Advanced Networked Systems Architecture ANSR

24 Hills Road
CAMBRIDGE
United Kingdom CB2 1JP

TELEPHONE: Cambridge (0223) 323010
INTERNATIONAL: +44 223 323010
TELEX: 817343 BLUCAM G

ANSA Reference Manual

Release 0.0

Abstract:

This is the title page and document list for the ANSA Reference Manual Release 0.0

Number: A0.52.00

Date: 9th January 1987 11:04 am

Area: Ad (Architectural Overview)

Status: 0 (Open Document)

Classification: U (Unrestricted)

Distribution: G (General Public)

The following documents comprise Release 0.0:

Advanced Networked Systems Architecture
ANSR

24 Hills Road
CAMBRIDGE
United Kingdom CB2 1JP

TELEPHONE : Cambridge (0223) 323010
INTERNATIONAL: + 44 223 323010
TELEX: 817343 BLUCAM G

ANSA Reference Manual (Pre-release) Contents and Preamble

Abstract:

This document is the contents page and preamble for the ANSA Reference Manual Release 0.0.

Number: A0.46.00

Date: 9th January 1987 10:23 am

Area: AO (Architectural Overview)

Status: 0 (Open Document)

Classification: U (Unrestricted)

Distribution: G (General Public)

CONTENTS

Page

PART II
Technical Assumptions for Distributed Systems

PART III
Definitions and concepts

1 General
- 1.1 Introduction
- 1.2 Scope
- 1.3 Format

2 Types and objects AJH
- 2.1 Introduction
- 2.2 Type concept
- 2.3 Information type concept
- 2.4 Operation concept
- 2.5 Operational abstraction
- 2.6 Object concept
- 2.7 Subtype concept

3 Interfaces Joe Sventek
- 3.1 Introduction
- 3.2 Operational interface concept
- 3.3 Indirect binding concept
- 3.4 Trader concept
- 3.5 Invocation concept (announce, interrogate, listen, reaction, break)
- 3.6 Invocation mode concept (reliable, sequenced, bandwidth etc)
- 3.7 Interface coupling concept (cf Giiord packages etc)

4 Abstract machines John Wratten
- 4.1 Introduction
- 4.2 Abstract machine concept (instructions, exceptions, nesting)
- 4.3 The local abstract machine (operating system transparency)
- 4.4 The distributed abstract machine (distribution transparency)
- 4.5 The nucleus (objects for building the DAM from the LAM)

5 Physical environment
- 5.1 Introduction
- 5.2 Processors (primitives, events, execution) (JM/AJH)
- 5.3 Storage (virtual memory)
- 5.9 I/O devices (control and data channel model) (GOE)
- 5.5 Networks (latency, rate control) (DO)

6 Logical environment DO + EdO
- 6.1 Introduction
- 6.2 Threads (virtual machine emulation,synchronization, ordering)
- 6.3 Storage (volatile, stable, versioned, persistant, virtual)
- 6.4 Transformers (from Sventek paper)
- 6.6 Resources (upcall arguments)

7 The human user (from JM/YH papers - discussion paper statement)
7.1 Introduction
(Dialogue, conceptual models, context, service style)

8 Transparency
8.1 Introduction
a.2 Access transparency
8.3 Location transparency
8.4 Migration transparency (from Delta 4)
8.5 Replication transparency (from Delta 4 and JPW paper)
8.6 Concurrency transparency (from JPW paper)
8.7 Failure transparency (from Delta 4, Newcastle work and JPW paper)
3.8 Performance transparency
8.9 Scaling transparency

9 Naming
9.1 Introduction
(context independent, catalogues)

10 Security
11.1 Introduction
11.2 Security models (from RvdL paper)
11.3 Separation
11.4 Accountability (from DASE spec)
11.5 Authentication
11.5 Mechanisms (access control, encryption, capabilities)

11 Data
11.1 Introduction (from RvdL/EO papers - discussion paper status)
11.2 Data model (from RvdL/EO papers - discussion paper status)
11.3 Data access (from RvdL/EO papers - discussion paper status)
11.4 Data management (from RvdL/EO papers - discussion paper status)
11.5 Data structures (document architecture from JDW paper - draft status)

12 Management (from RvdL/HJW papers - discussion paper status - needs liaison with CIM/OSA)
12.1 Introduction
12.2 Management model
(management things not picked up by object model, transparency, naming, security or data)

PART IV
Architectural framework

(these headings are provisional pending description of architectural methodology)

PART V Object and protocol descriptions

PART VI Specifications

18 Message passing protocol

19 Stream protocol

PART VII Examples

APPENDIX A: References

Editorial history:

Issue	Notes
01	Open Document status in Release 0.0 Approved AJH 9/1/87

PREAMBLE

1 GENERAL

1.1 Introduction

The *ANSA Reference Manual* is the principle technical output of the Alvey ANSA project. It describes, in detail, an architecture for advanced networked systems that embody and exploit distributed computing concepts.

This manual is directed towards a technical audience. Less technical information about ANSA can be found in the *ANSA Information Pack* which can be obtained freely upon application to the ANSA project office.

1.2 Structure

The manual is divided into nine major parts as follows:

- part I (Overview) is a short overview of ANSA containing a brief requirements statement for the architecture, a review of distributed system concepts and a guide to the content of ANSA
- part II (Technical assumptions) reviews the technical assumptions that have been made during the development of ANSA and focuses in particular on the advanced research results that have been absorbed into the architecture
- part III (Concepts and definitions) gives definitions for the technical terms used in ANSA and explains important ANSA concepts, as a prelude to description of the architecture in detail
- part IV (Architectural framework) introduces a formal classification framework for ANSA and the rules by which the framework may be used.
- part V (Object and protocol descriptions) gives a full textual description of every object and protocol in the architecture, using the concepts and definitions from part III and the framework from part IV.
- part VI (Formal specifications) gives formal specifications for the framework introduced in part IV and the objects described in part V
- part VII (Examples) gives some worked examples of the use of the architecture in practical networked systems
- Appendix A (References) give a consolidated list of journal articles, technical reports and standards referenced throughout the manual.

The formal specifications in part VI are authoritative. Descriptions and explanations in other parts of the manual are intended as corroborative interpretation of part VI.

1.3 Copyright issues

There are no restrictions on the access to, or distribution of, this manual. Error reports or requests for explanation should be directed to the ANSA project.

2 CHANGE CONTROL

The ***ANSA Reference Manual*** is an evolving document and is therefore under the formal change control of the ANSA project. Each release of the Manual has a release number of the form M.N. The first number, M, is incremented every time there is a substantial change to the contents of the manual; the addition of a new part for example. The second number, N, is incremented when more modest changes, such as the addition or revision of a section, occurs. The release number is recorded on the title sheet at the front of the Manual.

The various parts and sections of the ANSA ***Reference Manual*** are kept as separate ANSA project documents for convenience of editing and management. ANSA project, documents are numbered as in, for example A0.46.05. The two letter code indicates in which phase of the project the document was issued. The first number (46) is the serial number assigned to the particular document and the second number (05) indicates the version.

Every document is labelled with the ***ANSA Reference Manual*** Release to which it belongs. The status of the document within the release is shown by the status code on the document, from sheet. Status codes are to be interpreted as follows:

- P (Discussion Paper)
  The document, is in an early stage of drafting as should treated as purely informative about the intended content for that part of the Reference Manual

- 0 (Open Document)
  The document is a draft awaiting validation before formal inclusion in the release

- R (Released Document)
  The document, is stable within the release and has been validated for inclusion in the release.

(In IS0 standards terminology, these status codes are roughly equivalent to 'expert contribution', 'draft proposal (DP)' and 'draft international standard (DIS)'.

