can therefore easily store the snapshot on disk or send it out on the network without casting values into low level disk or network bit patterns. The conversion of values between application specific types and the type of a snapshot is done via procedures which are generated from an IDL file by the stub compiler.

The application programmer needs to write a considerable amount of code which implements interface instance recording, operations for producing and installing the application specific state, and the generic snapshot operations. However, appendix B shows that most of this code can be separated out from the original server's code, so it should be possible to generate the code automatically. A separate paper describes how to automate the generation of snapshot operations in server objects.

3.11 Acknowledgements

Thanks to David Iggulden of the ISA Core Team and Nigel Edwards of Hewlett-Packard Laboratories for commenting on an earlier draft of this paper.

3.12 References

[AIM 91]
"ANSAware 3.0 Implementation Manual", Report No.: RM.097, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Birrell 87]
Andrew D. Birrell, Michael B. Jones, Edward P. Wobber, ”A Simple and Efficient Implementation for Small Databases”, Operating Systems Review, Vol. 21, No. 5, (1987).

[Copeland 84]
George Copeland, David Maier, "Making Smalltalk a Database System", *SIGMOD’84*, Sigmod Record, Vol. 14, No. 2, (1984).

[Olsen 91a]
Michael Hoffmann Olsen, “A Model for Storage and resource Management”, Report No.: WP-DCG-91-14, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol, England, (1991).

[Olsen 91b]
Michael Hoffmann Olsen, “The storage activity: objectives, tasks, timescales, and status”, Report No.: RC.281, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Shrivastava 89]
S. K. Shrivastava et al, “The Treatment of Persistent Objects in Arjuna”, THE COMPUTER JOURNAL, VOL. 32, NO. 4, (1989).

[Warne 91]
John Warne, "ANSA: Assumptions, Principles, and Structure", Conference proceedings of Software Engineering Environments 1991, University College of Wales, Aberystwyth, (1991).

4 Appendix: The example application code

This appendix contains the code for the example application referred to in this document; the last section contains example executions of the application.

The application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It exports a reference to a PhoneBookOp interface instance to a trader, which enables a client to insert, remove, lookup and list names and phone numbers.

4.1 idl files

4.1.1 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

4.1.2 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

4.2 include files

4.2.1 ConcControl.h

```
extern void initConcControl();
extern void cleanUpConcControl();
extern void getSingleWriteLock();
extern void releaseSingleWriteLock();
extern void getMultipleReadLock();
```

```
extern void releaseMultipleReadLock();
```

4.2.2 ConcControl.c

```
#include "ansa.h"
#include "opsys.h"
#include "machine.h"
#include "ecs.h"
#include "ConcControl.h"

ansa_EventCount doneReadorWrite, doneNext;
ansa_Sequencer nextReadorWrite, next;

void initConcControl()
{ doneReadorWrite = ecs_makeEventCount((ansa_Cardinal) 0);
  doneNext        = ecs_makeEventCount((ansa_Cardinal) 0);
  nextReadorWrite = ecs_makeSequencer((ansa_Cardinal) 0);
  next            = ecs_makeSequencer((ansa_Cardinal) 0);
}

void cleanUpConcControl()
{ ecs_freeEventCount(doneReadorWrite);
  ecs_freeEventCount(doneNext);
  ecs_freeSequencer(nextReadorWrite);
  ecs_freeSequencer(next);
}

void getSingleWriteLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_await(doneNext, ecs_ticket(next));
}

void releaseSingleWriteLock()
{ ecs_advance(doneReadorWrite);
  ecs_advance(doneNext);
}

void getMultipleReadLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_ticket(next);
  ecs_advance(doneReadorWrite);
}

void releaseMultipleReadLock()
{ ecs_advance(doneNext);
}
```

4.3 dpl files

4.3.1 server.dpl

```
! USE Trader
! USE PhoneBookOp
! DECLARE { phoneBook, pbRef } : PhoneBookOp SERVER

#include <stdio.h>
#include "ansa.h"
```

```
#include "capsule.h"
#include "system.h"
#include "ConcControl.h"

#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16

GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

PBEntryList entryList;
ansa_InterfaceRef phoneBook;

void copyRef(new, old)
  ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

void body()
{ Result tres;
  char hn[64], bf[64 + 11];
  ansa_InterfaceRef pbRef;

! {pbRef} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  copyRef(&phoneBook, &pbRef);
  initConcControl();
  entryList.length = 0;
  entryList.data   = (PBEntry *)0;
  gethostname(hn, sizeof(hn));
  (void)sprintf(bf, "NodeName '%s'", hn);
! {tres} <- traderRef$Register("PhoneBookOp", "/ansa/PhoneBook",
\
                                bf, phoneBook)
}

void allocateEntryList()
{ if (entryList.length == 0)
  { if ((entryList.data = (PBEntry *)
         system_allocate(sizeof(PBEntry))) == NULL)
    fprintf(stderr,"ERROR while allocating memory for
entryList\n");
  }
  else
  {
   if ((entryList.data = (PBEntry *)system_extend((ansa_Cardinal
*)
       entryList.data, (entryList.length + 1) * sizeof(PBEntry)))
        == NULL)
    instruct_Abort("allocateEntryList", insufficientMemory);
  }
}

void removeAnEntry(toBeRemoved)
  int toBeRemoved;
{ PBEntry *this, *next;
  int i;

  this = entryList.data + toBeRemoved;
```

```
  next = this + 1;
  for (i=toBeRemoved; i<(entryList.length - 1);i++,this++,next++)
  { strcpy((char *) this->name, (char *) next->name);
    strcpy((char *) this->no, (char *) next->no);
  }
  entryList.length--;
  if (entryList.length == 0)
  { system_free((ansa_Cardinal *) entryList.data);
  }
  else
  { if ((entryList.data = (PBEntry *)system_extend((ansa_Cardinal
*)
         entryList.data, (entryList.length) * sizeof(PBEntry)))
        == NULL)
    instruct_Abort("extendEntryList", insufficientMemory);
  }
}

int PhoneBookOp_insert(_attr, newent)
  ansa_InterfaceAttr *_attr;
  PBEntry newent;
{ PBEntry *ept, *entry;
  int toBeChanged, index;
  int toBeRemoved;

  getSingleWriteLock();
  toBeChanged = -1;
  entry = entryList.data;
  for (index=0; (toBeChanged == -1) && (index<entryList.length);
       index++, entry++)
    if (strcmp(newent.name, entry->name) == 0)
    toBeChanged = index;
  if (toBeChanged != -1)
  { ept = entryList.data + toBeChanged;
  }
  else
  { allocateEntryList(&entryList);
    ept = entryList.data + entryList.length;
    entryList.length++;
  }
  strcpy((char *) ept->name, (char *) newent.name);
  strcpy((char *) ept->no, (char *) newent.no);
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
Name name;
{ int   j, i, index, toBeRemoved, processedLast;
  PBEntry *entry, *first, *next;
  ansa_InterfaceRef *ifrpt;

  getSingleWriteLock();
  toBeRemoved = -1;
  entry = entryList.data;
  for (index=0; (toBeRemoved == -1) && (index<entryList.length);
       index++, entry++)
```

```
      if (strcmp(name, entry->name) == 0)
      toBeRemoved = index;
  if (toBeRemoved != -1)
  removeAnEntry(toBeRemoved);
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  *entList = entryList;
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ int   i, index, toBeReturned;
  PBEntry ent, *ept;

  getMultipleReadLock();
  toBeReturned = -1;
  ept = entryList.data;
 for (index=0; (toBeReturned == -1) && (index<entryList.length);
        index++, ept++)
      if (strcmp(name, ept->name) == 0)
      toBeReturned = index;
  if (toBeReturned != -1)
  { ept = entryList.data + toBeReturned;
    strcpy(ent.name, ept->name);
    strcpy(ent.no, ept->no);
  }
  else
  { strcpy(ent.name, "0");
    strcpy(ent.no, "0");
  }
  *entry = ent;
  releaseMultipleReadLock();
  return 1;
}
```

4.3.2 client.dpl

```
! USE Capsule
! DECLARE {cref, crefimport} : Capsule    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook, pbRef} : PhoneBookOp CLIENT

#include   <stdio.h>
#include "capsule.h"
#include "system.h"

#define MAXANSWERLENGTH 31
```

```
ansa_InterfaceRef phoneBook;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "insert",
  "remove",
  "lookup",
  "list",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {4, 3, 3, 2};
int maxargs[] = {4, 3, 3, 2};

#define INVALID_CMD -1

PBEntryList eList;

void listEntryList(entryList)
  PBEntryList *entryList;
{ int        i;
  char       buffer[128];
  PBEntry  *ifr = entryList->data;

  fprintf(stdout, "\nENTRIES IN PHONEBOOK:\n");
  for (i=0; i<entryList->length; i++, ifr++)
   fprintf(stdout, "  %s %s\n", ifr->name, ifr->no );
}

void PrintUsageAndDie(prog)
char *prog;
{ fprintf(stderr, "usage: %s insert name no\n", prog);
  fprintf(stderr, "usage: %s remove name\n", prog);
  fprintf(stderr, "usage: %s lookup name\n", prog);
  fprintf(stderr, "usage: %s list\n", prog);
! capsule$Terminate()
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ ansa_InterfaceRef pbRef;
  int i, cmd_no;

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }
```

```
  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  /* get reference to PhoneBookOp */
! {pbRef} <- traderRef$Import("PhoneBookOp", "/ansa/PhoneBook",\
                              "NodeName == 'audrey'")
  system_allocateRef(pbRef.ir_id, pbRef.ir_ah, &phoneBook);

  switch(cmd_no)
  {   case 0:
        { PBEntry entry;

          strcpy(entry.name, argv[2]);
          strcpy(entry.no, argv[3]);
!         {} <- phoneBook$insert(entry)
        }
        break;
      case 1:
        { Name name;

          strcpy(name, argv[2]);
!         {} <- phoneBook$remove(name)
        }
        break;
      case 2:
        { PBEntry reply;

!         {reply} <- phoneBook$lookup(argv[2])
          if (strcmp(reply.name, "0") == 0)
         { fprintf(stdout, "Sorry, no entry for %s\n", argv[2]);
          }
          else
          { fprintf(stdout, "No: %s\n", reply.no);
          }
        }
        break;
      case 3:
        { PBEntryList entryList;

!         {entryList} <- phoneBook$list()
        }
     break;
  }
}
```

4.4 Imakefile

```
        DEFINES =
       INCLUDES =
       IDLFLAGS =
       DPLFLAGS =
```

```
       LINTLIBS = $(LINTANSALIB) -lc
       LOCALLIB = $(ANSALIB)
      LOCALLIBD = $(ANSALIBD)
        INSTALL = bsdinstall.sh

IDLFILES = PhoneBookOp.idl
SIFFILES = PhoneBookOp.sif
DPLFILES = server.dpl client.dpl
SRCS     = ConcControl.c
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)

PROGS = server client

# compile idl files
all:: $(SIFFILES)

# compile dpl files
DPLDepend(server)
DPLDepend(client)

all:: $(PROGS)

# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)

SingleProgramTarget(server, server.o sPhoneBookOp.o \
                    ConcControl.o,$(LOCALLIB),)
SingleProgramTarget(client, client.o
cPhoneBookOp.o,$(LOCALLIB),)

InstallProgram(server,$(TEMPLATEDIR))

NormalLintTarget(*.c)
DependTarget()

IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

4.5 Example executions

This section shows the results of some executions of the client while a server is running. Output from client executions are printed in italics.

4.5.1 server execution

```
server
```

4.5.2 client executions

```
client insert Michael_Olsen  0223323010
client insert Audrey_Hepburn +14156666660
client insert Ronald_Reagan +14156478231
client list
ENTRIES IN PHONEBOOK:
  Michael_Olsen  0223323010
  Audrey_Hepburn +14156666660
```

```
  Ronald_Reagan +14156478231
client lookup Michelle_Pheiffer
Sorry, no entry for Michelle_Pheiffer
client remove Michael_Olsen
client list
ENTRIES IN PHONEBOOK:
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231
Session lookup Audrey_Hepburn
No: +14156666660
```

5 Appendix: The extended example application

This appendix contains the code for the extended example application which supports the snapshot operations described in this document; the last section of the appendix contains example executions of the application.

5.1 include files

5.1.1 ConcControl.h

```
extern void initConcControl();
extern void cleanUpConcControl();
extern void getSingleWriteLock();
extern void releaseSingleWriteLock();
extern void getMultipleReadLock();
extern void releaseMultipleReadLock();
```

5.1.2 ConcControl.c

```
#include "ansa.h"
#include "opsys.h"
#include "machine.h"
#include "ecs.h"
#include "ConcControl.h"

ansa_EventCount doneReadorWrite, doneNext;
ansa_Sequencer nextReadorWrite, next;

void initConcControl()
{ doneReadorWrite = ecs_makeEventCount((ansa_Cardinal) 0);
  doneNext        = ecs_makeEventCount((ansa_Cardinal) 0);
  nextReadorWrite = ecs_makeSequencer((ansa_Cardinal) 0);
  next            = ecs_makeSequencer((ansa_Cardinal) 0);
}

void cleanUpConcControl()
{ ecs_freeEventCount(doneReadorWrite);
  ecs_freeEventCount(doneNext);
  ecs_freeSequencer(nextReadorWrite);
  ecs_freeSequencer(next);
}

void getSingleWriteLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_await(doneNext, ecs_ticket(next));
}

void releaseSingleWriteLock()
{ ecs_advance(doneReadorWrite);
```

```
  ecs_advance(doneNext);
}

void getMultipleReadLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_ticket(next);
  ecs_advance(doneReadorWrite);
}

void releaseMultipleReadLock()
{ ecs_advance(doneNext);
}
```

5.1.3 RecordMyInterfaces.h

```
#include "tInterfaceSet.h"

extern InterfaceSet myInterfaces;
extern void initMyInterfaces();
extern void addMyInterfaces();
extern void remMyInterfaces();
```

5.1.4 RecordMyInterfaces.c

```
#include "tInterfaceSet.h"
#include "capsule.h"

InterfaceSet myInterfaces = {0, (InterfaceInfo *) 0};

void copyIfRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

void initMyInterfaces(myIfSet)
InterfaceSet *myIfSet;
{ InterfaceInfo *ifInfo;
  int i;

  ifInfo = myIfSet->data;
  for (i=0; i < myIfSet->length; i++, ifInfo++)
  system_freeRef(&(ifInfo->ir));
  system_free(myIfSet->data);
  myIfSet->length = 0;
  myIfSet->data = (InterfaceInfo *) 0;
}

void addMyInterfaces(myIfSet, new, type, concurrency)
InterfaceSet *myIfSet;
ansa_InterfaceRef  *new;
InterfaceTypeName type;
ansa_Integer concurrency;
{ InterfaceInfo *freeslot;

  if (myIfSet->length == 0)
  { if ((myIfSet->data = (InterfaceInfo *)
       system_allocate(sizeof(InterfaceInfo))) == NULL)
    instruct_Abort("addMyInterfaces", insufficientMemory);
```

```
  }
  else
  { if ((myIfSet->data = (InterfaceInfo *)
       system_extend((ansa_Cardinal *) myIfSet->data,
       (myIfSet->length + 1) *
       sizeof(InterfaceInfo))) == NULL)
    instruct_Abort("addMyInterfaces", insufficientMemory);
  }
  freeslot = (InterfaceInfo *) myIfSet->data + myIfSet->length;
  copyIfRef(&(freeslot->ir), new);
  strcpy((char *) (freeslot->type), (char *) type);
  freeslot->concurrency = concurrency;
  myIfSet->length++;
}

void remMyInterfaces(myIfSet, old)
InterfaceSet *myIfSet;
ansa_InterfaceRef  *old;
{ InterfaceInfo *ifInfo, *next;
  int          i, index, toBeRemoved;

  toBeRemoved = -1;
  ifInfo = myIfSet->data;
  for (index=0; (toBeRemoved == -1) && (index<myIfSet->length);
       index++, ifInfo++)
  if (system_cmpIfID((ifInfo->ir).ir_id, old->ir_id) == 0)
  toBeRemoved = index;
  if (toBeRemoved != -1)
  { ifInfo = myIfSet->data + toBeRemoved;
    next = ifInfo + 1;
    for (i=toBeRemoved; i<((myIfSet->length) - 1); i++, ifInfo++,
         next++)
    { system_freeRef(&(ifInfo->ir));
      copyIfRef(&(ifInfo->ir), next);
    }
    myIfSet->length = (myIfSet->length) -1;
    system_freeRef(&(ifInfo->ir));
    if ((myIfSet->data = (InterfaceInfo *)
        system_extend((ansa_Cardinal *)
      myIfSet->data,(myIfSet->length +1)*sizeof(InterfaceInfo)))
        == NULL)
    instruct_Abort("remMyInterfaces", insufficientMemory);
  }
}
```

5.1.5 StateRefs.h

```
extern PBEntryList entryList;
```

5.1.6 ObjRepOp.h

```
extern int ObjRepOp_produceObjRep();
extern int ObjRepOp_installObjRep();
```

5.1.7 ObjRepOp.c

```
#include "tObjRepOp.h"
#include "tPhoneBookOp.h"
#include "StateRefs.h"
```

```
#include "RecordMyInterfaces.h"
#include "capsule.h"
#include "system.h"

int ObjRepOp_produceObjRep( _attr, objRep)
ansa_InterfaceAttr *_attr;
ObjRep *objRep;
{ objRep->entryList  = entryList;
  objRep->interfaces = myInterfaces;
  return 1;
}

int ObjRepOp_installObjRep( _attr, objRep)
ansa_InterfaceAttr *_attr;
ObjRep objRep;
{ int i;
  PBEntry *ifr, *destifr;
  InterfaceInfo *row;

  system_free(entryList.data);
  entryList.length = 0;
  if ((entryList.data = (PBEntry *)
                      system_allocate(objRep.entryList.length *
                                      sizeof(PBEntry))) == NULL)
  instruct_Abort("installObjRep", insufficientMemory);
  ifr = objRep.entryList.data;
  destifr = entryList.data;
  for (i=0; i<objRep.entryList.length; i++, ifr++, destifr++)
  { strcpy((char *) destifr->name, (char *) ifr->name);
    strcpy((char *) destifr->no, (char *) ifr->no);
  }
  entryList.length = objRep.entryList.length;
  initMyInterfaces(&myInterfaces);
  row = objRep.interfaces.data;
  for (i=0; i < objRep.interfaces.length; i++, row++)
  addMyInterfaces(&myInterfaces, &(row->ir), row->type,
                  row->concurrency);
  return 1;
}
```

5.1.8 SnapshotOp.h

```
extern int PhoneBookOp_installSnapshot();
extern int PhoneBookOp_produceSnapshot();
```

5.1.9 SnapshotOp.c

```
#include "ansa.h"
#include "MoveMacs.h"
#include "MemAlloc.h"
#include "tObjRepOp.h"
#include "tPhoneBookOp.h"
#include "ObjRepOp.h"

int PhoneBookOp_installSnapshot( _attr, snapshot )
ansa_InterfaceAttr *_attr;
Snapshot snapshot;
{ ObjRep objRep;
  ansa_BufferLink _buf;
```

```
  long _sz;
  ansa_Octet *_bp;
  ansa_BufferLink buffer_make();
  ansa_Status _stat;
  void S_ObjRepOp_installObjRep();

  _sz =2048;
  while (_sz < snapshot.length)
  _sz *= 4;
  _buf = buffer_make((ansa_Cardinal) _sz);
  if (_buf == (ansa_BufferLink)0)
  {  /* report a heapAllocation failure */
  }
  memcpy(_buf->data, snapshot.data, (int) snapshot.length);
  _buf->used = (int) snapshot.length;
  _bp = (ansa_Octet *)(_buf->data);
  _bp += UnmarshallEnum(_bp, &(_stat));
  if (_stat)
  goto exit;
  S_ObjRepOp_installObjRep(_attr, _buf, _bp);
exit:
  if (_buf != (ansa_BufferLink)0)
  buffer_free(_buf);
  return 1;
}

int PhoneBookOp_produceSnapshot( _attr, snapshot )
ansa_InterfaceAttr *_attr;
Snapshot *snapshot;
{ Snapshot pbuf;
  ansa_BufferLink _buf;
  long _sz;
  ansa_Octet *_bp;
  ansa_BufferLink buffer_make();
  void   S_ObjRepOp_produceObjRep();

  _sz = 2048;
  _buf = buffer_make((ansa_Cardinal) _sz);
  if (_buf == (ansa_BufferLink)0)
  {  /* report a heapAllocation failure */
  }
  _bp = (ansa_Octet *)(_buf->data);  /* just to initialize it */
  S_ObjRepOp_produceObjRep(_attr, _buf, _bp);
  pbuf.length = (ansa_Cardinal)(_buf->used);
  pbuf.data = (ansa_Octet *)(_buf->data);
  *snapshot = pbuf;
  return 1;
}
```

5.2 idl files

5.2.1 InterfaceSet.idl

```
InterfaceSet : INTERFACE =
BEGIN
   InterfaceTypeName : TYPE = ARRAY 32 OF CHAR;
   InterfaceInfo : TYPE = RECORD
```

```
                              [ ir : InterfaceRef,
                                type : InterfaceTypeName,
                                concurrency : INTEGER
                              ];
   InterfaceSet     : TYPE = SEQUENCE OF InterfaceInfo;
END.
```

5.2.2 SnapshotOp.idl

```
SnapshotOp : INTERFACE =
BEGIN
  Snapshot : TYPE = SEQUENCE OF OCTET;
  produceSnapshot : OPERATION []
                    RETURNS [snapshot : Snapshot];
  installSnapshot : OPERATION [snapshot : Snapshot]
                    RETURNS [];
END.
```

5.2.3 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

5.2.4 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
IS COMPATIBLE WITH SnapshotOp;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

5.2.5 ObjRepOp.idl

```
ObjRepOp : INTERFACE =
NEEDS PhoneBookOp;
NEEDS InterfaceSet;
BEGIN
ObjRep : TYPE = RECORD
                [entryList : PBEntryList,
                 interfaces : InterfaceSet
                ];
produceObjRep : OPERATION []
                RETURNS [objRep : ObjRep];
```

```
installObjRep : OPERATION [objRep : ObjRep]
                RETURNS [];
END.
```

5.3 dpl files

5.3.1 server.dpl

```
! USE Trader
! USE PhoneBookOp
! DECLARE { phoneBook, pbRef } : PhoneBookOp SERVER
#include <stdio.h>
#include "ansa.h"
#include "capsule.h"
#include "system.h"
#include "ConcControl.h"
#include "RecordMyInterfaces.h"
#include "SnapshotOp.h"

#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16

GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

PBEntryList entryList;
ansa_InterfaceRef phoneBook;

void copyRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

void body()
{ Result tres;
  char hn[64], bf[64 + 11];
  ansa_InterfaceRef pbRef;

! {pbRef} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  addMyInterfaces(&myInterfaces, &pbRef, "PhoneBookOp",
                  POTENTIALCONCURRENCY);
  copyRef(&phoneBook, &pbRef);
  initConcControl();
  entryList.length = 0;
  entryList.data   = (PBEntry *)0;
  gethostname(hn, sizeof(hn));
  (void)sprintf(bf, "NodeName ‘%s’", hn);
! {tres} <- traderRef$Register("PhoneBookOp", "/ansa/PhoneBook",
\
                                bf, phoneBook)
}

void allocateEntryList()
{ if (entryList.length == 0)
  { if ((entryList.data = (PBEntry *)
         system_allocate(sizeof(PBEntry))) == NULL)
```

```
    fprintf(stderr,"ERROR while allocating memory for
entryList\n");
  }
  else
  {
   if ((entryList.data = (PBEntry *)system_extend((ansa_Cardinal
*)
       entryList.data, (entryList.length + 1) * sizeof(PBEntry)))
        == NULL)
    instruct_Abort("allocateEntryList", insufficientMemory);
  }
}

void removeAnEntry(toBeRemoved)
int toBeRemoved;
{ PBEntry *this, *next;
  int i;

  this = entryList.data + toBeRemoved;
  next = this + 1;
  for (i=toBeRemoved; i<(entryList.length - 1);i++,this++,next++)
  { strcpy((char *) this->name, (char *) next->name);
    strcpy((char *) this->no, (char *) next->no);
  }
  entryList.length--;
  if (entryList.length == 0)
  { system_free((ansa_Cardinal *) entryList.data);
  }
  else
  { if ((entryList.data = (PBEntry *)system_extend((ansa_Cardinal
*)
        entryList.data, (entryList.length) * sizeof(PBEntry)))
        == NULL)
    instruct_Abort("extendEntryList", insufficientMemory);
  }
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ PBEntry *ept, *entry;
  int toBeChanged, index;
  int toBeRemoved;

  getSingleWriteLock();
  toBeChanged = -1;
  entry = entryList.data;
  for (index=0; (toBeChanged == -1) && (index<entryList.length);
       index++, entry++)
    if (strcmp(newent.name, entry->name) == 0)
    toBeChanged = index;
  if (toBeChanged != -1)
  { ept = entryList.data + toBeChanged;
  }
  else
  { allocateEntryList(&entryList);
    ept = entryList.data + entryList.length;
    entryList.length++;
```

```
  }
  strcpy((char *) ept->name, (char *) newent.name);
  strcpy((char *) ept->no, (char *) newent.no);
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
Name name;
{ int   j, i, index, toBeRemoved, processedLast;
  PBEntry *entry, *first, *next;
  ansa_InterfaceRef *ifrpt;

  getSingleWriteLock();
  toBeRemoved = -1;
  entry = entryList.data;
  for (index=0; (toBeRemoved == -1) && (index<entryList.length);
       index++, entry++)
     if (strcmp(name, entry->name) == 0)
     toBeRemoved = index;
  if (toBeRemoved != -1)
  removeAnEntry(toBeRemoved);
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  *entList = entryList;
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ int   i, index, toBeReturned;
  PBEntry ent, *ept;

  getMultipleReadLock();
  toBeReturned = -1;
  ept = entryList.data;
  for (index=0; (toBeReturned == -1) && (index<entryList.length);
         index++, ept++)
     if (strcmp(name, ept->name) == 0)
     toBeReturned = index;
  if (toBeReturned != -1)
  { ept = entryList.data + toBeReturned;
    strcpy(ent.name, ept->name);
    strcpy(ent.no, ept->no);
  }
  else
  { strcpy(ent.name, "0");
    strcpy(ent.no, "0");
```

```
  }
  *entry = ent;
  releaseMultipleReadLock();
  return 1;
}
```

5.3.2 client.dpl

```
! USE Capsule
! DECLARE {cref, crefimport} : Capsule    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook, pbRef} : PhoneBookOp CLIENT

#include   <stdio.h>
#include "capsule.h"
#include "system.h"

#define MAXANSWERLENGTH 31

ansa_InterfaceRef phoneBook;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "insert",
  "remove",
  "lookup",
  "list",
  "snapshottest",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {4, 3, 3, 2, 3};
int maxargs[] = {4, 3, 3, 2, 3};

#define INVALID_CMD -1

PBEntryList eList;

void listEntryList(entryList)
PBEntryList *entryList;
{ int        i;
  char       buffer[128];
  PBEntry  *ifr = entryList->data;

  fprintf(stdout, "\nENTRIES IN PHONEBOOK:\n");
  for (i=0; i<entryList->length; i++, ifr++)
  fprintf(stdout, "  %s %s\n", ifr->name, ifr->no );
}

void PrintUsageAndDie(prog)
char *prog;
{ fprintf(stderr, "usage: %s insert name no\n", prog);
  fprintf(stderr, "usage: %s remove name\n", prog);
  fprintf(stderr, "usage: %s lookup name\n", prog);
  fprintf(stderr, "usage: %s list\n", prog);
  fprintf(stderr, "usage: %s snapshottest name\n", prog);
! capsule$Terminate()
```

```
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ ansa_InterfaceRef pbRef;
  int i, cmd_no;

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  /* get reference to PhoneBookOp */
! {pbRef} <- traderRef$Import("PhoneBookOp", "/ansa/PhoneBook",\
                              "NodeName == 'audrey'")
  system_allocateRef(pbRef.ir_id, pbRef.ir_ah, &phoneBook);

  switch(cmd_no)
  {   case 0:
        { PBEntry entry;

          strcpy(entry.name, argv[2]);
          strcpy(entry.no, argv[3]);
!         {} <- phoneBook$insert(entry)
        }
        break;
      case 1:
        { Name name;

          strcpy(name, argv[2]);
!         {} <- phoneBook$remove(name)
        }
        break;
      case 2:
        { PBEntry reply;

!         {reply} <- phoneBook$lookup(argv[2])
          if (strcmp(reply.name, "0") == 0)
         { fprintf(stdout, "Sorry, no entry for %s\n", argv[2]);
          }
```

```
            else
            { fprintf(stdout, "No: %s\n", reply.no);
            }
          }
          break;
        case 3:
          { PBEntryList entryList;

!           {entryList} <- phoneBook$list()
            listEntryList(&entryList);
          }
      break;
        case 4:
          { Name name;
            Snapshot snapshot;
            PBEntryList entryList;

!           {snapshot} <- phoneBook$produceSnapshot()
            strcpy(name, argv[2]);
!           {} <- phoneBook$remove(name)
!           {entryList} <- phoneBook$list()
            listEntryList(&entryList);
!           {} <- phoneBook$installSnapshot(snapshot)
!           {entryList} <- phoneBook$list()
            listEntryList(&entryList);

          }
      break;
  }
}
```

5.4 Imakefile

```
                DEFINES =
         INCLUDES =
         IDLFLAGS =
         DPLFLAGS =
         LINTLIBS = $(LINTANSALIB) -lc
         LOCALLIB = $(ANSALIB)
        LOCALLIBD = $(ANSALIBD)
          INSTALL = bsdinstall.sh
IDLFILES = PhoneBookOp.idl InterfaceSet.idl ObjRepOp.idl
SIFFILES = PhoneBookOp.sif InterfaceSet.sif ObjRepOp.sif
DPLFILES = server.dpl client.dpl
SRCS     = ConcControl.c RecordMyInterfaces.c SnapshotOp.c
           ObjRepOp.c
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)

PROGS = server client

# compile idl files
all:: $(SIFFILES)

# compile dpl files
DPLDepend(server)
DPLDepend(client)
```

```
all:: $(PROGS)

# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)
IDLDepend(InterfaceSet)
IDLDepend(ObjRepOp)

SingleProgramTarget(server, server.o sPhoneBookOp.o
ConcControl.o
                            RecordMyInterfaces.o SnapshotOp.o
                            sObjRepOp.o ObjRepOp.o,$(LOCALLIB),)
SingleProgramTarget(client, client.o
cPhoneBookOp.o,$(LOCALLIB),)

InstallProgram(server,$(TEMPLATEDIR))

NormalLintTarget(*.c)
DependTarget()

IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

5.5 Example run

This section shows the results of some executions of the client while a server is running. Output from client executions are printed in italics.

5.5.1 server execution

```
server
```

5.5.2 client executions

When given *snapshottest* as argument, the client performs the following invocations on the server to demonstrate the installation of a previously obtained snapshot: (i) *produceSnapshot*, (ii) *remove <parameter name>*, (iii) *list*, (iv) *installSnapshot*, and (v) *list*.

```
client insert Michael_Olsen  0223323010
client insert Audrey_Hepburn +14156666660
client insert Ronald_Reagan +14156478231
client list
ENTRIES IN PHONEBOOK:
  Michael_Olsen  0223323010
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231
Session snapshottest Michael_Olsen
ENTRIES IN PHONEBOOK:
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231
ENTRIES IN PHONEBOOK:
  Michael_Olsen 0223323010
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231
```

3.

6 Automated generation of snapshot operations for an object

6.1 Introduction

This paper is the second in a series of eight papers describing a prototype of the ANSA Storage Model [Olsen 91] developed in the ANSA Testbench 3.0 [AIM 91]. An overview of the prototype is given in [Olsen 92a].

6.1.1 Terminology and background

A *persistent object system* is an infrastructure which enables application systems to treat objects which are stored on secondary storage as in-memory objects. It automatically moves objects, on demand, between secondary storage and main memory and thus eliminates the *impedance mismatch* between the type system of application systems and the type system of secondary storage abstractions like databases and file systems [Copeland 84]. Invocations performed on interface instances of stored objects can be extremely slow because stored objects must be moved in-memory before they can provide a requested service; the challenge is therefore to make persistent object systems run fast.

To move an in-memory object to secondary storage, a persistent object system must change the object's representation so it complies with the type system of a secondary storage abstraction. To facilitate the migration of objects, the chosen secondary storage representation should comply with the type systems of networks. A non-interpreted sequence of bytes is an example of such a representation.

A sequence of bytes which represents a single object is termed a *snapshot* of that particular object. The snapshots considered in this paper comprises three parts: (i) an object type component denoting an implementation template; (ii) a state component denoting the current state of an object; and (iii) a schema component denoting the set of interface instances currently supported by an object. A snapshot of an object is different from a *checkpoint*; whereas a snapshot represents a single object, a checkpoint is a set of mutually consistent snapshots. A checkpoint of an object is recursively defined to consist of a snapshot of that object and a checkpoint of each object which has an interface instance referred to by a reference in the initial object. If an object has no references to interface instances in other objects, a snapshot of that object is also a checkpoint.

By representing an object which currently performs no activities as a snapshot stored on secondary storage, the in-memory resources occupied by the object can be freed; the in-memory representation need not be available until the object must perform activities again.

6.1.2 Problem statement

In [Olsen 91b] it is shown how an application programmer can extend an example application so an object supports an operation which produces a snapshot of its state, and an operation which installs a snapshot as its state. However, to do this, the application programmer needs to write a considerable amount of code.

6.1.3 Purpose

The purpose of this paper is to describes how the code for supporting snapshot operations can be generated automatically, so the application programmer does not have to write it.

6.1.4 Main ideas

The snapshot operations are generated by the prepc preprocessor. They are implemented in terms of calls to marshal and unmarshal procedures which are generated by the stub compiler. The application specific details required for generating the snapshot operations must be provided by the application programmer in a single statement which is processed by the preprocessor. These details are (i) which variables refer to snapshot state; (ii) which IDL types these variables have; (iii) which interface the snapshot operations are to be provided in; and (iv) the file name of a template from which the snapshot producing object was instantiated.

6.1.5 Limitations and related work

In general, a snapshot of an object consists of an object type component, a state component, and a schema component. In this paper a snapshot is restricted to be of idle objects only. This simplifies the contents of a snapshot as the state component merely consists of that object state which is globally accessible from operations in the object's interfaces. In addition, the object type component is required to be available as an executable via NFS; this also simplifies the contents of a snapshot as the object type component need only be a file name. When a snapshot is installed in an object, the schema component will not be used for establishing a set of interface instances in the object. This is because a location service is needed to enable clients to replace references to interface instances of objects from which a snapshot has been produced, with references to interface instances of objects in which the snapshot has been installed. A separate paper describes a prototype of a location service, see [Olsen 92c].

Concurrency control mechanisms are required to ensure that snapshots are only produced when no other activities are in progress in an object except the activity required for producing a snapshot. Such mechanisms are currently being developed within the ISA project [Reese 91], therefore the work presented in this paper does not provide any application independent concurrency control.

6.1.6 Related work

The restricted snapshot is similar to the support in Arjuna for storing object state [Shrivastava 89]. However, an Arjuna object has one access point only (a single interface) and therefore Arjuna does not have to deal with the problems of a dynamic set of access points (multiple interfaces).

The approach of converting application values into sequences of bytes by using routines similar to those found in stubs for remote procedure calls was suggested in [Birrell 87] where the mechanism is termed *pickles*.

6.1.7 Audience and prerequisites

This paper targets system developers and programmers who are designing and implementing persistent object systems. It also addresses application programmers who need support for snapshot operations. The reader should be acquainted with ANSA [Warne 91], The ANSA Testbench3.0 [AIM 91], The ANSA Storage Model [Olsen 91], and the prototype overview [Olsen 92a].

6.1.8 Organisation

Section 2 describes how the stub compiler, stubc, is modified so it generates separate marshal and unmarshal procedures for IDL types, which can be used to convert application state into a sequence of bytes and vice versa. Section 3 describes how the snapshot operations can be implemented in terms of calls to these procedures. In section 4 and 5 it is described how the preprocessor, prepc, is modified so it generates snapshot operations and calls for recording the creation and destruction of interface instances. In section 6 an example application is used to show what the application programmer needs to specify to get the snapshot operations generated. Section 7 concludes the paper. The extensions made to stubc and prepc are described in detail in appendix A. Appendix B and C demonstrate in detail how to get snapshot operations generated in an object.

6.2 Marshal and unmarshal procedures for IDL types

Remote procedure call mechanisms employ marshalling and unmarshalling stubs to transform application data structures into sequences of bytes and vice versa. As argued in [Olsen 92b], it is necessary that a snapshot is represented in terms of a type which is generic to the contents of a snapshot, and it is convenient that this format conforms to the format supported by networks and storage devices. This is achieved by representing a snapshot as a sequence of bytes. To some extent, this format also prevents that the state encapsulation principle of the object-oriented paradigm is violated. To convert application state into a sequence of bytes and vice versa, [Olsen 92b] exploited the S_ procedures in the server stub that stubc generated from a .idl file which specifies an object's representation in terms of IDL. An application programmer does not have to specify the object representation in terms of IDL, if stubc is modified so it generates separate marshal and unmarshal procedures for IDL types.

6.2.1 Modifying stubc to generate marshal and unmarshal procedures

stubc is modified so it produces two additional files: mFoo.c and mFoo.h from a file Foo.idl. The mFoo.c file contains two procedures M_T and U_T for each IDL type T which is defined in Foo.idl. The mFoo.h file contains external declarations for the M_ and U_ procedures. Informally, the semantics of the M_T and U_T procedures are as follows for a given IDL type T:

```
void M_T(var, _buf)
T var;
ansa_BufferLink _buf;
```

```
{ ansa_BufferLink _buf';

  marshall var into _buf';
  append _buf' to _buf;
}

void U_T(var, _buf)
T *var;
ansa_BufferLink _buf;
{ garbage collect memory which has been allocated dynamically for
  state referred to by var;
  unmarshall _buf into *var;
  remove the unmarshalled front bytes from _buf;
}
```

The M_ and U_ procedures differ from ordinary marshal and unmarshal stubs in that they do not operate on memory managed in a stub. Typically, a stub records all the memory it allocates dynamically during unmarshalling in order to free the memory when an upcall to the application procedure returns. The memory allocated dynamically by the U_ procedure has not been freed when a call to U_ returns, because this would free the memory which holds the application state which has just been unmarshalled into an application variable. Instead, the U_ procedure frees the dynamically allocated memory which holds the application state which is to be replaced by unmarshalled snapshot state. This means that references (except the variable into which a snapshot is unmarshalled) which refer to memory segments which holds snapshot state, are no longer valid after unmarshalling. Appendix A describes the required modifications to stubc in detail.