Every document contains an editorial history at, its head that record the changes between issues. Changes between releases of the manual as a whole are record in Chapter 3 of the Premable.

Editorial comments about the manual are shown in the text thus:

Editorial: this is an editorial comment for the guidance of the reader.

Editorial comments are used to indicate inconsistencies or shortcomings in the text for later resolution.

5 RELEASE HISTORY

Advanced Networked Systems Architecture

ANSR

24 Hills Road
CAMBRIDGE
United Kingdom CB2 1JP

TELEPHONE: Cambridge (0223) 323010
INTERNATIONAL: + 44 223 323010
TELEX: 817343 BLUCAM G

ANSA Reference Manual

Part I: Overview

Abstract:

This document is Part 1 the ANSA Reference Manual Release 0.0.

It provides a brief summary of the purpose of the architecture, its orientation and content.

Much of this draft is taken from ECMA/TC32-TG2/86/48 "Distributed Application Support Environment (DASE) Overview", Draft 0.

Number: A0.47.00

Date: 9th January 1987 11:58 am

Area: AO (Architectural Overview)

Status: 0 (Discussion Paper)

Classification: U (Unrestricted)

Distribution: G (General Public)

Editorial history:

Issue	Notes
01	Discussion paper status for Release 0.0 Approved AJH 9/1/87

1 GENERAL

Editorial: this overview is indicative of the sort of material that could be used to make up an overview. The overview will evolve and stablize as subsequent parts of the manual become better developed.

1.1 Introduction

ANSA defines an abstract, programming language-independent, computational model for the construction of distributed information processing systems. This model encourages the design of well-structured systems which are amenable to generation from formal system specifications.

ANSA provides the essential functional definitions needed to support use of the computational model over a wide range of quality attributes, in a wide field of application. These essential functions are called the ANSA nucleus.

Figure 1.1: Position of ANSA

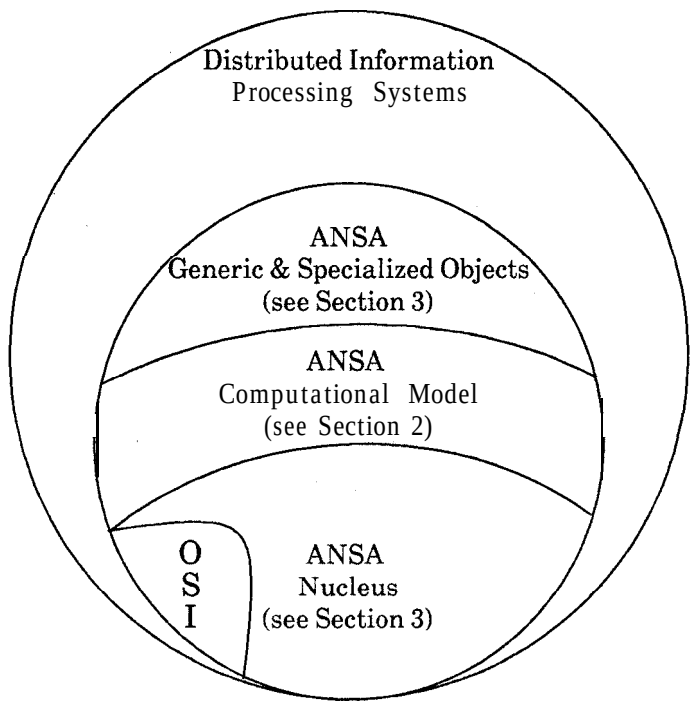


Figure 1 shows that the ANSA computational model and nucleus are complementary to, and may make use of, existing networking standards, particularly those defined by the International Standards Organization (ISO) within the Open Systems Interconnection (OSI) model. Alternatively, ANSA can be realized in the context of proprietary communication architectures, or in a closely-coupled multiprocessor host environment.

1.2 Scope of this overview

This overview is part of the *ANSA Reference Manual* and addresses three topics:

- the identified fields of application and the required functional and quality attributes for ANSA (§1)
- the distributed computing concepts which ANSA supports (§2)
- the content of ANSA (§3)

1.3 Field of application

ANSA is applicable to all commercial and industrial distributed systems. The field of application specifically meets requirements from:

- telecommunications
- integrated video, image, voice, graphics, text and data systems
- factory automation
- office automation
- command, control and communication systems
- distributed data-processing systems
- real-time systems
- embedded military systems

ANSA support for real-time systems include both transaction processing systems and industrial process control systems.

1.4 Requirements

The distinctive characteristic of a distributed system is that services are provided by a set of cooperating, logically separate, information processing units or components.

The components of a distributed system may be at physically remote sites or co-located at a single site but always remain logically separate from the point of view of system organization. ANSA defines a set of concepts by which the logically separate components of distributed systems can be integrated into a unified structure.

1.4.1 Required functional attributes

Interaction between components can be synchronous interactions: the so-called client / server interaction in which a client component specifies the operation to be performed by the server and later receives a reply from a server. In the course of executing that operation, the server component may

interact with other components (including the client component). Synchronous interactions are oriented towards **imperative** operations.

Alternatively, a component may simply send information to a destination component with an optional guarantee of delivery. This is the producer / consumer model of interaction. Such one-way **asynchronous** interaction can be generalized to a multi-endpoint interaction in which information is disseminated to an arbitrarily wide audience of components. Asynchronous interactions are oriented towards **advisory** interactions.

These primitive interactions form the basis for all distributed systems designs. To satisfy the wide field of applications targeted by ANSA, these interactions need to be supported by additional facilities which are now described:

- real-time systems are distinguished by the need to respect time deadlines. In the context of distributed systems, it is necessary to extend techniques such as priority scheduling, and the recognition of real-time from the local operating system environment into the distributed environment
- transaction processing systems require the property of atomicity i.e., that all or none of the operations within a transaction are performed. Specific protocols to connect or abort transactions are required in ANSA
- it is probable that the demand for dependable system operation, already familiar in the transaction processing and process control fields, will be met in future by the provision of redundant components within a distributed system. This will require support for logical component replication and coordination which, in turn, involves communications between replicates

The requirement to preserve the real-time attributes of local operating systems and so allow the construction of distributed operating systems is believed to be of particular significance. This requirement is satisfied by ANSA.

1.4.2 Required quality attributes

The principal quality attribute identified for ANSA is that of performance. While this attribute has general appeal, it has particular significance to real-time systems in general and process control systems in particular. Other quality attributes are required to meet the diverse requirements of the envisaged broad field of application. It is a basic tenet of ANSA that performance can often be surrendered to obtained other quality attributes whereas it is difficult to add performance if it is not designed in initially.

1.4.2.1 Performance

There is a natural relationship between distribution and performance. Distribution provides opportunities for parallelism in both closely and loosely-coupled multi-computer systems. ANSA fully exploits opportunities for parallelism in either of these environments in an efficient and dependable way.

how is that to be provided

1.4.2.2 Dimensioning and scaling

ANSA is applicable to a diverse range of network sizes and does not preclude the growth of a small network to a very large network while maintaining efficiency. ANSA specifically solves problems associated with uncertainty of styles of service provision and locality which arise in large networks.

1.4.2.3 *Portability*

ANSA is defined at a high level of abstraction, so that the specifications and designs of the interactive aspects of distributed applications are independent of particular networks, operating systems and programming languages.

1.4.2.4 *Compatibility*

ANSA permits interactions between system components to have strong assurances of compatibility. The strong "type" concept associated with object-oriented computational models is an intrinsic part of ANSA.