Stubc only needs to produce M_ and U_ procedures for those IDL types in an interface specification which are used to declare variables which will refer to snapshot state. The current practice of generating M_ and U_ procedures for each IDL type was chosen because it required less changes to stubc than other alternatives. Alternatively, the programmer could attribute the IDL types of which there will be variables referring to snapshot state, or, the specification of an interface could indicate that M_ and U_ procedures should be generated for all IDL types defined in it. Both of these alternatives would require modifications to the IDL grammar.

The time for building executables for an application and its memory requirement can be reduced if M_ and U_ procedures for the basic IDL types exists in a library which is linked in with M_ and U_ procedures for non-basic IDL types.

6.3 The produceSnapshot and installSnapshot procedures

If a snapshot is to contain the application state referred to by the variables v1, v2, and v3 which has IDL types IDL1, IDL2, and IDL3 respectively, then the produceSnapshot and installSnapshot procedures can now be implemented as follows:

```
void produceSnapshot(_buf)
ansa_BufferLink _buf;
{ init(_buf);
  M_IDL1(v1, _buf);
  M_IDL2(v2, _buf);
  M_IDL3(v3, _buf);
```

```
  M_InterfaceSet(myInterfaceSet, _buf);
}

void installSnapshot(_buf)
ansa_BufferLink _buf;
{ U_IDL1(&v1, _buf);
  U_IDL2(&v2, _buf);
  U_IDL3(&v3, _buf);
  U_InterfaceSet(&myInterfaceSet, _buf);
  /* object should now create interfaces corresponding
   * to those recorded in myInterfaceSet */
}
```

In section 4 it is described how the produceSnapshot and installSnapshot operations can be generated automatically by prepc, if the application programmer specifies application variables and their IDL types in a prepc statement[1].

6.4 A prepc statement for declaring a snapshot

Using the following SNAPSHOT prepc statement, the application programmer must declare what variables refer to the application state which is to be part of a snapshot:

```
! SNAPSHOT OF {v1 IDLType1; ...; vN IDLTypeN} IN ifType OF objType
```

From this statement, prepc generates the produceSnapshot and installSnapshot operations. They will be provided in interface instances of type ifType. objType must be the name of the file from which the object containing the SNAPSHOT statement is instantiated. Thus objType is the object type component of a snapshot. The interaction part of a snapshot is recorded by code generated by prepc as described in section 5. Appendix A describes the modifications to prepc in detail.

It is the programmers responsibility to ensure that memory referred to by the variables declared in a SNAPSHOT statement is not referred to by other variables. If some memory is referred to by other variables, then these variables will no longer be valid when installSnapshot has been invoked, because the memory will have been garbage collected and replaced by some other memory holding the installed state. If two or more variables in a SNAPSHOT statement refer to the same memory segment (i.e. the state referred to is shared), then the produced snapshot will contain separate copies of the memory segments for each variable. This means, that state which was shared before a snapshot is produced, will not be shared after a snapshot has been installed.

In addition specifying a SNAPSHOT statement, the programmer must also include the line

```
IS COMPATIBLE WITH SnapshotOp
```

in the IDL file which defines ifType in the SNAPSHOT statement. SnapshotOp is a library interface defined by:

```
SnapshotOp: INTERFACE =
BEGIN
  Snapshot: TYPE = SEQUENCE OF OCTET;
```

1. The application programmer therefore writes less code than in Arjuna where the application code must implement save_state and restore_state operations in terms of pack and unpack operations for non-basic types [Shrivastava 89].

```
  produceSnapshot: OPERATION []
                   RETURNS [snapshot: Snapshot];
  installSnapshot: OPERATION [snapshot: Snapshot]
                   RETURNS [];
END.
```

6.5 Recording interaction state

In [Olsen 92b] it is explained that an object must record its interface instances so they can be made part of a snapshot's schema component. A library of operations that operate on a data structure which contains interface instances is implemented for this purpose, and the application programmer must call addMyInterfaces or remMyInterfaces whenever an interface instance is created or destroyed in an object. These calls can be automatically generated by modifying prepc. When a SNAPSHOT statement is preprocessed, the generated code includes library procedures for adding and removing interface instances recorded in a data structure. The prepc statement for creating an interface instance

```
! {ifRef} :: InterfaceTypeName$Create(concurrency)
```

is preprocessed by the modified prepc so the generated code makes the call

```
addMyInterfaces(&myInterfaces, &ifRef, "InterfaceTypeName", \
                concurrency)
```

The prepc statement for destroying an interface instance

```
! {} :: InterfaceTypeName$Destroy(ifRef)
```

is preprocessed by the modified prepc so the generated code makes the call

```
remMyInterfaces(&myInterfaces, &ifRef)
```

6.6 An example application

An example application, which was used in [Olsen 92b] to demonstrate how an application programmer can extend an application to support snapshot operations, is re-used here to demonstrate, what the application programmer needs to specify, in order that snapshot operations are automatically generated by stubc and prepc. The example application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It exports a reference to a PhoneBookOp interface instance to a trader, which enables a client to insert, remove, lookup and list names and phone numbers. Appendix B contains the complete code for the example application.

6.6.1 Type conformance

The application programmer needs to make PhoneBookOp compatible with snapshot operations as follows

```
PhoneBookOp: INTERFACE =
NEEDS PhoneBookTypes;
IS COMPATIBLE WITH SnapshotOp;
BEGIN
  insert: OPERATION [entry: PBEntry]
          RETURNS [];
  remove: OPERATION [name: Name]
          RETURNS [];
```

```
    lookup: OPERATION [name: Name]
            RETURNS [entry : PBEntry];
    list:   OPERATION []
            RETURNS [entryList: PBEntryList];
  END.
```

The "IS COMPATIBLE WITH SnapshotOp;" statement implies that the produceSnapshot and installSnapshot operations must be implemented by two operations PhoneBookOp_produceSnapshot and PhoneBookOp_installSnapshot in the server. The SnapshotOp interface is defined in a .idl file in /master/include/idl in the master tree of Testbench 3.0.

6.6.2 Stating the need for snapshot operations

The application programmer must add a SNAPSHOT statement in the server code so the preprocessor can generate the implementation of PhoneBookOp_produceSnapshot and PhoneBookOp_installSnapshot in the server. Therefore the following must be added to server.dpl:

```
! SNAPSHOT OF {entryList PBEntryList} IN PhoneBookOp OF server
```

Appendix C contains the code for the example application equipped with the statements an application programmer needs to specify to get snapshot operations generated.

6.7 Conclusion

It has been shown how snapshot operations can be automatically generated by prepc and stubc, if the application programmer specifies (i) which variables refer to snapshot state, (ii) which IDL types these variables have, (iii) which interface the snapshot operations are to be provided in, and (iv) the file name of the template from which the object which provides the interface is to be instantiated from. The snapshot operations are generated by prepc, and are implemented in terms of calls to procedures generated by stubc. With a few modifications to prepc, a SNAPSHOT statement is preprocessed to generate the snapshot operations. The preprocessing of interface creation and destruction has been extended to generate calls that modify a data structure which records the interface instances of an object. Some simple modifications to stubc generate appropriate files which define marshal and unmarshal procedures for each IDL type in a .idl file. Appendix A describes the extensions made to prepc and stubc.

Compared to the way an application programmer can handcraft snapshot operations for an object, as described in [Olsen 92b], the programmer's involvement in producing the snapshot code is now reduced to the specification of two statements: the SNAPSHOT statement in a server's .dpl file, and the "IS COMPATIBLE WITH SnapshotOp" statement in an .idl file. Appendix B and C demonstrate in detail how to get snapshot operations generated in an object.

The modified stubc and prepc implementations are available from the author; the code has prototype status and comes with no warranty.

6.8 Acknowledgements

Thanks to David Iggulden of the ISA Core Team and Nigel Edwards of Hewlett-Packard Laboratories for commenting on an earlier draft of this paper.

6.9 References

[AIM 91]
"ANSAware 3.0 Implementation Manual", Report No.: RM.097, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Birrell 87]
Andrew D. Birrell, Michael B. Jones, Edward P. Wobber, ”A Simple and Efficient Implementation for Small Databases”, Operating Systems Review, Vol. 21, No. 5, (1987).

[Copeland 84]
George Copeland, David Maier, "Making Smalltalk a Database System", *SIGMOD’84*, Sigmod Record, Vol. 14, No. 2, (1984).

[Olsen 91]
Michael Hoffmann Olsen, “A Model for Storage and resource Management”, Report No.: WP-DCG-91-14, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol, England, (1991).

[Olsen 92a]
Michael Hoffmann Olsen, “A Persistent Object Infrastructure for Heterogeneous Distributed Systems”, Report No.: RC.343, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92b]
Michael Hoffmann Olsen, “Handcrafting Snapshot Operations for an Object”, Report No.: RC.280, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92c]
Michael Hoffmann Olsen, “Roadmap to the Storage Prototype Deliverables T18, T21 and T30”, Report No.: RC.342, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Reese 91]
Owen Reese, “Using path expressions as concurrency guards”, Report No.: RC.248, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Shrivastava 89]
S. K. Shrivastava et al, “The Treatment of Persistent Objects in Arjuna”, THE COM-

PUTER JOURNAL, VOL. 32, NO. 4, (1989).

[Warne 91]
John Warne, "ANSA: Assumptions, Principles, and Structure", Conference proceedings of Software Engineering Environments 1991, University College of Wales, Aberystwyth, (1991).

7 Appendix: Automated generation of snapshot operations for an object

7.1 stubc and prepc changes

This appendix describes the extensions which have been made to stubc and prepc to generate the code for snapshot operations.

7.1.1 stubc

The two files gram.y and outsubs.c are extended as follows:

7.1.1.1 gram.y

- main has been modified to accept an additional option m/M. If this flag is set or no flags are set then main will call the routine outputmarshallcode

7.1.1.2 outsubs.c

- the routine outputmarshallcode has been added. It calls outputmarshallexternals and then generates the content of a file mIDLTYPE.c, where IDLTYPE is the name of the .idl file which was given as argument to stubc. First the M_ and U_ routines for basic idl types are generated by calling declaremarshallfunc for each type, then the M_ and U_ routines for user defined idl types are generated by calling declaremarshallfunc. Each of the two sets of generated routines are guarded by #ifndef following the conventions for stub generation
- the routine outputmarshallexternals has been added. It generates the file mIDLTYPE.h containing the external declarations for the routines in mIDLTYPE.c
- the routine declaremarshallfunc has been added. It takes a representation of a parsed IDL type as argument and generates the M_ and U_ routines for it. It ignores intermediate types generated by the parser; these are detected by a call to the existing function istemp. Intermediate types are not declared in a .idl file and thus no M_ or U_ routines will be needed for them. declaremarshallfunc first generates a M_ function by (i) generating its header and local variable declarations, (ii) generating the loop statement and the initialization of variables, (iii) calling the existing routine marshall, (iv) generating the end of the loop, and (v) generating the end of the M_ routine; then an U_ routine is generated by (i) generating its header and local variable declarations, (ii) calling the routine garbageCollect, (iii) generating the initialization of local variables, (iv) calling the existing routine unmarshall, and (v) generating the code for constructing the result
- the routine garbageCollect has been added. It takes a representation of a parsed IDL type and a variable name and frees any memory which has been dynamically allocated for the variable. It works by recursively descending the structure of the IDL type. Types other than ARRAY,

SEQUENCE, RECORD, CHOICE, ALIAS and INTERFACEREF are ignored and thus stops the recursion

7.1.2 prepc

The file gram.y is extended as follows:

- the global variables TypeListHead and VarListHead of type ListHead have been added
- the grammar for the non-terminal statement has been extended to enable the derivation of:

```
SNAPSHOT OF statelist IN TOKEN OF TOKEN
```

- the following new non-terminals with their associated derivations have been added to the grammar:

```
statelist : '{' vtlist '}' ;
vtlist: pair | vtlist ',' pair ;
pair: TOKEN TOKEN {AddStringToList(&VarListHead, $1);
                   AddStringToList(&TypeListHead, $2);}
```

- the routine translate has been extended with the following two statements:

```
InitListHead(&VarListHead);
InitListHead(&TypeListHead);
```

- the routine yylex has been extended so the lines:

```
case SNAPSHOT:
case OF:
case IN:
```

is added to the switch alternatives for value

- the routine dolex has been extended with the following lines for constructing a return value:

```
if (strcmp(tok, "SNAPSHOT") == 0)
   return SNAPSHOT;
if (strcmp(tok, "OF") == 0)
   return OF;
if (strcmp(tok, "IN") == 0)
   return IN;
```

- the routine freetokens has been extended with the following lines for freeing memory allocated to VarListHead and TypeListHead:

```
FreeList(&VarListHead, 0);
FreeList(&TypeListHead, 0);
```

- the routine processsnapshot has been added. It takes two strings as arguments; the first is the interface name which followed IN in a SNAPSHOT statement, the second is the template name which followed OF in that statement. It first checks that the interface name as been declared as SERVER, otherwise stubc terminates with an error. Then some include statements are generated:

```
#include "mifname.c"
#include "RecordMyInterfaces.c"
```

where ifname is the interface name.

Then a produceSnapshot routine is generated containing a call to a M_ stub routine for each pair of type name and variable name in TypeListHead and VarListHead, and an installSnapshot routine is generated containing a call to an U_ routine for each pair of type name and variable name in TypeListHead and VarListHead..

- The routine outputinstantiation has been extended so a CREATE operation generates a call to addMyInterfaces, and a DESTROY operation generates a call to remMyInterfaces

7.2 The example application code

This appendix contains the code for the example application referred to in this paper. The application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It exports a reference to a PhoneBookOp interface instance to a trader, which enables a client to insert, remove, lookup and list names and phone numbers. The implementation of the server's operations has been omitted as they are independent of the snapshot mechanisms which are added in appendix C. The last section of the appendix contains example executions of the application.

7.2.1 include files

7.2.1.1 ConcControl.h

```
extern void initConcControl();
extern void cleanUpConcControl();
extern void getSingleWriteLock();
extern void releaseSingleWriteLock();
extern void getMultipleReadLock();
extern void releaseMultipleReadLock();
```

7.2.1.2 ConcControl.c

```
#include "ansa.h"
#include "opsys.h"
#include "machine.h"
#include "ecs.h"
#include "ConcControl.h"

ansa_EventCount doneReadorWrite, doneNext;
ansa_Sequencer nextReadorWrite, next;

void initConcControl()
{ doneReadorWrite = ecs_makeEventCount((ansa_Cardinal) 0);
  doneNext        = ecs_makeEventCount((ansa_Cardinal) 0);
  nextReadorWrite = ecs_makeSequencer((ansa_Cardinal) 0);
  next            = ecs_makeSequencer((ansa_Cardinal) 0);
}

void cleanUpConcControl()
{ ecs_freeEventCount(doneReadorWrite);
  ecs_freeEventCount(doneNext);
  ecs_freeSequencer(nextReadorWrite);
  ecs_freeSequencer(next);
}

void getSingleWriteLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_await(doneNext, ecs_ticket(next));
}

void releaseSingleWriteLock()
```

```
{ ecs_advance(doneReadorWrite);
  ecs_advance(doneNext);
}

void getMultipleReadLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_ticket(next);
  ecs_advance(doneReadorWrite);
}

void releaseMultipleReadLock()
{ ecs_advance(doneNext);
}
```

7.2.2 idl files

7.2.3 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

7.2.3.1 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

7.2.4 dpl files

7.2.4.1 server.dpl

```
! MANAGED
! USE Trader
! USE PhoneBookOp
! DECLARE { phoneBook, pbRef } : PhoneBookOp SERVER

#include <stdio.h>
#include "ConcControl.h"
#include "capsule.h"
#include "system.h"

#define POTENTIALCONCURRENCY  16
```

```
#define ACTUALCONCURRENCY     24 + 16
GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

PBEntryList entryList;
ansa_InterfaceRef phoneBook;

void copyIfRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

ansa_InterfaceRef ref[1];

ansa_StatePtr Create__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ Result tres;
  char hn[64], bf[64 + 11];

  initConcControl();
! {phoneBook} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  entryList.length = 0;
  entryList.data   = (PBEntry *)0;
  gethostname(hn, sizeof(hn));
  (void)sprintf(bf, "NodeName '%s'", hn);
! {tres} <- traderRef$Register("PhoneBookOp", "/ansa/PhoneBook",
\
                               bf, phoneBook)
  copyIfRef(&(ref[0]), &phoneBook);
  results->length = 1;
  results->data = ref;
  return (ansa_StatePtr)0;
}

void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
Name name;
{ getSingleWriteLock();
  /* remove entry identified by name from entryList */;
  releaseSingleWriteLock();
  return 1;
```

```
}

int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

7.2.4.2 client.dpl

```
! USE Capsule
! DECLARE {cref} : Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook, pbRef} : PhoneBookOp CLIENT

#include   <stdio.h>
#include "capsule.h"
#include "system.h"

#define MAXANSWERLENGTH 31

ansa_InterfaceRef phoneBook;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "create",
  "insert",
  "remove",
  "lookup",
  "list",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {3, 4, 3, 3, 2};
int maxargs[] = {3, 5, 4, 4, 3};

#define INVALID_CMD -1

PBEntryList eList;

void listEntryList(entryList)
PBEntryList *entryList;
{ int        i;
```

```
  char       buffer[128];
  PBEntry  *ifr = entryList->data;

  fprintf(stdout, "\nENTRIES IN PHONEBOOK:\n");
  for (i=0; i<entryList->length; i++, ifr++)
  fprintf(stdout, "  %s %s\n", ifr->name, ifr->no );
}

void PrintUsageAndDie(prog)
char *prog;
{ fprintf(stderr, "usage: %s create machine\n", prog);
  fprintf(stderr, "usage: %s insert name no <property>\n", prog);
  fprintf(stderr, "usage: %s remove name <property>\n", prog);
  fprintf(stderr, "usage: %s lookup name <property>\n", prog);
  fprintf(stderr, "usage: %s list <property>\n", prog);
! capsule$Terminate()
}

void copyIfRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ ansa_InterfaceRef pbRef;
  int i, cmd_no;
  char property[128];

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  strcpy(property, "");
  switch(cmd_no)
  {  case 1: /* insert */
       { if (argc = 5)
         strcpy(property, argv[4]);
```

```
        }
        break;
      case 2: /* remove */
        { if (argc = 4)
          strcpy(property, argv[3]);
        }
        break;
      case 3: /* lookup */
        { if (argc = 4)
          strcpy(property, argv[3]);
        }
        break;
      case 4: /* list */
        { if (argc = 3)
          strcpy(property, argv[2]);
        }
        break;
  }

  if (cmd_no != 0)
  { /* get reference to PhoneBookOp */
!   {pbRef} <- traderRef$Import("PhoneBookOp", "/ansa/
PhoneBook",\
                                property)
    copyIfRef(&phoneBook, &pbRef);
  }

  switch(cmd_no)
  {   case 0: /* create */
        { InstantiateResult res;
          ObjectId id;
          ansa_InterfaceRef fref, cref;
          char machine[128];

          sprintf(machine, "NodeName == '%s'", argv[2]);
!         {fref} <- traderRef$Import("Factory", "/", machine)
!         {cref} <- fref$Instantiate("PhoneBook", "", "")
!         {id, res} <- cref$Instantiate(nullRef, "0", "")
        }
        break;
      case 1: /* insert */
        { PBEntry entry;

          strcpy(entry.name, argv[2]);
          strcpy(entry.no, argv[3]);
!         {} <- phoneBook$insert(entry)
        }
        break;
      case 2: /* remove */
        { Name name;

          strcpy(name, argv[2]);
!         {} <- phoneBook$remove(name)
        }
        break;
      case 3: /* lookup */
        { PBEntry reply;
```

```
!         {reply} <- phoneBook$lookup(argv[2])
          if (strcmp(reply.name, "0") == 0)
         { fprintf(stdout, "Sorry, no entry for %s\n", argv[2]);
          }
          else
          { fprintf(stdout, "No: %s\n", reply.no);
          }
        }
        break;
      case 4: /* list */
        { PBEntryList entryList;

!         {entryList} <- phoneBook$list()
          listEntryList(&entryList);
        }
        break;
  }
}
```

7.2.5 Imakefile

```
          DEFINES =
         INCLUDES =
         IDLFLAGS =
         DPLFLAGS =
         LINTLIBS = $(LINTANSALIB) -lc
         LOCALLIB = $(ANSALIB)
        LOCALLIBD = $(ANSALIBD)
          INSTALL = bsdinstall.sh

IDLFILES = PhoneBookOp.idl
SIFFILES = PhoneBookOp.sif
DPLFILES = server.dpl client.dpl
SRCS     = ConcControl.c
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)

PROGS = server client

# compile idl files
all:: $(SIFFILES)

# compile dpl files
DPLDepend(server)
DPLDepend(client)

all:: $(PROGS)

# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)

SingleProgramTarget(server, server.o sPhoneBookOp.o \
                    ConcControl.o,$(LOCALLIB),)
SingleProgramTarget(client, client.o
cPhoneBookOp.o,$(LOCALLIB),)

InstallProgram(server,$(TEMPLATEDIR))
```

```
NormalLintTarget(*.c)
DependTarget()

IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

7.2.6 Example executions

This section shows the results of some executions of the client while a server is running. Output from client executions are printed in italics.

7.2.6.1 server creation

```
client create
```

7.2.6.2 client executions

```
client insert Michael_Olsen  0223323010
client insert Audrey_Hepburn +14156666660
client insert Ronald_Reagan +14156478231
client list
ENTRIES IN PHONEBOOK:
  Michael_Olsen  0223323010
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231
client lookup Michelle_Pheiffer
Sorry, no entry for Michelle_Pheiffer
client remove Michael_Olsen
client list
ENTRIES IN PHONEBOOK:
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231
Session lookup Audrey_Hepburn
No: +14156666660
```

7.3 Providing snapshot operations in the server

This appendix contains the code for the example application in appendix B which is modified with statements that result in snapshot operations being provided. In addition, the client has been modified to enable it to test the snapshot operations supported by the server. The last section contains example executions.

7.3.1 include files

7.3.1.1 ConcControl.h

```
extern void initConcControl();
extern void cleanUpConcControl();
extern void getSingleWriteLock();
extern void releaseSingleWriteLock();
extern void getMultipleReadLock();
extern void releaseMultipleReadLock();
```

7.3.1.2 ConcControl.c

```
#include "ansa.h"
#include "opsys.h"
#include "machine.h"
```

```
#include "ecs.h"
#include "ConcControl.h"

ansa_EventCount doneReadorWrite, doneNext;
ansa_Sequencer nextReadorWrite, next;

void initConcControl()
{ doneReadorWrite = ecs_makeEventCount((ansa_Cardinal) 0);
  doneNext        = ecs_makeEventCount((ansa_Cardinal) 0);
  nextReadorWrite = ecs_makeSequencer((ansa_Cardinal) 0);
  next            = ecs_makeSequencer((ansa_Cardinal) 0);
}

void cleanUpConcControl()
{ ecs_freeEventCount(doneReadorWrite);
  ecs_freeEventCount(doneNext);
  ecs_freeSequencer(nextReadorWrite);
  ecs_freeSequencer(next);
}

void getSingleWriteLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_await(doneNext, ecs_ticket(next));
}

void releaseSingleWriteLock()
{ ecs_advance(doneReadorWrite);
  ecs_advance(doneNext);
}

void getMultipleReadLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_ticket(next);
  ecs_advance(doneReadorWrite);
}

void releaseMultipleReadLock()
{ ecs_advance(doneNext);
}
```

7.3.2 idl files

7.3.2.1 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

7.3.2.2 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
```

```
IS COMPATIBLE WITH SnapshotOp;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

7.3.3 dpl files

7.3.3.1 server.dpl

```
! MANAGED
! USE Trader
! USE PhoneBookOp
! DECLARE { phoneBook, pbRef } : PhoneBookOp SERVER

#include <stdio.h>
#include "ConcControl.h"
#include "capsule.h"
#include "system.h"

#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16
GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

PBEntryList entryList;
ansa_InterfaceRef phoneBook;

! SNAPSHOT OF {entryList PBEntryList} IN PhoneBookOp OF server

ansa_InterfaceRef ref[1];

ansa_StatePtr Create__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ Result tres;
  char hn[64], bf[64 + 11];

  initConcControl();
! {phoneBook} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  entryList.length = 0;
  entryList.data   = (PBEntry *)0;
  gethostname(hn, sizeof(hn));
  (void)sprintf(bf, "NodeName '%s'", hn);
! {tres} <- traderRef$Register("PhoneBookOp", "/ansa/PhoneBook",
\
                               bf, phoneBook)
  copyIfRef(&(ref[0]), &phoneBook);
  results->length = 1;
  results->data = ref;
  return (ansa_StatePtr)0;
}
```

```
void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
Name name;
{ getSingleWriteLock();
  /* remove entry identified by name from entryList */;
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

7.3.3.2 client.dpl

```
! USE Capsule
! DECLARE {cref} : Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook, pbRef} : PhoneBookOp CLIENT

#include   <stdio.h>
#include "capsule.h"
#include "system.h"

#define MAXANSWERLENGTH 31

ansa_InterfaceRef phoneBook;
```

```
/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "create",
  "insert",
  "remove",
  "lookup",
  "list",
  "movesnapshot",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {3, 4, 3, 3, 2, 4};
int maxargs[] = {3, 5, 4, 4, 3, 4};

#define INVALID_CMD -1

PBEntryList eList;

void listEntryList(entryList)
PBEntryList *entryList;
{ int        i;
  char       buffer[128];
  PBEntry  *ifr = entryList->data;

  fprintf(stdout, "\nENTRIES IN PHONEBOOK:\n");
  for (i=0; i<entryList->length; i++, ifr++)
  fprintf(stdout, "  %s %s\n", ifr->name, ifr->no );
}

void PrintUsageAndDie(prog)
char *prog;
{ fprintf(stderr, "usage: %s create machine\n", prog);
  fprintf(stderr, "usage: %s insert name no <property>\n", prog);
  fprintf(stderr, "usage: %s remove name <property>\n", prog);
  fprintf(stderr, "usage: %s lookup name <property>\n", prog);
  fprintf(stderr, "usage: %s list <property>\n", prog);
  fprintf(stderr, "usage: %s snapshottest name <property>\n",
prog);
! capsule$Terminate()
}

void copyIfRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ ansa_InterfaceRef pbRef;
  int i, cmd_no;
  char property[128];

  /* check number of parameters */
```

```
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  strcpy(property, "");
  switch(cmd_no)
  {  case 1: /* insert */
       { if (argc = 5)
         strcpy(property, argv[4]);
       }
       break;
     case 2: /* remove */
       { if (argc = 4)
         strcpy(property, argv[3]);
       }
       break;
     case 3: /* lookup */
       { if (argc = 4)
         strcpy(property, argv[3]);
       }
       break;
     case 4: /* list */
       { if (argc = 3)
         strcpy(property, argv[2]);
       }
       break;
  }

  if ((cmd_no != 0) && (cmd_no != 5))
  { /* get reference to PhoneBookOp */
!   {pbRef} <- traderRef$Import("PhoneBookOp", "/ansa/PhoneBook",
\
                                  property)
    copyIfRef(&phoneBook, &pbRef);
  }

  switch(cmd_no)
  {   case 0: /* create */
        { InstantiateResult res;
          ObjectId id;
          ansa_InterfaceRef fref, cref;
          char machine[128];
```

```
          sprintf(machine, "NodeName == '%s'", argv[2]);
!         {fref} <- traderRef$Import("Factory", "/", machine)
!         {cref} <- fref$Instantiate("PhoneBook", "", "")
!         {id, res} <- cref$Instantiate(nullRef, "0", "")
        }
        break;
      case 1: /* insert */
        { PBEntry entry;

          strcpy(entry.name, argv[2]);
          strcpy(entry.no, argv[3]);
!         {} <- phoneBook$insert(entry)
        }
        break;
      case 2: /* remove */
        { Name name;

          strcpy(name, argv[2]);
!         {} <- phoneBook$remove(name)
        }
        break;
      case 3: /* lookup */
        { PBEntry reply;

!         {reply} <- phoneBook$lookup(argv[2])
          if (strcmp(reply.name, "0") == 0)
         { fprintf(stdout, "Sorry, no entry for %s\n", argv[2]);
          }
          else
          { fprintf(stdout, "No: %s\n", reply.no);
          }
        }
        break;
      case 4: /* list */
        { PBEntryList entryList;

!         {entryList} <- phoneBook$list()
          listEntryList(&entryList);
        }
        break;
      case 5:
        { Name name;
          Snapshot snapshot;
          PBEntryList entryList;

!         {phoneBook} <- traderRef$Import("PhoneBookOp", \
                                 "/ansa/PhoneBook", argv[2])
!         {snapshot} <- phoneBook$produceSnapshot()
!         {phoneBook} <- traderRef$Import("PhoneBookOp", \
                                "/ansa/PhoneBook", argv[3])
!         {} <- phoneBook$installSnapshot(snapshot)
        }
        break;
  }
}
```

7.3.4 Imakefile

```
           STIDIR = ../../../../snapshot/transparency/include
           STLDIR = ../../../../snapshot/transparency/idl
            STUBC = ../../../../snapshot/stubc/stubc
            PREPC = ../../../../snapshot/prepc/prepc
          DEFINES =
         INCLUDES = -I$(STIDIR)
         IDLFLAGS = -I$(STLDIR)
         DPLFLAGS =
         LINTLIBS = $(LINTANSALIB) -lc
         LOCALLIB = $(ANSALIB)
        LOCALLIBD = $(ANSALIBD)
          INSTALL = bsdinstall.sh

IDLFILES = PhoneBookOp.idl
SIFFILES = PhoneBookOp.sif
DPLFILES = server.dpl client.dpl
SRCS     = ConcControl.c
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)

PROGS = server client

# compile idl files
all:: $(SIFFILES)

# compile dpl files
DPLDepend(server)
DPLDepend(client)

all:: $(PROGS)

# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)

SingleProgramTarget(server, server.o sPhoneBookOp.o \
                    ConcControl.o,$(LOCALLIB),)
SingleProgramTarget(client, client.o
cPhoneBookOp.o,$(LOCALLIB),)

InstallProgram(server,$(TEMPLATEDIR))
NormalLintTarget(*.c)
DependTarget()
IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

7.3.5 Example run

This section shows the results of some executions of the client while a server is running. Output from client executions are printed in italics.

7.3.5.1 server creation

```
client create machine1
```

7.3.5.2 client executions

When given *movesnapshot* as argument, the client obtains a snapshot from one server and installs it in another.

```
client insert Michael_Olsen  0223323010 "NodeName == 'machine1'"
client insert Audrey_Hepburn +14156666660 "NodeName ==
'machine1'"
client insert Ronald_Reagan +14156478231 "NodeName == 'machine1'"
client list "NodeName == 'machine1'"
ENTRIES IN PHONEBOOK:
  Michael_Olsen  0223323010
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231

client create machine2
client list machine2
ENTRIES IN PHONEBOOK:


client movesnapshot "NodeName == 'machine1'" "NodeName ==
'machine2'"
client list "NodeName == 'machine2'"
ENTRIES IN PHONEBOOK:
  Michael_Olsen 0223323010
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231

client list "NodeName == 'machine1'"
ENTRIES IN PHONEBOOK:
  Michael_Olsen  0223323010
  Audrey_Hepburn +14156666660
  Ronald_Reagan +14156478231
```

8 Design and Implementation of a SnapshotBase

8.1 Introduction

This paper is the third in a series of eight papers describing a prototype of the ANSA Storage Model [Olsen 91a] developed in the ANSA Testbench 3.0 [AIM 91]. An overview of the prototype is given in [Olsen 92a].

8.1.1 Terminology and background

A *persistent object system* is an infrastructure which enables application systems to treat objects which are stored on secondary storage as in-memory objects. It automatically moves objects, on demand, between secondary storage and main memory and thus eliminates the *impedance mismatch* between the type system of application systems and the type system of secondary storage abstractions like databases and file systems [Copeland 84]. Invocations performed on interface instances of stored objects can be extremely slow because stored objects must be moved in-memory before they can provide a requested service; the challenge is therefore to make persistent object systems run fast.

To move an in-memory object to secondary storage, a persistent object system must change the object's representation so it complies with the type system of a secondary storage abstraction. To facilitate the migration of objects, the chosen secondary storage representation should comply with the type systems of networks. A non-interpreted sequence of bytes is an example of such a representation.

A sequence of bytes which represents a single object is termed a *snapshot* of that particular object. The snapshots considered in this paper comprises three parts: (i) an object type component denoting an implementation template; (ii) a state component denoting the current state of an object; and (iii) a schema component denoting the set of interface instances currently supported by an object. A snapshot of an object is different from a *checkpoint*; whereas a snapshot represents a single object, a checkpoint is a set of mutually consistent snapshots. A checkpoint of an object is recursively defined to consist of a snapshot of that object and a checkpoint of each object which has an interface instance referred to by a reference in the initial object. If an object has no references to interface instances in other objects, a snapshot of that object is also a checkpoint.

By representing an object which currently performs no activities as a snapshot stored on secondary storage, the in-memory resources occupied by the object can be freed; the in-memory representation need not be available until the object must perform activities again.

8.1.2 Problem statement

[Olsen 92b] describes how snapshots can be produced and installed in objects by invoking automatically generated operations. Persistent snapshot repositories are needed to enable snapshots to be stored independently of application objects; a *snapshot base* is a persistent repository for snapshots. The name "snapshot base" was chosen because the properties of a snapshot base are similar to those of a traditional database: it enables multiple clients to share the snapshot repository; it handles a large number of snapshots in a large storage space, and it is stable to the extent that the snapshot repository is maintained in the face of process failure.

8.1.3 Purpose

The purpose of this paper is to describe the design and implementation of a snapshot base which is simple, efficient and recoverable from process failures.

8.1.4 Main ideas

The implementation is based on [Birrell 87] which describes an implementation technique for databases with properties suitable for a snapshot base: no need for advanced information modelling and manipulation facilities. Using this technique, a snapshot base manages a complete cache in virtual memory of a repository of snapshots, which is frequently checkpointed to secondary storage. Transactions on a snapshot base take the form of single updates which are atomic and serialized with all other transactions. To ensure a snapshot base is stable updates performed between two checkpoints are written to secondary storage. Hence, if a snapshot base suffers a process failure, it can be recovered by reading a checkpoint followed by replaying logged updates.

8.1.5 Limitations

A snapshot base should be dependable, be operating in a secure manner, and be subject to garbage collection schemes. However, these issues are not addressed by this paper except the provision of stability in the face of process failures. No complicated solutions which addresses the performance of data transfer to and from secondary storage, such as hashing, clustering and cache fragmentation are employed. The limitations subject to the design of a snapshot base are detailed in section 2.

8.1.6 Related work

The aim of building a snapshot base, rather than using an existing object store like O_2 [Deux 91], ObjectStore [Lamb 91], GemStone [Maier 86], or Mneme [Moss 90] is to avoid the potential overheads of such sophisticated systems. A snapshot base is intended to provide only the necessary and sufficient features for storing and retrieving snapshots.

8.1.7 Audience and prerequisites

This paper targets system developers and programmers who are designing and implementing persistent object systems. It also addresses application programmers who need to explicitly store (snapshots of) objects on secondary storage. Readers should be acquainted with ANSA [Warne 91], The ANSA

Testbench3.0 [AIM 91], The ANSA Storage Model [Olsen 91a], and the prototype overview [Olsen 92a].

8.1.8 Organisation

The paper is organised as follows: section 2 describes the design requirements and limitations; section 3 describes the design of a snapshot base; section 4 describes the implementation details; section 5 shows how to start-up and use a snapshot base; section 6 discusses the implementation and its current status, and section 7 concludes the paper.

8.2 Design requirements and limitations

The goal for implementing a snapshot base is to develop a simple, yet efficient, initial prototype. This allows practical experience of its use to be obtained in order to discover which basic features and implementation details are critical to its performance. Because of this, we have made no efforts to provide features which are not basic (such as authorizing client access, making a snapshot base highly available and reliable, and configuring a system of federated snapshot bases which span a distributed system). The following sections describe the requirements and limitations of the design.

8.2.1 Distribution, autonomy, and scaling

A snapshot base is intended to be used by liveness transparencies, i.e. activation and passivation functions (see [Olsen 92a]), which store and retrieve snapshots. There will be a number of snapshot bases in a distributed system, each covering a storage domain. *Storage domains* are a means of dividing a distributed system into smaller more manageable local domains similar to the notion of trading domains. This prevents a distributed system from relying on a single service for the persistent storage of snapshots, and it also allows snapshot bases to contain fewer snapshots thus enabling the operations of a snapshot base to be more efficient. Snapshots can be moved and replicated between snapshot bases which cover different storage domains.

A system of snapshot bases has several advantages; it improves the performance of operations on individual snapshot bases and provides for scaling. It also allows for localization which can reduce the remote access delays in networks because it stores snapshots accessed by a given client in the snapshot base which is geographically closest to that client. Localization can also be used for grouping related snapshots. Note, that within a system of snapshot bases, each snapshot base is a homogeneous and autonomous entity. There is no single federated snapshot base, i.e. as in ORION-2 [Kim 91], which gives access to all the snapshots in the system of snapshot bases. This could, however, be achieved by making each snapshot base a member of a functionally distributed group [Olsen 91b].

Federation, partitioning of a distributed system into storage domains, and the choice of which snapshot is stored in which snapshot base will not be covered further in this paper.

8.2.2 Transactions

Because snapshots are immutable, there is no need for transactional operations covering multiple updates as in [Oki 85]. If a client fails before

completing a set of intended updates, the consistency of the snapshot base is not affected because updates are independent of each other. Each single update is guaranteed to be transaction-consistent (i.e., all-or-none update semantics), and recoverable in the event of process failure (see section 2.3).

8.2.3 Stability

A snapshot base must be stable so it can recover from process and media failures without losing performed updates. A process failure occurs when the process handling the management of the snapshot repository fails. A media failure occurs when a secondary storage device is damaged so that data is lost. Recovery from process failure is possible if checkpoints of the snapshot repository are written and if updates performed between checkpoints are logged to secondary storage. A snapshot base can also be made tolerant to process failures by replicating it, e.g. using a group abstraction, over several processes. Recovery from media failure is possible if checkpoints and logs are replicated on several independent storage devices. Recovery from media failure is not further covered in this paper.

8.2.4 Availability

A snapshot base should not be unavailable for unacceptable long periods of time, for example due to process, media, and network failures. High availability can be achieved by replicating a snapshot base on several nodes in a distributed system, for example by implementing it as a passive or active replica group [Olsen 91b]. We chose not to provide high availability in this initial prototype. Note, that because snapshots are immutable, they can be replicated in several snapshot bases, for example by using a lazy replication strategy as in [Weibe86].