1.4.2.5 *Resilience*

The design and qualitative characteristics of ANSA are such that application designers can quickly specify, develop and deliver distributed application services, independent of communication infrastructures.

1.4.2.6 *Multi-mediacapability*

ANSA provides support for interactions that involve the integrated use of several communications media within the interaction, including video, image, voice, data, text and graphics.

1.4.2.7 *Manageability*

management ? ie in terms of mgt/supervision ?

ANSA permits distributed application systems and subsystems to be readily installed, changed and maintained.

1.4.2.8 *Security*

The sociological model for ANSA allows for systems which contain a number of autonomous components, interacting on a-peer-to-peer basis, without the

need for there to be a single centre of authority. Security and related issues take place in a context of decentralized control and multiple administrative authorities. This model opens up the possibility of subversive system components defeating system-wide security policies. ANSA provides means for taking precautions against this threat.

1.4.2.9 Controllability

ANSA supports control mechanisms to permit interaction to be handled with a priority regime to enable processing and communication traffic control during periods of congestion or following resource failure.

1.5 Relation to existing initiatives

1.5.1 Introduction

ANSA is not an architecture in isolation. It has three major sources of input:

- well-established research into distributed systems
- existing operating system and communications standards
- standards development work

The technical background to ANSA in terms of established research results is analyzed in §4 of Part II of the ***ANSA Reference Manual.*** These results are important since they bring new ideas to bear on systems design and are the key contributors to the achievement of the ambitious requirements set, in §1.4.

The ANSA project recognizes the importance of building on previous standards work. It, is, therefore, central to the design of ANSA that applications conforming to the defined distributed computational model are presented with an interface which can be supported by existing standards as well as ANSA specifics. This aspect of ANSA provides:

- an evolution path for existing applications to use the enhanced ANSA facilities
- an option to run newly developed ANSA applications on existing systems

ANSA therefore makes effective use of existing standards and the investment in them by industry and users.

1.5.2 Communication standards

The primary communications standards focus of ANSA is towards the IS0 Open Systems Interconnection standards (OSI). This focus is demonstrated in the following ways:

- inclusion of the current Remote Operations Service [CCITT X.ros1] in ANSA, mapped onto CCITT Recommendation X.410 Reliable Tranfer Server [X.410]
- adopting the Remote Operations technology as a fundamental ingredient of the ANSA design
- use of the IS0 8824 Presentation Layer ASN/1 abstract syntax [ISO 8824] and the corresponding IS0 8825 / CCITT C.409 basic encodings [ISO 8825, CCITT X.4091

The focus is not exclusively towards OSI. ANSA has been designed to permit mappings onto other open communication standards, such as TCP/IP [DOD 85], and onto proprietary network standards.

A major concern of ANSA lies in the support of multi-media applications which requires alignment, of data-communications standards such as OSI with emerging public switched communications (ie ISDN).

1.5.3 Operating system standards

ANSA provides a bridge between communication standards and operating system standards and consequently use of ANSA will have an impact. on operating systems.

ANSA has been designed to permit its use in the context of a wide range of operating system designs. Particular, but not exlusive, attention has been given to the Unix† operating system.

The ANSA nucleus can be provided within the Unix kernel or as a library positioned above the kernel.

Unix is not able to satisfy the possible range of quality attributes supported by ANSA and therefore is not, applicable in all circumstances. ANSA provides a convenient framework within which to design extensions or modifications of Unix to meet quality attribute requirements presently beyond its capabilities.

1.5.3 ECMA standards

The ANSA project has liaised closely with ECMA TC32-TG2. This group is preparing a technical report on a "Distributed Applications Support Environment (DASE)". DASE is closely derived from the ANSA nucleus design. The DASE technical Report and the *ANSA Reference Manual* share common text.

---

†Unix is a trademark of AT& T Bell Laboratories.

The scope of ANSA is wider than DASE, since ANSA includes application structures whereas DASE encompasses only the compuational model and nucleus.

1.5.5 ESPRIT and RACE

Many ESPRIT and RACE projects are enhancing or adding to OSI by conducting research and development to meet the needs of specific functional areas, such as office automation and factory automation.

The ANSA project has formed technical liaisons with several of these ESPRIT and RACE projects as follows:

- Communications Systems Architecture (CSA - ESPRIT 237)
- Definition and Design of an Open Dependable Distributed Computer System Architecture (DELTA-4 - ESPRIT 818)
- Computer Integrated Manufacturing - Open Systems Architecture (CIM-OSA - ESPRIT 955)
- Subscriber Premises Networks and Terminals (RACE 1006)

The purpose of these liaisons has been to ensure technical convergence between overlapping aspects of these projects and ANSA.

2 DISTRIBUTED SYSTEMS

2.1 Introduction

This section briefly introduces distributed system concepts and indicates the problems in programming such systems. An object-oriented computational model is introduced to overcome these problems. The computational model is not restricted to distributed applications. Also, the computational model does not preclude existing standards defined using the remote operations concepts of [CCITT X.rosO].

2.2 What is a distributed system?

Many applications systems consist of several parts which interwork. Such systems can be said to be distributed when not all parts of the application execute at the same site. When a system is physically distributed over several sites, the parts of the system interact by means of a communications network. There are several problems introduced by physically distributed applications which are outlined in section 2.3.

A distributed system should be distinguished from a networked system. The latter is only concerned with communications between autonomous end-systems. For example, the transfer of a file between remote filestores. Any distributed processing in a networked system is only necessary to handle communications, whereas distributed systems are complete systems that require communications between their components, and where all forms of processing, not just communications processing, are potentially distributed.

2.3 Problems with distribution

There are several problems with physically distributed systems that are not encountered in non-distributed systems. Therefore, the distributed system builder requires different models of computation and more powerful tools.

2.3.1 Disjoint storage

A distributed system executes over several sites connected by a communications network. Each site has its own local memory. A part of a distributed system only has access to the data residing in the local memory in the site at which it is executing. information at other sites is not directly accessible. Any model of distributed computation must recognize this disjoint memory property and include some concept of modularity.

2.3.2 Partial failures

A system executing at a single site fails as a whole when the site fails. Any recovery can only occur once the site has been restarted. In contrast the

failure of one site will only cause part of a physically distributed system to fail. The other parts of the distributed system can take immediate recovery action and not wait for the failed site to restart. In fact, one of the advantages of physical distributed systems is that they can be made tolerant of site failures. However, to do so increases the complexity of distributed systems and a model for distributed computation should aid the writing of systems tolerant of host failures.

23.3 Parallelism

The model of computation most frequently used in non-distributed systems is a single thread of execution. Concurrency is usually restricted to system programs. A physically distributed system executes over several hosts and it is wasteful for there to be only one thread of execution. The computational model should allow the parallelism to be exploited within distributed systems and not restricted to use solely in system programs. The computational model should allow for the different threads of execution to be synchronized where necessary.

2.3.4 Communication

The parts of a distributed application must be able to interact. Therefore, the model of computation must include some concept of communication.

2.3.5 Decentralized control

The components of a distributed system may be autonomous with no single point of control. For both administrative and engineering reasons it may not be appropriate to impose a single control point. Therefore the model of computation must permit multiple management domains to exist and to provide structures in which management domains can cooperate.

2.4 Object-oriented computational models

This is the central concept of ANSA which allows the unification of many different distributed application areas. As the object model concepts have been widely used in research projects and products over the past 10 years, the ANSA Project can be confident it is suitable for distributed systems.

An object consists of an encapsulated data structure and a set of operations to modify and interrogate the data structure. The data structure can only be accessed via the operations associated with the object and is generally referred to as the state of the object. The definition of this set of operations forms the operational interface specification of an object. Operations may have parameters and results. When there are results, requesting an operation is similar to a procedure call. However, there is no shared data between objects. Therefore, a new primitive, called **invoke,** is introduced for starting operations.