8.2.5 Authorization

A snapshot base should require client authorization to control the accessibility of snapshots. We chose not to provide authorization in this first prototype. Possession of a reference to an interface of a snapshot base gives access to all the snapshots it stores. Localization can be used as a means of separating the accessibility of snapshots.

8.2.6 Concurrent access

A snapshot base must support access from multiple clients and prevent concurrent updates from interfering.

8.2.7 Size and number of snapshots

A snapshot base should have no artificial limits on the number or size of snapshots which it can store.

8.2.8 Performance

There is no practical experience to justify it, but the assumptions of the technique for implementing small databases which is described in [Birrell 87] appears to be appropriate for a snapshot base: a relatively small amount of data (<10 mega bytes); a moderate rate of updates (a burst rate of up to 10 transactions per second, and up to 10000 transactions per day); no updates

composed of multiple client actions, and no client visibility of any intermediate state during updates.

8.2.9 Garbage collection

Snapshots must be explicitly deleted from a snapshot base when they are no longer needed by applications. The garbage collection criteria is thus defined externally to the snapshot base, and should be administered externally by entities which posses sufficient information to detect when a snapshot becomes garbage.

8.2.10 Portability

A snapshot base should be portable to all machine types for which the ANSA Testbench has been ported. To ease porting to a new machine, the implementation of a snapshot base should use only a few of the most common I/O primitives available in operating systems.

8.3 Design

A snapshot base manages a repository of stored snapshots, and clients can interact with it via two interfaces.

8.3.1 The interfaces

A snapshot base has two interfaces. The SBControl interface has operations for shutting the snapshot base down and for listing its contents by snapshot identifier and type. The SBStore interface has operations for saving, fetching, and removing a snapshot.

Computationally, a snapshot base can be regarded as having an interface for each snapshot it stores, so a particular snapshot can be retrieved by invoking a particular interface. In practice it is far too expensive to implement it this way. Imagine the cost of checkpointing all the interfaces of a snapshot base to secondary storage, recreating the interfaces on recovery, and redirecting references to these interfaces, compared to having a simple integer identifier for each snapshot.

8.3.2 The repository

A snapshot base represents each snapshot it stores as an *object denotation* which consists of an uninterpreted sequence of bytes (the snapshot), a snapshot identifier, and a type name. No operations can be performed on an object denotation (it is immutable), and there are no cross-references between object denotations. The snapshot identifier, which is also the name for accessing a snapshot, is defined relative to a snapshot base. This means that if a snapshot is moved from one snapshot base to another, its snapshot identifier in the latter snapshot base is likely to be different from its snapshot identifier in the former. Snapshots must therefore be named by a reference to a snapshot base interface (SBI) and a snapshot identifier (SID) within that snapshot base.

It is the responsibility of a *snapshot locator* [Olsen 92a] to map interfaces of application objects to a (SBI, SID) pair in order to name a snapshot which represents a passive application object. Snapshot bases cannot redirect the addressing of moved snapshots, for example by containing a (SBI, SID) pair of

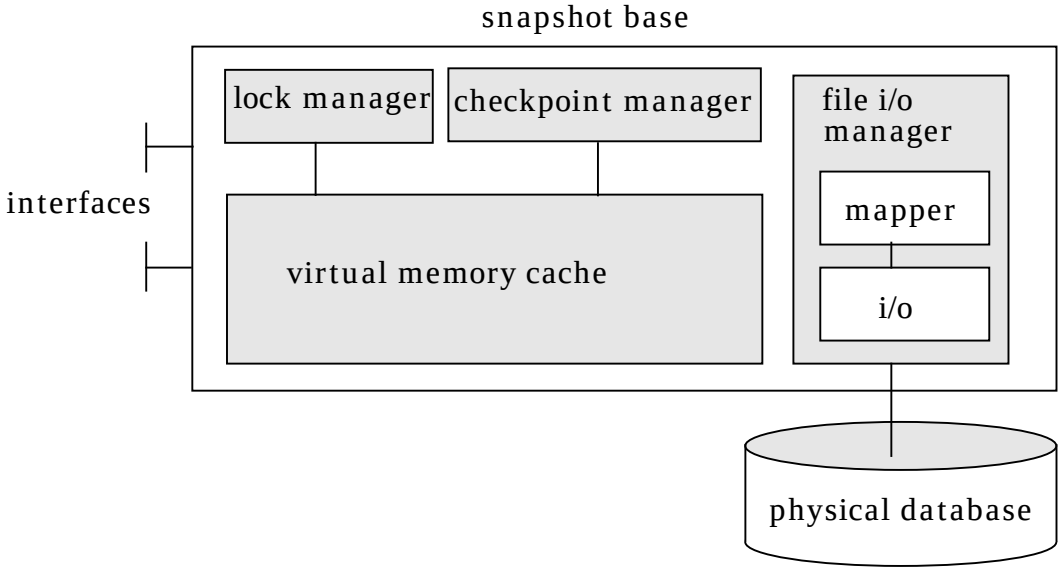

Figure 1: Structure of a snapshot base

a moved snapshot. Such redirection is to be achieved outside snapshot bases via snapshot locators.

8.3.3 Design structure

A snapshot base consists of a virtual memory cache of stored snapshots, a lock manager, a checkpoint manager, and a file input/output manager (see figure 1). The interfaces of a snapshot base can manipulate the virtual memory cache. Before invocations of their operations can update the cache, a lock must be obtained from the lock manager. The file input/output manager contains a collection of routines for data mapping and a collection of routines for transferring data between the cache and the physical database. The mapping routines correspond to marshal and unmarshal operations, and the transfer routines correspond to file input/output operations. The checkpoint manager ensures that checkpoints are frequently taken of the virtual memory cache.

The file input/output manager provides physical data independence between the cache and the physical database so the organisation of the physical database can be modified without affecting the cache. This allows subsequent tuning for efficiency of the physical database and the file i/o manager without affecting other parts of the snapshot base.

8.4 Implementation

The snapshot base is implemented as a stand-alone server which maintains a repository of snapshots as a virtual memory cache, which is frequently checkpointed to a disk file.

The approach is to make *simple* implementations of the lock manager, the cache, the file i/o manager and the physical database so they subsequently can be tuned individually for improved efficiency. The resulting implementation thus provides a skeleton structure which partitions the snapshot base into independent modules, each consisting of code which is likely to be replaced by more sophisticated code in later versions of the snapshot base prototype.

Rather than implementing the snapshot base as a separate server, it can be implemented as a local interface in each client. The operations in this interface can then be implemented by linked-in library routines which operate on the snapshot repository on secondary storage via a loaded cache as in [Moss 90]. This alternative avoids the overheads of remote communication between a client and a snapshot base, but it introduces the problem of coordinating multiple clients without a central coordinator.

8.4.1 The interfaces

The SBControl and the SBStore interfaces are defined in terms of a collection of types which define the structure of the cache as a set of object denotations each of which contains a snapshot:

```
SnapshotBaseTypes: INTERFACE =
BEGIN
  Snapshot          : TYPE = SEQUENCE OF OCTET;
  SnapshotIdentifier: TYPE = INTEGER;
  ObjectDenotation  : TYPE = RECORD
                             [ snapshot  : Snapshot,
                               objectType: ARRAY 32 OF CHAR,
                               sid       : SnapshotIdentifier
                             ];
  Cache             : TYPE = SEQUENCE OF ObjectDenotation;
END.
```

8.4.1.1 The SBStore interface

The SBStore interface is defined by

```
SBStore: INTERFACE =
NEEDS SnapshotBaseTypes;
BEGIN
  save  : OPERATION [objectDenotation: ObjectDenotation]
          RETURNS [sid: SnapshotIdentifier];
  remove: OPERATION [sid: SnapshotIdentifier]
          RETURNS [objectDenotation: ObjectDenotation];
  fetch : OPERATION [sid: SnapshotIdentifier]
          RETURNS [objectDenotation: ObjectDenotation];
END.
```

The save operation takes an ObjectDenotation, adds it to the cache, and returns a snapshot identifier which identifies the ObjectDenotation in the cache. Snapshot identifiers are reused, so if the sid field in the argument ObjectDenotation has a value different from zero, this value becomes the identifier for the ObjectDenotation in the cache if it is not already used for an object denotation in the cache, otherwise a new identifier is generated. The remove operation takes a snapshot identifier, removes an ObjectDenotation with the given snapshot identifier from the cache, and returns the ObjectDenotation. If there is no ObjectDenotation in the cache with an sid equal to the argument snapshot identifier, an ObjectDenotation with a snapshot of length 0, an object type of value "", and an sid of value 0 is returned. The fetch operation differs from the remove operation by not removing an ObjectDenotation from the cache.

8.4.1.2 The SBControl interface

The SBControl interface is defined by

```
SBControl: INTERFACE
NEEDS SnapshotBaseTypes;
```

```
BEGIN
  Entry       : TYPE = RECORD
                       [ sid       : SnapshotIdentifier,
                         objectType: ARRAY 32 OF CHAR
                       ];
  ContentsList: TYPE = SEQUENCE OF Entry;
  list        : OPERATION[]
                RETURNS[contentsList: ContentsList];
  shutdown    : ANNOUNCEMENT OPERATION []
                RETURNS [];
END.
```

The list operation returns a sequence of pairs of snapshot identifier and object type name. The shutdown operation requests the snapshot base to checkpoint its cache and snapshot identifier counter to a file, to withdraw its SBStore and SBControl interfaces from the local trader, and then to terminate itself.

8.4.2 The virtual memory cache

The cache consists of a sequence of object denotations each of which consists of a number of consecutive bytes in virtual memory. The snapshot and object type fields of an object denotation have values which were assigned outside the snapshot base, whereas the snapshot identifier field is assigned by the snapshot base. This field is used for identifying a particular object denotation in the cache. A snapshot identifier is simply an integer value generated by increasing a counter each time an object denotation is added to the cache. If snapshot identifiers of removed object denotations are not reused, a snapshot base can allocate no more than 2^{31} snapshot identifiers.

A new object denotation is added to the cache by allocating the appropriate number of bytes at the end of the cache and then initializing them with the values of the object denotation. An object denotation is removed by over-writing its values with the values of the neighbouring object denotation and repeating this to the end of the cache where the bytes allocated for the last object denotation are removed.

Because the cache is organized as a heap, a lookup requires a scan of at least half the cache on average, and of the whole cache at worst, until a snapshot is found. Removal requires data swapping of at least half the cache on average, and of the remaining cache following the removal point in the worst case. Addition requires additional memory to be allocated at the end of the cache using system_extend which in the worst case causes a complete memory re-allocation for the whole cache.

By using a more sophisticated data structure for the cache, for example a double linked list where each list-element contains an object denotation and is ordered by the value of the snapshot identifier in the object denotation, the performance of addition, removal and copying of records may be improved. Note, however, because the cache is residing in virtual memory, pointers should not span disk pages as this could cause more page swapping than when the cache is represented in contiguous memory segments.

If it turns out that keeping a full virtual memory cache of all stored snapshots and relying on virtual memory management is too inefficient or that virtual memory is not large enough, it may be necessary to fragment the cache into smaller segments.

8.4.3 The lock manager

The lock manager is implemented as a collection of routines which operate on event counters and sequencers. The routines provide a pessimistic locking scheme (rather than an optimistic commit scheme) to ensure that only one save, remove or shutdown is performed at a time. When no save, remove or shutdown operation is being performed, any number of fetch and list operations may be performed concurrently. This concurrency control scheme is stronger than strictly necessary because it serializes concurrent non-conflicting updates. If concurrent non-conflicting updates were allowed and occurred frequently enough, the average update time could be improved. Concurrent non-conflicting updates could be allowed by having a locking scheme which locks individual records in the cache by using an event counter and a sequencer for each record. The chosen implementation of the cache implies that any two updates will interfere because the memory allocated for the cache may be affected by each update.

8.4.4 The file input/output manager

The file input/output manager is implemented as a two modules. The transfer module contains a collection of routines which performs file input/output operations. The mapper module contains a collection of routines which converts between the data types in the cache and the sequence of bytes stored in files representing the physical database.

8.4.5 The checkpoint manager

If a snapshot base crashes due to a process failure, it must be possible to construct another snapshot base which has a cache and a snapshot identifier counter with values identical to those of the original snapshot base, immediately before it crashed. This can be achieved by frequently writing a checkpoint of the cache and the snapshot identifier counter to secondary storage. To prevent updates to the repository from being lost if a snapshot base crashes before making its next checkpoint, updates are logged so they can be replayed when a checkpoint has been restored.

The frequency of checkpoints is determined by a checkpoint policy. Checkpoints are time consuming to make, and the recovery time is dependent on the number of entries in the log file. A checkpointing policy should therefore make a trade-off between requirements to recovery time and the cost of replaying logged updates. In the current implementation, an invocation of a save or remove operation causes the checkpoint manager to request the file input/output manager to log the invocation; every 10th invocation of a save or remove operation causes the checkpoint manager to request the file input/output manager to write a checkpoint.

Checkpointing and logging should be controlled by a separate thread in the snapshot base, enabling timer based checkpointing policies. This can also prevent the client performing the 10th update from suffering a delay before the invocation returns, but it means that the client performing the subsequent update may be delayed if a checkpoint is in progress.

8.4.5.1 The checkpoint and log scheme

The writing of checkpoints and logging of updates follows the scheme in [Birrell 87]. A snapshot base has three associated files on disk: the ckp, the log and the version file. The ckp file contains the latest checkpoint of the snapshot

base's cache; the log file contains chronologically ordered entries for each save and remove operation performed by the snapshot base since the checkpoint in the ckp file was made, and the version file contains an integer denoting which ckp file contains the latest checkpoint and which log file contains the logged updates since last checkpoint.

In the following we assume that ckp42 contains the latest checkpoint and that log42 contains an entry for each save and remove performed since last checkpoint. The contents of version is thus "42". A new checkpoint is made by first reading "42" from the version file, then a file named ckp43 is created to which the checkpoint is written. In addition, a file named log43 is created. Then a file named newversion is created which contains "43", and finally newversion is renamed to version. The creation of the newversion file is the commit point for the new checkpoint. This is because the algorithm for recovering a snapshot base first searches for a newversion file to obtain the identifier for the ckp file in case a crash occurred before newversion was renamed to version. If a newversion file does not exist the identifier will be obtained from the version file. If a crash occurred before the newversion file was created, the snapshot base can recover from the ckp42 and log42 files as these will only be removed after the creation of newversion. Having determined the version number to be "43", a snapshot base recovers by first reading the checkpoint from the ckp43 file and then replaying each entry in the log43 file.

8.4.6 Mapping between cache and physical database

A checkpoint is easily constructed and installed in a snapshot base using the produceSnapshot and installSnapshot routines which are generated automatically by the SNAPSHOT prepc statement described in [Olsen 92b]. This avoids dealing with the impedance mismatch [Copeland 84] between the snapshot representation in the cache and on secondary storage. Note, that the SNAPSHOT statement can only be used because the data structure for the cache can be described in IDL. If a double linked list was used as the data structure for the cache instead, the cache could not be described in IDL due to IDL's lack of pointers.

A log entry for a remove invocation simply consists of "r sid" where sid is the value of the snapshot identifier of the snapshot which is to be removed. A log entry for a save operation consists of "save n denotation", where n is the number of bytes occupied by denotation, which is a marshalled ObjectDenotation obtained by invoking an automatically generated produceSnapshot operation.

To detect failed writes to the ckp and log files, the number of bytes of a checkpoint and the number of bytes in an update entry are recorded as a prefix in checkpoints and in each log entry. There are currently no means of recovering if writes to the ckp and log files have failed.

8.5 Start-up and use

A snapshot base is started up by the following command:

```
SnapshotBase <pathname>
```

pathname must specify an absolute path to the directory in which the snapshot base can create ckp and log files; if no pathname is specified /tmp/SnapshotBase becomes the default. The snapshot base registers its SBStore and SBControl interfaces with the property "NodeName HOSTMACHINE" in the local trader. HOSTMACHINE is the name of the machine on which the snapshot base is

running. A client can use the services of a snapshot base as any other service by importing its interfaces.

8.6 Discussion and status of work

In this section we discuss the internals of the snapshot base and to what degree they accommodate the design requirements.

8.6.1 Stability

The stability of the snapshot base is achieved by a simple strategy for checkpointing the cache and logging updates between checkpoints on secondary storage. However, the stability only covers process crashes. Recovery from a process crash must be done manually by starting up a new snapshot base and providing it with the name of the directory in which the *ckp* and *log* files are kept. Clients must then obtain references to the interfaces of the new snapshot base. Detection of a snapshot base crash is currently based on invocation failure. If a client experiences an invocation failure, the client must wait for the interfaces of a new snapshot base to become available in the local trader.

8.6.2 Availability

The services provided by a snapshot base are unavailable from when it crashes until the interfaces of a new snapshot base have been exported to the local trader. The service availability can be increased if a number of snapshot bases are running simultaneously on different machines, enabling a client to import an interface reference to an alternative snapshot base. There is, however, currently no support for replicating the physical database on different machines or for more than one snapshot base to have access to a single physical database. It is possible to implement the snapshot base service as a passive or active replica group with members being the interfaces of snapshot bases on different machines, but this has not yet been done.

8.6.3 Portability

The system dependent implementation, i.e. the use of the operating system facilities for file i/o, is currently only available for HP series 300 machines running HP-UX 7.0 and HP series 425s machines running HP-UX 8.0.

8.6.4 Efficiency

Because the cache is implemented as a heap data structure and because checkpoints and logs are read and written using the *fscanf* and *fprintf* primitives, there is plenty of room for improving the efficiency. The performance of the current implementation is illustrated by some raw figures on a HP series 300 machine (not in single user mode):

snapshot base start-up time	size of ckp file/number of snapshots
0.20 sec.	0 bytes/0
1.00 sec.	33222 bytes/100
1.82 sec.	66422 bytes/200
4.18 sec.	166022 bytes/500

time for 100 save invocations, with checkpoint being made for every 10th save	size of cache/number of snapshots
6 sec.	0 bytes/0
10 sec.	33222 bytes/100
15 sec.	66422 bytes/200
31 sec.	166022 bytes/500

The integration of the snapshot base with the prototypes of the other architectural entities of the ANSA storage model will enable the pattern of use for a snapshot base to be determined so its performance can be improved accordingly.

8.6.5 Simplicity

The principal implementation goal was simplicity. Much of the simplicity of the current implementation is due to the fact that a complete cache of the physical database is maintained in virtual memory. This is feasible if large virtual memories are available. If this is not the case, the cache must be fragmented introducing complexities, such as those caused by caching and clustering strategies.

8.7 Conclusion

It has been shown how to design and implement a simple snapshot base which enables clients to store and retrieve snapshots on secondary storage. The implementation maintains a virtual memory cache of the complete physical database on secondary storage. The implementation is partitioned into modules which can be fine-tuned independently for improved efficiency. A snapshot base is stable to the extent that it can be recovered from process failures. To achieve this, checkpoints of the cache and updates performed on the cache between checkpoints are written to secondary storage.

The snapshot base prototype is available from the author; the code has prototype status and comes with no warranty.

8.8 Acknowledgements

Thanks to David Iggulden of the ISA Core Team and Nigel Edwards of Hewlett-Packard Laboratories for commenting on an earlier draft of this paper.

8.9 References

[AIM 91]
"ANSAware 3.0 Implementation Manual", Report No.: RM.097, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Birrell 87]
Andrew D. Birrell, Michael B. Jones, Edward P. Wobber, " A Simple and Efficient Implementation for Small Databases*", Operating Systems Review*, Vol. 21, No. 5, (1987).

[Copeland 84]
George Copeland, David Maier, "Making Smalltalk a Database System", *SIGMOD'84*, Sigmod Record, Vol. 14, No. 2, (1984).

[Deux 91]
O. Deux et al, "The O_2 System", *Communications of the ACM*, Vol. 34, No. 10, (1991).

[Kim 91]
Won Kim, Nat Ballou, Jorge F. Garza, Darrell Woelk, "A Distributed Object-Oriented Database System Supporting Shared Private Databases", *ACM Transactions on Information Systems*, Vol. 9. No. 1, (1991).

[Lamb 91]
Charles Lamb, Gordon Landis, Jack Orenstein, Dan Weinreb, "The Object-Store Database System", *Communications of the ACM*, Vol. 34, No. 10, (1991).

[Maier 86]
David Maier, Jacob Stein, Allen Otis, Alan Purdy, "Development of an Object-Oriented DBMS*", OOPSLA'86 Proceedings*, ACM, (1986).

[Moss 90]
J. Eliot B. Moss, "Design of the Mneme Persistent Object Store",*ACM Transactions on Information Systems*, Vol. 8, No. 2, (1990).

[Oki 85]
Brian M. Oki, Barbara H. Liskov, Robert W. Scheifler, "Reliable Object Storage to Support Atomic Actions*", Proceedings of the Tenth ACM Symposium on Operating System Principles*, (1985).

[Olsen 91a]
Michael Hoffmann Olsen, "A Model for Storage and Resource Management", *Report No.: WP-DCG-91-14,* Hewlett-Packard Laboratories, Filton Road, Stoke Gifford,Bristol BS12 6QZ, UK, (1991).

[Olsen 91b]
Michael H. Olsen, Ed Oskiewicz, John P. Warne, "A Model for Interface Groups", *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, IEEE, (1991).

[Olsen 92a]
Michael Hoffmann Olsen, "A Persistent Object Infrastructure for Heterogeneous Distributed Systems", Report No.: RC.343, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92b]
Michael Hoffmann Olsen, "Automated Generation of Snapshot Operations for an Object*", ISA Report No.: RC.288,* ISA project, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, (1992).

[Warne 91]
John Warne, "ANSA: Assumptions, Principles, and Structure", Conference proceedings of Software Engineering Environments 1991, University College of Wales, Aberystwyth, (1991).

[Weibe 86]
Douglas Wiebe, "A Distributed Repository for Immutable Persistent Objects", *OOPSLA'86 Proceedings*, ACM, (1986).

9 Design and Implementation of a StorageLocator

9.1 Introduction

This paper is the fourth in a series of eight papers describing a prototype of the ANSA Storage Model [Olsen 91a] developed in the ANSA Testbench 3.0 [AIM 91]. An overview of the prototype is given in [Olsen 92a].

9.1.1 Terminology and background

A *persistent object system* is an infrastructure which enables application systems to treat objects which are stored on secondary storage as in-memory objects. It automatically moves objects, on demand, between secondary storage and main memory and thus eliminates the *impedance mismatch* between the type system of application systems and the type system of secondary storage abstractions like databases and file systems [Copeland 84]. Invocations performed on interface instances of stored objects can be extremely slow because stored objects must be moved in-memory before they can provide a requested service; the challenge is therefore to make persistent object systems run fast.

To move an in-memory object to secondary storage, a persistent object system must change the object's representation so it complies with the type system of a secondary storage abstraction. To facilitate the migration of objects, the chosen secondary storage representation should comply with the type systems of networks. A non-interpreted sequence of bytes is an example of such a representation. A sequence of bytes which represents a single object is termed a *snapshot* of that particular object. The snapshots considered in this paper comprises three parts: (i) an object type component denoting an implementation template; (ii) a state component denoting the current state of an object; and (iii) a schema component denoting the set of interface instances currently supported by an object. A snapshot of an object is different from a *checkpoint*; whereas a snapshot represents a single object, a checkpoint is a set of mutually consistent snapshots. A checkpoint of an object is recursively defined to consist of a snapshot of that object and a checkpoint of each object which has an interface instance referred to by a reference in the initial object. If an object has no references to interface instances in other objects, a snapshot of that object is also a checkpoint.

By representing an object which currently performs no activities as a snapshot stored on secondary storage, the in-memory resources occupied by the object can be freed. A service which enables snapshots to be stored on and retrieved from secondary storage is termed a *snapshot base*. The process of creating a snapshot of an object, storing it in a snapshot base and then terminating the object is termed *passivation*. The in-memory representation of a passive object need not be available until the object is required to perform activities again. The process of creating an in-memory object and installing a snapshot in it is

termed *activation*. *Migration* is the process of moving an active object from one location in a distributed system to another. By migrating objects, a distributed system can be dynamically reconfigured.

9.1.2 Problem statement

Clients which posses references to interfaces of a passive object will not be able to invoke them due to the fact that the interfaces are no longer represented by in-memory sockets. If a client knew how to access the snapshot representing the passive object, it could activate the object, and after rebinding interface references to the sockets of the active object's interfaces, the client would be able to use references to the object's interfaces successfully (until the object is passivated again). Clients which posses references to interfaces of an object which have been passivated and subsequently activated will not be able to invoke them, unless they have rebound their references to the in-memory sockets of the active object's interfaces. To do this, they need to know which sockets to rebind to. References to interfaces may not have been rebound between several sequences of pair-wise passivation and activation of an object, because they have not been used for invocation; it must be possible to rebind such references to the sockets of the interfaces of the current active object. The problem of rebinding a reference to an interface of an object after it has been passivated and subsequently activated is equivalent to rebinding a reference to an interface of an object which has changed its in-memory location, i.e. a migrated object.

9.1.3 Purpose

The purpose of this paper is to describe the design and implementation of a service which can return replacement references to interfaces which have become unusable for invocation because an object has changed (i) its in-memory location to a secondary storage location, (ii) its secondary storage location to an in-memory location, (iii) its in-memory location to another in-memory location, or (iv) its secondary storage location to another secondary storage location. An object which provides this service is termed a *snapshot locator*. If a client uses a reference to an interface for invocation and experiences a failure, the client can request a snapshot locator for a replacement reference which is either

- a reference to an interface of a snapshot base and a snapshot identifier which enables a snapshot to be retrieved, as part of activating a passive object; this is required if the object currently is passive;
- a replacement reference which can be used for invoking the given interface; this is required because the object supporting the interface, has undergone passivation and activation (one or more times) or it has migrated since last time the reference was successfully used for invocation;
- a null reference which indicates that there is no replacement reference; this implies that the invocation failure is not due to passivation, activation or migration, but something else caused it (e.g. a failed object).

The process of requesting a snapshot locator for a replacement reference is termed *locating an interface*.

9.1.4 Main ideas

The snapshot locator manages a list of entries each of which associates an out-of-date interface references with an up-to-date replacement references. A replacement reference can either be a reference to an interface of an active object, or a reference to the SBStore interface of a snapshot base and a snapshot identifier which identifies a passive object (see [Olsen 92b]). The snapshot locator thus enables interfaces of both active and passive objects to be located. To ensure the snapshot locator's look up operation returns an up-to-date replacement interface reference, the snapshot locator's register and deregister operations perform transitive closures on associations between out-of-date references and up-to-date references.

9.1.5 Limitations

This paper does not address how a client discovers that a reference to an interface is no longer usable for invocation, how a passive object is activated, how an in-memory object is passivated, or how a client rebinds reference names to replacement references; this is covered by other papers (see [Olsen 92c]). For simplicity, the paper only covers interfaces supported at the level of capsules, and if not otherwise made explicit, the term object is synonymous with a capsule; we do not expect major problems in extending our work to cover interfaces supported at the level of objects within capsules (or further levels of nesting).

A general location scheme requires a locator which handles the interfaces of active objects, and a snapshot locator which handles the interfaces of passive objects. In this paper the snapshot locator both handles the interfaces of active and passive objects. The signature of the snapshot locator described in this paper is different from the signature of the location service described in [Herbert 91]. This is because a snapshot locator handles the addresses of snapshots, which are composed of an interface reference to a snapshot base's SBStore interface and a snapshot identifier. The details of this and what implications it has are described in more detail in section 8.

A snapshot locator should be dependable, be operating in a secure manner, and be subject to garbage collection schemes. However, these issues are not addressed by this paper. The limitations subject to the design of a snapshot locator are detailed in section 2.

9.1.6 Related work within ANSA

Within the ANSA framework, the issue of locating interfaces has previously been implemented in the context of trading, see [Nicolaou 91a]. Our paper differs by considering location issues particular to the context of interfaces of passive objects. For this reason our location service is termed a snapshot locator. It is not a general solution to all possible location issues in a distributed system. Our work also deviates from the trader locator in being designed for and implemented in ANSA Testbench3.0 rather than Testbench4.0. We use Testbench3.0 because Testbench4.0 was still being developed while we were developing the snapshot locator. Because ANSA Testbench3.0 does not support the new architecture for interface references [Nyong 91] [Nicolaou 91b], it will have to be simulated; simulation, however, is merely a matter of explicitly keeping track of the details which are abstracted away by higher level abstractions, so it will not prevent the demonstration of the principles of the ANSA Storage Model, only make the demonstration more

complex. For the same reason, we do not expect major difficulties in porting this work to Testbench4.0.

9.1.7 Audience and prerequisites

This paper targets system developers and programmers who are designing and implementing persistent object systems. It also addresses application programmers who choose to store, retrieve and migrate objects. Readers should be acquainted with ANSA [Warne 91], The ANSA Testbench3.0 [AIM 91], The ANSA Storage Model [Olsen 91a], and the prototype overview [Olsen 92a].

9.1.8 Organisation

The paper is organised as follows: section 2 describes the design requirements and limitations; section 3 describes the design of a snapshot locator; section 4 describes the implementation details; section 5 shows how to start up and use a snapshot locator; section 6 discusses the implementation and its current status; section 7 relates to other work; section 8 discusses how the prototyped snapshot locator deviates from an earlier design, and section 9 concludes the paper.

9.2 Design requirements and limitations

The goal for implementing the snapshot locator is to develop a simple initial prototype. This allows practical experience to be obtained of the proposed addressing scheme for handling passivation, activation and migration [Olsen 91a]. Since the addressing scheme is the focus, we have made no efforts to make a snapshot locator reliable, highly available, federated with other snapshot locators to span a distributed system, and to authorize client access. These issues must be addressed by any snapshot locator which is going to form part of a persistent object infrastructure for applications in practice. The requirements and limitations of the design cover issues similar to those of the snapshot base, and they are described in the following sections which to some extent overlap with the corresponding sections in [Olsen 92b].

9.2.1 Functionality and intended use

A snapshot locator is intended to be used by liveness and migration transparencies, i.e. activation, passivation and migration functions. A transparency layer in server objects is required to register and deregister references to interfaces with a snapshot locator, when they are relocated. This enables a transparency layer in clients to look up replacement references to interfaces of objects which have undergone passivation, activation, and migration. A server object which changes its in-memory location needs to manufacture new references to its interfaces because new sockets are associated with them. The server can do this by requesting its nucleus for a new reference to each of its interfaces. The server must then be able to request a snapshot locator to register the new references as being replacements for old references. A client will experience a run-time error if it attempts to use one of the old references to invoke an interface and it must therefore be able to request a snapshot locator to look up a replacement reference for the old reference. After receiving the replacement reference, the client can retry invoking the interface. To avoid that a look up returns a reference for which a

more up-to-date replacement reference exists, the registration of replacement references must update previous registered replacements which are out dated by a new registration. When a server destroys an interface or terminates itself, the replacement references to its interfaces become redundant and should be removed from snapshot locators. Servers must therefore be able to request a snapshot locator to de-register references.

9.2.2 Distribution, autonomy, and scaling

There will be a number of snapshot locators in a distributed system, each covering a location domain. *Location domains* are a means of dividing a distributed system into smaller more manageable local domains similar to the notion of trading domains (typically it will be sensible to make a location domain correspond to a storage domain [Olsen 92b]). This prevents a distributed system from relying on a single centralized service for the location of all interfaces, and it means that snapshot locators can contain fewer replacement references thus enabling the operations of a snapshot locator to be more efficient. A system of snapshot locators has several advantages; it improves the performance of operations on individual snapshot locators and provides for scaling. It also allows for localization which can reduce the remote access delays in networks because it enables replacement references to be accessed from that snapshot locator which is geographically closest to a client. Localization can also be used for grouping related replacement references, i.e. all references to interfaces of a given object. Note, that within a system of snapshot locators, each snapshot locator is a homogeneous and autonomous entity. There is no single federated snapshot locator which gives access to all the replacement references in the system of snapshot locators. This could, however, be achieved by making each snapshot locator a member of a functionally distributed group [Olsen 91b].

Federation, partitioning of a distributed system into location domains, and the choice of which replacement reference is registered in which snapshot locator will not be covered further in this paper.

9.2.3 Transactions

Because passivation, activation, and migration of an object make references to all its interfaces unusable, it is required that a snapshot locator can update sets of replacement references atomically, i.e. without other invocations interfering.

9.2.4 Stability

A snapshot locator must be stable so it can recover from process and media failures without losing performed updates. A process failure occurs when the process which executes a snapshot locator fails. A media failure occurs when a secondary storage device is damaged so that data is lost. Recovery from process failure is possible (i) if checkpoints of replacement references are written and if updates performed between checkpoints are logged to secondary storage, or (ii) if a snapshot locator is implemented as a (passive or active) replica group [Olsen 91b] which only fails if all replicas fail. While enabling recovery from process failure, the writing of checkpoints and logs to secondary storage are subject to the possibility of media failures. Recovery from media failure is possible if checkpoints and logs are replicated on several independent storage devices. Recovery from media failure is not further covered in this paper.

9.2.5 Availability

A snapshot locator should not be unavailable for unacceptable long periods of time, for example due to process, media, and network failures. High availability can be achieved by replicating a snapshot locator on several nodes in a distributed system, for example by implementing it as a passive or active replica group [Olsen 91b]. We chose not to provide high availability in the initial prototype.

9.2.6 Authorization

A snapshot locator should have a means of requiring client authorization to control the accessibility of replacement references. We chose not to provide authorization in this first prototype under the assumption that if a client possesses a reference to an interface in the first place, then the client should not be prohibited from obtaining a replacement reference. This assumption may not be valid for all applications. Possession of a reference to an interface of a snapshot locator enables a client to obtain any replacement reference registered in the snapshot locator. Localization can be used as a means of separating the accessibility of replacement references.

9.2.7 Concurrent access

A snapshot locator must support access from multiple clients and prevent concurrent updates from interfering.

9.2.8 Number of replacement references

A snapshot locator should have no artificial limits on the number of replacement references which it keeps. Our underlying assumption is that a snapshot locator covers a location domain which in principle could constantly consist of a single interface, but in practice it will probably be more sensible for a location domain to vary in size reflecting the creation and termination of interfaces.

9.2.9 Garbage collection

A replacement reference to a given interface must be kept by a snapshot locator as long as there exists a client with a reference to that interface. If a replacement interface is removed from a snapshot locator while at least one client possesses a reference to that interface, that client will never be able to invoke the operations of the interface. If an object destroys an interface or terminates itself it should remove references to its (destroyed) interfaces from snapshot locators, because if any one of these reference are used for invocation by a client, the invocation will fail. It should therefore be possible for an object to remove references to its interfaces from snapshot locators as part of destroying an interface or as part of terminating itself. The real problem is to detect failed objects and remove references to their interfaces from snapshot locators. This may be done by components (perhaps part of each snapshot locator) within the persistent object system which at given intervals polls the interfaces which snapshot locators keep references to. When a replacement reference to an interface of a passive object can be removed, the snapshot which represents it can be removed from the snapshot base within which it resides. Garbage collection is not further covered in this paper.

9.3 Design

A snapshot locator manages a list of entries each of which associates out-of-date interface references with replacement references. Each entry consists of (i) an out-of-date reference; (ii) an up-to-date reference; (iii) the previous up-to-date reference to the interface in an active object; (iv) a type name; and (v) a node name. The up-to-date reference is composed of a reference and an identifier. The identifier is used to distinguish between two interpretations of the reference: a replacement reference to an interface in an active object, or a reference to a snapshot base. In the latter case, the identifier locates the snapshot stored within the snapshot base. The purpose of recording the previous up-to-date reference is to identify which of the interfaces in a passive object's snapshot corresponds to a given out-of-date reference. The type name denotes what type of object a given out-of-date interface instance was instantiated in. The node name denotes where that object is active, or previously was active if currently passive.

Clients can interact with a snapshot locator via two interfaces. The SLControl interface has operations for shutting a snapshot locator down and for obtaining a listing of the its registered entries.

The SLLocate interface has operations for registering, deregistering, and looking up replacement references.

Like a snapshot base, a snapshot locator consists of a virtual memory cache, a lock manager, a checkpoint manager, and a file input/output manager (see figure 1). The interfaces of a snapshot locator can manipulate the virtual memory cache which contains mappings from out-of-date to replacement references. Before operation invocations can update the cache, each operation must obtain a lock from the lock manager. The file input/output manager contains a collection of routines for data mapping and a collection of routines for transferring data between the cache and the physical database. The mapping routines correspond to marshal and unmarshal operations, and the transfer routines correspond to file input/output operations. The checkpoint manager ensures that checkpoints are frequently taken of the virtual memory cache. The file input/output manager provides physical data independence between the cache and the physical database so the organisation of the physical database can be modified without affecting the cache. This allows

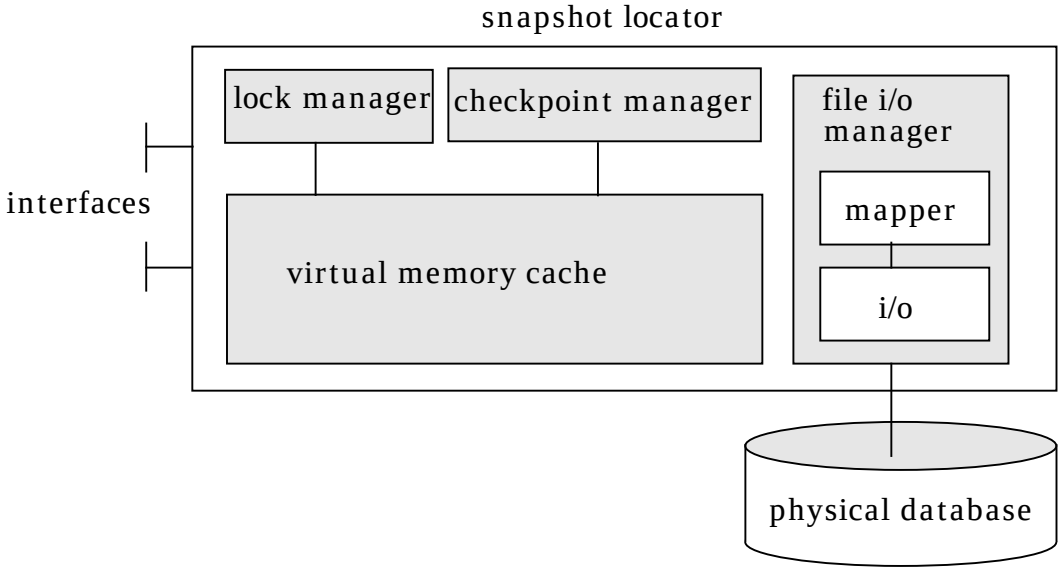

Figure 1: Structure of a snapshot locator

subsequent tuning for efficiency of the physical database and the file i/o manager without affecting other parts of the snapshot locator.

9.4 Implementation

The snapshot locator is implemented as a stand-alone server. The approach taken is to develop a *simple* implementation of a snapshot locator, which subsequently can be tuned for improved efficiency. The lock manager, the checkpoint manager, and the file input/output manager from the snapshot base implementation are reused to implement the snapshot locator. Their implementations are described in [Olsen 92b].

9.4.1 The interfaces

The SLLocate and SLControl interfaces are defined in terms of a collection of types which define the structure of the snapshot locator's list of entries:

```
LocatorTypes: INTERFACE =
NEEDS SnapshotIdentifier;
BEGIN
  Entry             : TYPE = RECORD
                             [old: InterfaceRef,
                              new: InterfaceRef,
                              sid: SnapshotIdentifier,
                              last: InterfaceRef,
                              objectType: ARRAY 32 OF Char,
                              nodeName: ARRAY 32 OF Char
                             ];
  EntryList         : TYPE = SEQUENCE OF Entry;
  IfRefList         : TYPE = SEQUENCE OF InterfaceRef;
END.
```

9.4.2 The SLLocate interface

The SLLocate interface has operations for looking up an entry, and adding and removing a set of entries in the entry list. The register operation takes a list of entries as argument in order to enable replacement references to all the

interfaces of an object to be registered atomically. Similarly, the deregister operation takes a list of references as argument in order to enable the removal of all references to interfaces of an object to be deregistered atomically. The SLLocate interface is defined by

```
SLLocate: INTERFACE =
NEEDS LocatorTypes;
BEGIN
  register  : OPERATION [entries: EntryList]
              RETURNS [];
  deregister: OPERATION [ifrefs: IfRefList]
              RETURNS [];
  lookup    : OPERATION [old: InterfaceRef]
              RETURNS [entry: Entry];
END.
```

9.4.2.1 The register operation

The register operation takes a list of entries and for each entry E, the entry list in the snapshot locator is updated in the following steps

- if E.old = E.new then the entry is redundant and it is ignored;
- if there is an entry F in the snapshot locator's entry list with F.old = E.old then F is removed from the entry list;
- add E as a new entry to the snapshot locator's entry list;
- each entry G in the snapshot locator's entry list which has G.new = E.old is updated so G = <G.old, E.new, E.sid, E.last, G.objectType, E.nodeName>;
- each entry H in the snapshot locator's entry list which has H.new = F.new, H.last = F.last, and H.sid = F.sid is updated so H = <H.old, E.new, E.sid, H.new, H.objectType, E.nodeName>;
- Each entry J in the snapshot locator's entry list which has J.old = J.new is removed from the entry list.

9.4.2.2 The deregister operation

The deregister operation takes a list of references to interfaces and for each reference R it updates the snapshot locator in the following steps

- each entry E in the snapshot locator's entry list with E.old = R is removed from the entry list;
- each entry F in the snapshot locator's entry list with F.new = R is removed from the entry list.

9.4.2.3 The lookup operation

The lookup operation takes a reference R to an interface and returns an entry E which has E.old = R if one exists in the snapshot locator's entry list, otherwise a void entry F is returned which has F.old = NULL, F.new = NULL, F.sid = -1, F.last = NULL, F.objectType = "", and F.nodeName = "".

9.4.3 The SLControl interface

The SLControl interface has an operation for returning a copy of its entry list and an operation for shutting a snapshot locator down. The shutdown operation requests the snapshot locator to checkpoint its cache to a file, to withdraw its SLLocate and SLControl interfaces from the local trader, and then to terminate itself. The SLControl interface is defined by

```
SLControl: INTERFACE
NEEDS LocatorTypes;
BEGIN
  list    : OPERATION []
            RETURNS [entryList: EntryList];
  shutdown: ANNOUNCEMENT OPERATION []
                         RETURNS [];
END.
```

9.5 The virtual memory cache

The entry list is cached in virtual memory. It contains a sequence of entries, implemented as a number of consecutive bytes in virtual memory. The size of the entry list is thus restricted by the virtual memory available. Each entry in the entry list is a record which represents a mapping between a reference to an interface and its replacement reference. A new entry is added to the entry list by allocating the appropriate number of bytes at the end of the list and then initializing them with the values of the entry which is to be added. An entry is removed by over-writing its values with the values of the neighbouring entry and repeating this to the end of the entry list where the bytes allocated for the end entry are removed.

Organizing the entry list as an unorganized heap means that lookup requires a scan of at least half the list on average, and of the whole list at worst, until an entry which contains a source reference which matches a request reference is found. In addition, removal requires data swapping of at least half the entry list on average, and of the remaining list following the removal point in the worst case. Addition requires additional memory to be allocated at the end of the list using system_extend causing a complete memory re-allocation for the whole list in the worst case. Some raw performance figures are reported in section 6.3. By using a more sophisticated data structure for the entry list, for example a double linked list of records which are ordered by hashing source references to indexes in the list, the performance of addition, removal and lookup of entries may be improved.

9.6 Start-up and use

A snapshot locator is started up by the following command:

```
SnapshotLocator <pathname>
```

pathname must specify an absolute path to the directory in which the snapshot locator can create ckp and log files; if no pathname is specified /tmp/SnapshotLocator becomes the default. The snapshot locator registers its SLLocate and SLControl interfaces with the property "NodeName HOSTMACHINE" in the local trader. HOSTMACHINE is the name of the machine on which the snapshot locator is running. A client can use the services of a snapshot locator as any other service by importing its interfaces. Below, some example invocations performed on a snapshot locator by a client demonstrates the effect of transitive closures on associated old and new interface references. The output from the client is shown in italics. Note, that only the interface identifier of an interface reference is shown in the example.

```
> SLclient list
ENTRIES IN SNAPSHOTLOCATOR:
  old: c005fe112af8c005fe203f0029fd4614000000fd
```

```
  new: c005fe112af8c005fe203f2329fd4881000000fd
  sid: 0
  last: c005fe112af8c005fe203f0729fd46de000000fd
  objectType: PhoneBook
  nodeName: audrey

  old: c005fe112af8c005fe203f0729fd46de000000fd
  new: c005fe112af8c005fe203f2329fd4881000000fd
  sid: 0
  last: c005fe112af8c005fe203f0729fd46de000000fd
  objecType: PhoneBook
  nodeName: audrey

SLclient register c005fe112af8c005fe203f0729fd46de000000fd
                  c005fe112af8c005fe203a8a29f83e7c000000fc
                  11

> SLclient list
ENTRIES IN SNAPSHOTLOCATOR:
  old: c005fe112af8c005fe203f0029fd4614000000fd
  new: c005fe112af8c005fe203a8a29f83e7c000000fc
  sid: 11
  last: c005fe112af8c005fe203f2329fd4881000000fd
  objecType: PhoneBook
  nodeName: audrey

  old: c005fe112af8c005fe203f0729fd46de000000fd
  new: c005fe112af8c005fe203a8a29f83e7c000000fc
  sid: 11
  last: c005fe112af8c005fe203f2329fd4881000000fd
  objecType: PhoneBook
  nodeName: audrey

  old: c005fe112af8c005fe203f2329fd4881000000fd
  new: c005fe112af8c005fe203a8a29f83e7c000000fc
  sid: 11
  last: c005fe112af8c005fe203f2329fd4881000000fd
  objectType: PhoneBook
  nodeName: audrey
```

9.7 Discussion and status of work

In this section we discuss the internals of the snapshot locator and to what degree they accommodate the design requirements.

9.7.1 Stability

The stability of the snapshot locator is achieved by a simple strategy for checkpointing the cache and logging updates between checkpoints on secondary storage. However, the stability only covers process crashes. Recovery from a process crash must be done manually by starting up a new snapshot locator and providing it with the name of the directory in which the *ckp* and *log* files are kept. Clients must then obtain references to the interfaces of the new snapshot locator. Detection of a snapshot locator crash is currently based on invocation failure. If a client experiences an invocation failure, the

client must wait for the interfaces of a new snapshot locator to become available in the local trader.

9.7.2 Availability

The services provided by a snapshot locator are unavailable from when it crashes until the interfaces of a new snapshot locator have been exported to the local trader. The service availability can be increased if a number of snapshot locators are running simultaneously on different machines, enabling a client to import an interface reference to an alternative snapshot locator. There is, however, currently no support for replicating the physical database on different machines or for more than one snapshot locator to have access to a single physical database. It is possible to implement the snapshot locator service as a passive or active replica group with members being the interfaces of snapshot locators on different machines, but this has not yet been done.

9.7.3 Performance

The performance of the current implementation is illustrated below by some raw performance figures for a client invoking the register, deregister and lookup operations as a function of the size of the entry list. The figures for the deregister and lookup operations are for the worst cases, i.e. the whole entry list needed scanning. The performance was measured on a HP 425s running HP-UX version 8.01, and both the snapshot locator and the client were running on the same machine.

operation	entries in entry list	time in useconds
register	100	22844
	500	34112
	1000	48108
deregister	100	21337
	500	23264
	1000	29084
lookup	100	18003
	500	18188
	1000	18792

The performed register and delete invocations did not require any transitive closures to be made on the entry list. An unsystematic test of the effect of a transitive closure on an entry list with 2000 entries showed the time for invoking a register operation was increased from 48108 useconds to between 194661 and 334127 useconds, corresponding to a transitive closure making between 1 and 10 changes to the entry list.

9.7.4 Optimization

The current implementation can be optimised in several ways by using a data structure for the entry list, which improves the performance of the register, deregister and lookup operations. For example, in the current implementation there will be an entry *<X,Y,Z>* for each of a passive object's interfaces, where the values of *Y* and *Z* are repeated in all those entries. If an entry instead had the structure *<X, pointer to replacement>* where a replacement has the structure *<Y,Z>*, then for a passive object a snapshot locator would have an

entry *<X,P>* for each of its interfaces, where *X* is a reference to an interface and *P* has the same value for each interface namely a pointer to the replacement *<Y,Z>*. Apart from reducing the memory requirements for the entries associated with the interfaces of a passive object, it would also improve the performance of the register, lookup, and deregister operations.

9.8 Related work

The traditional approaches to updating references to interfaces of objects which have changed their location apply typically to centralised systems, where a centralized infrastructure, a so called *object manager*, keeps track of the addresses of objects. Many such systems, see [Cockshott 84] for an example, incur the overhead of checking if an object is located in-memory prior to each invocation of its operations. Typically, the check is based on one of two techniques [Hosking 91]: (i) references to interfaces of objects which reside on secondary storage are *tagged*, or (ii) objects which resides on secondary storage are *marked*, i.e. represented by in-memory proxies. Reference tagging can be achieved by setting a bit in a reference which is checked prior to each invocation. Object marking can be achieved by creating an in-memory residing *fault stub* which serve as a proxy (or "stand in") for an object which resides on secondary storage [Ford 88].

The advantage of object marking is that many references may refer to the interfaces of the same fault stub which brings an object in-memory when required. The disadvantage is the additional in-memory space required for each fault stub and the introduction of additional entities which can be subject to failure. Reference tagging eliminates the space consumed by fault stubs and the level of indirection associated with them, but the reference to the interfaces of an object which has been made in-memory resident will still be tagged as residing on secondary storage until they are used for invocation, even after an object has been moved in-memory.

In most existing persistent object systems (mostly non-distributed) in-memory objects refer to each other by absolute addresses, therefore two kinds of addresses are in use: addresses for in-memory objects (internal addresses) and addresses for objects on secondary storage (external addresses). An object which is moved to secondary storage will therefore have all its references converted from internal to external addresses; and an object which is moved from secondary storage to in-memory will have its references converted to internal ones. This conversion of references is known as *pointer swizzling* and various strategies for swizzling (i.e., what to swizzle, when to swizzle, and how to unswizzle) are discussed in [Moss 92]. Pointer swizzling not only affects the objects which are moved, the objects which contains references to interfaces of moved objects must have those references converted to external ones, too. This requires a "backward" link between an interface and all the references to it. In some systems, *remembered sets* maintains backward links from interfaces to their references.

ANSA uses references to interfaces as a location independent addressing mechanism. Pointer swizzling is therefore not an issue unless local optimizations have been made within an object. The approach in ANSA is to rely on invocation failure before locating an object. snapshot locators provide the location functionality which object managers provide in centralized systems. Rather than marking objects or tagging certain references, the ANSA addressing architecture provide each reference with (at least one) associated

reference to a snapshot locator which enables a client, upon invocation failure, to request the snapshot locator to look up a replacement reference.

9.9 Architectural deviations

Architecturally, a locator advises clients about up-to-date interface references [Herbert 91]. To be able to do this, each object which changes its location must register new references to its relocated interfaces with its locator. A locator service therefore has two operations; (i) the register operation takes an old interface reference and a new interface reference, returning a termination which indicates whether registering these references with the locator succeeded or failed; (ii) the lookup operation takes an interface reference and returns either a replacement interface reference, or a termination indicating no replacement reference has been registered.

The lookup and register operations of the snapshot locator which is described in this paper do not conform to the signatures of the locator's lookup and register operations. This is because the snapshot locator has the responsibility of advising clients about the up-to-date location of snapshots in terms of an interface reference to a snapshot base and a snapshot identifier which is local to that snapshot base.

To avoid typing problems, a location service and a snapshot locator must be clearly distinguished as separate services; both services are required in a system which provides location transparency and liveness transparency. In this paper, the snapshot locator has been overloaded to provide an addressing scheme for interfaces of active objects. To conform to the architecture, a snapshot locator should only provide an addressing scheme for passive objects, and it should be consulted separately by the protocol which provides for liveness transparency. The location transparency protocol contains the following algorithm:

- a transparency layer in the client unpicks a reference to a locator from an interface reference which has failed;
- the lookup operation is invoked on the locator interface, which returns a termination to the transparency layer in the client, so if the termination contains a replacement reference, the failed invocation can be retried.

To provide liveness transparency, the lookup operation should have a table where it can be checked if the locator reference maps to a snapshot locator reference

- if the locator reference does not map to a snapshot locator no further action is taken;
- if it does then the snapshot locator is requested for an interface reference to a snapshot base and a snapshot identifier, and then an activator is called to activate the passive object. The resulting active object registers new references to its interfaces with the locator, and it removes the mapping from its locator to the snapshot locator.

When an interface is provided by an active object, the locator can search for a replacement reference for the argument interface reference. Note the last two bullets can either be part of the algorithm for the locators lookup operation, or be part of a client's transparency layer. The advantage of the latter is that location transparency is independent from liveness transparency, so if only location transparency is required, the locator carries no overhead imposed to

support liveness transparency. A separate paper which describes a prototype for passivation and activation addresses these protocols in more detail, see [Olsen 92c].

9.10 Conclusion

This paper has shown how to implement a simple snapshot locator which enables a liveness transparency layer in server objects to register and deregister references to interfaces, and a liveness transparency layer in clients to lookup replacement references to interfaces of objects which have undergone passivation and activation. To ensure the snapshot locator's look up operation returns an up-to-date replacement interface reference, the snapshot locator's register and deregister operations perform transitive closures on associations between out-of-date references and up-to-date references.

The snapshot locator described in this paper has been overloaded to provide an addressing scheme for interfaces of both active and passive objects. It was shown how a snapshot locator which only deals with the interfaces of passive objects, and a locator which deals with the interfaces of active objects are part of protocols for location and liveness transparency.

The paper has not covered how clients discover that a reference to an interface is unusable, and how a passive object is activated, nor has passivation been described. These are issues which are addressed by other papers as described in [Olsen 92c], as is the issue of making objects deregister references when they destroy an interface and when they terminate themselves.

The snapshot locator prototype is available from the author; the code has prototype status, and comes with no warranty.

9.11 Acknowledgements

Thanks to Andrew Herbert of the ISA Core Team for suggesting separation of locating interfaces of passive objects from locating interfaces of active objects.

9.12 References

[AIM 91]
"ANSAware 3.0 Implementation Manual", Report No.: RM.097, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Cockshott 84]
W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, R. Morrison, "Persistent Object Management System", Software - Practice and Experience, Vol. 14, (1984).

[Copeland 84]
George Copeland, David Maier, "Making Smalltalk a Database System", SIGMOD'84, Sigmod Record, Vol. 14, No. 2, (1984).

[Ford 88]
S. Ford, J. Joseph, D. E. Langworthy, D. F. Lively, G. Pathak, E. R. Perez, R. W. Peterson, D. M. Sparacin, S. Thatte, D. L. Wells, S. Agarwala, "Zeitgeist: Database Support for Object Oriented Programming", Proceedings of 2nd International Workshop on Object-Oriented Database Systems, Bad Munster, West Germany, (1988).

[Herbert 91]

Andrew Herbert,"Engineering Model: Conceptual Framework", Report No.: RC.282, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Hosking 90]

Antony L. Hosking and J. Eliot Moss, "Towards Compile-Time Optimisations for Persistence", The Fourth International Workshop on Persistent Object Systems, Ed.: Dearle, Shaw, and Zdonik, Morgan Kaufmann (1990).

[Moss 92]

J. Eliot B. Moss,"Working with persistent objects: To swizzle or not to swizzle", to appear in: IEEE Transactions on Computers, 1992 or 1993.

[Nicolaou 91a]

Cosmos Nicolaou,"ANSAware 4.0 Location and Trader Stale Offer Purging Mechanisms", Report No.: RC.283, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Nicolaou 91b]

Cosmos Nicolaou,"ANSAware 4.0 Interface References", Report No.: RC.268, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Nyong 91]

Dennis Nyong,"The ANSA Interface Reference: An Engineering Model", Report No.: TR.016, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91a]

Michael Hoffmann Olsen,"A Model for Storage and Resource Management", Report No.: WP-DCG-91-14, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6Qz, England, (1991).

[Olsen 91b]

Michael H. Olsen, Ed Oskiewicz, John P. Warne, "A Model for Interface Groups", Proceedings of the Tenth Symposium on Reliable Distributed Systems, IEEE, (1991).

[Olsen 92a]

Michael Hoffmann Olsen, “A Persistent Object Infrastructure for Heterogeneous Distributed Systems”, Report No.: RC.343, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92b]

Michael Hoffmann Olsen,"Design and Implementation of a Snapshot Base", Report No.: RC.301, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92c]

Michael Hoffmann Olsen,"Roadmap to the Storage Prototype Deliverables T18, T21 and T30", Report No.: RC.342, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Warne 91]

John Warne, "ANSA: Assumptions, Principles, and Structure", Conference proceedings of Software Engineering Environments 1991, University College of Wales, Aberystwyth, (1991).

10 Handcrafting passivation and activation

10.1 Introduction

This paper is the fifth in a series of eight papers describing a prototype of the ANSA Storage Model [Olsen 91a] developed in the ANSA Testbench 3.0 [AIM 91]. An overview of the prototype is given in [Olsen 92a].

10.1.1 Terminology and background

A *persistent object system* is an infrastructure which enables application systems to treat objects which are stored on secondary storage as in-memory objects. It automatically moves objects, on demand, between secondary storage and main memory and thus eliminates the *impedance mismatch* between the type system of application systems and the type system of secondary storage abstractions like databases and file systems [Copeland 84]. Invocations performed on interface instances of stored objects can be extremely slow because stored objects must be moved in-memory before they can provide a requested service; the challenge is therefore to make persistent object systems run fast.

To move an in-memory object to secondary storage, a persistent object system must change the object's representation so it complies with the type system of a secondary storage abstraction. To facilitate the migration of objects, the chosen secondary storage representation should comply with the type systems of networks. A non-interpreted sequence of bytes is an example of such a representation. A sequence of bytes which represents a single object is termed a *snapshot* of that particular object. The snapshots considered in this paper comprises three parts: (i) an object type component denoting an implementation template; (ii) a state component denoting the current state of an object; and (iii) a schema component denoting the set of interface instances currently supported by an object. A snapshot of an object is different from a *checkpoint*; whereas a snapshot represents a single object, a checkpoint is a set of mutually consistent snapshots. A checkpoint of an object is recursively defined to consist of a snapshot of that object and a checkpoint of each object which has an interface instance referred to by a reference in the initial object. If an object has no references to interface instances in other objects, a snapshot of that object is also a checkpoint.

By representing an object which currently performs no activities as a snapshot stored on secondary storage, the in-memory resources occupied by the object can be freed. A service which enables snapshots to be stored on and retrieved from secondary storage is termed a *snapshot base*. A service which enables objects to obtain replacements for references to interface instances in objects which have migrated or moved to secondary storage is termed a *snapshot locator*.

Passivation is the process of making a snapshot of an object, storing it in a snapshot base, inserting a reference to the snapshot in a snapshot locator as

the replacement for references to the object's interface instances, and terminating the object. An object which has undergone passivation has been *passivated* and is said to be a *passive* object. A passive object must undergo activation before its services are available. *Activation* is the process of instantiating an object, retrieving a passive object's snapshot from a snapshot base, installing the snapshot, creating interface instances in the instantiated object, and inserting references to these interface instances in a snapshot locator as replacements for references to the passive object's interface instances. A passive object which has undergone activation has been *activated* and is said to be an *active* object.

Each object controls its own passivation, but it may allow other objects to request it to passivate. An object cannot passivate until it has completed all activities except the one associated with passivation. If an object possesses a reference to an interface instance of a passive object and uses the reference for invoking an operation, an invocation failure occurs; this kind of failures must be handled by activating the passive object, rebinding the reference to an interface instance in the active object, and retrying the invocation.

Migration is the process of moving an active object from one location in a distributed system to another. By migrating objects, a distributed system can be dynamically reconfigured.

10.1.2 Problem statement

To preserve the computational integrity of distributed applications, each object must persist for at least as long as some other object possesses a reference to one of its interface instances. Apart from the implications this has for object dependability and garbage collection, it also implies that the number of persistent objects can increase to an extent which goes beyond the capacity of in-memory resources available to the nodes in a distributed system.

To cater for a growing number of persistent objects, inactive objects can be passivated, i.e. stored as snapshots on secondary storage, until they need to be activated again. Passive objects persist independently of capsules in which they have been active, so passivation and activation can enable a potentially unlimited number of objects to take part in one or more applications independently of when or where the application is run in a distributed system.

Mechanisms for passivation and activation require a means for turning an active object into a snapshot which is stored on secondary storage, a means for discovering that an object is passive, a means for activating a passive object, and a means for replacing references to interface instances of passive objects with references to interface instances of their active counterparts.

Compared to an active object, a passive object has a higher degree of stability by virtue of residing on secondary storage, but its availability is reduced by the time it takes to activate it. Therefore, passivation and activation may not be feasible in applications which require a high degree of availability.

10.1.3 Purpose

The purpose of this paper is to describe how an application programmer can handcraft extensions to an application so an object can passivate itself and so it is activated when a client invokes its interface instances. The findings are to serve as the basis for generating the infrastructural support for passivation and activation automatically.

The goal for prototyping passivation and activation is to obtain practical experience of the overhead which passivation and activation impose on applications, both in terms of application programming complexity and in terms of performance. A separate paper (see [Olsen 92b]) covers the subject of making passivation and activation transparent to application programmers.

10.1.4 Main ideas

Each object has a passivator which executes a separate thread that periodically checks if the object should passivate. The passivator passivates the object by storing its snapshot in a snapshot base, inserting a reference to the snapshot in a snapshot locator, and terminating the object. Each object also has an activator which activates passive objects when invocation requests fail. The activator activates a passive object by obtaining a reference to its snapshot from a snapshot locator, instantiating an object which uses the snapshot for setting up the state and interface instances of the passive object; the new object inserts references to its interface instances in a snapshot locator as replacements for references to the passive object's interface instances, and returns the replacement for a failed reference to the activator which instigated the activation.

10.1.5 Limitations

Passivation and activation should be dependable, be performed in a secure manner, and be subject to garbage collection schemes. However, these issues and database issues concerned with the management of large numbers of objects (see [Bloom 87]) are not addressed in this paper. Though there may be many areas where passivation and activation mechanisms are useful, this paper focuses on passivation and activation alone as a technique for accommodating a large number of persistent objects in distributed systems.

For simplicity, the paper only considers single object capsules, and if not otherwise made explicit, the term object is synonymous with a capsule; we do not expect problems in extending our work to cover multiple objects in capsules or further levels of nesting.

10.1.6 Audience and prerequisites

This paper targets system developers and programmers who are designing and implementing persistent object systems. It also addresses application programmers who choose to store, retrieve and migrate objects. Readers should be acquainted with ANSA [Warne 91], The ANSA Testbench3.0 [AIM 91], The ANSA Storage Model [Olsen 91a], and the prototype overview [Olsen 92a].

10.1.7 Organisation

An example application is briefly summarized in section 2, and section 3 describes how it is extended with passivation and activation mechanisms. Section 4 compares the performance of the extended application to the original, and section 5 discusses implications passivation and activation have on program complexity. Some related work is presented in section 6, and section 7 concludes the paper. Appendix A contains the code for the example application, and appendix B contains the code for the extended application.

10.2 An example application

The example application which was used in [Olsen 92c] is re-used in this paper to demonstrate how an application programmer can extend an application with passivation and activation mechanisms. The example application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It registers a reference to an interface instance in a trader; this interface enables a client to insert, remove, lookup and list names and phone numbers.

10.3 Adding passivation and activation

The server must be extended so (i) it passivates itself according to a passivation policy, (ii) it can receive passivate requests from the client, and (iii) it can establish itself as an activated object. The client must be extended so it can request the server to passivate itself, and so it obtains replacement references for out-of-date references from a snapshot locator. The extensions comprise:

- simulate a new interface reference format which extends an interface reference to contain a reference to a snapshot locator;
- extend the server with a passivator which passivates the server according to a passivation policy;
- extend the server so it has an interface with a passivate operation which enables the client to request the server to passivate itself;
- extend the client so it can request a snapshot locator for a replacement reference for an out-of-date reference;
- extend the client with an activator which can activate a passive server
- extend the server so it can install a passive object's state and create interface instances when it is instantiated;
- extend the server so it replaces installed references to a passive object's interface instances with references to its own interface instances;
- extend the server so it inserts replacement references for out-of-date references in a snapshot locator;
- enable the client to invoke a server's passivate operation.

10.3.1 Simulating a new interface reference format

A new format for interface references which enables an interface reference to contain references to locators is described in [Nicolaou 91]. The reason for embedding a locator reference in an interface reference is to enable an object which experiences an invocation failure when it invokes a moved object, to request the locator for an up-to-date reference. The new interface reference format has to be simulated in the prototype as it is not supported in Testbench3.0; it will be used in Testbench4.0 so we have chosen only to simulate it to an extent which is sufficient for demonstrating the addressing principles pertaining to passivation and activation. We simulate the format by replacing each variable of type InterfaceRef with a variable of type NewInterfaceRef which has the following IDL definition:

```
NewInterfaceRef: INTERFACE =
BEGIN
```

```
    NewInterfaceRef: TYPE = RECORD
                            [ ir: InterfaceRef,
                              locator: InterfaceRef
                            ];
  END.
```

A variable of type InterfaceRef is represented by a variable of type NewInterfaceRef where the ir field contains the value of the InterfaceRef it represents. The locator field must be assigned when an interface instance is created. To simulate trading of interfaces without modifying the trader implementation in Testbench3.0, a server must register an offer using the ir field of a NewInterfaceRef as the interface reference being registered and the locator field of that NewInterfaceRef as a property of the offer. When a client imports an offer from a trader, it must assign the selected offer's interface reference and its locator property to the fields of a NewInterfaceRef variable. Because the trader in TestBench3.0 does not provide an operation which returns both an interface reference and the properties associated with it, the property which is associated with an imported offer must be obtained by searching the trader separately.

Simulation of interface instance creation and offer registration in a trader:

```
! USE Trader
! USE PhoneBookOp
! DECLARE { phoneBook.ir }: PhoneBookOp SERVER
! USE SLLocate
! DECLARE { phoneBook.locator }: SLLocate CLIENT
! USE NewInterfaceRef
NewInterfaceRef phoneBook;
Result tres;
char hn[64], lr[256], bf[11 + 64 + 256];

! {phoneBook.ir} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
! {phoneBook.locator} <- traderRef$Import("SLLocate", \
                            "/ansa/SnapshotLocator", "")
gethostname(hn, sizeof(hn));
system_putIfRef(lr, &(phoneBook.locator));
(void)sprintf(bf, "NodeName '%s' Locator '%s'", hn, lr);
! {tres} <- traderRef$Register("PhoneBookOp", \
              "/ansa/PhoneBook", bf, phoneBook.ir)
```

Simulation of offer import from a trader:

```
! USE Trader
! USE PhoneBookOp
! DECLARE {phoneBook.ir}: PhoneBookOp CLIENT
! USE NewInterfaceRef
NewInterfaceRef phoneBook;

! {phoneBook.ir} <- traderRef$Import("PhoneBookOp", \
  "/ansa/PhoneBook", "(NodeName == 'audrey') and (Locator != '
')")
/* get locator reference */
{ char id[256], bf[17 + 256];
  SearchResult sresult;
  PropRecord *prp;
  int i;

  system_putIfID(id, phoneBook.ir.ir_id);
  (void)sprintf(bf, "InterfaceId == '%s'", id);
```

```
! {sresult} <- traderRef$Search("PhoneBookOp", "/ansa/
PhoneBook", bf)
  if (sresult.length > 0)
  { prp = ((sresult.data)->or_prop).data;
    for (i = 0; i < ((sresult.data)->or_prop).length; i++, prp++)
     if (strcmp("Locator", prp->p_name) == 0)
     system_getIfRef( prp->p_value, &(phoneBook.locator));
  }
  else
  { fprintf(stderr, "Import failure - no matching offer\n");
!   capsule$Terminate()
  }
}
```

10.3.1.1 Announcements and relocation

An announcement has no delivery guarantee, so the invoker has no means of knowing whether an announcement has been received by a server. This causes problems when a server moves because the client cannot know when an interface reference which is used for sending an announcement to a server is out-of-date. The problem can be solved by providing announcements with a delivery result which indicates whether a server has received an announcement or not. We have chosen to limit the prototype so announcements are not allowed.

10.3.2 The passivator

To control passivation, an object has a passivator which passivates the object according to a passivation policy. To maintain computational consistency, the passivator must not passivate an object before all its activities, except the one associated with passivation, have completed, and new activities must not be initiated while passivation is in progress. To be able to store a snapshot, a passivator must have a reference to a snapshot base. An algorithm for passivation is given below

- decide to passivate or sanction external passivate request;
- Wait till current activities have completed;
- Prevent processing of invocation requests;
- produce a snapshot;
- store the snapshot in a snapshot base;
- update a snapshot locator;
- terminate.

For the prototype, we have chosen to (i) co-locate a passivator with the server, (ii) continue passivation even if invocation requests are waiting to be processed, and (iii) provide a timer based passivation policy. The timer based passivation policy requires the passivator to check periodically whether its server has been inactive for more than a certain period of time, in which case it is passivated. To perform periodical checks, the passivator has a separate thread which executes concurrently with activities in the server. The passivator has an interface of the following type:

```
Passivator: INTERFACE =
BEGIN
  startTimer: ANNOUNCEMENT OPERATION [period: CARDINAL]
              RETURNS [];
```

```
END.
```

The server must spawn a thread for the passivator by creating a Passivator interface instance and invoke its startTimer operation. This can be done by calling the following procedure:

```
! USE Passivator
! DECLARE {lrc} : Passivator CLIENT
! DECLARE {lrs} : Passivator SERVER

void startPassivator(sleepPeriod)
ansa_Cardinal sleepPeriod;
{ ansa_InterfaceRef lrc, lrs;

  ! {lrs} :: Passivator$Create(1)
  copyIfRef(&lrc, &lrs);
! {} <- lrc$startTimer(sleepPeriod)
}
```

The startTimer operation executes an endless loop in which it periodically checks whether the server's interfaces instances have been invoked. To prevent incoming invocations from interfering, startTimer obtains an exclusive lock before making its check and before starting to passivate. startTimer is implemented as follows:

```
int Passivator_startTimer(_attr, period)
ansa_InterfaceAttr *_attr;
ansa_Cardinal period;
{   for(;;)
  { getSingleWriteLock();
    hasBeenInvoked = 0;
    releaseSingleWriteLock();
    timer_Sleep(TSeconds, period);
    getSingleWriteLock();
    if (hasBeenInvoked == 0)
    passivate();
    releaseSingleWriteLock();
  }
}
```

The value of the variable hasBeenInvoked shows whether a service operation has been invoked during a time interval of period length. passivate is implemented as follows:

```
void passivate()
{ ansa_InterfaceAttr *_attr;
  ObjectDenotation od;
  ansa_InterfaceRef myStore;
  SnapshotIdentifier sid;
  EntryList subs;
  InterfaceInfo *next;
  Entry *entryptr;
  int i;
  char hn[64];

  PhoneBookOp_produceSnapshot(_attr, &(od.snapshot));
  strcpy(od.objectType, "PhoneBook");
  od.sid = 0;
! {myStore} <- traderRef$Import("SBStore", "/ansa/SnapshotBase",
"")
! {sid} <- myStore$save(od)
```

```
    if ((subs.data = (Entry *)system_allocate(
        myInterfaces.length * sizeof(Entry))) == NULL)
    instruct_Abort("passivate", insufficientMemory);
    next = myInterfaces.data;
    entryptr = subs.data;
    gethostname(hn, sizeof(hn));
    subs.length = myInterfaces.length;
    for (i = 0; i < myInterfaces.length; i++, next++, entryptr++)
    { copyIfRef(&(entryptr->old), &(next->ir));
      copyIfRef(&(entryptr->new), &myStore);
      entryptr->sid = (SnapshotIdentifier) sid;
      strcpy(entryptr->objectType, "PhoneBook");
      strcpy(entryptr->nodeName, hn);
    }
  ! {} <- phoneBook.locator$register(subs)
    cleanUpConcControl();
  ! capsule$Terminate()
  }
```

The passivate procedure stores objectDenotation, which contains the server's snapshot, in a snapshot base. Each of the server's interface instances, which are recorded in myInterfaces, is registered in a snapshot locator with (i) a replacement reference (i.e. a pair consisting of the snapshot identifier which was returned when the snapshot was stored, and the snapshot base reference), (ii) the name of the server's executable, and (iii) the name of the node on which the server is executing. The name of the server's executable enable an activator to instantiate an appropriate object when activating the passive server; the node name enables an activator to activate the passive server on the node on which it was passivated. When the server has registered the replacement references in a snapshot locator, it terminates itself and the capsule in which it resides.

10.3.2.1 The stale offer problem

When an object passivate, any offers it has registered in traders will be removed because a trader's notification service regards the offers as stale. To prevent this from happening, it must be checked whether a stale offer was registered by a passive object before the offer is removed. For simplicity, the prototype assumes no stale offers are removed.

10.3.2.2 Shallow versus deep passivation

Object passivation only results in the passivation of a single object. If, an object is the root of a tree in which a branch is an object which possesses references to other objects in the tree, and a leaf is an object which do not contain references to other objects in the tree, then the whole tree can be passivated as follows: (i) the root object invokes the passivate operation concurrently on each of its leaves and branches, (ii) each invoked branch acts as the root for the sub-tree it is an ancestor for (ii) when a leaf has passivated itself the passivate request returns to the invoker which then passivate itself. The whole tree has been passivated when all the root object's passivate requests have returned and the root object subsequently passivate itself. The passivated tree is not necessarily a checkpoint, as no attempt is made to maintain mutual consistency between the objects in the tree.

10.3.3 External passivate requests

To enable external passivate requests, the passivator has an interface instance of the following type:

```
Passivate: INTERFACE =
BEGIN
  passivate: ANNOUNCEMENT OPERATION []
             RETURNS [];
END.
```

An invocation of passivate informs the passivator that some other object wants the server to passivate. The passivator's passivation policy determines whether to perform passivation immediately, delay passivation to a more convenient point in time, or discard the request. In the prototype, an external passivate invocation results in the passivate routine being executed without interference from the passivator:

```
int Passivate_passivate(_attr)
ansa_InterfaceAttr *_attr;
{ getSingleWriteLock();
  passivate();
  releaseSingleWriteLock();
}
```

10.3.3.1 Which interface?

In the example application, the passivate operation is made available to a client by making the server's interface compatible with the Passivate interface. In general, the Passivate interface should be a separate management interface.

10.3.4 The activator

An activator activates a passive object when a client requires its service. Activation can be lazy, i.e. a passive object is activated after an invocation failed, or eager, i.e. a passive object is activated prior to invoking its interface instances[1]. Because passivation terminates a capsule, activation must instantiate a capsule. An algorithm for activation is given below

- obtain a reference to a snapshot from a snapshot locator
- instantiate a new capsule via a capsule factory
- instantiate a new object via a factory; the new object receives the snapshot reference via the instantiation argument; it installs the snapshot, creates interfaces, and updates the snapshot locator
- rebind the failed interface reference to its replacement reference

If passivation followed by activation must preserve the location of the active object, then a passive object should always be instantiated in the capsule in which the object was first instantiated. This may not always be possible, for example if capsules are passivated. There are various reasons why it can be useful to change or override the location where a passive object will be activated, for example if a node has become unavailable. In the prototype, each capsule contains a single object, and when it has passivated, the capsule is terminated. Activation on the "same" location where an object was passivated is then reduced to be the node on which the passivated object was instantiated. To be able to instantiate capsules and objects, the activator

1. Eager activation do not eliminate the need for handling invocation failures caused by passivation, because of the possibility of race between activation and invocation.

requires a factory to be running on the node where the passive object is to be activated, and the factory must be able to access an executable which suits the machine which supports the node.