Invoke causes the specified operation **on** the specified object to be executed with parameters and results passed where necessary. For synchronous interactions, the destination object returns a **normal result** or, alternatively, an **exception result** to indicate a reason for failure when the execution of the operation completes. This is the same interaction model found in Remote Operation Service. Asynchronous operations do not return normal results or exeption results and therefore any checks for completion of an asynchronous operation must be made by a subsequent synchronous interaction.

The lack of shared memory between objects makes the object-oriented model ideal for distributed systems. Further, the clear interface to an object is an asset in developing systems. The operational interface can be defined in the ASN.l /X.409 abstract syntax [CCITT X.ros, ISO 8824] or syntax integrated with a programming language. Clearly, the concrete syntax must be the same between communicating objects.

Synchronous operations are total in that exceptions are possible. This aids developing robust distributed applications as exceptional returns can be handled as cleanly as normal returns.

When an invoke request arrives at the destination object, the required operation is executed. This is similar to the execution of a procedure following a procedure call. Should the object already be executing an operation, the invoke request may be queued. Alternatively, another thread of execution may be started. The choice of single or multiple threads is determined by the object programmer. Multiple threads allow concurrency within an object.

The object-oriented computation model naturally permits concurrency between objects. Interaction between objects is only possible by invoking operations and operation execution only affects data belonging to the object in which the execution occurs. Therefore, executions in separate objects will not interfere with each other and can occur in parallel.

When an object is created, its availability must be announced to its potential users. This is achieved by publishing or **exporting** the availability of the new object to undertake operations defined by an operational interface specification. Before invoking an operation, the destination object must verify the availability and location of the new object. This is achieved by importing or referencing the information previously published.

2.5 Suitability of object-oriented models

The object-oriented model is a suitable candidate for physically distributed computation because it either overcomes the problems outlined in section 2.3 or provides help in solving them.

First, the object-oriented model naturally suits the disjoint memory property. The state of an object is only accessed by operations of that

instance. Therefore, all object-to-object interaction is by operation invocation. This maps naturally on to the communication service as a request message and a possible reply message.

Second, the data accessed by an operation execution is entirely local to one object so concurrency is permitted between separate objects. Concurrency can be further increased by permitting concurrent operation executions within a single object. In this case, the object programmer must provide concurrency control for access to the state. Finally, operations are can be made total by the use of exceptions.

The object-oriented model has other advantages. It is a proven software methodology which is useful for separating programming-in-the-small from programming-in-the-large. This is a consequence of the modularity between object and clear interfaces. Programming-in-the-small is concerned with programming the object representation needed to support its operational interface while programming-in-the-large is concerned with structuring object-to-object interactions to form a complete system by considering the operational interfaces and not their implementions.

Several of the techniques for programming physically distributed systems in an object-oriented style already exist. For example integration of object-oriented invocation into programming languages is straight forward as it is similar to a procedure call. Also, the protocol for remote operation invocation is very similar to remote procedure call [BIRRELL 84].

In summary, it is the opinion of the ANSA project that the object-oriented computational model is suitable for systems where the components of that system may be arbitrarily co-located or physically distributed. This opinion is reinforced by the large numbers of distributed computing research groups who have adopted an object-oriented style.

2.6 Support for object-oriented models

This section describes in abstract terms the support necessary for the objects forming a distributed system. Further details can be found in section 3. First, support is necessary for operation execution and access to state. This is provided by the local site at which the object instance resides. Also, support for operation invocation is necessary.

Figure 2.1 illustrates two objects *A* and *B* with object *A* invoking an operation on *B*. The object interaction support ensures the parameters are passed to operation B and the operation execution started. When the execution is complete, the object interaction support will pass back the results to *A,* In ANSA the term **distributed abstract machine** is used for the operations provided by the object interaction support. The distributed abstract machine in an idealized model of a distributed computer system; it provides the primitives for supporting the use of objects in a distributed environment.

Figure 2.1: Operation invocation

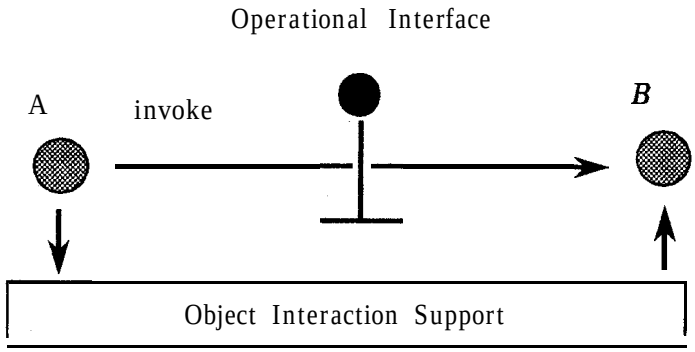
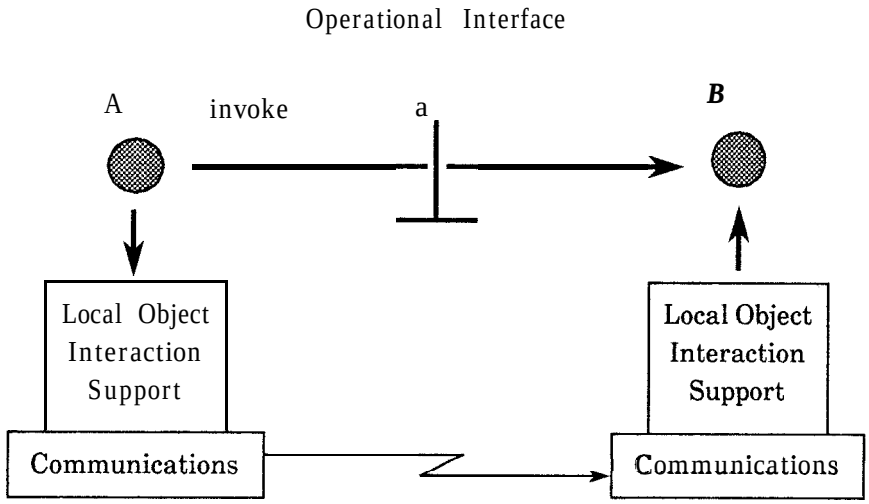


Figure 2.2: Remote operation invocation



When A and *B* are not co-located on the same host, their respective local object interaction support accesses the communication network to pass invoke requests and return parameters between them. However, objects are provided with the same interface specifications whether operations are invoked within a site or between sites. Thus the distributed abstract machine gives objects location independence. There are other forms of independence from the consequences of distributed which can be exhibited by the distributed abstract machine. These are discussed under distribution transparency in Part II, §3.

Figure 2.2 shows how the distributed abstract machine is realized: at each site in the distributed system some form of local object interaction support is built upon a basic communications facility. This local support is called the nucleus in ANSA. Interactions between objects at remote sites requires the use of communications between separate nuclei, although this is not visible to the objects themselves. Thus the distributed abstract machine gives objects independence from the details of the communications network.

The object-oriented computational model is applicable to the nucleus, but with the proviso that interaction can only occur with objects that are part of the same nucleus. The term local abstract machine is used to describe the support for the objects that make up a nucleus. One (at least) of the local objects in a nucleus must provide an operational interface to the communications network so that the nucleus can achieve the distributed functionality of the distributed abstract machine. Thus the local abstract machine gains objects independence from the details of the local environment (e.g. operating system) at a site.