The activator could be co-located with the snapshot locator which then invokes the activator when it receives a lookup request for a reference to an interface instance in a passive object. In the prototype, activation has been made client specific by co-locating the activator with each client, and the snapshot locator is invoked by the activator. The activator has an interface of the following type:

```
Activator: INTERFACE =
NEEDS NewInterfaceRef
BEGIN
  activate: OPERATION [old: NewInterfaceRef]
            RETURNS [new: NewInterfaceRef];
END.
```

An invocation of activate takes an old reference and returns a replacement reference. If the old reference refers to an interface instance in a passive object, activation takes place, and the returned reference refers to the active object's interface instance[1] which corresponds to the interface instance in the passive object referred to by the old reference. If no activation took place, then the returned interface reference is identical to the argument interface reference. The activate operation is implemented as follows:

```
void Activator_activate(_attr, oldRef, newRef)
ansa_InterfaceAttr *_attr;
NewInterfaceRef *oldRef;
NewInterfaceRef *newRef;
{ InstantiateResult res;
  ObjectId id;
  ansa_InterfaceRef fref, cref;
  char nodeName[64], aSnapshotBase[256], anIr[256],
       aLocator[256], activateArgs[64 + 256 + 256 + 256];
  Entry ent;

! {ent} <- oldRef->locator$lookup(oldRef->ir)
  if (ent.sid > 0)
  { /* need to activate */
    system_putIfRef(aSnapshotBase, &(ent.new));
    system_putIfRef(anIr, &(ent.last));
    system_putIfRef(aLocator, &(oldRef->locator));
    sprintf(activateArgs, "%d %s %s %s", ent.sid, aSnapshotBase,
                                         anIr, aLocator);
    sprintf(nodeName, "NodeName == '%s'", ent.nodeName);
!   {fref} <- traderRef$Import("Factory", "/", nodeName)
!   {cref} <- fref$Instantiate(ent.objectType, "", "")
!   {id, res} <- cref$Instantiate(nullRef, activateArgs, "")
    copyIfRef(&(newRef->ir), res.data);
  }
  if (ent.sid == 0)
  { /* assign up-to-date reference to newRef.ir */
    copyIfRef(&(newRef->ir), &(ent.new));
  }
  if (ent.sid == -1)
```

1. In general, there is no guarantee that the object will be active when the new reference is received by the invoker of activate, as it may have passivated.

```
    { /* can't activate - try again later
       */
      copyIfRef(&(newRef->ir), &(oldRef->ir));
    }
    copyIfRef(&(newRef->locator), &(oldRef->locator));
  }
```

The activate operation looks up oldRef.ir in its snapshot locator; if the returned result has a snapshot identifier greater than zero then oldRef is provided by a passive object which must be activated. If the snapshot identifier is equal to zero then the returned result contains a replacement reference which is assigned to newRef; if the snapshot identifier is equal to -1 then oldRef cannot be replaced at the moment.

Activation is carried out by first getting a reference to a factory on the node where the passive object were previously active; the name of this node is obtained from the result returned by the snapshot locator. The factory is used for instantiating a new capsule in which an object of the required type can be instantiated; the type is obtained from the result returned by the snapshot locator. An object is instantiated in the capsule, and it is given instantiation arguments which enable it to establish itself as an activated object. The arguments are (i) the snapshot identifier, (ii) the snapshot base reference, (iii) an interface reference representing[1] the failed reference in oldRef, (iv) and the snapshot locator reference in oldRef.

The instantiated object uses the snapshot identifier's value to discriminates between ordinary instantiation and activation; if it is non-zero then the object was instantiated for activation. The snapshot identifier and the snapshot base reference enable the object to retrieve a snapshot. The interface reference representing the failed reference enables the object to determine which of its references to its new interface instances must be returned to the activator. The object assigns the snapshot locator reference to each new interface instance it creates, and it also uses it to register replacement references.

Rather than giving the snapshot locator reference as argument to object instantiation, it could be given as argument to capsule instantiation. Each object which is instantiated in a capsule will then have the same snapshot locator provided as part of its environment. As the prototype is limited to single object capsules, we found it simplest to provide the reference as an argument to object instantiation.

10.3.4.1 The problem with standalone servers

The server in the example application is a managed object, that is, it is instantiated by invoking a factory. If an application contains standalone servers, that is, servers which are started up by executing their executables directly as an operating system command, there is no executable which can be used for activation via a factory. To enable passivation and activation of standalone objects, an executable must be manufactured for a managed version of the object; this can be done by a preprocessor. Architecturally, all objects are managed, so standalone servers are specific to Testbench3.0. For simplicity, we chose to limit passivation and activation to managed objects.

1. This reference is the value of the last field of the result returned by the locator; it is the latest valid reference to the interface referred to by oldRef. It is necessary to provide the object with this reference as the object itself does not maintain associations between all previously valid references to its interface instances.

10.3.5 Object instantiation for activation

An object must be able to distinguish between being instantiated as a new object and being instantiated for the purpose of activating a passive object. In the latter case, the instantiated object must install the passive object's snapshot, set up interface instances, and register replacement references for the passive object's interface instances in snapshot locators. In general, when an object is instantiated for the purpose of activation, it should not execute the initialization code which a new object normally executes, e.g. if this happened for an object which exports an offer to a trader as part of its initialization, then each activation would result in a new offer being exported.

Each object must have a reference to a snapshot locator which it associates with each interface instance it creates. Each object which can passivate itself must also possess a reference to a snapshot base in which it can store its snapshot when it passivates. These references are given to each object when it is instantiated. When an object is activated, its snapshot base reference must be the one which enables the object to fetch the snapshot which represents the passive object. Therefore, if objects can be activated in other capsules than those in which they were passivated, the snapshot base reference cannot in general be provided as a reference common to all objects in a capsule. snapshot locators are part of interface references so when an object installs a snapshot, it can pick out a snapshot locator reference from an installed interface reference.

When an activator instantiates an object it receives instantiation arguments which enable it to establish its state and interface instances from a snapshot which denotes a passive object, and to register replacement references in a snapshot locator. The instantiation of an object results in its Create__Object routine being executed, the parameters of which contain the instantiation arguments provided by the activator. The PhoneBook's Create__Object routine is implemented as follows:

```
ansa_StatePtr Create__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ SnapshotIdentifier sid;

  initConcControl();
  entryList.length = 0;
  entryList.data   = (PBEntry *)0;
  sscanf(argv[0],"%d", &sid);
  if (sid == 0) /* I'm the first PhoneBook */
  { exportNewInterface(&phoneBook);
    results->length = 1;
    results->data = (ansa_InterfaceRef *) &(phoneBook.ir);
  }
  else /* I'm an activated PhoneBook */
  { ObjectDenotation od;
    ansa_InterfaceAttr *_attr;
    ansa_InterfaceRef myStore, failedRef, newRef, myLocator;
    EntryList subs;

    system_getIfRef(argv[1], &myStore);
    system_getIfRef(argv[2], &failedRef);
    system_getIfRef(argv[3], &myLocator);
```

```
!   {od} <- myStore$fetch(sid)
    PhoneBookOp_installSnapshot(_attr, od.snapshot);
    setUpInterfaces(&myInterfaces, &subs);
!   {} <- myLocator$register(subs)
    getReplacement(&failedRef, &subs, &newRef);
    results->length = 1;
    results->data = (ansa_InterfaceRef *) &newRef;
    assignReplacements(&subs, &myLocator);
    garbageCollectSubs(&subs);
  }
  startPassivator((ansa_Cardinal) 30);
}
```

The received snapshot identifier enables ordinary object instantiation to be distinguished from object activation. When Create__Object is called for ordinary object instantiation, exportNewInterface is called to create the object's initial interface instance and simulate the export of a new interface references as described in section 3.1. A reference to the object's initial interface instance is returned to the client which requested the object instantiation. Before returning, Create__Object calls startPassivator which was described in section 3.2.

When Create__Object is called for object activation, the instantiation arguments supplied by the calling activator must be extracted from a string argument. Then a snapshot is fetched from a snapshot base and installed as the state of the instantiated object by calling PhoneBookOp_installSnapshot which is generated from a SNAPSHOT statement by the preprocessor (see [Olsen 92c]). The installation assigns a list which represents each interface instance in the passive object to myInterfaces. setUpInterfaces (explained in more detail below) uses this list to create the active object's interface instances, and returns a result which contains a list of the interface reference in myInterfaces and their replacement references. The returned list is used to register the replacement references in the snapshot locator. getReplacement is called to get the replacement for the reference which represents the failed reference in the object instantiation arguments; this replacement is returned to the activator. The interface references which are part of the object's installed state and which refer to the passive object's interface instances are out-of-date, so to avoid having to look up replacements for them in a snapshot locator, the replacements are assigned by calling assignReplacements (explained in more detail below). Before returning, Create__Object calls startPassivator.

10.3.5.1 Creating new interface instances

The interface instances which are recorded in myInterfaces are recreated by setUpInterfaces which is implemented as follows:

```
void setUpInterfaces(myIfs, regList)
InterfaceSet *myIfs;
EntryList *regList;
{ InterfaceSet ifSet;
  Entry *eptr;
  InterfaceInfo *iptr;
  int i, if_type, ifref_name;
  ansa_InterfaceRef pbRef;

  ifSet.length = myIfs->length;
  ifSet.data = myIfs->data;
  myIfs->length = 0;
  myIfs->data = (InterfaceInfo *) 0;
```

```
  if ((regList->data = (Entry *)
      system_allocate(ifSet.length * sizeof(Entry))) == NULL)
  instruct_Abort("setUpInterfaces", insufficientMemory);
  regList->length = ifSet.length;
  eptr = regList->data;
  iptr = ifSet.data;
  for (i=0; i<ifSet.length; i++, iptr++, eptr++)
  { if_type = if_locate_name(iptr->type, myIfTypes);
    switch(if_type)
    { case 0:
        {
!         {pbRef} :: PhoneBookOp$Create(iptr->concurrency)
        }
        break;
    }
    copyIfRef(&(eptr->old), &(iptr->ir));
    copyIfRef(&(eptr->new), &pbRef);
    copyIfRef(&(eptr->last), &(iptr->ir));
    eptr->sid = (SnapshotIdentifier) 0;
  }
  garbageCollect(&ifSet);
}
```

Interface instances creation is dispatched on the interface type name for an interface instance which is recorded in myInterfaces. Dispatching is necessary because the statement for creating interface instances does not allow the interface type to be specified as an expression.

10.3.5.2 Swizzling interface references

The interface references which are part of the object's installed state and which refer to the passive object's interface instances are out-of-date. If these references are used for invocation, invocation failures will occur which must be handled by obtaining replacement references from snapshot locators. However, because Create__Object possesses all the information necessary for assigning replacement references to installed references, it is possible to avoid requesting snapshot locators for their replacements. The additional activation overhead imposed by Create__Object assigning replacements should be traded-off against the likelihood of the out-of-date references being used for invocations and the frequency of passivation and activation. Replacement of out-of-date references with up-to-date references is generally known as reference swizzling (see [Moss 92] which contains a detailed evaluation of the cost of various strategies for when and how to swizzle).

In the prototype we have chosen to make Create__Object swizzle out-of-date references by calling assignReplacements which is implemented as follows:

```
void getReplacement(ifRef, regList, newRef)
ansa_InterfaceRef *ifRef, *newRef;
EntryList *regList;
{ Entry *eptr;
  int i;

  eptr = regList->data;
  for (i=0; i < regList->length; i++, eptr++)
  { if (system_cmpIfID((eptr->old).ir_id, ifRef->ir_id) == 0)
    { copyIfRef(newRef, &(eptr->new));
    }
  }
```

```
}

void assignReplacements(regList, locatorRef)
EntryList *regList;
ansa_InterfaceRef *locatorRef;
{ ansa_InterfaceRef newRef;

  getReplacement(&(phoneBook.ir), regList, &newRef);
  system_freeRef(&(phoneBook.ir));
  copyIfRef(&(phoneBook.ir), &newRef);
  system_freeRef(&newRef);
  system_freeRef(&(phoneBook.locator));
  copyIfRef(&(phoneBook.locator), locatorRef);
}
```

The references which are swizzled by assignReplacements must be part of a snapshot, so they are specified in a SNAPSHOT statement [Olsen 92c]. It is therefore possible generate the assignReplacements procedure by a preprocessor. In the example application, only the state of the phoneBook interface reference is part of a snapshot.

10.3.6 Triggering activation

Because the prototype's terminates a capsule when an object is passivated, failed invocations manifest themselves as timed out invocations. A client must handle a timed out invocation (assuming the time out is due to a passive object) by requesting its activator for a replacement reference. Example code for handling invocation time out in a client is given below:

```
PBEntry entry;

{ ansa_Status _stat = ok;
  int _retries = 0;
  NewInterfaceRef _newRef;
  ansa_InterfaceAttr *_attr;

  I_PhoneBookOp_insert(&(phoneBook.ir), (ansa_Voucher *)0,
                       CALLtype, (ansa_Dispatch *)0, entry);
  _stat = C_PhoneBookOp_insert(&(phoneBook.ir),(ansa_Voucher
*)0);
  while (_stat != ok)
  { Activator_activate(_attr, &(phoneBook), &(_newRef));
    system_freeRef(&(phoneBook.ir));
    system_freeRef(&(phoneBook.locator));
    copyIfRef(&(phoneBook.ir), &(_newRef.ir));
    copyIfRef(&(phoneBook.locator), &(_newRef.locator));
    I_PhoneBookOp_insert(&(phoneBook.ir), (ansa_Voucher *)0,
                         CALLtype, (ansa_Dispatch *)0, entry);
    _stat = C_PhoneBookOp_insert(&(phoneBook.ir),(ansa_Voucher
*)0);
    if (_stat != ok)
    { retries++;
      if (retries == 10)
      { binder_terminate();
        instruct_Abort("PhoneBookOp.insert", _stat);
      }
    }
  }
}
```

An invocation time out results in at most 10 retries where each retry is preceded by a call to Activator_activate in order to get a replacement reference. There are several ways to avoid having to wait for a time out. In the example application this is done by preceding each initial invocation by a call to Activator_activate, i.e. activation is eager. However, for simplicity the client in the example application does not retry failed invocations, though it should to avoid race between invocation and passivation.

10.3.7 Preventing simultaneous activations

More than one client can simultaneously attempt to call Activator_activate, so it must be ensured that only one activation takes place, otherwise the computational integrity of an application can be lost. To prevent simultaneous activations, activation requests must be serialized across clients. This is for example achieved if the snapshot locator blocks lookup requests for references to interface instances of passive objects which are undergoing activation. To do this, a flag is simply set when the first lookup request for an interface instance of a passive object is received by the snapshot locator, and the flag is cleared when a replacement reference has been registered. However, failures may occur during activation which prevents a flag from being cleared. This problem can be overcome by setting a deadline for how long an activation flag can be set; when the deadline is reached, it is assumed that activation has failed, so the flag is cleared. The example application does not prevent simultaneous activations.

In theory, it is possible that an object is passivated and activated in a way which prevents a client from performing a successful invocation. This can occur if other clients continuously make the object alternate between being passive and active, without the former client being able to obtain and use an up-to-date references while the object is active. The extended application has no means which prevent a situation like this from occurring.

10.4 Performance

There are five performance aspects to the way passivation and activation is carried out in the example application:

- the cost of checking whether the server is active or passive prior to invoking its interface instance;
- the cost of fetching and storing a snapshot in a snapshot bases;
- the cost of installing and producing a snapshot;
- the cost of setting up new interface instances and registering them in a snapshot locator;
- the cost of swizzling interface references.

The performance of the current implementation is illustrated below by some raw figures (in seconds) for invoking the insert operation on the PhoneBook

	invoca-tions	original application		extended application			
				no activation		activation	
		total time	average	total time	average	total time	average

server. The performance was measured on a HP 425s running HP-UX version 8.01, and both the snapshot locator, the snapshot base, the client, and the server were running on the same machine.

The figures, given in seconds, show the invocation overhead to be a factor 3 when an invocation is preceded by a check to determine that an active object is in fact active, and they show the invocation overhead to be a factor 150 when each invocation is preceded by a check which gives rise to activation of a passive object. The increase in average time when activation is necessary, is probably due to the fact that the time for invoking the snapshot locator and the snapshot base increase as the data which they maintain increase. Because no garbage collection is carried out, each passivation and activation will increase the number of snapshots in the snapshot base by one, and the number of entries in the snapshot locator by two. Performance figures for the snapshot base and the snapshot locator are reported in [Olsen 92d] and [Olsen 92e].

10.5 Programming complexity

In addition to performance overhead, passivation and activation also impose overhead on application programming complexity. This is illustrated by the fact that the number of code lines in the extended application is 3 times that of the original application. A similar figure is reported in [Atkinson 83].

By making passivation and activation transparent to the application programmer, it is possible to reduce the programming overhead of passivation and activation. Ideally, the application programmer should not have to write any code concerned with passivation and activation except when overriding default policies. This means the passivation and activation overhead is limited to a reduction in service availability, possibly only noticeable when the first invocation in a sequence of invocations is performed. In practice, there can be several ways in which passivation and activation are visible to the application programmer. For example, because passivation and activation is restricted to whole objects, large objects can be a problem, so an application programmer may be forced to design applications in which the average object size is sufficiently small. A separate paper describes how to make passivation and activation transparent (to a high degree) at the application level.

10.6 Related work

Arjuna [Shrivastava 89] implements persistent objects in terms of passivation and activation mechanisms which are integrated with transaction mechanisms for controlling concurrent use of persistent objects. The application programmer is required to specify how programmer defined object types are mapped between secondary and in-memory representations.

The principles for supporting persistent objects in Commandos [Marques 89] are very similar to the principles in the ANSA Storage Model. The implementation of passivation and activation for ANSA objects are complicated by the fact that objects have dynamically instantiated interface instances.

Ps-algol [Atkinson 83] pioneered type-orthogonal persistence in terms of a persistent heap, using name servers and database objects which extends the scope of objects to span multiple program executions. In contrast to Arjuna, Comandos, and our work, ps-algol focuses on the language aspect of persistence rather than distribution and heterogeneity aspects.

Passivation and activation can be thought of as object-based virtual memory, where objects are swapped in and out as their services are required. Some virtual memory systems, e.g. Chorus [Habert 90], provide support for persistent objects; however these systems restrict heterogeneity as all machines must support a particular virtual memory system.

10.7 Conclusion

It has been shown how an application can be extended with mechanisms for passivating inactive objects and activating passive objects when their services are required.

Testbench3.0 has significant deviations from the engineering model which is described in [Herbert 91]. The major deviations are (i) the new interface reference format is not simulated properly, (ii) each capsule contains only a single object, (iii) passivation terminates a capsule, (iv) activation creates a new capsule, and (v) activation is carried out by an activator which calls a locator rather than the other way around.

The paper motivated passivation and activation as a means for accommodating an increasing number of persistent objects in distributed systems. One of the overheads of passivation and activation is a reduction in availability, which, for some real-time applications, can be unacceptable. The extended example application showed the marginal cost of activation to be a factor 150 with subsequent invocations carrying a factor 3 overhead if they do not result in activation. We expect performance to improve as new versions of the prototype are developed.

The application programming overhead imposed by passivation and activation can be reduced by making a preprocessor generate passivation and activation code, thus providing passivation and activation as a transparency known as liveness transparency. A separate paper (see [Olsen 92b]) describes the implementation of liveness transparency for the prototype.

10.8 Acknowledgements

Thanks to Nigel Edwards of Hewlett-Packard Laboratories for commenting on an earlier draft of this paper.

10.9 References

[AIM 91]
"ANSAware 3.0 Implementation Manual", Report No.: RM.097, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Atkinson 83]
M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P.W. Cockshott, and R. Morrison, "An approach to persistent programming", *THE COMPUTER JOURNAL,* Volume 26, No. 4, (1983).

[Bloom 87]
T. Bloom and S. B. Zdonik, "Issues in the design of object-oriented database programming languages", *OOPSLA'87*, (1987).

[Copeland 84]
George Copeland, David Maier, "Making Smalltalk a Database System", *SIGMOD'84*, Sigmod Record, Vol. 14, No. 2, (1984).

[Habert 90]
S. Habert, V. Abrossimov and L. Mosseri, "COOL: Kernel Support for Object-Oriented Environments", *OOPSLA'90*, (1990).

[Herbert 91]
Andrew Herbert, "Engineering Model: Conceptual Framework", Report No.: RC.282, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Marques 89]
J. A. Marques and P. Guedes,"Extending the Operating System to Support an Object-Oriented Environment", *OOPSLA'89*, (1989).

[Moss 92]
J. Eliot B. Moss,"Working with persistent objects: To swizzle or not to swizzle", to appear in: *IEEE Transactions on Computers*, 1992 or 1993.

[Nicolaou 91]
Cosmos Nicolaou,"ANSAware 4.0 Interface References", Report No.: RC.268, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91a]
Michael Hoffmann Olsen,"A Model for Storage and Resource Management", Report No.: WP-DCG-91-14, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6Qz, England, (1991).

[Olsen 92a]

Michael Hoffmann Olsen, “A Persistent Object Infrastructure for Heterogeneous Distributed Systems”, Report No.: RC.343, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92b]
Michael Hoffmann Olsen,"Roadmap to the Storage Prototype Deliverables T18, T21 and T30", Report No.: RC.342, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92c]
Michael Hoffmann Olsen,"Automated Generation of Snapshot Operations for an Object", Report No.: RC.288, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92d]
Michael Hoffmann Olsen,"Design and Implementation of a Snapshot Base", Report No.: RC.301, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92e]
Michael Hoffmann Olsen,"Design and Implementation of a Snapshot Locator", Report No.: RC.307, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Shrivastava 89]
S. K. Shrivastava et al, “The Treatment of Persistent Objects in Arjuna”, *The Computer Journal*, Vol. 32, No. 4, (1989).

[Warne 91]
John Warne, "ANSA: Assumptions, Principles, and Structure", *Conference proceedings of Software Engineering Environments 1991*, University College of Wales, Aberystwyth, (1991).

11 Appendix: Handcrafting passivation and activation

11.1 The example application code

This appendix contains the code for the example application referred to in the main part of the paper. The application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It exports a reference to a PhoneBookOp interface instance to a trader, which enables a client to insert, remove, lookup and list names and phone numbers. The implementation of the server's operations has been omitted as they are independent of the snapshot mechanisms which are added in appendix B.

11.1.1 include files

11.1.1.1 ConcControl.h

```
extern void initConcControl();
extern void cleanUpConcControl();
extern void getSingleWriteLock();
extern void releaseSingleWriteLock();
extern void getMultipleReadLock();
extern void releaseMultipleReadLock();
```

11.1.1.2 ConcControl.c

```
#include "ansa.h"
#include "opsys.h"
#include "machine.h"
#include "ecs.h"
#include "ConcControl.h"

ansa_EventCount doneReadorWrite, doneNext;
ansa_Sequencer nextReadorWrite, next;

void initConcControl()
{ doneReadorWrite = ecs_makeEventCount((ansa_Cardinal) 0);
  doneNext        = ecs_makeEventCount((ansa_Cardinal) 0);
  nextReadorWrite = ecs_makeSequencer((ansa_Cardinal) 0);
  next            = ecs_makeSequencer((ansa_Cardinal) 0);
}

void cleanUpConcControl()
{ ecs_freeEventCount(doneReadorWrite);
  ecs_freeEventCount(doneNext);
  ecs_freeSequencer(nextReadorWrite);
  ecs_freeSequencer(next);
}

void getSingleWriteLock()
```

```
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_await(doneNext, ecs_ticket(next));
}

void releaseSingleWriteLock()
{ ecs_advance(doneReadorWrite);
  ecs_advance(doneNext);
}

void getMultipleReadLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_ticket(next);
  ecs_advance(doneReadorWrite);
}

void releaseMultipleReadLock()
{ ecs_advance(doneNext);
}
```

11.1.2 idl files

11.1.2.1 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

11.1.2.2 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

11.1.3 dpl files

11.1.3.1 server.dpl

```
! MANAGED
! USE Trader
! USE PhoneBookOp
! DECLARE { phoneBook, pbRef } : PhoneBookOp SERVER

#include <stdio.h>
#include "ConcControl.h"
```

```
#include "capsule.h"
#include "system.h"

#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16
GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

PBEntryList entryList;
ansa_InterfaceRef phoneBook;

void copyIfRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

ansa_InterfaceRef ref[1];

ansa_StatePtr Create__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ Result tres;
  char hn[64], bf[64 + 11];

  initConcControl();
! {phoneBook} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  entryList.length = 0;
  entryList.data   = (PBEntry *)0;
  gethostname(hn, sizeof(hn));
  (void)sprintf(bf, "NodeName '%s'", hn);
! {tres} <- traderRef$Register("PhoneBookOp", "/ansa/PhoneBook",
\
                               bf, phoneBook)
  copyIfRef(&(ref[0]), &phoneBook);
  results->length = 1;
  results->data = ref;
  return (ansa_StatePtr)0;
}

void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
Name name;
```

```
{ getSingleWriteLock();
  /* remove entry identified by name from entryList */;
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

11.1.3.2 client.dpl

```
! USE Capsule
! DECLARE {cref} : Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook, pbRef} : PhoneBookOp CLIENT

#include   <stdio.h>
#include "capsule.h"
#include "system.h"

#define MAXANSWERLENGTH 31

ansa_InterfaceRef phoneBook;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "create",
  "insert",
  "remove",
  "lookup",
  "list",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {3, 4, 3, 3, 2};
int maxargs[] = {3, 5, 4, 4, 3};

#define INVALID_CMD -1

PBEntryList eList;
```

```
void listEntryList(entryList)
PBEntryList *entryList;
{ int        i;
  char       buffer[128];
  PBEntry  *ifr = entryList->data;

  fprintf(stdout, "\nENTRIES IN PHONEBOOK:\n");
  for (i=0; i<entryList->length; i++, ifr++)
  fprintf(stdout, "  %s %s\n", ifr->name, ifr->no );
}

void PrintUsageAndDie(prog)
char *prog;
{ fprintf(stderr, "usage: %s create machine\n", prog);
  fprintf(stderr, "usage: %s insert name no <property>\n", prog);
  fprintf(stderr, "usage: %s remove name <property>\n", prog);
  fprintf(stderr, "usage: %s lookup name <property>\n", prog);
  fprintf(stderr, "usage: %s list <property>\n", prog);
! capsule$Terminate()
}

void copyIfRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ ansa_InterfaceRef pbRef;
  int i, cmd_no;
  char property[128];

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  strcpy(property, "");
```

```
  switch(cmd_no)
  {  case 1: /* insert */
       { if (argc = 5)
         strcpy(property, argv[4]);
       }
       break;
     case 2: /* remove */
       { if (argc = 4)
         strcpy(property, argv[3]);
       }
       break;
     case 3: /* lookup */
       { if (argc = 4)
         strcpy(property, argv[3]);
       }
       break;
     case 4: /* list */
       { if (argc = 3)
         strcpy(property, argv[2]);
       }
       break;
  }

  if (cmd_no != 0)
  { /* get reference to PhoneBookOp */
!   {pbRef} <- traderRef$Import("PhoneBookOp", "/ansa/
PhoneBook",\
                                property)
    copyIfRef(&phoneBook, &pbRef);
  }

  switch(cmd_no)
  {   case 0: /* create */
        { InstantiateResult res;
          ObjectId id;
          ansa_InterfaceRef fref, cref;
          char machine[128];

          sprintf(machine, "NodeName == '%s'", argv[2]);
!         {fref} <- traderRef$Import("Factory", "/", machine)
!         {cref} <- fref$Instantiate("PhoneBook", "", "")
!         {id, res} <- cref$Instantiate(nullRef, "0", "")
        }
        break;
      case 1: /* insert */
        { PBEntry entry;

          strcpy(entry.name, argv[2]);
          strcpy(entry.no, argv[3]);
!         {} <- phoneBook$insert(entry)
        }
        break;
      case 2: /* remove */
        { Name name;

          strcpy(name, argv[2]);
!         {} <- phoneBook$remove(name)
        }
```

```
            break;
          case 3: /* lookup */
            { PBEntry reply;

!             {reply} <- phoneBook$lookup(argv[2])
              if (strcmp(reply.name, "0") == 0)
               { fprintf(stdout, "Sorry, no entry for %s\n", argv[2]);
              }
              else
              { fprintf(stdout, "No: %s\n", reply.no);
              }
            }
            break;
          case 4: /* list */
            { PBEntryList entryList;

!             {entryList} <- phoneBook$list()
              listEntryList(&entryList);
            }
            break;
    }
}
```

11.1.4 Imakefile

```
            DEFINES =
           INCLUDES =
           IDLFLAGS =
           DPLFLAGS =
           LINTLIBS = $(LINTANSALIB) -lc
           LOCALLIB = $(ANSALIB)
          LOCALLIBD = $(ANSALIBD)
            INSTALL = bsdinstall.sh

IDLFILES = PhoneBookOp.idl
SIFFILES = PhoneBookOp.sif
DPLFILES = server.dpl client.dpl
SRCS     = ConcControl.c
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)

PROGS = server client

# compile idl files
all:: $(SIFFILES)

# compile dpl files
DPLDepend(server)
DPLDepend(client)

all:: $(PROGS)

# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)

SingleProgramTarget(server, server.o sPhoneBookOp.o \
                    ConcControl.o,$(LOCALLIB),)
```

```
SingleProgramTarget(client, client.o
cPhoneBookOp.o,$(LOCALLIB),)

InstallProgram(server,$(TEMPLATEDIR))

NormalLintTarget(*.c)
DependTarget()

IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

11.2 The extended example application

This appendix contains the code for the extended example application which supports passivation and activation as described earlier in the paper. In addition, the client has been modified to enable it to test the passivate and activate operations operations supported by the server.

11.2.1 include files

11.2.1.1 ConcControl.h

```
extern void initConcControl();
extern void cleanUpConcControl();
extern void getSingleWriteLock();
extern void releaseSingleWriteLock();
extern void getMultipleReadLock();
extern void releaseMultipleReadLock();
```

11.2.1.2 ConcControl.c

```
#include "ansa.h"
#include "opsys.h"
#include "machine.h"
#include "ecs.h"
#include "ConcControl.h"

ansa_EventCount doneReadorWrite, doneNext;
ansa_Sequencer nextReadorWrite, next;

void initConcControl()
{ doneReadorWrite = ecs_makeEventCount((ansa_Cardinal) 0);
  doneNext        = ecs_makeEventCount((ansa_Cardinal) 0);
  nextReadorWrite = ecs_makeSequencer((ansa_Cardinal) 0);
  next            = ecs_makeSequencer((ansa_Cardinal) 0);
}

void cleanUpConcControl()
{ ecs_freeEventCount(doneReadorWrite);
  ecs_freeEventCount(doneNext);
  ecs_freeSequencer(nextReadorWrite);
  ecs_freeSequencer(next);
}

void getSingleWriteLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_await(doneNext, ecs_ticket(next));
}
```

```
void releaseSingleWriteLock()
{ ecs_advance(doneReadorWrite);
  ecs_advance(doneNext);
}

void getMultipleReadLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_ticket(next);
  ecs_advance(doneReadorWrite);
}

void releaseMultipleReadLock()
{ ecs_advance(doneNext);
}
```

11.2.2 idl files

11.2.2.1 SnapshotBaseTypes.idl

```
SnapshotBaseTypes: INTERFACE =
NEEDS Snapshot;
NEEDS SnapshotIdentifier;
BEGIN
  ObjectDenotation: TYPE = RECORD
                           [ snapshot: Snapshot,
                             objectType: ARRAY 32 OF CHAR,
                             sid: SnapshotIdentifier
                           ];
  Cache: TYPE = SEQUENCE OF ObjectDenotation;
END.
```

11.2.2.2 SBStore.idl

```
SBStore: INTERFACE =
NEEDS SnapshotBaseTypes;
BEGIN
  save:   OPERATION [ objectDenotation: ObjectDenotation ]
          RETURNS [ sid: SnapshotIdentifier];
  remove: OPERATION [ sid: SnapshotIdentifier]
          RETURNS [ objectDenotation: ObjectDenotation];
  fetch:  OPERATION [ sid: SnapshotIdentifier]
          RETURNS [ objectDenotation: ObjectDenotation];
END.
```

11.2.2.3 LocatorTypes.idl

```
LocatorTypes: INTERFACE =
NEEDS SnapshotIdentifier;
BEGIN
  Entry:     TYPE = RECORD
                    [ old: InterfaceRef,
                      new: InterfaceRef,
                      sid: SnapshotIdentifier,
                      last: InterfaceRef,
                      objecttype: ARRAY 32 OF Char,
                      nodeName: ARRAY 32 OF Char
                    ];
  EntryList: TYPE = SEQUENCE OF Entry;
  IfRefList: TYPE = SEQUENCE OF InterfaceRef;
```

```
END.
```

11.2.2.4 SLLocate.idl

```
SLLocate: INTERFACE =
NEEDS LocatorTypes;
BEGIN
  register:   OPERATION [entries: EntryList]
              RETURNS [];
  deregister: OPERATION [ifrefs: IfRefList]
              RETURNS [];
  lookup:     OPERATION [old: InterfaceRef]
              RETURNS [entry: Entry];
END.
```

11.2.2.5 NewInterfaceRef.idl

```
NewInterfaceRef: INTERFACE =
BEGIN
  NewInterfaceRef: TYPE = RECORD
                          [ ir: InterfaceRef,
                            locator: InterfaceRef
                          ];
END.
```

11.2.2.6 Passivator.idl

```
Passivator: INTERFACE =
BEGIN
  startTimer: ANNOUNCEMENT OPERATION [period: CARDINAL]
              RETURNS [];
END.
```

11.2.2.7 Passivate.idl

```
Passivate: INTERFACE =
BEGIN
  passivate: ANNOUNCEMENT OPERATION []
                RETURNS [];
END.
```

11.2.2.8 PhoneBookTypes.idl

```
PhoneBookTypes: INTERFACE =
NEEDS NewInterfaceRef;
BEGIN
  Name: TYPE = ARRAY 64 OF CHAR;
  No:   TYPE = ARRAY 64 OF CHAR;
  PBEntry:     TYPE = RECORD
                      [ name: Name,
                        no  : No
                      ];
  PBEntryList: TYPE = SEQUENCE OF PBEntry;
END.
```

11.2.2.9 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
IS COMPATIBLE WITH Passivate;
BEGIN
  insert: OPERATION [entry : PBEntry]
          RETURNS [];
```

```
  remove: OPERATION [name : Name]
          RETURNS [];
  lookup: OPERATION [name : Name]
          RETURNS [entry : PBEntry];
  list:   OPERATION []
          RETURNS [entryList : PBEntryList];
END.
```

11.2.3 dpl files

11.2.3.1 PhoneBook.dpl

```
! MANAGED
! USE Trader
! USE PhoneBookOp
! DECLARE { pbRef } : PhoneBookOp SERVER

#include <stdio.h>
#include "ConcControl.h"
#include "capsule.h"
#include "system.h"

#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16

GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

PBEntryList entryList;
NewInterfaceRef phoneBook;

! SNAPSHOT OF {entryList PBEntryList, phoneBook NewInterfaceRef}
\
  IN PhoneBookOp OF PhoneBook

! USE SLLocate
! DECLARE {locatorRef,phoneBook.locator,myLocator}: SLLocate
CLIENT
! USE SBStore
! DECLARE { storeRef, myStore } : SBStore CLIENT
! USE Passivator
! DECLARE {lrc} : Passivator CLIENT
! DECLARE {lrs} : Passivator SERVER
int hasBeenInvoked;

void createNewInterface(newRef)
NewInterfaceRef *newRef;
{ ansa_InterfaceRef pbRef, locatorRef;

! {pbRef} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  copyIfRef(&(newRef->ir), &pbRef);
! {locatorRef} <- traderRef$Import("SLLocate", \
                                   "/ansa/SnapshotLocator", "")
  copyIfRef(&(newRef->locator), &locatorRef);
}

void exportNewInterface(newRef)
NewInterfaceRef *newRef;
{ char hn[64], lr[256], bf[11 + 64 + 256];
  Result tres;
```

```
  createNewInterface(newRef);
  gethostname(hn, sizeof(hn));
  system_putIfRef(lr, &(newRef->locator));
  (void)sprintf(bf, "NodeName '%s' Locator '%s'", hn, lr);
! {tres} <- traderRef$Register("PhoneBookOp", "/ansa/PhoneBook",
\
                               bf, newRef->ir)
}

void passivate()
{ ansa_InterfaceAttr *_attr;
  ObjectDenotation od;
  ansa_InterfaceRef myStore;
  SnapshotIdentifier sid;
  EntryList subs;
  InterfaceInfo *next;
  Entry *entryptr;
  int i;
  char hn[64];

  PhoneBookOp_produceSnapshot(_attr, &(od.snapshot));
  strcpy(od.objectType, "PhoneBook");
  od.sid = 0;
! {myStore} <- traderRef$Import("SBStore", "/ansa/SnapshotBase",
"")
! {sid} <- myStore$save(od)
  if ((subs.data = (Entry *)system_allocate(
      myInterfaces.length * sizeof(Entry))) == NULL)
  instruct_Abort("passivate", insufficientMemory);
  next = myInterfaces.data;
  entryptr = subs.data;
  gethostname(hn, sizeof(hn));
  subs.length = myInterfaces.length;
  for (i = 0; i < myInterfaces.length; i++, next++, entryptr++)
  { copyIfRef(&(entryptr->old), &(next->ir));
    copyIfRef(&(entryptr->new), &myStore);
    entryptr->sid = (SnapshotIdentifier) sid;
    strcpy(entryptr->objectType, "PhoneBook");
    strcpy(entryptr->nodeName, hn);
  }
! {} <- phoneBook.locator$register(subs)
  cleanUpConcControl();
! capsule$Terminate()
}

int Passivator_startTimer(_attr, period)
ansa_InterfaceAttr *_attr;
ansa_Cardinal period;
{ for(;;)
  { getSingleWriteLock();
    hasBeenInvoked = 0;
    releaseSingleWriteLock();
    timer_Sleep(TSeconds, period);
    getSingleWriteLock();
    if (hasBeenInvoked == 0)
    passivate();
    releaseSingleWriteLock();
```

```
  }
}

! USE Passivator
! DECLARE {lrc} : Passivator CLIENT
! DECLARE {lrs} : Passivator SERVER

void startPassivator(sleepPeriod)
ansa_Cardinal sleepPeriod;
{ ansa_InterfaceRef lrc, lrs;

! {lrs} :: Passivator$Create(1)
  copyIfRef(&lrc, &lrs);
! {} <- lrc$startTimer(sleepPeriod)
}

static char *myIfTypes[] =
{ "PhoneBookOp",
  (char *)0
};

int if_locate_name(name, vec)
char *name;
char *vec[];
{ int i;
  int strcmp();

  for (i = 0; vec[i] != (char *)0; i++)
  if (strcmp(name, vec[i]) == 0)
  return i;
  return -1;
}

void setUpInterfaces( myIfs, regList)
InterfaceSet *myIfs;
EntryList *regList;
{ InterfaceSet ifSet;
  Entry *eptr;
  InterfaceInfo *iptr;
  int i, if_type, ifref_name;
  ansa_InterfaceRef pbRef;

  ifSet.length = myIfs->length;
  ifSet.data = myIfs->data;
  myIfs->length = 0;
  myIfs->data = (InterfaceInfo *) 0;
  if ((regList->data = (Entry *)
      system_allocate(ifSet.length * sizeof(Entry))) == NULL)
  instruct_Abort("setUpInterfaces", insufficientMemory);
  regList->length = ifSet.length;
  eptr = regList->data;
  iptr = ifSet.data;
  for (i=0; i<ifSet.length; i++, iptr++, eptr++)
  { if_type = if_locate_name(iptr->type, myIfTypes);
    switch(if_type)
    { case 0:
        {
!         {pbRef} :: PhoneBookOp$Create(iptr->concurrency)
```

```
        }
        break;
    }
    copyIfRef(&(eptr->old), &(iptr->ir));
    copyIfRef(&(eptr->new), &pbRef);
    copyIfRef(&(eptr->last), &(iptr->ir));
    eptr->sid = (SnapshotIdentifier) 0;
  }
  /* garbage collect ifSet */
  iptr = ifSet.data;
  for (i=0; i<ifSet.length; i++, iptr++)
  system_freeRef(&(iptr->ir));
  system_free(ifSet.data);
}

void getReplacement(ifRef, regList, newRef)
ansa_InterfaceRef *ifRef, *newRef;
EntryList *regList;
{ Entry *eptr;
  int i;

  eptr = regList->data;
  for (i=0; i < regList->length; i++, eptr++)
  { if (system_cmpIfID((eptr->old).ir_id, ifRef->ir_id) == 0)
    { copyIfRef(newRef, &(eptr->new));
    }
  }
}

void assignReplacements(regList, locatorRef)
EntryList *regList;
ansa_InterfaceRef *locatorRef;
{ ansa_InterfaceRef newRef;

  getReplacement(&(phoneBook.ir), regList, &newRef);
  system_freeRef(&(phoneBook.ir));
  copyIfRef(&(phoneBook.ir), &newRef);
  system_freeRef(&newRef);
  system_freeRef(&(phoneBook.locator));
  copyIfRef(&(phoneBook.locator), locatorRef);
}

void garbageCollectSubs(regList)
EntryList *regList;
{ int i;
  Entry *eptr;

  eptr = regList->data;
  for (i=0; i < regList->length; i++, eptr++)
  { system_freeRef(&(eptr->old));
    system_freeRef(&(eptr->new));
    system_freeRef(&(eptr->last));
  }
  system_free(regList->data);
}

ansa_StatePtr Create__Object(argc, argv, envp, results)
int argc;
```

```
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ SnapshotIdentifier sid;

  initConcControl();
  entryList.length = 0;
  entryList.data   = (PBEntry *)0;
  sscanf(argv[0],"%d", &sid);
  if (sid == 0) /* I'm the first PhoneBook */
  { exportNewInterface(&phoneBook);
    results->length = 1;
    results->data = (ansa_InterfaceRef *) &(phoneBook.ir);
  }
  else /* I'm an activated PhoneBook */
  { ObjectDenotation od;
    ansa_InterfaceAttr *_attr;
    ansa_InterfaceRef myStore, failedRef, newRef, myLocator;
    EntryList subs;

    system_getIfRef(argv[1], &myStore);
    system_getIfRef(argv[2], &failedRef);
    system_getIfRef(argv[3], &myLocator);
!   {od} <- myStore$fetch(sid)
    PhoneBookOp_installSnapshot(_attr, od.snapshot);
    setUpInterfaces(&myInterfaces, &subs);
!   {} <- myLocator$register(subs)
    getReplacement(&failedRef, &subs, &newRef);
    results->length = 1;
    results->data = (ansa_InterfaceRef *) &newRef;
    assignReplacements(&subs, &myLocator);
    garbageCollectSubs(&subs);
  }
  startPassivator((ansa_Cardinal) 30);
}

void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
Name name;
{ getSingleWriteLock();
  /* remove entry identified by name from entryList */;
  releaseSingleWriteLock();
  return 1;
}
```

```
int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

11.2.3.2 client.dpl

```
! USE Trader
! USE Capsule
! DECLARE {cref}: Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook.ir, pbRef}: PhoneBookOp CLIENT
! USE NewInterfaceRef
! DECLARE {newRef->ir}: PhoneBookOp CLIENT
! USE SLLocate
! DECLARE {newRef->locator}: SLLocate CLIENT

#include <stdio.h>
#include "capsule.h"
#include "system.h"

#define MAXANSWERLENGTH 31

#include "RecordMyInterfaces.c"
NewInterfaceRef phoneBook;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "create",
  "insert",
  "remove",
  "lookup",
  "list",
  "passivate",
  "activate",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {2, 4, 3, 3, 2, 2, 2};
int maxargs[] = {2, 4, 3, 3, 2, 2, 2};
```

```
#define INVALID_CMD -1

PBEntryList eList;

void listEntryList(entryList)
  PBEntryList *entryList;
{ int        i;
  char       buffer[128];
  PBEntry  *ifr = entryList->data;

  fprintf(stdout, "\nENTRIES IN PHONEBOOK:\n");
  for (i=0; i<entryList->length; i++, ifr++)
  { fprintf(stdout, "  %s %s\n", ifr->name, ifr->no );
  }
}

void PrintUsageAndDie(prog)
char *prog;
{ fprintf(stderr, "usage: %s create\n", prog);
  fprintf(stderr, "usage: %s insert name no\n", prog);
  fprintf(stderr, "usage: %s remove name\n", prog);
  fprintf(stderr, "usage: %s lookup name\n", prog);
  fprintf(stderr, "usage: %s list\n", prog);
  fprintf(stderr, "usage: %s passivate\n", prog);
  fprintf(stderr, "usage: %s activate\n", prog);
! capsule$Terminate()
}

void Activator_activate(_attr, oldRef, newRef)
ansa_InterfaceAttr *_attr;
NewInterfaceRef *oldRef;
NewInterfaceRef *newRef;
{ InstantiateResult res;
  ObjectId id;
  ansa_InterfaceRef fref, cref;
  char nodeName[64], aSnapshotBase[256], anIr[256],
       aLocator[256], activateArgs[64 + 256 + 256 + 256];
  Entry ent;

! {ent} <- oldRef->locator$lookup(oldRef->ir)
  if (ent.sid > 0)
  { /* need to activate */
    system_putIfRef(aSnapshotBase, &(ent.new));
    system_putIfRef(anIr, &(ent.last));
    system_putIfRef(aLocator, &(oldRef->locator));
    sprintf(activateArgs, "%d %s %s %s", ent.sid, aSnapshotBase,
                                         anIr, aLocator);
    sprintf(nodeName, "NodeName == '%s'", ent.nodeName);
!   {fref} <- traderRef$Import("Factory", "/", nodeName)
!   {cref} <- fref$Instantiate(ent.objectType, "", "")
!   {id, res} <- cref$Instantiate(nullRef, activateArgs, "")
    copyIfRef(&(newRef->ir), res.data);
  }
  if (ent.sid == 0)
  { /* assign up-to-date reference to newRef.ir */
    copyIfRef(&(newRef->ir), &(ent.new));
  }
  if (ent.sid == -1)
```

```
  { /* can't activate - try again later
     */
    copyIfRef(&(newRef->ir), &(oldRef->ir));
  }
  copyIfRef(&(newRef->locator), &(oldRef->locator));
}

void importNewReference(newRef)
NewInterfaceRef *newRef;
{ ansa_InterfaceRef pbRef;

  /* get reference to PhoneBookOp */
! {pbRef} <- traderRef$Import("PhoneBookOp", \
  "/ansa/PhoneBook", "(NodeName == 'audrey') and (Locator != '
')")
  copyIfRef(&(newRef->ir), &pbRef);
  /* get the locator */
  { char id[256], bf[17 + 256];
    SearchResult sresult;
    PropRecord *prp;
    int i;

    system_putIfID(id, (newRef->ir).ir_id);
    (void)sprintf(bf, "InterfaceId == '%s'", id);
!   {sresult} <- traderRef$Search("PhoneBookOp", \
                  "/ansa/PhoneBook", bf)
    if (sresult.length > 0)
    { prp = ((sresult.data)->or_prop).data;
     for (i = 0; i < ((sresult.data)->or_prop).length; i++, prp++)
      if (strcmp("Locator", prp->p_name) == 0)
      system_getIfRef( prp->p_value, &(newRef->locator));
    }
    else
    { fprintf(stderr, "Import failure - no matching offer\n");
!     capsule$Terminate()
    }
  }
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ int i, cmd_no;

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }
```

```
  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  if (cmd_no != 0)
  { /* get reference to PhoneBookOp */
    importNewReference(&phoneBook);
  }

  switch(cmd_no)
  { case 0:
      { InstantiateResult res;
        ObjectId id;
        ansa_InterfaceRef fref, cref;

!       {fref} <- traderRef$Import("Factory", "/",\
                                        "NodeName == 'audrey'")
!       {cref} <- fref$Instantiate("PhoneBook", "", "")
!       {id, res} <- cref$Instantiate(nullRef, "0", "")
      }
      break;
    case 1:
      { PBEntry entry;
        NewInterfaceRef newRef;;

        strcpy(entry.name, argv[2]);
        strcpy(entry.no, argv[3]);
        Activator_activate(_attr, &phoneBook, &newRef);
        copyIfRef(&(phoneBook.ir), &(newRef.ir));
        copyIfRef(&(phoneBook.locator), &(newRef.locator));
!       {} <- phoneBook.ir$insert(entry)
      }
      break;
    case 2:
      { Name name;
        NewInterfaceRef newRef;;

        strcpy(name, argv[2]);
        Activator_activate(_attr, &phoneBook, &newRef);
        copyIfRef(&(phoneBook.ir), &(newRef.ir));
        copyIfRef(&(phoneBook.locator), &(newRef.locator));
!       {} <- phoneBook.ir$remove(name)
      }
      break;
    case 3:
      { PBEntry reply;
        NewInterfaceRef newRef;;

        Activator_activate(_attr, &phoneBook, &newRef);
        copyIfRef(&(phoneBook.ir), &(newRef.ir));
        copyIfRef(&(phoneBook.locator), &(newRef.locator));
!       {reply} <- phoneBook.ir$lookup(argv[2])
        if (strcmp(reply.name, "0") == 0)
        { fprintf(stdout, "Sorry, no entry for %s\n", argv[2]);
```

```
            }
            else
            { fprintf(stdout, "No: %s\n", reply.no);
            }
          }
          break;
        case 4:
          { PBEntryList entryList;
            NewInterfaceRef newRef;;

            Activator_activate(_attr, &phoneBook, &newRef);
            copyIfRef(&(phoneBook.ir), &(newRef.ir));
            copyIfRef(&(phoneBook.locator), &(newRef.locator));
!           {entryList} <- phoneBook.ir$list()
            listEntryList(&entryList);
          }
          break;
        case 5:
          { NewInterfaceRef newRef;;

            Activator_activate(_attr, &phoneBook, &newRef);
            copyIfRef(&(phoneBook.ir), &(newRef.ir));
            copyIfRef(&(phoneBook.locator), &(newRef.locator));
!           {} <- phoneBook.ir$passivate()
!           {} <- phoneBook.ir$passivate()
!           {} <- phoneBook.ir$passivate()
          }
          break;
        case 6:
          { NewInterfaceRef newRef;;

            Activator_activate(_attr, &phoneBook, &newRef);
            copyIfRef(&(phoneBook.ir), &(newRef.ir));
            copyIfRef(&(phoneBook.locator), &(newRef.locator));
          }
          break;
      }
    }
```

11.2.4 Imakefile

```
             STIDIR = ../../../../snapshot/transparency/include
             STLDIR = ../../../../snapshot/transparency/idl
              STUBC = ../../../../snapshot/stubc/stubc
              PREPC = ../../../../snapshot/prepc/prepc
            DEFINES =
           INCLUDES = -I$(STIDIR)
           IDLFLAGS = -I$(STLDIR)
           DPLFLAGS =
           LINTLIBS = $(LINTANSALIB) -lc
           LOCALLIB = $(ANSALIB)
          LOCALLIBD = $(ANSALIBD)
            INSTALL = bsdinstall.sh

    IDLFILES = NewInterfaceRef.idl PhoneBookOp.idl SLLocate.idl \
               SBStore.idl Passivator.idl
    SIFFILES = NewInterfaceRef.sif PhoneBookOp.sif SLLocate.sif \
               SBStore.sif Passivator.sif
```

```
DPLFILES = server.dpl client.dpl
SRCS     = ConcControl.c
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)

PROGS = server client

# compile idl files
all:: $(SIFFILES)

# compile dpl files
DPLDepend(server)
DPLDepend(client)

all:: $(PROGS)

# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)

SingleProgramTarget(server, server.o sPhoneBookOp.o cSLLocate.o
\
                        cSBStore.o sPassivator.o cPassivator.o \
                            ConcControl.o,$(LOCALLIB),)
SingleProgramTarget(client, client.o cPhoneBookOp.o \
                            cSLLocate.o,$(LOCALLIB),)

InstallProgram(server,$(TEMPLATEDIR))

NormalLintTarget(*.c)
DependTarget()

IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

12 Automated Generation of Passivation and Activation

12.1 Introduction

This paper is the sixth in a series of eight papers describing a prototype of the ANSA Storage Model [Olsen 91] developed in the ANSA Testbench 3.0 [AIM 91]. An overview of the prototype is given in [Olsen 92a].

12.1.1 Terminology

A *persistent object system* is an infrastructure which enables application systems to treat objects which are stored on secondary storage as in-memory objects. It automatically moves objects, on demand, between secondary storage and main memory and thus eliminates the *impedance mismatch* between the type system of application systems and the type system of secondary storage abstractions like databases and file systems [Copeland 84]. Invocations performed on interface instances of stored objects can be extremely slow because stored objects must be moved in-memory before they can provide a requested service; the challenge is therefore to make persistent object systems run fast.

To move an in-memory object to secondary storage, a persistent object system must change the object's representation so it complies with the type system of a secondary storage abstraction. To facilitate the migration of objects, the chosen secondary storage representation should comply with the type systems of networks. A non-interpreted sequence of bytes is an example of such a representation. A sequence of bytes which represents a single object is termed a *snapshot* of that particular object. The snapshots considered in this paper consists of three parts: (i) an object type component denoting an implementation template, (ii) a state component denoting the current state of an object, and (iii) a schema component denoting the set of interface instances currently supported by an object. A snapshot of an object is different from a *checkpoint*; whereas a snapshot represents a single object, a checkpoint is a set of mutually consistent snapshots. A checkpoint of an object is recursively defined to consist of a snapshot of that object and a checkpoint of each object which has an interface instance referred to by a reference in the initial object. If an object has no references to interface instances in other objects, a snapshot of that object is also a checkpoint.

By representing an object which currently performs no activities as a snapshot stored on secondary storage, the in-memory resources occupied by the object can be freed. A service which enables snapshots to be stored on and retrieved from secondary storage is termed a *snapshot base*. A service which enables objects to obtain replacements for references to interface instances in objects which have migrated or moved to secondary storage is termed a *snapshot locator*.

Passivation is the process of making a snapshot of an object, storing it in a snapshot base, inserting a reference to the snapshot in a snapshot locator as the replacement for references to the object's interface instances, and terminating the object. An object which has undergone passivation has been *passivated* and is said to be a *passive* object. A passive object must undergo activation before its services are available. *Activation* is the process of instantiating an object, retrieving a passive object's snapshot from a snapshot base, installing the snapshot, creating interface instances in the instantiated object, and inserting references to these interface instances in a snapshot locator as replacements for references to the passive object's interface instances. A passive object which has undergone activation has been *activated* and is said to be an *active* object.

Each object controls its own passivation, but it may allow other objects to request it to passivate. An object cannot passivate until it has completed all activities except the one associated with passivation. If an object possesses a reference to an interface instance of a passive object and uses the reference for invoking an operation, an invocation failure occurs; this kind of failures must be handled by activating the passive object, rebinding the reference to an interface instance in the active object, and retrying the invocation. Compared to an active object, a passive object has a higher degree of stability by virtue of residing on secondary storage, but its availability is reduced by the time it takes to activate it. Therefore, passivation and activation may not be feasible in applications which require a high degree of availability.

Migration is the process of moving an active object from one location in a distributed system to another. By migrating objects, a distributed system can be dynamically reconfigured.

12.1.2 Problem statement

To preserve the computational integrity of distributed applications, each object must persist for at least as long as some other object possesses a reference to one of its interface instances. Apart from the implications this has for object dependability and garbage collection, it also implies that the number of persistent objects can increase to an extent which goes beyond the capacity of in-memory resources available to the nodes in a distributed system.

To cater for a growing number of persistent objects, inactive objects can be passivated, i.e. stored as snapshots on secondary storage, until they need to be activated again. Passive objects persist independently of capsules in which they have been active, so passivation and activation can enable a potentially unlimited number of objects to take part in one or more applications independently of when or where the application is run in a distributed system.

Passivation and activation impose programming complexity and performance overheads on applications, as reported in [Olsen 92b]. The programming complexity overhead for an example application is a factor 3, so there is a considerable gain at the application programming level if the code which implements passivation and activation is generated automatically. The automatic provision of a passivation and activation infrastructure is known as *liveness transparency* [Herbert 91].

12.1.3 Purpose

The purpose of this paper is to describe how a passivation and activation infrastructure can be generated automatically, thus reducing the complexity of

applications which need to passivate and activate objects. The paper does not describe how the passivation and activation infrastructure is implemented.

12.1.4 Main ideas

The passivation and activation mechanisms which are described in [Olsen 92b are used as the basis for the passivation and activation infrastructure. The application independent passivation and activation code is separated out into include files, and the prepc preprocessor is extended in order to include these files and to generate the application dependent parts of the passivation and activation infrastructure. Application specific details which are required by the preprocessor must be specified by the application programmer in a prepc statement in a server object. These details are (i) which program variables refer to state which must be part of a snapshot; (ii) which IDL types these variables have; (iii) the name of the server's executable; (iv) the period of time the server must have been idle before it passivates itself, and (v) which external interface a passivate operation can be made available in. In addition, the application programmer is required to follow some programming conventions in order (i) to facilitate the simulation of a new interface reference format; (ii) to create an object; (iii) to control concurrency; and (iv) to enable idle time detection.

12.1.5 Limitations

For simplicity, the paper only considers single object capsules, and if not otherwise made explicit, the term object is synonymous with a capsule; we do not expect problems in extending our work to cover multiple objects in capsules or further levels of nesting.

The passivation and activation infrastructure does not tolerate failures and does not collect garbage, so the application programmer will have to deal with any problems this gives rise to.

The simulation of a new interface reference format is considerably simplified; existing services in Testbench3.0 do not support this format, therefore the application programmer is required to participate in the simulation. The new interface reference format will be supported in Testbench4.0, so the simulation is only carried out to an extent which is sufficient to support the addressing principles which pertain to passivation and activation.

Services which are part of Testbench3.0, i.e. traders and factories, and services which are part of the passivation and activation infrastructure, i.e. snapshot bases and snapshot locators, cannot be passivated or activated in the present prototype. The prototype has a fixed activation policy, but the passivation policy is parameterized.

12.1.6 Audience and prerequisites

This paper targets system developers and programmers who are designing and implementing persistent object systems. It also addresses application programmers who choose to store, retrieve and migrate objects. Readers should be acquainted with ANSA [Warne 91], The ANSA Testbench3.0 [AIM 91], The ANSA Storage Model [Olsen 91], and the prototype overview [Olsen 92a].

12.1.7 Organisation

A new prepc statement for declaring server passivation is presented in section 2, and section 3 describes activation triggering. Section 4 describes how the simulation of a new interface reference format affects the application programmer. Programming conventions which are imposed by the passivation and activation infrastructure, and conventions for building applications are described in section 5 and 6. Section 7 examines the complexity of an example application for which the passivation and activation infrastructure is generated. Some related work is presented in section 8, and section 9 concludes the paper. Extensions made to the prepc preprocessor are detailed in Appendix A, and Appendix B and C contain a complete example application for which the passivation and activation infrastructure is generated.

12.2 Passivation triggering

To enable a server to passivate, the application programmer must declare passivation by the following new prepc PASSIVATION statement:

```
! PASSIVATION OF {v1 IDLType1, ... , vN IDLTypeN} IN objType \
  INTERNALLY period EXTERNALLY ifType
```

from which the prepc preprocessor generates application dependent code for the passivation and activation infrastructure. The program variables and their associated IDL types which are listed in {v1 IDLType1, ... , vN IDLTypeN} specify which application state must be part of a snapshot; objType specifies which executable the activation infrastructure must use for activating the server; \ is a new line separator; period declares the time interval during which the server must have been idle before being passivated by its infrastructure, and ifType declares which type of interface a passivate operation can be made available in. See Appendix A for the extensions which have been made to the preprocessor in order to preprocess PASSIVATION statements.

The passivation policy is to passivate a server when it has been idle for a given period of time. This period is parameterized by the period argument in the PASSIVATION statement. The PASSIVATION statement is an extension of the SNAPSHOT statement which is described in [Olsen 92c], and its restrictions are inherited. It is the programmers responsibility to ensure that memory referred to by the variables declared in a PASSIVATION statement is not referred to by other variables. If some memory is referred to by other variables, then these variables will no longer be valid after activation, because the memory will have been garbage collected and replaced by some other memory containing state which activation has installed. If two or more variables in a PASSIVATION statement refer to the same memory segment (i.e. state is shared), then the snapshot produced by passivation will contain separate copies of the memory segments for each variable. This means that state which was shared before passivation, will not be shared after activation.

To make a passivate operation externally available in interface instances of type ifType, the programmer must include the line

```
IS COMPATIBLE WITH Passivate;
```

in the IDL file which defines ifType. Passivate is a library interface type which is defined by:

```
Passivate: INTERFACE =
BEGIN
  passivate: ANNOUNCEMENT OPERATION []
```

```
                RETURNS [];
END.
```

12.3 Activation triggering

The triggering of activation is generic to applications, so the application programmer is not required to declare any application dependent activation triggering details; it is an integral part of the code generated for a prepc invocation which simulates a new interface reference format, see section 4.

When an interface instance is invoked, before making a service request, the client's infrastructure attempts to activate the server in order to avoid the time out which results from making a request to a passive object[1]. If activation takes place, the passive object will be activated in a new capsule on the machine on which the passive object was made passive. In the present prototype, this activation policy cannot be parameterized.

12.4 New interface reference format simulation

As explained in [Olsen 92b], uniform addressing of the interface instances of passive and active objects require a new interface reference format. The prototype simulates a new interface reference format in Testbench3.0, but the application programmer must participate in the simulation. The following sections describe how the new interface reference format is simulated, and how it affects the application programmer. When the prototype has been ported to Testbench4.0, simulation is no longer necessary, so the application programming conventions described in this section can be removed.

12.4.1 Using the new interface reference format

In order to switch to the new interface reference format, the application programmer must specify the following line in clients and servers:

```
#include "NewInterfaceRef.c"
```

This statement should be hidden in the .ah file which is generated by the stub compiler.

12.4.2 Declaring an interface reference

The new interface reference format is simulated by a new C type:

```
NewInterfaceRef
```

The programmer must use NewInterfaceRef in place of ansa_InterfaceRef when declaring interface references.

12.4.3 Interface instance creation

Before any interface instances can be created, the programmer must call

```
initNewInterfaceRef();
```

from the body of an object; this procedure provides an object with a reference to a snapshot locator and a reference to a snapshot base. Both a snapshot base

1. A time out occurs because the capsule in which an active object resides, is terminated when the object is passivated.

and a snapshot locator must be running and their services must be registered in the local trader.

The call to initNewInterfaceRef() should be hidden as part of the initialization of a capsule.

The prepc preprocessor has been modified to generate new code for the Create statement, so the programmer can create interface instances in the usual way by:

```
! USE IfTypeName
! DECLARE {ref}: IfTypeName SERVER
NewInterfaceRef ref;

initNewInterfaceRef();
! {ref} :: IfTypeName$Create(concurrency)
```

12.4.4 Registering an offer in a trader

A server must export an offer by

```
registerNewInterface(&ref, "ifTypeName", "somecontext",
"someprop");
```

A call to this procedure, which must be preceded by creation of the interface instance which is the subject of the offer being registered, replaces the use of the Export statement. The preprocessor should be modified to generate new code for the Export statement.

12.4.5 Importing an offer

A client must import an offer by

```
importNewReference(&ref, "ifTypeName", "somecontext",
"someprop");
```

A call to this procedure replaces the use of the Import statement. The preprocessor should be modified to generate new code for the Import statement.

12.4.6 Operation invocation

The preprocessor has been modified to generate new code for invocations; the result is that the programmer can invoke operations in the usual way by:

```
! USE IfTypeName
! DECLARE {ref}: IfTypeName CLIENT
NewInterfaceRef ref;
Res res;
Arg arg;

importNewReference(&ref, "IfTypeName", "somecontext",
"someprop");
! {res} <- ref$someOp(arg)
```

A factory must be running which can be used for activation in case the object which provides the referred interface instance is passive, otherwise an invocation which gives rise to activation will fail. As a general rule, there should be a factory running on every machine on which parts of a distributed application is to be executing.

12.4.7 Receiving results of type ansa_InterfaceRef

Some existing services in Testbench3.0 return results of type ansa_InterfaceRef. The application programmer is required to convert ansa_InterfaceRef values into NewInterfaceRef values. For example, when an object is instantiated via a factory, the factory returns a sequence of ansa_InterfaceRef as part of the instantiation result. The application programmer must therefore convert each ansa_InterfaceRef into a NewInterfaceRef. A NewInterfaceRef has the following IDL specification:

```
NewInterfaceRef: INTERFACE =
BEGIN
  NewInterfaceRef: TYPE = RECORD
                          [ ir: InterfaceRef,
                            locator: InterfaceRef
                          ];
END.
```

It is the application programmers responsibility to assign the ir field and locator fields when converting an ansa_InterfaceRef into a NewInterfaceRef. The value of the ansa_InterfaceRef should be assigned to the ir field, and a reference to a snapshot locator should be assigned to the locator field. In the case of the factory, the application programmer needs to coordinate what is assigned to the locator field at the client-end with what is assigned to the corresponding locator field at the server-end, i.e. by the server's Create__Object routine.

Each object's passivation and activation infrastructure contains the variable _myLocator which defines that object's snapshot locator.

12.4.8 Manipulating NewInterfaceRef variables

NewInterfaceRef is not an abstract datatype. This means that there are no operations, apart from registerNewInterface, importNewReference and modified prepc statements, which operate on NewInterfaceRef. If the application programmer needs to manipulate a NewInterfaceRef, e.g. copy it, it is necessary to operate on the ir and locator fields explicitly. All internal interfaces supported by Testbench3.0 still operates on ansa_InterfaceRef, in particular, the system independent interfaces which contain operations for manipulating ansa_InterfaceRef are unchanged.

12.4.9 Destroy, Discard, Initiation, Redeem, Import, Export, and Withdraw

Currently the application programmer must use the ir field of a NewInterfaceRef in Destroy, Discard, Initiation, Redeem, Import, Export and Withdraw prepc statements.

12.5 Additional application programming conventions

In addition to the programming conventions which pertain to the simulation of the new interface reference format, the application programmer must also adopt some programming conventions necessary for the proper function of the passivation and activation infrastructure.

12.5.1 Object creation via factories

When instantiating an object via a factory, the instantiated object receives the second argument given to the Instantiate operation as argv[]. The application programmer must ensure that argv[0] is "0" as this informs the Create__Object

operation in the instantiated object that ordinary instantiation should take place. Create__Object has become part of the activation and passivation infrastructure, and it multiplexes upon the value of argv[0] in order to distinguish between ordinary instantiation and activation, see section 5.2. An example of object instantiation, which also illustrates the required conversion of an ansa_InterfaceRef into a NewInterfaceRef is given by:

```
InstantiateResult res;
ObjectId id;
NewInterfaceRef fref, cref;
ansa_InterfaceRef impRef;

importNewReference(&fref, "Factory", "/", "NodeName ==
'audrey'");
! {impRef} <- fref$Instantiate("PhoneBook", "", "")
copyIfRef(&(cref.ir), &impRef);
copyIfRef(&(cref.locator), &(_myLocator) NULL);
! {id, res} <- cref$Instantiate(nullRef, "0", "")
```

Note that cref.locator is actually not used because an object factory cannot move. Similarly, snapshot locators, snapshot bases, and Testbench3.0 services like traders and node managers cannot move, so no locator reference need to be made part of references to these services.

12.5.2 NewCreate__Object

In place of the Create__Object routine, the application programmer must specify the NewCreate__Object routine. The purpose of the NewCreate__Object routine is the same as that of Create__Object. Create__Object is now reserved for use by the activation infrastructure.

Rather than using a capsule's object instantiation operation to also carry out object activation and thus making Create__Object part of the passivation and activation infrastructure, a capsule should have a separate activate operation which calls an Activate__Object routine in the passivation and activation infrastructure. This would remove the application programming requirement that Create__Object is replaced with NewCreate__Object, and that the second argument to object instantiation, argv, has argv[0] = "0".

12.5.3 Concurrency control

To prevent interference between passivation and ongoing activities in an server, the application programmer is required to include a call to getMultipleReadLock as the first action and a call to releaseMultipleLock as the last action in each operation in an object's interfaces. In addition, the application programmer must include the statement hasBeenInvoked++ between getMultipleReadLock and releaseMultipleReadLock in each operation. These conventions are necessary in order to prevent passivation from being carried out while an operation invocation is being carried out.

The lock operations are part of a concurrency infrastructure which the application programmer can use for general application programming purposes. If the application programmer needs to prevent interference between invocations of operations in its interfaces, a getMultipleReadLock call should be replaced by a call to getSingleWriteLock and the corresponding releaseMultipleReadLock call should be replaced by a call to releaseSingleWriteLock.

It is possible to hide the required concurrency control calls in the stubs of a server, but if a server creates separate threads, the application programmer

still has responsibility of preventing passivation from taking place while a thread performs activity; this can be done by making appropriate calls to getMultipleReadLock (getSingleWriteLock) and executing hasBeenInvoked++ for preventing passivation, and releaseMultipleReadLock (releaseSingleWriteLock) for allowing passivation. Note, that these calls only prevent passivation from being carried out while the locks are held.

12.5.4 Forcing activation

To avoid the possible delay imposed by the time out before activation takes place, the application programmer can force activation by calling

```
Activator_activate(_attr, &ref, &newRef);
```

This forces the object which provides the interface instance referred to by ref to be activated, if passive. The up-to-date reference to the interface instance of the active object is returned in newRef. If a client creates a separate thread which makes appropriate calls to Activator_activate while the client performs its main activities simultaneously, the thread can trigger activation by calling Activator_activate before the client's main activities performs invocations. However, the invocations made by the client's main activities may still result in activation being carried out, if the time between activation and invocation exceeds the server's maximum idle time. Therefore, pre-activation must be carefully carried out in order to get a performance gain.

12.6 Building an application

The application programmer needs to follow some conventions for building applications which must have the passivation and activation infrastructure.

The application programmer must use an Imakefile which defines the path of the include files and some files to be linked in, when building the activation and passivation infrastructure for an application; the Imakefile must also define the path to the new preprocessor.

Before an application is executed, a factory must be running on each machine which is to host managed objects of the application, and snapshot locators and snapshot bases must have been started up on selected machines.

12.7 An example application

The example application which is used in [Olsen 92b] to demonstrate how an application programmer can handcraft passivation and activation mechanisms for objects in an application, is reused in Appendix B and C to show what the application programmer needs to specify, in order to get the passivation and activation infrastructure generated. The reader is encouraged to compare the code in Appendix C with the code for handcrafting passivation and activation in Appendix C of [Olsen 92b]. The difference between the original example application and its extension which contains handcrafted passivation and activation mechanisms, is 3 times as many lines of code as in the original. The application in Appendix C for which the passivation and activation infrastructure is automatically generated has 17 lines of code less than the original; this is because the concurrency control module of the original application has been made part of the passivation and activation infrastructure, and therefore it is no longer part of the application code. If it

were, then the application in Appendix C would have 24 lines of code more than the original.

The additional programming complexity which is represented by the 24 lines of additional code will be reduced when the prototype matures and when it has been ported to Testbench4.0.

12.8 Related work

Arjuna [Shrivastava 89] implements persistent objects in terms of passivation and activation mechanisms which are integrated with transaction mechanisms for controlling concurrent use of persistent objects. The application programmer is required to specify how programmer defined object types are mapped between secondary and in-memory representations. The application programmer is also required to use certain classes as super classes for objects which need to be persistent. This inheritance scheme corresponds to our requirement for making an interface compatible with the Passivate library interface type.

The principles for supporting persistent objects in Comandos [Marques 89] are very similar to the principles in the ANSA Storage Model. [Currently, we are not aware to what degree liveness is transparent at the application programming level in Comandos]

Ps-algol [Atkinson 83] pioneered type-orthogonal persistence in terms of a persistent heap, using name servers and database objects which extends the scope of objects to span multiple program executions. In contrast to Arjuna, Comandos, and our work, ps-algol focuses on the language aspect of persistence rather than distribution and heterogeneity aspects. It is fully transparent whether an object is stored or in-memory.

Passivation and activation can be thought of as object-based virtual memory, where objects are transparently swapped in and out as their services are required. Some virtual memory systems, e.g. Chorus [Habert 90], provide support for persistent objects; however these systems restrict heterogeneity as all machines must support a particular virtual memory system.

12.9 Conclusion

It has been shown how an application can be provided with an automatically generated infrastructure which passivate inactive objects and activate passive objects when their services are required. The infrastructure is generated by the preprocessor which has been modified to preprocess a PASSIVATION statement.

The paper motivated passivation and activation as a means for accommodating an increasing number of persistent objects in distributed systems. One of the overheads of passivation and activation is a reduction in availability, which, for some real-time sensitive applications, can be unacceptable. The application programming overhead imposed by passivation and activation is limited to the PASSIVATION statement, and some programming conventions which we expect to vanish as the prototype matures and is ported to Testbench4.0.

Major areas which have not been covered are dependability and garbage collection. The implications of this are that there is no support for fault tolerance, and the programmer must collect garbage.

The passivation and activation infrastructure, and the example applications are available from the author; the code has prototype status and comes with no warranty.

12.10 Acknowledgements

Thanks to Nigel Edwards of Hewlett-Packard Laboratories for commenting on an earlier draft of this paper.

12.11 References

[AIM 91]
"ANSAware 3.0 Implementation Manual", Report No.: RM.097, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Atkinson 83]
M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P.W. Cockshott, and R. Morrison, "An approach to persistent programming", *THE COMPUTER JOURNAL,* Volume 26, No. 4, (1983).

[Copeland 84]
George Copeland, David Maier, "Making Smalltalk a Database System", *SIGMOD'84,* Sigmod Record, Vol. 14, No. 2, (1984).

[Habert 90]
S. Habert, V. Abrossimov and L. Mosseri, "COOL: Kernel Support for Object-Oriented Environments", *OOPSLA'90,* (1990).

[Herbert 91]
Andrew Herbert, "Engineering Model: Conceptual Framework", Report No.: RC.282, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Marques 89]
J. A. Marques and P. Guedes,"Extending the Operating System to Support an Object-Oriented Environment", *OOPSLA'89,* (1989).

[Olsen 91]
Michael Hoffmann Olsen,"A Model for Storage and Resource Management", Report No.: WP-DCG-91-14, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6Qz, England, (1991).

[Olsen 92a]
Michael Hoffmann Olsen, "A Persistent Object Infrastructure for Heterogeneous Distributed Systems", Report No.: RC.343, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92b]

Michael Hoffmann Olsen,"Handcrafting Passivation and Activation", Report No.: RC.321, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Olsen 92c]
Michael Hoffmann Olsen,"Automated Generation of Snapshot Operations for an Object", Report No.: RC.288, ISA project, Poseidon House, Castle Park, Cambridge, England, (1992).

[Shrivastava 89]
S. K. Shrivastava et al, “The Treatment of Persistent Objects in Arjuna”, *THE COMPUTER JOURNAL*, VOL. 32, NO. 4, (1989).

[Warne 91]
John Warne, "ANSA: Assumptions, Principles, and Structure", *Conference proceedings of Software Engineering Environments 1991*, University College of Wales, Aberystwyth, (1991).

13 Appendix: Prepc preprocessor extensions

This appendix describes the extensions which have been made to the prepc preprocessor to simulate the new interface reference format, and to preprocess the new PASSIVATION statement. It is assumed that the preprocessor has been modified to preprocess the SNAPSHOT statement as described in [Olsen 91d].

The file gram.y is extended as follows:

- the grammar for the non-terminal statement has been extended to enable the derivation of:

```
PASSIVATION OF statelist IN TOKEN INTERNALLY TOKEN EXTERNALLY
```

TOKEN

- the routine yylex has been extended so the lines:

```
case PASSIVATION:
case INTERNALLY:
case EXTERNALLY:
```

is added to the switch alternatives for value

- the routine dolex has been extended with the following lines for constructing a return value:

```
if (strcmp(tok, "PASSIVATION") == 0)
   return PASSIVATION;
if (strcmp(tok, "INTERNALLY") == 0)
   return INTERNALLY;
if (strcmp(tok, "EXTERNALLY") == 0)
```

return EXTERNALLY;

- the routine processpassivation has been added. It takes three strings as arguments; the first is the template name which followed IN in a PASSIVATION statement, the second is the idle period argument which followed INTERNALLY in the statement, and the third is the interface name which followed IN in that statement. It first checks that the interface name as been declared as SERVER, otherwise stubc terminates with an error. Then some include statements are generated:

```
#include "mifname.c"
#include "RecordMyInterfaces.c"
```

where ifname is the interface name.

Then a produceSnapshot routine is generated containing a call to a M_ stub routine for each pair of type name and variable name in TypeListHead and VarListHead, and an installSnapshot routine is generated containing a call to an U_ routine for each pair of type name and variable name in TypeListHead and VarListHead.

Include statements for the storage and location files are then generated, followed by the procedures copyIfRef, ifType_passivate, Passivator_startTimer, produceMySnapshot, installMySnapshot, myIfTypes, assignreplacements, and AC_createIf.

The routine outputinvocation has been extended so for a remote invocation the following is generated: A call to Activator_activate followed by calls to the I_ and C_ procedures in the client stub.

A loop which calls Activator_activate followed by call to I_ and C_ stub routines until the C_ call is successful or until the loop has been iterated 10 times; in the latter case the client is terminated.

- The routine outputinstantiation has been modified so a CREATE operation assigns the ir and location fields of a NewInterfaceRef

14 Appendix: The example application code

This appendix contains the code for the example application referred to in the main part of the paper. The application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It exports a reference to a PhoneBookOp interface instance to a trader, which enables a client to insert, remove, lookup and list names and phone numbers. The implementation of the server's operations has been omitted as they are independent of the snapshot mechanisms which are added in appendix C.

14.1 include files

14.1.1 ConcControl.h

```
extern void initConcControl();
extern void cleanUpConcControl();
extern void getSingleWriteLock();
extern void releaseSingleWriteLock();
extern void getMultipleReadLock();
extern void releaseMultipleReadLock();
```

14.1.2 ConcControl.c