The distributed abstract machine provides the same set of primitives as the local abstract machine, although with different constraints. This is an important feature since it enables objects designed for use in the nucleus to be re-used in the distributed system and vice-versa with resultant savings in design (and implementation) costs.

It is necessary to manage objects and structure applications. This is primarily achieved by managing the operational interfaces. Before object ***A*** can invoke an operation on object ***B***, it must be able to reference ***B***. When ***B*** is created it publishes it availability to perform operations specified by particular operational interfaces. A references this published information and imports knowledge of ***B***. Clearly by controlling which objects can publish availability and where, access to objects can be controlled.

3 WHAT ANSA PROVIDES

3.1 The nucleus

Editorial: outline local abstrcat machine for OS transparency, distributed abstract machine for distribution transparency.

3.2 Generic objects

Editorial: explain how the design of applications depends upon the sorts of transparency and quality attributes wanted and therefore the architecture provides a stock of generic objects to generate these attributes above the DAM.

3.3 Ancilliary objects

Editorial: explain how a number of particular objects such as catalogues and authentication servers are needed to support the generic objects.

3.4 Framework, rules and guidelines

Editorial: explain how the architecture has a framework for explaining how to put together the objects and make a system run. Framework is the box you put it in, Guidelines are rules about how you take it out again.

3.5 Limits to ANSA

Editorial: put a lower boundary on the nucleus (= LAM) - MPS, local coordinator, memory management, synchronization, protection, etc and an upper bound on the generic objects in terms of distance from actual applications.

Advanced Networked Systems Architecture
ANSR

24 Hills Road
CAMBRIDGE
United Kingdom CB2 1 JP

TELEPHONE: Cambridge (0223) 323010
INTERNATIONAL: + 44 223 323010
TELEX: 817343 BLUCAM G

ANSA Reference Manual

Part II: Technical Assumptions

Abstract:

This document is Part **2 of the ANSA** Reference Manual Release 0.0.

This part outlines the fundamental assumptions about future technology directions that have been taken in the development of ANSA,discusses assumptions about key architectural issues for distributed systems, based on a simple distributed system model, a survey of the research results that have contributed important technical concepts to ANSA, and assumptions about the implications of open standardization for ANSA.

Much of this section is derived from an ECMA/TC32-TG2 contribution to ISO on "Proposed Technical Assumptions for ODP" and PR.13 *ANSA:* Approach to ***Distributed Systems.***

Number: **A0.48.00**

Date: **9th January 1987 11: 16 am**

Area: AO **(Architectural Overview)**

Status: **0** **(Open Document)**

Classification: **U** **(Unrestricted)**

Distribution: **G** **(General Public)**

Editorial history:

Issue	Notes
01	Open Document status in Release 0.0 Approved AJH 9/1/87

1 GENERAL

1.1 Introduction

OSI has provided standards for interconnection between systems. The ANSA project expects that the next phase of standardization will provide open standards for building distributed systems. These standards will emphasize integration between diverse applications and across diverse technologies.

Table 1.1 lists several important fields of application for information technology. Previously, such fields had relatively little technology in common. With modern developments, a common technological core spanning all such fields of application has emerged. Its fundamental basis is the transformation of all kinds of information into and out of digital signals, which can be stored, communicated, manipulated and interpreted.

Table 1.1: Some fields of application for information technology.

Administrative Systems
Business Management
Command and Control
Engineering Computation
Factory Automation
Image Manipulation
Knowledge Engineering
Radio /TV / Hi-fi
Office Systems
Process Control
Scientific Computation
Telecommunications
Technical Design

This common core of information technology draws on the principles and practices of:

- computer science
- systems design
- software engineering
- hardware engineering
- communications engineering

ANSA is based on the assumption that these areas of information technology have reached a point of convergence, such that it is possible to construct a corpus of distributed system concepts sufficient to achieve a near universal basis for information systems.

1.2 Scope

This discussion of technical assumptions for ANSA is part II of the ***ANSA Reference Manual*** and is broken into four sections:

(1) the technical changes and advances in information technology that will have greatest impact upon distributed systems

(2) the key architectural issues arising from distribution

(3) a survey of the research results that have contributed important technical concepts to ANSA

(4) the implications of open standardization for ANSA.

2 A NEW PARADIGM

2.1 Introduction

This section considers some technical advances which theANSA project asserts must affect the scope and content of standards for distributed systems. These advances are grouped below into three areas: programming languages, communications and multi-media systems.

The pace of change in information technology is very rapid, and probably nowhere more so than in the field of distributed processing. The impact of the changes have resulted in "paradigm shifts".

A paradigm shift occurs when changes overwhelm the basis for some existing pattern of thought and a different pattern takes its place. As explained in [KUHN 70], people naturally have great difficulty in recognizing and accommodating to such revolutionary changes in thinking.

When there is a paradigm shift, it is important to question previous assumptions when examining new ideas.

2.2 Programming languages

The changes considered under this heading are a combination of three factors:

- **language oriented approach -** distributed processing needs to be considered essentially in terms of application system design, programming-in-the-large, language structures, and integration into language libraries and compilation systems
- **type oriented approach -** there is an inexorable move into richly typed systems. At the leading edge of this are the object oriented approaches to system design. Advanced computer languages are strongly typed. Document architectures [ISO 8613] are another kind of richly typed structure
- **novel processing -** new languages and processor architectures will be important in the lifetime of ANSA. Examples include: non-procedural languages and reduction machines specialized for processing them; systems with very high parallelism via multi-processor architectures and lightweight processes; reduced instruction set computers (RISC) with special relationships between compilers and the runtime environment

2.3 Communications

Traditionally networking has mostly been in terms of data transmission at kilobit rates (e.g. over the PSTN). In the lifetime of ANSA, transmission rates orders of magnitude higher will come into general use. Moreover, the inherent reliability of networks will continue to increase.

Some of the implications for distributed system standards are:

- **distance independence -** the combination of general connectivity, increased transmission rates and lower cost thresholds is reaching a transition point, beyond which the use of distributed processing becomes essentially distance-independent and universally possible (as is already the case for postal, telephone and telex interactions)
- **system performance -** high transmission rate communications remove the traditional communications bottlenecks. The limiting performance factor then tends to be bottlenecks inside the endpoints of the communication and intermediate gateways (e.g. process context switching, scheduling overheads, buffer copying, disc access latency)
- **voice / data -** high speed digital networks typically support anisochronous data communication and isochronous voice communication together. This has important systems implications, particularly where voice, image, graphics, text and data form part of the same interaction
- flow **control -** rate-controlled communication is inherent in isochronous communication. It also becomes desirable for high transmission rate anisochronous communication. This is because the communications storage factor (speed **X** delay) is inherently large, and the traditional back pressure flow-control mechanisms used for data-communications do not perform well in such circumstances (due to over-running the receiver and consequent oscillations)

2.4 Multi-media systems

The traditional approach to distributed processing is exclusively in terms of data interactions (and data-encoded text or graphics); for example transaction processing and remote database access.