```
#include "ansa.h"
#include "opsys.h"
#include "machine.h"
#include "ecs.h"
#include "ConcControl.h"

ansa_EventCount doneReadorWrite, doneNext;
ansa_Sequencer nextReadorWrite, next;

void initConcControl()
{ doneReadorWrite = ecs_makeEventCount((ansa_Cardinal) 0);
  doneNext        = ecs_makeEventCount((ansa_Cardinal) 0);
  nextReadorWrite = ecs_makeSequencer((ansa_Cardinal) 0);
  next            = ecs_makeSequencer((ansa_Cardinal) 0);
}

void cleanUpConcControl()
{ ecs_freeEventCount(doneReadorWrite);
  ecs_freeEventCount(doneNext);
  ecs_freeSequencer(nextReadorWrite);
  ecs_freeSequencer(next);
}

void getSingleWriteLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
```

```
  ecs_await(doneNext, ecs_ticket(next));
}

void releaseSingleWriteLock()
{ ecs_advance(doneReadorWrite);
  ecs_advance(doneNext);
}

void getMultipleReadLock()
{ ecs_await(doneReadorWrite, ecs_ticket(nextReadorWrite));
  ecs_ticket(next);
  ecs_advance(doneReadorWrite);
}

void releaseMultipleReadLock()
{ ecs_advance(doneNext);
}
```

14.2 idl files

14.2.1 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

14.2.2 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

14.3 dpl files

14.3.1 server.dpl

```
! MANAGED
! USE Trader
! USE PhoneBookOp
! DECLARE { phoneBook, pbRef } : PhoneBookOp SERVER
```

```
#include <stdio.h>
#include "ConcControl.h"
#include "capsule.h"
#include "system.h"
#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16
GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

PBEntryList entryList;
ansa_InterfaceRef phoneBook;

void copyIfRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

ansa_InterfaceRef ref[1];

ansa_StatePtr Create__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ Result tres;
  char hn[64], bf[64 + 11];

  initConcControl();
! {phoneBook} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  entryList.length = 0;
  entryList.data   = (PBEntry *)0;
  gethostname(hn, sizeof(hn));
  (void)sprintf(bf, "NodeName '%s'", hn);
! {tres} <- traderRef$Register("PhoneBookOp", "/ansa/PhoneBook",
\
                               bf, phoneBook)
  copyIfRef(&(ref[0]), &phoneBook);
  results->length = 1;
  results->data = ref;
  return (ansa_StatePtr)0;
}

void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
```

```
ansa_InterfaceAttr *_attr;
Name name;
{ getSingleWriteLock();
  /* remove entry identified by name from entryList */;
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

14.3.2 client.dpl

```
! USE Capsule
! DECLARE {cref} : Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook, pbRef} : PhoneBookOp CLIENT

#include   <stdio.h>
#include "capsule.h"
#include "system.h"

#define MAXANSWERLENGTH 31

ansa_InterfaceRef phoneBook;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "create",
  "insert",
  "remove",
  "lookup",
  "list",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {3, 4, 3, 3, 2};
int maxargs[] = {3, 5, 4, 4, 3};
```

```
#define INVALID_CMD -1

PBEntryList eList;

void listEntryList(entryList)
PBEntryList *entryList;
{ int        i;
  char       buffer[128];
  PBEntry  *ifr = entryList->data;

  fprintf(stdout, "\nENTRIES IN PHONEBOOK:\n");
  for (i=0; i<entryList->length; i++, ifr++)
  fprintf(stdout, "  %s %s\n", ifr->name, ifr->no );
}

void PrintUsageAndDie(prog)
char *prog;
{ fprintf(stderr, "usage: %s create machine\n", prog);
  fprintf(stderr, "usage: %s insert name no <property>\n", prog);
  fprintf(stderr, "usage: %s remove name <property>\n", prog);
  fprintf(stderr, "usage: %s lookup name <property>\n", prog);
  fprintf(stderr, "usage: %s list <property>\n", prog);
! capsule$Terminate()
}

void copyIfRef(new, old)
ansa_InterfaceRef *new, *old;
{ if(!system_allocateRef(old->ir_id, old->ir_ah, new))
  instruct_Abort("copyRef", insufficientMemory);
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ ansa_InterfaceRef pbRef;
  int i, cmd_no;
  char property[128];

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
```

```
  PrintUsageAndDie(argv[0]);

  strcpy(property, "");
  switch(cmd_no)
  {  case 1: /* insert */
       { if (argc = 5)
         strcpy(property, argv[4]);
       }
       break;
     case 2: /* remove */
       { if (argc = 4)
         strcpy(property, argv[3]);
       }
       break;
     case 3: /* lookup */
       { if (argc = 4)
         strcpy(property, argv[3]);
       }
       break;
     case 4: /* list */
       { if (argc = 3)
         strcpy(property, argv[2]);
       }
       break;
  }

  if (cmd_no != 0)
  { /* get reference to PhoneBookOp */
!   {pbRef} <- traderRef$Import("PhoneBookOp", "/ansa/
PhoneBook",\
                                property)
    copyIfRef(&phoneBook, &pbRef);
  }

  switch(cmd_no)
  {   case 0: /* create */
        { InstantiateResult res;
          ObjectId id;
          ansa_InterfaceRef fref, cref;
          char machine[128];

          sprintf(machine, "NodeName == '%s'", argv[2]);
!         {fref} <- traderRef$Import("Factory", "/", machine)
!         {cref} <- fref$Instantiate("PhoneBook", "", "")
!         {id, res} <- cref$Instantiate(nullRef, "0", "")
        }
        break;
      case 1: /* insert */
        { PBEntry entry;

          strcpy(entry.name, argv[2]);
          strcpy(entry.no, argv[3]);
!         {} <- phoneBook$insert(entry)
        }
        break;
      case 2: /* remove */
        { Name name;
```

```
          strcpy(name, argv[2]);
!         {} <- phoneBook$remove(name)
        }
        break;
      case 3: /* lookup */
        { PBEntry reply;

!         {reply} <- phoneBook$lookup(argv[2])
          if (strcmp(reply.name, "0") == 0)
         { fprintf(stdout, "Sorry, no entry for %s\n", argv[2]);
          }
          else
          { fprintf(stdout, "No: %s\n", reply.no);
          }
        }
        break;
      case 4: /* list */
        { PBEntryList entryList;

!         {entryList} <- phoneBook$list()
          listEntryList(&entryList);
        }
        break;
  }
}
```

14.4 Imakefile

```
        DEFINES =
       INCLUDES =
       IDLFLAGS =
       DPLFLAGS =
       LINTLIBS = $(LINTANSALIB) -lc
       LOCALLIB = $(ANSALIB)
      LOCALLIBD = $(ANSALIBD)
        INSTALL = bsdinstall.sh

IDLFILES = PhoneBookOp.idl
SIFFILES = PhoneBookOp.sif
DPLFILES = server.dpl client.dpl
SRCS     = ConcControl.c
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)
PROGS = server client

# compile idl files
all:: $(SIFFILES)
# compile dpl files
DPLDepend(server)
DPLDepend(client)

all:: $(PROGS)

# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)
```

```
SingleProgramTarget(server, server.o sPhoneBookOp.o \
                    ConcControl.o,$(LOCALLIB),)
SingleProgramTarget(client, client.o
cPhoneBookOp.o,$(LOCALLIB),)
InstallProgram(server,$(TEMPLATEDIR))

NormalLintTarget(*.c)
DependTarget()
IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

15 Appendix: The extended example application

This appendix contains the code for the example application in appendix B which is modified in order that passivation and activation mechanisms are automatically generated. The client has been extended to enable testing of passivation and activation.

15.1 idl files

15.1.1 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

15.1.2 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
IS COMPATIBLE WITH Passivate;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

15.2 dpl files

15.2.1 server.dpl

```
! MANAGED
! USE Trader
! USE PhoneBookOp
! DECLARE { phoneBook } : PhoneBookOp SERVER

#include <stdio.h>
```

```
#include "ConcControl.h"
#include "capsule.h"
#include "system.h"

#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16

GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

#include "NewInterfaceRef.c"

PBEntryList entryList;
NewInterfaceRef phoneBook;

! PASSIVATION OF {entryList PBEntryList, phoneBook
NewInterfaceRef}\
  IN PhoneBook INTERNALLY 30 EXTERNALLY PhoneBookOp

ansa_StatePtr NewCreate__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ initNewInterfaceRef();
  initConcControl();
! {phoneBook} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  registerNewInterface(&phoneBook, "PhoneBookOp", "/ansa/
PhoneBook",
                       "");
  results->length = 1;
  results->data = (ansa_InterfaceRef *) &(phoneBook.ir);
  return (ansa_StatePtr)0;
}

void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
Name name;
{ getSingleWriteLock();
  /* remove entry identified by name from entryList */;
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_list(_attr, entList)
```

```
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

15.2.2 client.dpl

```
! USE Trader
! USE Capsule
! DECLARE {cref}: Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook}: PhoneBookOp CLIENT

#include <stdio.h>
#include "capsule.h"
#include "system.h"

#define MAXANSWERLENGTH 31

#include "NewInterfaceRef.c"
#include "Activator.c"

NewInterfaceRef phoneBook;
PBEntryList eList;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "create",
  "insert",
  "remove",
  "lookup",
  "list",
  "passivate",
  "activate",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {2, 4, 3, 3, 2, 2, 2};
int maxargs[] = {2, 4, 3, 3, 2, 2, 2};

#define INVALID_CMD -1
```

```
void listEntryList(entryList)
  PBEntryList *entryList;
{ int        i;
  char       buffer[128];
  PBEntry  *ifr = entryList->data;

  for (i=0; i<entryList->length; i++, ifr++)
  fprintf(stdout, "  %s %s\n", ifr->name, ifr->no );
}

void PrintUsageAndDie(prog)
char *prog;
{ fprintf(stderr, "usage: %s create\n", prog);
  fprintf(stderr, "usage: %s insert name no\n", prog);
  fprintf(stderr, "usage: %s remove name\n", prog);
  fprintf(stderr, "usage: %s lookup name\n", prog);
  fprintf(stderr, "usage: %s list\n", prog);
  fprintf(stderr, "usage: %s passivate\n", prog);
  fprintf(stderr, "usage: %s activate\n", prog);
  fprintf(stderr, "usage: %s test times\n", prog);
! capsule$Terminate()
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ int i, cmd_no;

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  initNewInterfaceRef();

  if (cmd_no != 0)
  { /* get reference to PhoneBookOp */
    importNewReference(&phoneBook, "PhoneBookOp",
                       "/ansa/PhoneBook", "");
  }
```

```
  switch(cmd_no)
  { case 0: /* create */
      { InstantiateResult res;
        ObjectId id;
        NewInterfaceRef fref, cref;
        ansa_InterfaceRef impRef;

        importNewReference(&fref, "Factory", "/",
                           "NodeName == 'audrey'");
!       {impRef} <- fref$Instantiate("PhoneBook", "", "")
        copyIfRef(&(cref.ir), &impRef);
        copyIfRef(&(cref.locator), &(_myLocator));
!       {id, res} <- cref$Instantiate(nullRef, "0", "")
      }
      break;
    case 1:   /* insert */
      { PBEntry entry;

        strcpy(entry.name, argv[2]);
        strcpy(entry.no, argv[3]);
!       {} <- phoneBook$insert(entry)
      }
      break;
    case 2: /* remove */
      { Name name;

        strcpy(name, argv[2]);
!       {} <- phoneBook$remove(name)
      }
      break;
    case 3:   /* lookup */
      { PBEntry reply;

!       {reply} <- phoneBook$lookup(argv[2])
        if (strcmp(reply.name, "0") == 0)
        { fprintf(stdout, "Sorry, no entry for %s\n", argv[2]);
        }
        else
        { fprintf(stdout, "No: %s\n", reply.no);
        }
      }
      break;
    case 4: /* list */
      { PBEntryList entryList;

!       {entryList} <- phoneBook$list()
        listEntryList(&entryList);
      }
      break;
    case 5: /* passivate */
      {
!       {} <- phoneBook$passivate()
      }
      break;
    case 6: /* activate */
      { ansa_InterfaceAttr *_attr;
        NewInterfaceRef newRef;
```

```
        Activator_activate(_attr, &phoneBook, &newRef);
      }
      break;
  }
}
```

15.3 Imakefile

```
         STIDIR = ../../../../snapshot/transparency/include
         STLDIR = ../../../../snapshot/transparency/idl
         LTIDIR = ../../../../liveness/transparency/include
         LTSDIR = ../../../../liveness/transparency/stub
         LTLDIR = ../../../../liveness/transparency/idl
          STUBC = ../../../../snapshot/stubc/stubc
          PREPC = ../../../../liveness/prepc/prepc
        DEFINES =
       INCLUDES = -I$(LTIDIR) -I$(STIDIR)
       IDLFLAGS = -I$(LTLDIR) -I$(STLDIR)
       DPLFLAGS =
       LINTLIBS = $(LINTANSALIB) -lc
       LOCALLIB = $(ANSALIB)
      LOCALLIBD = $(ANSALIBD)
        INSTALL = bsdinstall.sh

LTSRCS    = ConcControl.c cSLLocate.c cSBStore.c sPassivator.c \
            cPassivator.c
LTSOBJECTS = ConcControl.o cSLLocate.o cSBStore.o sPassivator.o \
            cPassivator.o
LTCOBJECTS = cSLLocate.o
IDLFILES = PhoneBookOp.idl
SIFFILES = PhoneBookOp.sif
DPLFILES = server.dpl client.dpl
SRCS     =
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)
PROGS = server client

# compile idl files
all:: $(SIFFILES)
      $(CP) $(LTSDIR)/cSLLocate.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/cSBStore.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/sPassivator.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/cPassivator.c $(CURRENT_DIR)
      $(CP) $(LTIDIR)/ConcControl.c $(CURRENT_DIR)
      $(CP) $(LTIDIR)/ConcControl.h $(CURRENT_DIR)

# compile dpl files
DPLDepend(server)
DPLDepend(client)
all:: $(PROGS)
#
# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)

SingleProgramTarget(server, server.o sPhoneBookOp.o \
                    $(LTSOBJECTS),$(LOCALLIB),)
```

```
SingleProgramTarget(client, client.o cPhoneBookOp.o \
                    $(LTCOBJECTS),$(LOCALLIB),)
InstallProgram(server,$(TEMPLATEDIR))
NormalLintTarget(*.c)
DependTarget()
IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

16 Handcrafting Object Migration

16.1 Introduction

This paper is the seventh in a series of nine papers which describe a prototype of the ANSA Storage Model [Olsen 91a] being developed in the ANSA Testbench 3.0 [AIM 91]. The plan for developing the prototype is described in [Olsen 91b].

16.1.1 Terminology

A *persistent object system* is an infrastructure which enables application systems to treat objects which are stored on secondary storage as in-memory objects. It automatically moves objects, on demand, between secondary storage and main memory and thus eliminates the *impedance mismatch* between the type system of application systems and the type system of secondary storage abstractions like databases and file systems [Copeland 84]. Invocations performed on interface instances of stored objects can be extremely slow because stored objects must be moved in-memory before they can provide a requested service; the challenge is therefore to make persistent object systems run fast.

In order to move an in-memory object to secondary storage, a persistent object system must change the object's representation so it complies with the type system of a secondary storage abstraction. To facilitate the migration of objects, the chosen secondary storage representation should comply with the type systems of networks. A non-interpreted sequence of bytes is an example of such a representation.

A sequence of bytes which represents a single object is termed a *snapshot* of that particular object. The snapshots considered in this paper consists of three parts: (i) an object type component which denotes the possible transformations an object's state can undergo, (ii) a state component which denotes the current state of an object in terms of name to value bindings and references to interface instances, and (iii) a schema component which denotes the set of interface types and instances currently supported by an object. A snapshot of an object is different from a *checkpoint*; whereas a snapshot represents a single object, a checkpoint is a set of mutually consistent snapshots. A checkpoint of an object is recursively defined to consist of a snapshot of that object and a checkpoint of each object which has an interface instance referred to by a reference in the initial object. If an object has no references to interface instances in other objects, a snapshot of that object is also a checkpoint.

A service which enables snapshots to be stored on and retrieved from secondary storage is termed a *SnapshotBase*; a service which enables objects to obtain replacements for references to interface instances in objects which have moved to secondary storage is termed a *SnapshotLocator*.

Passivation is the process of making a snapshot of an object, storing it in a SnapshotBase, inserting a reference to the snapshot in a SnapshotLocator as

the replacement for references to the object's interface instances, and terminating the object. An object which has undergone passivation has been *passivated* and is said to be a *passive* object. A passive object must undergo activation before its services are available. *Activation* is the process of instantiating an object, retrieving a passive object's snapshot from a SnapshotBase, installing the snapshot, creating interface instances in the instantiated object, and inserting references to these interface instances in a SnapshotLocator as replacements for references to the passive object's interface instances. A passive object which has undergone activation has been *activated* and is said to be an *active* object.

Each object controls its own passivation, but it may allow other objects to request it to passivate. An object cannot passivate until it has completed all activities except the one associated with passivation. If an object possesses a reference to an interface instance of a passive object and uses the reference for invoking an operation, an invocation failure occurs; this kind of failures must be handled by activating the passive object, rebinding the reference to an interface instance in the active object, and retrying the invocation.

Compared to an active object, a passive object has a higher degree of stability by virtue of residing on secondary storage, but its availability is reduced by the time it takes to activate it. Therefore, passivation and activation may not be feasible in applications which require a high degree of availability.

Migration is the process of moving an active object from one location in a distributed system to another. An object which has undergone migration has been *migrated*. If an object which is being requested to migrate is passive, the migration request fails. This kind of failures must be handled by activating the passive object on the new location. If an object which is being requested to migrate is active, the object carries out the migration itself by acting both as an activator for a clone of itself and as a SnapshotBase from which the clone can fetch a snapshot to install. During migration, the services of an object are temporarily unavailable until the clone has fetched the snapshot and registered its interfaces with a SnapshotLocator. It is possible to narrow the window of unavailability by fetching the snapshot in parts, if it is known that service provision only alters a small part of an object's state while migration is carried out.

16.1.2 Problem statement

There are many reasons why object migration is considered a useful feature in distributed systems:

- dynamic re-configuration of distributed systems require the ability to move objects between nodes in order to preserve service availability while nodes are unavailable
- the load on individual nodes can be balanced against the average load by migrating objects from heavily loaded nodes to lightly loaded nodes
- network traffic caused by remote invocations can be reduced by migrating server to client and vice versa
- fault tolerance can be increased by migrating objects from less reliable nodes to more reliable nodes
- local garbage collection strategies can be exploited by migrating objects from nodes where their interface instances are not referred, to nodes where their interface instances are referred

Each of these areas require more or less the same base mechanism for carrying out the migration, however, the application of policies for deciding what, when and where to migrate in each of the areas are not well understood. In general, the pattern of service usage across the whole of a distributed system needs to be taken into consideration in order to make optimal decisions about each object's location and to avoid that conflicting distributed views cause objects to oscillate between locations.

16.1.3 Purpose

The purpose of this paper is to describe how an application programmer can handcraft extensions to an application so a client can request an object to migrate to a new location. The findings are to serve as the basis for automatically generating the infrastructural support for being able to migrate an object. The paper does not cover migration policies, nor the provision of an infrastructure which migrates objects on it's own initiative transparently to applications. A separate paper, see [Olsen 91b], covers the subject of generating infrastructural support for object migration.

The goal for prototyping migration is to develop a test-bed which enables experimentation with migration policies layered on the basic migration mechanism.

16.1.4 Main ideas

An object has a migrate operation in one of its interfaces which allows a client to request the object to migrate. The only available migration policy (at the moment) is to accept the request by waiting for all other activities to terminate and prevent other activities from being started. If the object which is being requested to migrate is passive, then migration is simply carried out by activating the passive object on the new location. If the object is active, it creates a SnapshotBase interface instance which allows its snapshot to be fetched, and then it instantiates a new object which is passed a reference to this SnapshotBase interface instance. The instantiated object fetches the snapshot from the object using the SnapshotBase reference (exactly as for activation) and when it has set up its interface instances, it registers them in a locator. After instantiation has completed, the instantiated object will begin to receive invocation requests from clients which have updated their interface references to the interface instances of the original object. The original object can now terminate itself. Any client which has an invocation request blocked in the original object will experience loss of connection when the object terminates. They handle this by obtaining an up-to-date reference to the migrated object from the locator.

16.1.5 Limitations

Migration should be dependable and be performed in a secure manner. However, these issues are not addressed in this paper.

Though there may be many areas where migration is useful, this paper focuses on migration as an enabling technology principally used by system managers for relatively infrequent dynamic re-configurations of distributed systems to ensure service availability when nodes become unavailable. It is assumed that system managers have information about which nodes are available in a network to a degree which is sufficient for requesting objects to migrate from one node to another.

Rather than assuming a system model where machines in a network share a logical address space (distributed virtual memory) we assume a system model where separate machines have their own distinct address spaces between which object can be moved. For simplicity, the paper only considers single object address spaces, i.e. capsules, and if not otherwise made explicit, the term object is synonymous with a capsule; we do not expect problems in extending our work to cover multiple objects in capsules or further levels of nesting.

16.1.6 Audience and prerequisites

The paper targets system developers and programmers who are designing and implementing persistent object systems. It also addresses application programmers who choose to take control over migration.

The reader is assumed to be familiar with ANSA [Warne 91], The ANSA Testbench3.0 [AIM 91], and The ANSA Storage Model [Olsen 91a]. It may be useful for the reader to be acquainted with the preceding six papers on the storage prototype: [Olsen 91c] and [Olsen 91d] which describe how to produce and install snapshots in objects, [Olsen 91e] which describes the implementation of a SnapshotBase, [Olsen 91f] which describes the implementation of a SnapshotLocator, and [Olsen 92a] and [Olsen 92b] which describe how to passivate and activate objects.

16.1.7 Organisation

An example application is briefly summarized in section 2, and section 3 describes how it is extended with a migration mechanism. Section 4 discusses the programming overhead imposed on application programs by migration. Some related work is presented in section 5, and section 6 concludes the paper. Appendix A contains the code for the example application, and appendix B contains the code for the extended application.

16.2 An example application

The example application which was used in [Olsen 91c], [Olsen 91d], [Olsen 92a] and [Olsen 92b] is re-used in this paper to demonstrate how an application programmer can extend an application with a migration mechanism. The example application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It registers a reference to an interface instance in a trader; this interface enables a client to insert, remove, lookup and list names and phone numbers.

16.3 Adding migration

The server must be extended so it can receive migration requests from the client and migrate itself accordingly. The client must be extended so it can request the server to migrate. These extensions comprise:

- extend the server so it has an interface with a migrate operation
- extend the server so it can create a SnapshotBase interface instance
- extend the server so it can act as an activator which instantiates a new server on a remote node and passes it a reference to the SnapshotBase

interface instance; the new server's activation infrastructure uses the SnapshotBase reference as though the server is being activated

- enable the client to invoke the servers migrate operation if the server is active, or, if the server is passive, request an activator to activate the server on the location where the client wants it to be located

16.3.1 External migration requests

To enable migration request, the server must have an interface instance which is compatible with the following interface type:

```
Migrate: INTERFACE =
BEGIN
  migrate: OPERATION [newLocation: STRING]
           RETURNS [];
END.
```

16.3.1.1 Which interface?

In the example application, the migrate operation is made available to the client by making the server's interface compatible with the Migrate interface. In general, the Migrate interface should be a separate management interface.

16.3.2 Implementing migration

Migration can be achieved by passivating an object and then activating it on the required location. This method carries the overhead of first storing a snapshot in a SnapshotBase, which is then fetched by an instantiated object. If an object is indeed passive when it is being requested to migrate, migration should be carried out by activating it on the requested location, but if the object is active then it should carry out the migration by instantiating a new object which fetches the snapshot directly from the original object via a SnapshotBase interface instance.

Migration requires that a factory is running on the node to which an object will be requested to migrate.

16.3.2.1 Issuing the migrate request

The way migration is carried out depends on whether the target object is passive or active. The abstract migration request

```
!   {} <- ref$migrate(nodeName)
```

issued by the client is therefore implemented as follows

```
ansa_Status _stat = ok;
int _retries = 0;
NewInterfaceRef _newRef;
ansa_InterfaceAttr *_attr;
ansa_Boolean activated;

Activator_migrate(_attr, &ref, &(_newRef), nodeName,
&activated);
copyIfRef(&(ref.ir), &(_newRef.ir));
copyIfRef(&(ref.locator), &(_newRef.locator));
if (activated == FALSE)
{ /* phoneBook is active and needs to be migrated */
  I_ifType_migrate(&(ref.ir), (ansa_Voucher *)0,
                   CALLtype, (ansa_Dispatch *)0, nodeName);
  _stat = C_ifType_migrate(&(ref.ir), (ansa_Voucher *)0);
```

```
  while (_stat != ok)
  { /* ref may now be out-of-date */
    Activator_migrate(_attr, &ref, &(_newRef), nodeName,
&activated);
    copyIfRef(&(ref.ir), &(_newRef.ir));
    copyIfRef(&(ref.locator), &(_newRef.locator));
    if (activated == TRUE)
    { _stat == ok;
    }
    else
    { I_ifType_migrate(&(ref.ir), (ansa_Voucher *)0,
                       CALLtype, (ansa_Dispatch *)0, nodeName);
      _stat = C_ifType_migrate(&(ref.ir), (ansa_Voucher *)0);
      if (_stat != ok)
      { _retries++;
        if (_retries == 10)
        { binder_terminate();
          instruct_Abort("ifType.migrate", _stat);
        }
      }
    }
  }
}
```

The Activator_migrate call (see section 3.2.2) migrates a passive object by activating it on the new location. If the object is active, then the object's own migrate operation must be invoked. If this invocation does not succeed then it is assumed the reference used for the invocation is out-of-date, so Activator_migrate is called again followed, if necessary, by another migrate invocation until Activator_migrate or a migrate invocation has resulted in the required migration or until a 10th migration attempt has failed.

16.3.2.2 Migrating a passive object

If the object which is being requested to migrate is passive, migration is simply performed by activating the passive object on the new location. To carry out migration when an object is passive, the activator, see [Olsen 92a], is extended with a migrate operation:

```
Activator: INTERFACE =
NEEDS NewInterfaceRef
BEGIN
  activate: OPERATION [old: NewInterfaceRef]
            RETURNS [new: NewInterfaceRef];
  migrate:  OPERATION [old: NewInterfaceRef, newLocation: STRING]
            RETURNS [new: NewInterfaceRef, activated: BOOLEAN];
END.
```

An invocation of migrate takes an old reference and the location on which to activate the server; it returns a replacement reference and a boolean denoting whether activation was carried out or not. If the old reference refers to an interface instance in a passive object, activation takes place, and the returned boolean has the value TRUE; the returned reference then refers to the active object's interface instance[1] which corresponds to the interface instance in the passive object referred to by the old reference. If no activation took place (the returned boolean has the value FALSE), then the returned interface reference is

1. In general, there is no guarantee that the object will be active when the new reference is received by the invoker of activate, as it may have passivated.

identical to the argument interface reference, and the active object's own migrate operation must be invoked in order to migrate the object. The activator's migrate operation is implemented as follows:

```
void Activator_migrate(_attr, oldRef, newRef, newLocation,
activated)
ansa_InterfaceAttr *_attr;
NewInterfaceRef *oldRef, *newRef;
char newLocation[64];
ansa_Boolean *activated;
{ Entry ent;
  InstantiateResult res;
  char nodeName[80], activateArgs[1024];

  lookUp(oldRef->locator, oldRef->ir, &ent);
  if (ent.sid > 0) /* object is passive */
  { makeActivateArgs(ent.sid, &(ent.new), &(ent.last),
                     &(oldRef->locator), activateArgs);
    sprintf(nodeName, "NodeName == '%s'", newLocation);
    instantiateObject(nodeName, ent.objectType, activateArgs,
&res);
    copyIfRef(&(newRef->ir), res.data);
    *activated = TRUE;
  }
  if (ent.sid == 0) /* object is active */
  { copyIfRef(&(newRef->ir), &(ent.new));
    *activated = FALSE;
  }
  if (ent.sid == -1) /* no entry in locator */
  { copyIfRef(&(newRef->ir), &(oldRef->ir));
    *activated = FALSE;
  }
  copyIfRef(&(newRef->locator), &(oldRef->locator));
}
```

The lookup operation is invoked to get the latest location information for oldRef. If oldRef refers to an interface instance in a passive object then makeActivateArgs is called to construct a string containing the information necessary for activation, and instantiateObject is then called to instantiate an object which establishes itself as the active object on the location specified in the newLocation argument to Activator_migrate. If oldRef refers to an interface instance in an object which is currently active, then Activator_migrate merely returns the up-to-date reference of this object. If there is no entry in the locator for oldRef, then oldRef must be up-to-date and it must refer to an active object; it is therefore returned as newRef.

16.3.2.3 Migrating an active object

An active object is migrated by invoking its migrate operation. migrate creates a SnapshotBase interface instance which allows the object's snapshot to be fetched, and then it instantiates a new object which is passed a reference to this SnapshotBase interface instance. The instantiated object fetches the snapshot from the object using the SnapshotBase reference (exactly as for activation) and when it has set up its interface instances, it registers them in a locator. After instantiation has completed, the instantiated object will begin to receive invocation requests from clients which have updated their interface references to the interface instances of the original object. The original object can now terminate itself. Any client which has an invocation request blocked in this

object will experience a loss of connection when the object terminates. Clients handle this by obtaining an up-to-date reference to the migrated object from a locator.

The migrate operation for the example application's server is implemented as follows:

```
int PhoneBookOp_migrate(_attr, newLocation)
ansa_InterfaceAttr *_attr;
ansa_String newLocation;
{ char nodeName[64], hn[64];
  NewInterfaceRef storeRef;

  getSingleWriteLock();
  hasBeenInvoked++;
  gethostname(hn, sizeof(hn));
  strcpy(nodeName, newLocation);
  if (strcmp(nodeName, hn) != 0)
  {
!   {storeRef} :: SBStore$Create(1)
    makeActivateArgs(-1, &(_myStore), &(_myLocator),
                     &(storeRef->ir), activateArgs);
    instantiateObject(nodeName, "PhoneBookOp", activateArgs,
&res);
    binder_terminate();
    instruct_Abort("Binder.CreateIfRef", _stat);
  }
  releaseSingleWriteLock();
  return 1;
}
```

The getSingleWriteLock and hasBeenInvoked statements results in all other incoming invocations being blocked and prevents the object from passivating itself. If the object is already located at the location to which it is being requested to migrate then the migrate request is ignored. Otherwise, a SnapshotBase interface instance is created and a reference to it is passed to makeActivateArgs which constructs a string containing the information necessary for instantiating an object via instantiateObject. instantiateObject carries out activation by obtaining a snapshot via the SnapshotBase reference. When instantiateObject returns, the object terminates itself.

16.4 Programming overhead

The implementation of migration comprises less than 200 lines of code by virtue of being based on the functionality of the activation and passivation infrastructure. By generating the code which implements migration automatically, i.e. similar to the way the passivation and activation infrastructure is generated (see [Olsen 92b]), the programming overhead imposed on application programming by migration can be reduced. A separate paper describes how to generate the migration code automatically.

16.5 Related work

Arjuna [Shrivastava] implements persistent objects in terms of passivation and activation mechanisms which are integrated with transaction mechanisms for controlling concurrent use of persistent objects. Object

migration is carried out implicitly due to the fact that passive server objects are co-located with clients when activated.

COOL [Habert 90], an object-oriented layer on top of Chorus, enables objects to be migrated as part of invoking an object. Object migration is piggy backed on message transmission. Several objects can migrate along with a message, either on request or on reply message transmissions. The objects that should be migrated must be listed in an argument to an invocation. Although coupled with remote invocation, object migration is based on the distributed virtual memory management in Chorus. This corresponds to performing migration by passivation followed by activation. COOL has not yet been adapted for a heterogeneous environment.

In Comandos [Marques 89] object migration is carried out implicitly by the virtual object memory system which maps an object into an invoking client's context on a node. If an invoked object is active on a remote node, the virtual object memory system first unmaps the object from its context on its current node and then maps it into the invoking client's context on another node. This corresponds to performing migration by passivation followed by activation. However, the virtual object memory system can also decide to migrate an invoking object to an invoked object by transferring the invoker via RPC, similar to the way active objects are migrated in our prototype.

16.6 Conclusion

It has been shown how an application can be extended with mechanisms for migrating an object to another node. It is the responsibility of the invoker of an object's migrate operation to ensure that a factory is running on the remote node, otherwise migration cannot be carried out. The paper motivated migration as an enabling technology principally used by system managers for relatively infrequent dynamic re-configurations of distributed systems to ensure service availability when nodes become unavailable. If a node is to become unavailable, a system administrator should first terminate any factories running on the node to prevent new objects from being instantiated or activated on the node, and then invoke the migrate operation on all objects which reside on the node. When all objects have migrated, the node can be shut down. The paper has not addressed the issues of how a system administrator detects which objects (both active and passive) reside on a given node, nor how to choose which node to migrate each object to. A separate paper describes how to generate the migration code automatically.

16.7 References

[AIM 91]
"ANSAware 3.0 Implementation Manual", Report No.: RM.097, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Copeland 84]
George Copeland, David Maier, "Making Smalltalk a Database System", *SIGMOD'84*, Sigmod Record, Vol. 14, No. 2, (1984).

[Habert 90]

S. Habert, V. Abrossimov and L. Mosseri, "COOL: Kernel Support for Object-Oriented Environments", *OOPSLA'90*, (1990).

[Herbert 91]
Andrew Herbert, "Engineering Model: Conceptual Framework", Report No.: RC.282, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Marques 89]
J. A. Marques and P. Guedes,"Extending the Operating System to Support an Object-Oriented Environment", *OOPSLA'89*, (1989).

[Olsen 91a]
Michael Hoffmann Olsen,"A Model for Storage and Resource Management", Report No.: WP-DCG-91-14, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6Qz, England, (1991).

[Olsen 91b]
Michael Hoffmann Olsen,"The storage activity: objectives, tasks, timescales, and status", Report No.: RC.281, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91c]
Michael Hoffmann Olsen,"Handcrafting snapshot operations for an object", Report No.: RC.280, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91d]
Michael Hoffmann Olsen,"Automated generation of snapshot operations for an object", Report No.: RC.288, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91e]
Michael Hoffmann Olsen,"Design and implementation of a SnapshotBase", Report No.: RC.301, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91f]
Michael Hoffmann Olsen,"Design and implementation of a SnapshotLocator", Report No.: RC.307, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 92a]
Michael Hoffmann Olsen,"Handcrafting passivation and activation", Report No.: RC.321, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 92b]
Michael Hoffmann Olsen,"Automated generation of passivation and activation", Report No.: RC.326, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Shrivastava 89]
S. K. Shrivastava et al, "The Treatment of Persistent Objects in Arjuna", *THE COMPUTER JOURNAL*, VOL. 32, NO. 4, (1989).

[Warne 91]
John Warne, "ANSA: Assumptions, Principles, and Structure", *Conference proceedings of Software Engineering Environments 1991*, University College of Wales, Aberystwyth, (1991).

17 Appendix: The example application code

This appendix contains the code for the example application referred to in the main part of the paper. The application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It exports a reference to a PhoneBookOp interface instance to a trader, which enables a client to insert, remove, lookup and list names and phone numbers. The application uses the passivation and activation infrastructure.

The implementation of the server's operations have been omitted as they are independent of the migration mechanism which is added in appendix B; the implementations can be found in [Olsen 92b].

17.1 idl files

17.1.1 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

17.1.2 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
IS COMPATIBLE WITH Passivate;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

17.2 dpl files

17.2.1 server.dpl

```
! MANAGED
! USE PhoneBookOp
! DECLARE { phoneBook } : PhoneBookOp SERVER

#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16
GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

#include "NewInterfaceRef.c"

PBEntryList entryList;
NewInterfaceRef phoneBook;

! PASSIVATION OF {entryList PBEntryList, phoneBook
NewInterfaceRef}\
  IN PhoneBook INTERNALLY 30 EXTERNALLY PhoneBookOp

ansa_InterfaceRef ref[2];

ansa_StatePtr NewCreate__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ initNewInterfaceRef();
  initConcControl();
! {phoneBook} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  registerNewInterface(&phoneBook, "PhoneBookOp", "/ansa/
PhoneBook",
                       "");
  copyIfRef(&(ref[0]), &(phoneBook.ir));
  copyIfRef(&(ref[1]), &(phoneBook.locator));
  results->length = 2;
  results->data = ref;
  return (ansa_StatePtr)0;
}

void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  hasBeenInvoked++;
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
```

```
Name name;
{ getSingleWriteLock();
  hasBeenInvoked++;
  /* remove entry identified by name from entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  hasBeenInvoked++;
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  hasBeenInvoked++;
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

17.2.2 client.dpl

```
! USE Trader
! USE Capsule
! DECLARE {cref}: Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook}: PhoneBookOp CLIENT

#include <stdio.h>
#include "capsule.h"
#include "system.h"

#include "NewInterfaceRef.c"
#include "Activator.c"

NewInterfaceRef phoneBook;
PBEntryList eList;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "insert",
  "remove",
  "lookup",
  "list",
  "passivate",
  "activate",
  "create",
```

```
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {4, 3, 3, 2, 2, 2, 2};
int maxargs[] = {4, 3, 3, 2, 2, 2, 2};

#define INVALID_CMD -1

void listEntryList(entryList)
  PBEntryList *entryList;
{ /* print contents of entryList on stdout */
}

void PrintUsageAndDie(prog)
char *prog;
{ /* print error message and terminate */
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ int i, cmd_no;

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  initNewInterfaceRef();
  if (cmd_no != 6)
  importNewReference(&phoneBook, "PhoneBookOp", "", "");
  switch(cmd_no)
  { case 0:   /* insert */
      { PBEntry entry;

        strcpy(entry.name, argv[2]);
        strcpy(entry.no, argv[3]);
!       {} <- phoneBook$insert(entry)
      }
      break;
```

```
            case 1: /* remove */
              { Name name;

                strcpy(name, argv[2]);
!               {} <- phoneBook$remove(name)
              }
              break;
            case 2:   /* lookup */
              { PBEntry reply;

!               {reply} <- phoneBook$lookup(argv[2])
                if (strcmp(reply.name, "0") == 0)
                { fprintf(stdout, "No entry for %s\n", argv[2]);
                }
```

18 Automated Generation of a Migration Infrastructure

18.1 Introduction

This paper is the eighth in a series of nine papers which describe a prototype of the ANSA Storage Model [Olsen 91a] being developed in the ANSA Testbench 3.0 [AIM 91]. The plan for developing the prototype is described in [Olsen 91b].

18.1.1 Terminology and background

A *persistent object system* is an infrastructure which enables application systems to treat objects which are stored on secondary storage as in-memory objects. It automatically moves objects, on demand, between secondary storage and main memory and thus eliminates the *impedance mismatch* between the type system of application systems and the type system of secondary storage abstractions like databases and file systems [Copeland 84]. Invocations performed on interface instances of stored objects can be extremely slow because stored objects must be moved in-memory before they can provide a requested service; the challenge is therefore to make persistent object systems run fast.

In order to move an in-memory object to secondary storage, a persistent object system must change the object's representation so it complies with the type system of a secondary storage abstraction. To facilitate the migration of objects, the chosen secondary storage representation should comply with the type systems of networks. A non-interpreted sequence of bytes is an example of such a representation.

A sequence of bytes which represents a single object is termed a *snapshot* of that particular object. The snapshots considered in this paper consists of three parts: (i) an object type component which denotes the possible transformations an object's state can undergo, (ii) a state component which denotes the current state of an object in terms of name to value bindings and references to interface instances, and (iii) a schema component which denotes the set of interface types and instances currently supported by an object. A snapshot of an object is different from a *checkpoint*; whereas a snapshot represents a single object, a checkpoint is a set of mutually consistent snapshots. A checkpoint of an object is recursively defined to consist of a snapshot of that object and a checkpoint of each object which has an interface instance referred to by a reference in the initial object. If an object has no references to interface instances in other objects, a snapshot of that object is also a checkpoint.

A service which enables snapshots to be stored on and retrieved from secondary storage is termed a *SnapshotBase*; a service which enables objects to obtain replacements for references to interface instances in objects which have moved to secondary storage is termed a *SnapshotLocator*.

Passivation is the process of making a snapshot of an object, storing it in a SnapshotBase, inserting a reference to the snapshot in a SnapshotLocator as the replacement for references to the object's interface instances, and terminating the object. An object which has undergone passivation has been *passivated* and is said to be a *passive* object. A passive object must undergo activation before its services are available. *Activation* is the process of instantiating an object, retrieving a passive object's snapshot from a SnapshotBase, installing the snapshot, creating interface instances in the instantiated object, and inserting references to these interface instances in a SnapshotLocator as replacements for references to the passive object's interface instances. A passive object which has undergone activation has been *activated* and is said to be an *active* object.

Each object controls its own passivation, but it may allow other objects to request it to passivate. An object cannot passivate until it has completed all activities except the one associated with passivation. If an object possesses a reference to an interface instance of a passive object and uses the reference for invoking an operation, an invocation failure occurs; this kind of failures must be handled by activating the passive object, rebinding the reference to an interface instance in the active object, and retrying the invocation.

Compared to an active object, a passive object has a higher degree of stability by virtue of residing on secondary storage, but its availability is reduced by the time it takes to activate it. Therefore, passivation and activation may not be feasible in applications which require a high degree of availability.

Migration is the process of moving an active object from one location in a distributed system to another. An object which has undergone migration has been *migrated*. If an object which is being requested to migrate is passive, the migration request fails. This kind of failures must be handled by activating the passive object on the new location. If an object which is being requested to migrate is active, the object carries out the migration itself by acting both as an activator for a clone of itself and as a SnapshotBase from which the clone can fetch a snapshot to install.

During migration, the services of an object are temporarily unavailable until the clone has fetched the snapshot and registered its interfaces with a SnapshotLocator. It is possible to narrow the window of unavailability by fetching the snapshot in parts, if it is known that service provision only alters a small part of an object's state while migration is carried out.

18.1.2 Problem statement

There are many reasons why object migration is considered a useful feature in distributed systems:

- dynamic re-configuration of distributed systems require the ability to move objects between nodes in order to preserve service availability while nodes are unavailable;
- the load on individual nodes can be balanced against the average load by migrating objects from heavily loaded nodes to lightly loaded nodes
- network traffic caused by remote invocations can be reduced by migrating server to client and vice versa;
- fault tolerance can be increased by migrating objects from less reliable nodes to more reliable one;

- local garbage collection strategies can be exploited by migrating objects from nodes where their interface instances are not referred, to nodes where their interface instances are referred.

Each of these areas require more or less the same base mechanism for carrying out the migration, however, the application of policies for deciding what, when and where to migrate in each of the areas are not well understood. In general, the pattern of service usage across the whole of a distributed system needs to be taken into consideration in order to make optimal decisions about each object's location and to avoid that conflicting distributed views cause objects to oscillate between locations.

Object migration impose a programming overhead on application programming. In [Olsen 92c] it was show that the overhead is about 200 lines of additional code. The programming overhead can be reduced by generating the migration code automatically as a migration infrastructure in each migratable object. The automatic provision of a migration infrastructure is known as *migration transparency*.

18.1.3 Purpose

The purpose of this paper is to describe how a migration infrastructure can be generated automatically, thus reducing the programming overhead from applications which migrate objects. The paper does not describe how the migration infrastructure is implemented and it does not cover migration policies, nor does it cover the provision of an infrastructure which migrates objects on it's own initiative transparently to applications.

The goal for prototyping migration is to provide a test-bed which enables experimentation with migration policies layered on the basic migration mechanism.

18.1.4 Main ideas

The handcrafted migration mechanisms which are described in [Olsen 92b] are used as the basis for the migration infrastructure. The application independent migration code is separated out into include files, and the prepc preprocessor is extended in order to include these files and to generate the application dependent parts of the infrastructure. Application specific details which are required by the preprocessor must be specified by the application programmer in a prepc statement in objects which should be migratable. One application specific detail is which interface a migrate operation should be available in. Other application specific details which are necessary for migrating an object are inherited from a PASSIVATION statement, see [Olsen 92b], which must precede a prepc statement required for migration. The preprocessor generates the implementation of an object's migrate operation. The only available migration policy (at the moment) is to accept a migration request by waiting for all other activities to terminate and prevent other activities from being started. If the object which is being requested to migrate is passive, then it is activated on the location to which it is being requested to migrate. If the object is active, it will change its current location to the requested location. Clients will have their invocations blocked while the object they have invoked is being migrated; when migration has completed, the clients infrastructure will obtain an up-to-date reference before retrying the invocation.

18.1.5 Limitations

Migration should be dependable and be performed in a secure manner. However, these issues are currently not addressed by the migration infrastructure.

Though there may be many areas where migration is useful, the paper focuses on migration as an enabling technology principally used by system managers for relatively infrequent dynamic re-configurations of distributed systems to ensure service availability when nodes become unavailable. It is assumed that system managers have information about which nodes are available in a network to a degree which is sufficient for requesting objects to migrate from one node to another.

Rather than assuming a system model where machines in a network share a logical address space (distributed virtual memory) we assume a system model where separate machines have their own distinct address spaces between which object can be moved. For simplicity, the paper only considers single object address spaces, i.e. capsules, and if not otherwise made explicit, the term object is synonymous with a capsule; we do not expect problems in extending our work to cover multiple objects in capsules or further levels of nesting.

18.1.6 Audience and prerequisites

The paper targets application programmers who need support for object migration, and it also addresses system developers who are designing and implementing persistent object systems.

The reader is assumed to be familiar with ANSA [Warne 91], The ANSA Testbench3.0 [AIM 91], and The ANSA Storage Model [Olsen 91a]. It may be useful for the reader to be acquainted with the preceding six papers on the storage prototype: [Olsen 91c] and [Olsen 91d] which describe how to produce and install snapshots in objects, [Olsen 91e] which describes the implementation of a SnapshotBase, [Olsen 91f] which describes the implementation of a SnapshotLocator, [Olsen 92a] and [Olsen 92b] which describe how to passivate and activate objects, and [Olsen 92c] which describes how an application programmer can extend an application with a migration mechanism.

18.1.7 Organisation

Section 2 positions the migration infrastructure has a layer on top of the passivation and activation infrastructure, and it summarizes the programming conventions imposed by the passivation and activation infrastructure. A new prepc statement for declaring that an object is migratable is presented in section 3. Section 4 describes how an object can migrate another object, and section 5 describes some conventions for building applications. Section 6 describes an example application for which the migration infrastructure is generated. Some example prepc language constructs for facilitating location transparent migration, i.e. migrating relative to objects rather than absolute locations, are discussed in section 7. Some related work is presented in section 8, and section 9 concludes the paper. Extensions made to the prepc preprocessor are detailed in Appendix A, and Appendix B and C contain a complete example application for which the migration infrastructure is generated.

18.2 Extending the passivation and activation infrastructure

The migration infrastructure was handcrafted in [Olsen 92c]. To generate the infrastructure automatically, the generic parts are separated out into library functions, and the application specific parts are generated from a new prepc statement which is described in section 3. When an application has been built, each object which can migrate has a migration infrastructure which is layered on the snapshot infrastructure and the passivation and activation infrastructure as shown in figure 1.

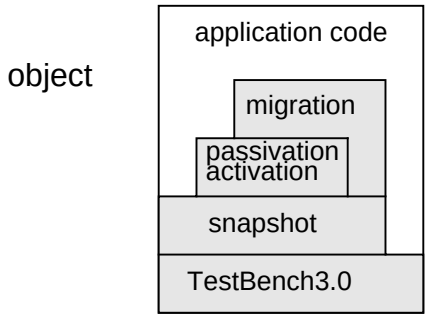

Figure 1.: *The application code for an object and the layering of its infrastructure (shaded).*

The passivation and activation infrastructure introduced some new programming conventions for building applications, see [Olsen 92b]. In summary the conventions are as follows:

- a new interface reference format must be simulated by declaring interface references as being of type NewInterfaceRef which is defined in the include file NewInterfaceRef.c;
- the procedure initNewInterfaceRef() must be called before any interface instances can be manipulated;
- service offers must be registered in the local trader by calling registerNewInterface;
- service offers must be imported by calling importNewReference;
- a NewInterfaceRef contains an ir and a locator field which must be manipulated explicitly when receiving results from existing TestBench3.0 services;
- the ir field of a NewInterfaceRef must be used in Destroy, Discard, Initiation, Redeem, Import, Export and Withdraw prepc statements.

18.3 Declaring a migratable object

To enable an object to migrate, the application programmer must declare that the object is migratable by the following MIGRATE statement in prepc

```
! MIGRATE USING ifType
```

From this statement, the prepc preprocessor generates the application dependent parts of the migration infrastructure. ifType declares which type of interface a “migrate” operation can be made available in. Appendix A describes the extensions to the prepc preprocessor which have been made in order to enable it to preprocess MIGRATE statements.

The MIGRATE statement must appear textually after a PASSIVATION statement, see [Olsen 92b], and all application state which persist across passivation and activation will also persist across migration, i.e.

```
! PASSIVATION OF {foo bar} IN pip INTERNALLY 42 EXTERNALLY pap
/* more declarations can appear here */
! MIGRATE USING pop
```

Here the variable foo of type bar makes up the application state which will persist across passivation and activation. foo will also be preserved when the object is migrated. Note that the passivate operation can be made available in an interface instance of type pap whereas a migrate operation can be made available in an interface instance of type pop. pop (or pap) could have been used in both statements. The object type pip which is specified in the PASSIVATION statement is used by the migration infrastructure for naming the executables for the object on the different machines; all executables for an object must have the same name.

To make a migrate operation externally available in interface instances of type ifType, the programmer must include the line

```
IS COMPATIBLE WITH Migrate;
```

in the IDL file which defines ifType. Migrate is a library interface type which is defined by:

```
Migrate: INTERFACE =
BEGIN
  migrate: OPERATION [newLocation: STRING]
           RETURNS [];
END.
```

The preprocessor generates the implementation of the migrate operation.

18.4 Migrating an object

A client can migrate an object by invoking a migrate operation in an interface instance of the object. It is the client's responsibility to ensure that the argument to a migrate invocation denotes a NodeName property of an offer exported by a factory.

A migration request can cause the client to time out and the server to fail or time out if (i) no Factory offer has a NodeName property which matches the migration argument, (ii) a Factory offer with a matching NodeName property is stale, or (iii) the Factory cannot access an executable with the name specified in a PASSIVATION statement.

18.5 Building an application

The application programmer needs to follow some conventions for building applications which must have the migration infrastructure. The application programmer must use an Imakefile which defines the path of the include files and some files to be linked in; the Imakefile must also define the path to the new preprocessor. Before an application is executed, a factory must be running on each machine which is to host managed objects of the application, and SnapshotLocators and SnapshotBases must have been started up on selected machines.

18.6 An example application

The example application which was used in [Olsen 92c] to demonstrate how an application programmer can handcraft passivation and activation mechanisms for objects in an application, is reused in Appendix B and C to show what the application programmer needs to specify, in order to get the migration infrastructure generated. The reader is encouraged to compare the code in Appendix C with the code for handcrafting object migration in Appendix C of [Olsen 92c]. The difference between the original example application and its extension which contains handcrafted migration mechanisms, is about 200 additional lines of code in the extended application. The application in Appendix C for which the migration infrastructure is automatically generated has 2 line more than the original: the MIGRATE statement and the “IS COMPATIBLE WITH Migrate;” statement in an IDL file.

18.7 Location transparent migration

Location transparent migration enables object migration to be carried out *relative* to other objects. For example (i) migrating an object to a server it is about to invoke, (ii) migrating a server to a client prior to an invocation, or (iii) migrating objects which are referred in arguments of an invocation to the server which is being passed the arguments.

It is quite easy to support location transparent migration either by changing the semantics of the prepc language or by enriching it with new kinds of invocation statements which are preprocessed into the appropriate migration invocations. Location transparent migration can be regarded as syntactic sugar which hides absolute locations.

In some situations it is beneficial that an application programmer is free from dealing with absolute locations, in others it can be a serious problem: i.e. it is impossible to make sure that objects are migrated away from a machine which is about to be power-cycled if objects can only be migrated relatively to each other. Another area where location transparent migration is undesirable is when replicas have to be residing on different machines in order to support a degree of fault-tolerance. It is possible to combine location transparent migration and absolute location knowledge by designating an object on each location which cannot migrate and which thus represents the location. A good candidate for such an object is a factory.

The conclusion is, that location transparency need to be selective to support both the situation where the absolute location of an object need to be controlled and the situation where a relative location of an object is sufficient.

The prototype does not support location transparent migration, but the following sections 7.1 and 7.2 suggests ways location transparent migration could easily be supported by making some minor changes to the prototype’s preprocessor.

18.7.1 Changing invocation semantics

The preprocessor can be changed so if a flag is set it will either

- process prepc statements unchanged;

- process prepc invocation statements so a server is migrated to a client which is about to invoke it, and so objects which have their interfaces referred in invocation arguments are migrated to the client; this requires the preprocessor to be able to map from the invoked interface to the interface which contains the migrate operation in the server and from each argument interface to the interface which provides the migrate operation of an object which provides the argument interface;
- process prepc invocation statements so a client is migrated to a server prior to an invocation and so objects which have their interfaces referred in invocation arguments are migrated to the server; the preprocessor has knowledge of which interface the client's migrate operation is available in from a MIGRATE statement, but it must be able to map from each argument interface to the interface which provides the migrate operation of an object which provides the argument interface.

18.7.2 Enriching prepc

The prepc language can be changed so

- `{r1 .. rN} <- ifref$op(a1 .. aN)` has the current invocation semantics;
- `{r1 .. rN} <<- ifref$op(a1 .. aN)` migrates the invoking object to the invoked object;
- `{r1 .. rN} <- ifref$op(a1, <- a2, .. aN)` migrates arguments prefixed by <- to the invoked object;
- `{r1 .. rN} <<- ifref$op(a1, <- a2, .. aN)` migrates the invoking object and arguments prefixed by <- to the invoked object;
- `{r1 .. rN} <-> ifref$op(a1 .. aN)` migrates the invoked object to the invoker;
- `{r1 .. rN} <-> ifref$op(a1 -> .. aN)` migrates the invoked object and arguments postfixed by -> to the invoker;
- `{r1 .. rN} <<<- ifref$op(a1 .. aN)` migrates the invoking object to the invoked and after the invocation has been carried out, the invoking object is migrated back to its previous location;
- `{r1 .. rN} <->> ifref$op(a1 .. aN)` migrates the invoked object to the invoking object and after the invocation has been carried out, the invoking object is migrated back to its previous location;
- similarly, arguments can be migrated and returned to their original location by using the <<- prefix and the ->> postfix.

In order to carry out location transparent migration by the statements above, each interface reference must have an associated location attribute. This can easily be achieved by requiring that each offer registered in a trader must have a NodeName property. Once an object has changed its location the NodeName property recorded in the trader is invalid, but a locator records the updated location.

These language extensions not only serves as syntactic sugar, they can also improve performance because the migration of objects can be piggy backed message transmission. This cannot in general be achieved by migrating objects manually.

A number of projects have designed language support for migration, some of which are summarized in the next section.

18.8 Related work

Emerald [Jul 88] supports fine-grained object migration in homogeneous distributed systems. Migration is object based rather than process based, i.e. individual objects within an address space can be migrated rather than the whole address space. While invocations are location transparent, language primitives can be used to find and manipulate the location of objects: objects can be declared as being “attached” to other objects so they migrate when the object to which they are attached migrate, and a “call-by-move” parameter passing mode permits an invocation’s argument objects to be moved along with the invocation request. Object migration is programmer controlled rather than policy or system controlled, however, the Emerald compiler can make automatic decisions to move objects in limited cases. The language primitives enable an object to (i) locate an object, (ii) co-locate an object with another object, (iii) fix an object on the location of another object, (iv) unfix an object thus making it mobile again, (v) refix an object by atomically unfixing, moving and fixing an object.

Arjuna [Shrivastava] implements persistent objects in terms of passivation and activation mechanisms which are integrated with transaction mechanisms for controlling concurrent use of persistent objects. Object migration is carried out implicitly due to the fact that passive server objects are co-located with clients when activated.

COOL [Habert 90], an object-oriented layer on top of Chorus, enables objects to be migrated as part of invoking an object. Object migration is piggy backed on message transmission. Several objects can migrate along with a message, either on request or on reply message transmissions. The objects that should be migrated must be listed in an argument to an invocation. Although coupled with remote invocation, object migration is based on the distributed virtual memory management in Chorus. This corresponds to performing migration by passivation followed by activation. COOL has not yet been adapted for a heterogeneous environment.

In Comandos [Marques 89] object migration is carried out implicitly by the virtual object memory system which maps an object into an invoking client’s context on a node. If an invoked object is active on a remote node, the virtual object memory system first unmaps the object from its context on its current node and then maps it into the invoking client’s context on another node. This corresponds to performing migration by passivation followed by activation. However, the virtual object memory system can also decide to migrate an invoking object to an invoked object by transferring the invoker via RPC, similar to the way active objects are migrated in our prototype.

18.9 Conclusion

It has been shown how what an application programmer needs to specify in order to be able to migrate objects. It is the responsibility of the invoker of an object’s migrate operation to ensure that a factory is running on the remote node otherwise migration cannot be carried out. The paper motivated migration as an enabling technology principally used by system managers for

relatively infrequent dynamic re-configurations of distributed systems to ensure service availability when nodes become unavailable. If a node is to become unavailable, a system administrator should first terminate any factories running on the node to prevent new objects from being instantiated or activated on the node, and then invoke the migrate operation on all objects which reside on the node. When all objects have migrated, the node can be shut down. The paper has not addressed the issues of how a system administrator detects which objects (both active and passive) reside on a given node, nor how to choose which node to migrate each object to.

18.10 References

[AIM 91]
"ANSAware 3.0 Implementation Manual", Report No.: RM.097, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Copeland 84]
George Copeland, David Maier, "Making Smalltalk a Database System", *SIGMOD'84*, Sigmod Record, Vol. 14, No. 2, (1984).

[Habert 90]
S. Habert, V. Abrossimov and L. Mosseri, "COOL: Kernel Support for Object-Oriented Environments", *OOPSLA'90*, (1990).

[Herbert 91]
Andrew Herbert, "Engineering Model: Conceptual Framework", Report No.: RC.282, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Jul 88]
Eric Jul, "Object Mobility in a Distributed Object-Oriented System", Technical report 88-12-06, Department of Computer Science University of Washington Seattle, (1988).

[Marques 89]
J. A. Marques and P. Guedes,"Extending the Operating System to Support an Object-Oriented Environment", *OOPSLA'89*, (1989).

[Olsen 91a]
Michael Hoffmann Olsen,"A Model for Storage and Resource Management", Report No.: WP-DCG-91-14, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6Qz, England, (1991).

[Olsen 91b]
Michael Hoffmann Olsen,"The storage activity: objectives, tasks, timescales, and status", Report No.: RC.281, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91c]
Michael Hoffmann Olsen,"Handcrafting snapshot operations for an object", Report No.: RC.280, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91d]

Michael Hoffmann Olsen,"Automated generation of snapshot operations for an object", Report No.: RC.288, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91e]
Michael Hoffmann Olsen,"Design and implementation of a SnapshotBase", Report No.: RC.301, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 91f]
Michael Hoffmann Olsen,"Design and implementation of a SnapshotLocator", Report No.: RC.307, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 92a]
Michael Hoffmann Olsen,"Handcrafting passivation and activation", Report No.: RC.321, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 92b]
Michael Hoffmann Olsen,"Automated generation of passivation and activation", Report No.: RC.326, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Olsen 92c]
Michael Hoffmann Olsen,"Handcrafting object migration", Report No.: RC.329, ISA project, Poseidon House, Castle Park, Cambridge, England, (1991).

[Shrivastava 89]
S. K. Shrivastava et al, “The Treatment of Persistent Objects in Arjuna”, *THE COMPUTER JOURNAL*, VOL. 32, NO. 4, (1989).

[Warne 91]
John Warne, "ANSA: Assumptions, Principles, and Structure", *Conference proceedings of Software Engineering Environments 1991*, University College of Wales, Aberystwyth, (1991).

19 Appendix: Prepc preprocessor extensions

This appendix describes the extensions which have been made to the prepc preprocessor to preprocess the new MIGRATE statement. It is assumed that the preprocessor has already been modified to preprocess the SNAPSHOT statement as described in [Olsen 91d] and the PASSIVATE statement as described in [Olsen 92b].

The file gram.y is extended as follows:

- the grammar for the non-terminal statement has been extended to enable the derivation of:

```
MIGRATE TOKEN USING TOKEN
```

- the routine yylex has been extended so the lines:

```
case MIGRATE:
case USING:
```

is added to the switch alternatives for value

- the routine dolex has been extended with the following lines for constructing a return value:

```
if (strcmp(tok, "MIGRATE") == 0)
   return MIGRATE;
if (strcmp(tok, "USING") == 0)
```

return USING;

the routine processmigration has been added. It takes two strings as arguments; the first is the template name which followed MIGRATE in a PASSIVATION statement and the second is the interface name which followed USING in that statement. It first generates the implementation of a SnapshotBase's Store interface, then it generates the migrate operation.

The routine outputstatement has been extended to generate the preprocessed code for a migrate operation invocation, i.e. calls to Activator.migrate for migrating a passive objects and calls to the I_migrate and C_migrate procedures in the client stub for migrating an active object.

- The routine outputredeem has been extended to check that the migrate operation is not invoked as an announcement.

20 Appendix: The example application code

This appendix contains the code for the example application referred to in the main part of the paper. The application consists of a client and a server. The server implements a telephone book by maintaining a table of names and telephone numbers. It exports a reference to a PhoneBookOp interface instance to a trader, which enables a client to insert, remove, lookup and list names and phone numbers. The application uses the passivation and activation infrastructure.

The implementation of the server's operations have been omitted as they are independent of the migration mechanism which is added in appendix B; the implementations can be found in [Olsen 92b].

20.1 idl files

20.1.1 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry           : TYPE = RECORD
                             [ name : Name,
                               no   : No
                             ];
  PBEntryList       : TYPE = SEQUENCE OF PBEntry;
END.
```

20.1.2 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
IS COMPATIBLE WITH Passivate;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
END.
```

20.2 dpl files

20.2.1 server.dpl

```
! MANAGED
! USE PhoneBookOp
! DECLARE { phoneBook } : PhoneBookOp SERVER

#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16
GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

#include "NewInterfaceRef.c"

PBEntryList entryList;
NewInterfaceRef phoneBook;

! PASSIVATION OF {entryList PBEntryList, phoneBook
NewInterfaceRef}\
  IN PhoneBook INTERNALLY 30 EXTERNALLY PhoneBookOp

ansa_InterfaceRef ref[2];

ansa_StatePtr NewCreate__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ initNewInterfaceRef();
  initConcControl();
! {phoneBook} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  registerNewInterface(&phoneBook, "PhoneBookOp", "/ansa/
PhoneBook",
                       "");
  copyIfRef(&(ref[0]), &(phoneBook.ir));
  copyIfRef(&(ref[1]), &(phoneBook.locator));
  results->length = 2;
  results->data = ref;
  return (ansa_StatePtr)0;
}

void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  hasBeenInvoked++;
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
```

```
Name name;
{ getSingleWriteLock();
  hasBeenInvoked++;
  /* remove entry identified by name from entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  hasBeenInvoked++;
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  hasBeenInvoked++;
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

20.2.2 client.dpl

```
! USE Trader
! USE Capsule
! DECLARE {cref}: Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook}: PhoneBookOp CLIENT

#include <stdio.h>
#include "capsule.h"
#include "system.h"

#include "NewInterfaceRef.c"
#include "Activator.c"

NewInterfaceRef phoneBook;
PBEntryList eList;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "insert",
  "remove",
  "lookup",
  "list",
  "passivate",
  "activate",
  "create",
```

```
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {4, 3, 3, 2, 2, 2, 2};
int maxargs[] = {4, 3, 3, 2, 2, 2, 2};

#define INVALID_CMD -1

void listEntryList(entryList)
  PBEntryList *entryList;
{ /* print contents of entryList on stdout */
}

void PrintUsageAndDie(prog)
char *prog;
{ /* print error message and terminate */
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ int i, cmd_no;

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  initNewInterfaceRef();
  if (cmd_no != 6)
  importNewReference(&phoneBook, "PhoneBookOp", "", "");
  switch(cmd_no)
  { case 0:   /* insert */
      { PBEntry entry;

        strcpy(entry.name, argv[2]);
        strcpy(entry.no, argv[3]);
!       {} <- phoneBook$insert(entry)
      }
      break;
```

```
        case 1: /* remove */
          { Name name;

            strcpy(name, argv[2]);
!           {} <- phoneBook$remove(name)
          }
          break;
        case 2:   /* lookup */
          { PBEntry reply;

!           {reply} <- phoneBook$lookup(argv[2])
            if (strcmp(reply.name, "0") == 0)
            { fprintf(stdout, "No entry for %s\n", argv[2]);
            }
            else
            { fprintf(stdout, "No: %s\n", reply.no);
            }
          }
          break;
        case 3: /* list */
          { PBEntryList entryList;

!           {entryList} <- phoneBook$list()
            listEntryList(&entryList);
          }
          break;
        case 4: /* passivate */
          {
!           {} <- phoneBook$passivate()
          }
          break;
        case 5: /* activate */
          { ansa_InterfaceAttr *_attr;

            Activator_activate(_attr, &phoneBook, &phoneBook);
          }
          break;
        case 6: /* instantiate */
          { InstantiateResult res;
            ObjectId id;
            NewInterfaceRef fref, cref;

!           {fref} <- traderRef$Import("Factory", \
                                       "NodeName == 'audrey'", "")
!           {cref} <- fref$Instantiate("PhoneBook", "", "")
!           {id, res} <- cref$Instantiate(nullRef, "0", "")
          }
          break;
      }
    }
```

20.3 Imakefile

```
          LTIDIR = ../master/livenessTransparency/include
          LTSDIR = ../master/livenessTransparency/stubc
         DEFINES =
        INCLUDES = -I$(LTIDIR)
```

```
        IDLFLAGS =
        DPLFLAGS =
        LINTLIBS = $(LINTANSALIB) -lc
        LOCALLIB = $(ANSALIB)
       LOCALLIBD = $(ANSALIBD)
         INSTALL = bsdinstall.sh

LTSRCS   = cSLLocate.c cSBStore.c sPassivator.c cPassivator.c
LTSOBJECTS = cSLLocate.o cSBStore.o sPassivator.o cPassivator.o
LTCOBJECTS = cSLLocate.o
IDLFILES = PhoneBookOp.idl
SIFFILES = PhoneBookOp.sif
DPLFILES = PhoneBook.dpl PBclient.dpl
SRCS     = ConcControl.c $(LTSRCS)
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)

PROGS = PhoneBook PBclient

# compile idl files
all:: $(SIFFILES)
      $(CP) $(LTSDIR)/cSLLocate.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/cSBStore.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/sPassivator.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/cPassivator.c $(CURRENT_DIR)

# compile dpl files
DPLDepend(PhoneBook)
DPLDepend(PBclient)

all:: $(PROGS)

#
# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)

SingleProgramTarget(PhoneBook, PhoneBook.o sPhoneBookOp.o \
ConcControl.o $(LTSOBJECTS),$(LOCALLIB),)
SingleProgramTarget(PBclient, PBclient.o cPhoneBookOp.o \
$(LTCOBJECTS),$(LOCALLIB),)

InstallProgram(PhoneBook,$(TEMPLATEDIR))

NormalLintTarget(*.c)
DependTarget()

IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

21 Appendix: The extended example application code

This appendix contains the code for the example application in appendix B which has been modified so a migration infrastructure is added to the server and so the client can test the migration operation.

21.1 idl files

21.1.1 PhoneBookTypes.idl

```
PhoneBookTypes : INTERFACE =
BEGIN
  Name : TYPE = ARRAY 64 OF CHAR;
  No   : TYPE = ARRAY 64 OF CHAR;
  PBEntry         : TYPE = RECORD
                           [ name : Name,
                             no   : No
                           ];
  PBEntryList     : TYPE = SEQUENCE OF PBEntry;
END.
```

21.1.2 PhoneBookOp.idl

```
PhoneBookOp : INTERFACE =
NEEDS PhoneBookTypes;
IS COMPATIBLE WITH Passivate;
IS COMPATIBLE WITH Migrate;
BEGIN
  insert      : OPERATION [entry : PBEntry]
                RETURNS [];
  remove      : OPERATION [name : Name]
                RETURNS [];
  lookup      : OPERATION [name : Name]
                RETURNS [entry : PBEntry];
  list        : OPERATION []
                RETURNS [entryList : PBEntryList];
  setColor    : OPERATION [ color : STRING]
                RETURNS   [];
END.
```

21.2 dpl files

21.2.1 server.dpl

```
! MANAGED
! USE PhoneBookOp
! DECLARE { phoneBook } : PhoneBookOp SERVER
```

```
#define POTENTIALCONCURRENCY  16
#define ACTUALCONCURRENCY     24 + 16
GLOBAL ansa_Cardinal Ansa_InitialTasks = ACTUALCONCURRENCY;

#include "NewInterfaceRef.c"
PBEntryList entryList;
NewInterfaceRef phoneBook;

! PASSIVATION OF \
  {entryList PBEntryList, phoneBook NewInterfaceRef} \
  IN PhoneBook INTERNALLY 30 EXTERNALLY PhoneBookOp

! MIGRATE PhoneBook USING PhoneBookOp

ansa_InterfaceRef ref[2];

ansa_StatePtr NewCreate__Object(argc, argv, envp, results)
int argc;
char *argv[];
char *envp[];
ansa_InterfaceSeq *results;
{ initNewInterfaceRef();
  initConcControl();
! {phoneBook} :: PhoneBookOp$Create(POTENTIALCONCURRENCY)
  registerNewInterface(&phoneBook, "PhoneBookOp",
                       "/ansa/PhoneBook", "");
  copyIfRef(&(ref[0]), &(phoneBook.ir));
  copyIfRef(&(ref[1]), &(phoneBook.locator));
  results->length = 2;
  results->data = ref;
  return (ansa_StatePtr)0;
}

void Destroy__Object(state)
ansa_StatePtr state;
{
}

int PhoneBookOp_insert(_attr, newent)
ansa_InterfaceAttr *_attr;
PBEntry newent;
{ getSingleWriteLock();
  hasBeenInvoked++;
  /* insert newent in entryList */
  releaseSingleWriteLock();
  return 1;
}

int PhoneBookOp_remove(_attr, name)
ansa_InterfaceAttr *_attr;
Name name;
{ getSingleWriteLock();
  hasBeenInvoked++;
  /* remove entry identified by name from entryList */
  releaseSingleWriteLock();
  return 1;
}
```

```
int PhoneBookOp_list(_attr, entList)
ansa_InterfaceAttr *_attr;
PBEntryList *entList;
{ getMultipleReadLock();
  hasBeenInvoked++;
  /* return pointer to entryList in entList */
  releaseMultipleReadLock();
  return 1;
}

int PhoneBookOp_lookup(_attr, name, entry)
ansa_InterfaceAttr *_attr;
Name name;
PBEntry *entry;
{ getMultipleReadLock();
  hasBeenInvoked++;
  /* return entry identified by name */
  releaseMultipleReadLock();
  return 1;
}
```

21.2.2 client.dpl

```
! USE Trader
! USE Capsule
! DECLARE {cref}: Capsule    CLIENT
! USE Factory
! DECLARE {fref}: Factory    CLIENT
! USE PhoneBookOp
! DECLARE {phoneBook}: PhoneBookOp CLIENT

#include <stdio.h>
#include "capsule.h"
#include "system.h"
#include "NewInterfaceRef.c"
#include "Activator.c"

NewInterfaceRef phoneBook;
PBEntryList eList;

/* commands that session can execute on PhoneBook */
char *cmd_list[] =
{ "insert",
  "remove",
  "lookup",
  "list",
  "passivate",
  "activate",
  "create",
  "migrate",
  (char *)0
};

/* Min and max number of arguments for each command */
int minargs[] = {4, 3, 3, 2, 2, 2, 2, 3};
int maxargs[] = {4, 3, 3, 2, 2, 2, 2, 3};
```

```
#define INVALID_CMD -1

void listEntryList(entryList)
  PBEntryList *entryList;
{ /* print contents of entryList on stdout */
}

void PrintUsageAndDie(prog)
char *prog;
{ /* print error message and terminate */
}

void body(argc, argv, envp)
int  argc;
char *argv[];
char *envp[];
{ int i, cmd_no;

  /* check number of parameters */
  if (argc < 2)
  PrintUsageAndDie(argv[0]);

  /* check valid command name */
  cmd_no = INVALID_CMD;
  for (i = 0; cmd_list[i] != (char *)0; i++)
  { if (strcmp(argv[1], cmd_list[i]) == 0)
    { cmd_no = i;
      break;
    }
  }

  /* if command not valid, print error and quit */
  if (cmd_no == INVALID_CMD)
  PrintUsageAndDie(argv[0]);

  /* if wrong number of arguments, print error and quit */
  if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
  PrintUsageAndDie(argv[0]);

  initNewInterfaceRef();
  if (cmd_no != 6)
  importNewReference(&phoneBook, "PhoneBookOp", "", "");
  switch(cmd_no)
  { case 0:   /* insert */
      { PBEntry entry;

        strcpy(entry.name, argv[2]);
        strcpy(entry.no, argv[3]);
!       {} <- phoneBook$insert(entry)
      }
      break;
    case 1: /* remove */
      { Name name;

        strcpy(name, argv[2]);
!       {} <- phoneBook$remove(name)
      }
      break;
```

```
        case 2:   /* lookup */
          { PBEntry reply;

!           {reply} <- phoneBook$lookup(argv[2])
            if (strcmp(reply.name, "0") == 0)
            { fprintf(stdout, "No entry for %s\n", argv[2]);
            }
            else
            { fprintf(stdout, "No: %s\n", reply.no);
            }
          }
          break;
        case 3: /* list */
          { PBEntryList entryList;

!           {entryList} <- phoneBook$list()
            listEntryList(&entryList);
          }
          break;
        case 4: /* passivate */
          {
!           {} <- phoneBook$passivate()
          }
          break;
        case 5: /* activate */
          { ansa_InterfaceAttr *_attr;

            Activator_activate(_attr, &phoneBook, &phoneBook);
          }
          break;
        case 6: /* instantiate */
          { InstantiateResult res;
            ObjectId id;
            NewInterfaceRef fref, cref;

!           {fref} <- traderRef$Import("Factory", \
                                       "NodeName == 'audrey'", "")
!           {cref} <- fref$Instantiate("PhoneBook", "", "")
!           {id, res} <- cref$Instantiate(nullRef, "0", "")
          }
          break;
        case 9: /* migrate */
          {
!           {} <- phoneBook$migrate(argv[2])
          }
          break;
      }
    }
```

21.3 Imakefile

```
LTIDIR = /usr/users/mho/PROTOTYPE/PASSIVATION/\
         livenesstransparency/include
LTSDIR = /usr/users/mho/PROTOTYPE/PASSIVATION/\
         livenesstransparency/stubc
LTLDIR = /usr/users/mho/PROTOTYPE/PASSIVATION/\
         livenesstransparency/idl
```

```
          DEFINES =
         INCLUDES = -I$(LTIDIR)
         IDLFLAGS = -I$(LTLDIR)
         DPLFLAGS =
         LINTLIBS = $(LINTANSALIB) -lc
         LOCALLIB = $(ANSALIB)
        LOCALLIBD = $(ANSALIBD)
          INSTALL = bsdinstall.sh

LTSRCS     = ConcControl.c cSLLocate.c cSBStore.c sSBStore.c\
             sPassivator.c cPassivator.c cKillMySelf.c
sKillMySelf.c
LTSOBJECTS = ConcControl.o cSLLocate.o cSBStore.o sSBStore.o\
             sPassivator.o cPassivator.o cKillMySelf.o
sKillMySelf.o
LTCOBJECTS = cSLLocate.o
IDLFILES = PhoneBookOp.idl
SIFFILES = PhoneBookOp.sif
DPLFILES = PhoneBook.dpl PBclient.dpl
SRCS     =
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS)

PROGS = PhoneBook PBclient

# compile idl files
all:: $(SIFFILES)
      $(CP) $(LTSDIR)/cSLLocate.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/cSBStore.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/sSBStore.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/sPassivator.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/cPassivator.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/sKillMySelf.c $(CURRENT_DIR)
      $(CP) $(LTSDIR)/cKillMySelf.c $(CURRENT_DIR)
      $(CP) $(LTIDIR)/ConcControl.c $(CURRENT_DIR)
      $(CP) $(LTIDIR)/ConcControl.h $(CURRENT_DIR)

# compile dpl files
DPLDepend(PhoneBook)
DPLDepend(PBclient)

all:: $(PROGS)

#
# Need these .o .c dependencies to keep make happy, since
# it will not spot implicit dependencies generated by this make
run!
IDLDepend(PhoneBookOp)

SingleProgramTarget(PhoneBook, PhoneBook.o sPhoneBookOp.o \
                    $(LTSOBJECTS),$(LOCALLIB), )
SingleProgramTarget(PBclient, PBclient.o cPhoneBookOp.o \
                    $(LTCOBJECTS),$(LOCALLIB),)

InstallProgram(PhoneBook,$(TEMPLATEDIR))

NormalLintTarget(*.c)
DependTarget()
```

```
IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```