The integration of processing, storage, transmission and use of multi-media information is likely to become widespread during the lifetime of ANSA. The main reasons are:

- market pull - most kinds of information-related activity involve mixtures of data, text, pictures, graphs, diagrams, charts, the spoken word and other sounds. People naturally work with multiple media, and have always been able to do so until information technology raised artificial barriers to this. The consequent latent market pull is waiting for these barriers to come down
- technology push - some of the technology needed for integrated multi-media distributed processing is already visible (multi-media document architectures, advanced workstations, desktop

publishing, optical storage, and integrated services LANs and WANs (e.g. ISDN and FDDI 2)

- **industrial convergence -** information technology is leading to the world-wide convergence of the data processing, office systems, telecommunications, consumer electronics and information services industries. Multi-media integrated distributed systems are inherent in this trend

3 DISTRIBUTED SYSTEM ISSUES

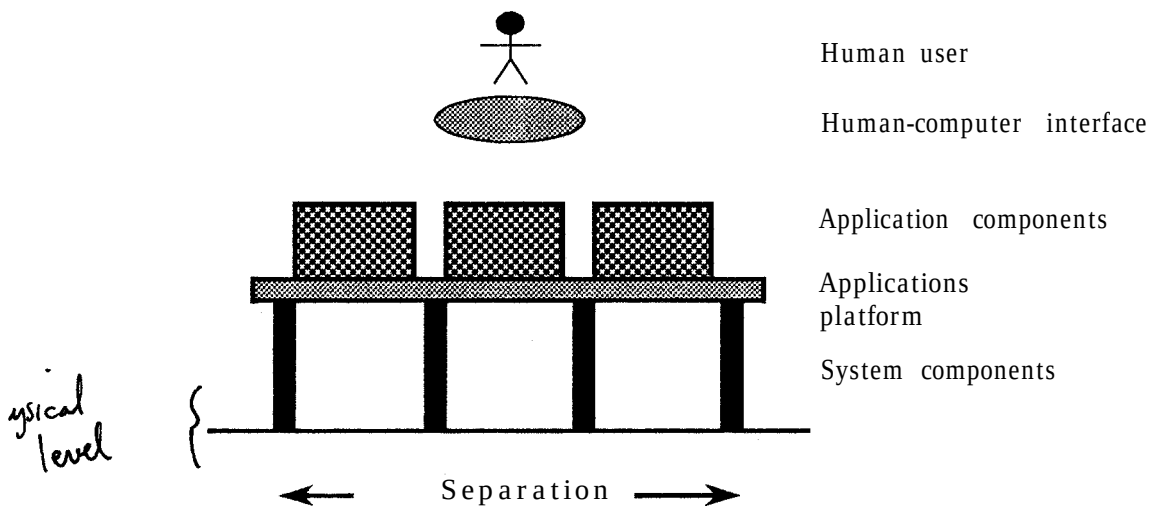
3.1 Introduction

This section states assumptions about the fundamental properties of distributed systems adopted in ANSA.

3.2 Separation

Figure 3.1 shows a simple model of a distributed system. In this model, system components, which are built up from physical resources and software, provide a supportive platform for application components; application components are specialized for specific user needs and designers will exploit the applications platform and use whatever system components are necessary to build the application components. This supportive platform is called the **distributed abstract machine** since it is an idealized abstraction of a distributed multi-computer system. Users perceive and interact with the system through a human-computer interface.

Figure 3.1: Simplified model of a distributed system



3.2 Separation

The key to understanding distributed system architecture is to consider the inherent separation of components in distributed systems.

- Separation requires explicit communications between interacting system components since their storage is disjoint
- Separation allows for the use of truly parallel execution in a distributed system
- Separation allows for the possibility of independent component faults and recovery from such faults

- Separation allows for the use of isolation as a method of enforcing security policies
- Separation allows for the incremental growth or contraction of a system, through the addition or subtraction of individual components

From these considerations, two questions emerge. The first is the how should the consequences of distribution (explicit communications, independent failure and parallelism) be reflected through to application components? The second question is how should the features of distribution be exploited to achieve desirable quality attributes (e.g. parallelism for efficiency, recoverability for resiliance, isolation for security, incremental change for scaleability)?

3.3 Distribution transparency

A major system and application design issue is whether or not to hide distributedness and its consequences within the distributed abstract machine. The term **distribution transparency** is widely used for discussing the visibility of distributedness within distributed systems.

- **Arguments for transparency.** It can be advantageous if all the consequences of distribution are hidden. This hides complexity, simplifies the task of applications designers and enhances the re-useability of system components between systems. The evolution of existing products based on centralized systems is inherently straightforward. Successful experiments with transparency are Unix United [BROWNBRIDGE 82] and the LOCUS distributed operating system [WALKER 83].

- **Arguments against transparency.** Full transparency, which completely hides distribution, can be relatively expensive in terms of the underlying implementation effort and performance overheads. Moreover, it denies designers the opportunity to exploit the consequences of distribution via decentralization, explicit fault, management or replication of control, or data, or both.

Thus transparency is not always necessary, and relaxing the degree of transparency enables new kinds of applications to be constructed. System design choices lead to different, transparency requirements. ANSA structures these choices and does not pre-empt them.

3.4 Kinds of distribution transparency

It is well established that transparency is made up of a number of separate elements [POPEK 81] which are described here in terms of the conditions necessary to achieve full distribution transparency:

- **access transparency -** concealing the use of communications when accessing remote resources (such as programs, data and devices)

- **location transparency** - enabling the use of a resource, independent of the placement of that resource in the distributed system
- **migration transparency** - enabling the migration or reconfiguration of resources in a distributed system
- **replication transparency** - enabling the use of multiple instances of a resource for such purposes as enhancing dependability and performance
- concurrency transparency - avoiding inconsistencies due to parallel execution by using concurrency control techniques
- fault transparency - concealing faults by using error processing techniques
- performance transparency - minimizing the performance penalties associated with using remote resources
- scaling transparency - concealing variations in system behaviour due to scaling up to large or busy systems, and scaling down to small systems

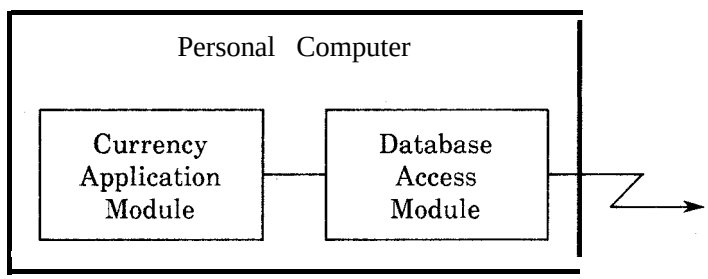
3.5 Transparency examples

An illustration of distribution transparency issues is shown in the following examples.

Imagine a network of small personal computers used by a group of currency dealers. The dealers need to share access to a simple database of currency prices and deals in progress.

The structure of the program in each personal computer is shown in Figure 3.2. There is a database access module that manages access to the shared database and a currency application module.

Figure 3.2: Example program structure



The currency application module should not be influenced by the distributed nature of the system, and this requires that the database access module should provide full distribution transparency.

Different strategies will be adopted for the database access module, depending upon how the system as a whole is structured.

- **centralized structure:** keep the database on a central database server which manages concurrent access to the database. In this structure the database access module has only to handle the communications with the server (i.e. access transparency)

- **partially distributed structure:** keep the database on a central 'remote disc' server. The database access modules must now cooperate with one another to preserve consistency of the database by setting and releasing locks (i.e. concurrency transparency) in addition to handling communications (i.e. access transparency)

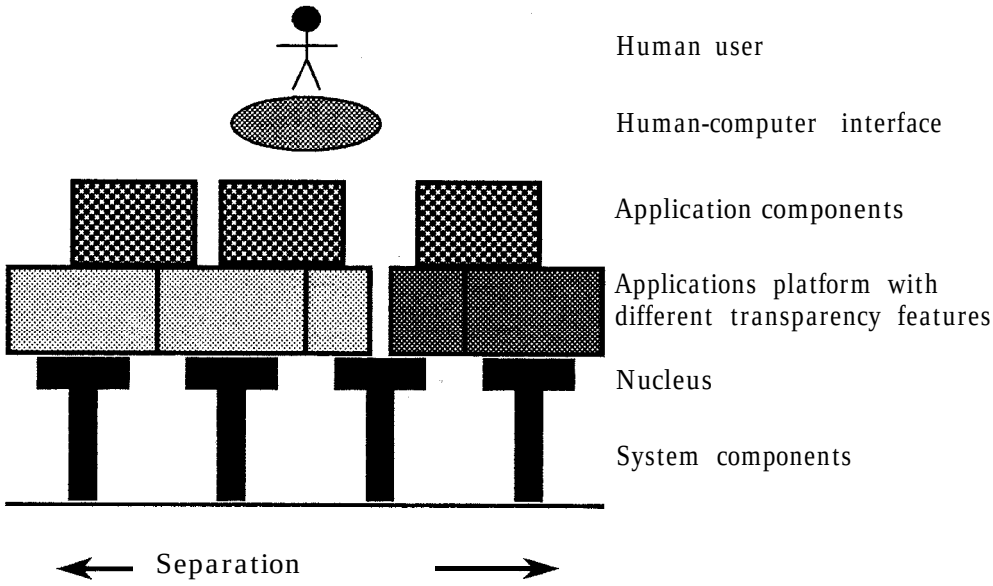
- **fully distributed structure:** store a copy of the database on the local disc of each personal computer (i.e. replication transparency) and arrange that the copies of the database are kept in step by using, for example, Birman's 'Bulletin Board' protocols [BIRMAN 86]. Since the database is replicated, it is possible for the system to remain in operation despite the failure of individual copies (i.e. fault transparency). The database may be too large to store on each local disc, in which case it can be partitioned and each local disc keeps only some fraction of the whole (i.e. scaling transparency). Responsiveness will be enhanced if each computer stores the partitions that the dealer uses most frequently (i.e. performance transparency), and if the partitioning can be reconfigured when usage patterns change (i.e. location transparency and migration transparency)

From these structures it is evident that as the degree of distribution increases, the database access module has to provide a greater number of transparency attributes to meet the transparency requirement of the currency application module. ANSA helps the system designer achieve these transparency attributes.

A revised distributed system model is shown in Figure 3.3 to illustrate distribution transparency in ANSA. Every system component in the figure is shown augmented by a basic set of distributed system functions, which are known as the ANSA nucleus. The nucleus is the set of essential functions necessary for a system component to be part of an ANSA-conforming system. The nucleus builds upon the local functions of each system component to provide the distributed abstract machine. These local functions are termed the local abstract machine.

Above the nucleus there are additional functions to provide the different sorts of transparency required for different sorts of applications.

Figure 3.3: A distributed system with selective transparency



3.6 Quality attributes

The commercial and technical viability of future distributed systems will be critically dependent on quality attributes. It should be possible for system designers to achieve high levels for selected quality attributes and to permit trade-offs between quality attributes and other commercial factors such as cost.

The following quality attributes are particularly stressed in ANSA:

- **efficiency -** every improvement in performance with a given resource is a potential increase in applicability
- **dependability -** every improvement in security, reliability, resilience and availability is a potential increase in applicability
- **scaling -** every improvement in the ability to scale up for large distributed systems and busy nodes and to scale down for small systems is a potential increase in applicability

For all such quality attributes there are also threshold levels which have to be achieved for viability in particular fields of application (e.g. for process-control applications there are critical requirements for response times, continuous operation and stability at peak loads).

The experience of the ANSA project is that achieving these quality attributes dominates system design and therefore these attributes dominate the architecture.

3.7 Generic functions

ANSA is supportive a wide range of independently selectable transparency and quality attributes. This is done by recognizing two sorts of function. First, there are primitive functions that are meaningful in all systems, no

matter what attributes are selected. Second, there are **generic** functions that can be applied to the primitive functions in order to operate the primitive function at particular transparency and quality attribute settings.

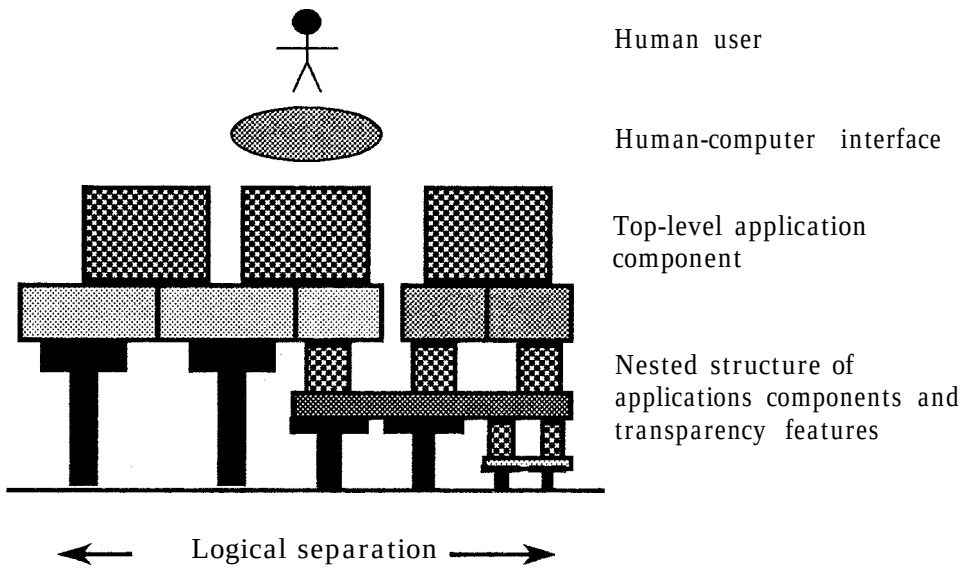
Primitive functions are concerned with what goes on in distributed systems; they are the building blocks for applications of distributed systems.

Generic functions are concerned with how distributed systems operate and are selected on the basis of system design policies relating to transparency, efficiency, dependability and scaling. These concepts are explained in [RANDELL 86].

3.8 Nesting

The distinction between system components and applications in the distributed system model of Figure 3.3 is artificial. An application component, which is specified as a composition of ANSA primitive functions and generic functions can be used as a component of a higher level distributed system. This concept, known as nesting, is illustrated in Figure *3.4.*

Figure 3.4: Nesting in an ANSA-conforming distributed system



The top-level system is made up of a combination of concrete system components and nested distributed sub-systems, each with ’different transparency selections. It is important to note, however, that at the bottom of every nested sequence there has to be a concrete system component to provide contact with the physical world.

The nesting may be a result of the application of ANSA to different technologies. For example, a system may include multi-processor systems

attached to LANs, which in turn are interconnected via WANs. Different transparency and quality attribute requirements may exist at each level.

Alternatively nesting may be the result of embedding one application within another. For example, a factory automation system may include an office automation component, a CAD component and manufacturing cells.

The nesting concept is valuable to ANSA in three ways.

Firstly, it demonstrates how the architecture can enable interworking between different specialized sub-systems. The solution is to build structures above each sub-system using ANSA primitive and generic functions until a common level of capability is achieved. Thus the applications of the individual sub-systems become embedded components of the higher-level system.

Secondly (and really the first argument in reverse), the system designer can design a system top-down and then look individually at system components and design each of them in turn as ANSA-conforming sub-systems.

Thirdly, nesting provides a route for coping with evolution from existing systems to fully ANSA-conforming systems. The system designer can place the applications platform above existing products and use ANSA to generate missing parts. Old products can be selectively removed from the structure as new ANSA-conforming products emerge and further interworking will be gained at lower levels in the structure. By a similar argument, the nesting concept enables system components from alien architectures to be incorporated into ANSA-conforming systems.

4 SURVEY OF RESEARCH

4.1 Introduction

There is a large body of techniques that have been developed and exploited in the distributed systems research community that have yet to appear in standards. These techniques have originated from a considerable experience with system design [LAMPSON 83] and are well matched to the technical requirements of ANSA.

4.2 Remote procedure call

Research into distributed systems has focussed on bringing software engineering techniques to bear on the problems of building, operating and managing distributed systems.

A major breakthrough came with the discovery of **remote procedure call (RPC)** [BIRRELL 84] as a way of providing a link between programming languages and communications via process-to-process communications. In an RPC system one program (the client) can call a procedure, defined at the language level, which is executed by another, potentially remote, program (the server>. An RPC package consists of three parts:

- a transfer protocol
- a runtime library
- a program module linker

The transfer protocol is responsible for the transfer of request and response messages between client and server. RPC protocols are optimized for short response times through minimization of packet exchanges. This is in contrast to traditional transport protocols which are optimized for continuous bulk transfer. RPC protocols mask communication failures so that in the absence of catastrophic failure remote calls are executed exactly once (i.e. like local calls). RPC protocols are based on simple recoverable connections with ultimate responsibility for recovery vested in the client.

The runtime library consists of two parts: marshalling and dispatching. Marshalling is the process of taking the arguments and results of a procedure call and assembling them into packets for transmission over the transfer protocol and then disassembling them. RPC is optimized for minimizing the complexity and overheads of buffer management to improve reponsiveness. Dispatching is the process of selecting the correct procedure to invoke on receipt of a remote call. Despatching relies upon a binding being set up between client and server. A server exports information about the procedures it offers and a client imports this information. The imported information specifies procedure identifiers to be inserted in requests for decoding by the despatcher.

The job of the program module linker is to replace calls of remote procedures by calls on the local marshalling routines and transfer service automatically

without explicit programmer involvement. These generated calls are often called **stubs.** The linker also generates the runtime binding information needed for imports and exports using programming language modularization features to delimit client and server procedures.

The major feature of RPC is that it allows the programmer to construct application protocols in terms of calls upon remote procedures and provides great flexibility in the configuration of distributed applications since the choice of co-location or remoteness for sets of procedures can be deferred. There is a close equivalence between RPC and the OSI Remote Operations concepts [CCITT X.410].

A notable feature of the Birrell RPC protocol is that it is very well integrated with an encryption system to provide secure communications [BIRRELL 85].

Since procedure calls wait until a result is returned, RPC systems are usually based on a lightweight process structure so that many remote procedure calls can be active simultaneously in systems where asynchrony is required.

4.3 Consistency

RPC and similar techniques have made possible the programming of distributed (i.e. multi-site) programs. This has led to much research into the problems of **consistency.** In a distributed program there is true parallelism and the execution of processes will overlap. **Correctness** requires that each process should see a consistent view of the internal data structures and state of the program and therefore the parallelism must be constrained.

Much work has been done on **transaction-based systems** using the concept of transactions from the database world [GRAY 79]. In the ARGUS system [LISKOV 82] sequences of program statements, including calls of remote procedures, can be labelled as **atomic actions.** An atomic action is 'all-or-nothing' in effect. The ARGUS compiler and runtime system are jointly responsible for providing stable storage, managing read/write locks and running two-phase commitment protocols to achieve atomicity. Atomic actions in ARGUS may be nested so that a programmer can build atomic actions around any sequence of statements, including nested sub-actions.

Following on from the ARGUS work other research groups have recognized that greater parallelism can result by using application-oriented locking strategies rather than by nested compositions of read/write locks. An example is the TABS system [SPECTOR 85]. An alternative approach to parallelism is that of optimistic concurrency strategies where processes are allowed to proceed until a conflict is detected and recovery invoked [KUNG 81].

Another feature of distributed programs is the potential for replicating parts of the program to increase dependability and performance. Birman's ISIS system [BIRMAN 85] provides an efficient implementation of resilient objects. If one of the resilient objects in the set fails, or becomes overloaded,

another in the set will take over. The system is based on a suite of optimized **atomic broadcast protocols.** A similar scheme is illustrated by Cooper's replicated distributed programs [COOPER 85] which is an RPC system that supports active replication of both client and server for increased dependability. The treatment of replication in distributed systems is now being made systematic by the recognition that the style of interaction between replicates is **advisory,** in comparison to the **imperative style** of client-server interactions. There are strong linkages between these techniques for replication and the techniques employed in fault-tolerant systems [LAPRIE 85].

4.4 Operating systems

As well as language oriented developments there have been many innovations in operating system technology to accomodate distribution.

The ACCENT system [RASHID 81] is an example of a network operating system. The local inter-process message system of the ACCENT kernel is extended across a network by a 'network process'. Remote services appear as local ports to the network process. When a message is sent to such a port via the ACCENT kernel, the network process packages up the message and transmits it to the network process at the remote site. The remote network process then extracts the message and sends it to a local port at that end where the message is processed. Thus the network process is responsible for bringing the remote ports into the local address space and for isolating the inter-process communication system from the details of network communications. Ports are treated as protected entities by ACCENT so that access to remote services can be controlled.

The V-system [CHERITON 84] has followed a different approach and put the communications in as an integral part of the kernel to optimize performance. Because of the efficiency of its communications, the V-system is able to use the local area network for page swapping between disc-less workstation and file servers. The V-system includes the notion of a 'process group' as a set of processes that can be addressed as a single entity, even if they are distributed across several sites. This notion therefore provides system level support for some of the replication techniques mentioned in ***94.3*** above.

A number of operating systems have been based on the object-oriented model of computation since the disjoint address spaces of multiple sites matches the 'encapsulation of state' concept that underpins 'objects' [JONES 79]. The best known example of this approach to operating systems is Eden [BLACK 85]. All programs in Eden are objects with well-defined external interfaces. Interaction takes the form of one object invoking an operation at another, using RPC-like protocols. In this systems-oriented perspective, the emphasis of the object-oriented model is on the interactions and relationships between objects rather than on the details of objects themselves and their execution, as is the case in programming language perspectives on the object-oriented model. In Eden objects have logical addresses so that they can be accessed without knowledge of their location, enabling dynamic reconfiguration of the

system. Eden specifies a number of generic functions that can be applied to any object. These generic functions are mostly to do with various aspects of system management and permit management of many different sorts of object to be unified.

The Cambridge Distributed System [NEEDHAM 82] explored the possibility of dynamic instantiation of services upon demand using a pool of uncommitted processors. Requests for service are directed to a resource manager which finds an appropriate free processor, loads it with the required service and transparently reconnects the user to the newly made service. The operation of this processor pool is dependent upon remote debugging, automated service management and security.

The LOCUS system [WALKER 83] is an example of how a derivative of Unix can be implemented as a distributed operating system. The LOCUS kernel goes to great lengths to insulate the applications programmer completely from any of the effects of distribution. This feature, called distribution transparency, has the great merit that applications previously written for ordinary Unix can run unchanged in the distributed environment, which offers greater performance and dependability than a single node Unix system. The negative aspect of complete transparency is that new applications cannot exploit distribution and that system management (particularly failure diagnosis and reconfiguration) are in conflict with transparency. There has been much debate on what kinds of transparency should or should not be provided in distributed systems. This debate reinforces the recommendation in §3.2 that standards be flexible enough to support a variety of transparency attributes.

4.5 Protocols

Many aspects of protocol design have been revisited using systems engineering techniques rather than traditional communications engineering. The application of the 'end-to-end' principle [SALTZER 84] has led to a focus on reducing buffer management and processing overheads at network nodes. The outcome has been a move away from strict layering of protocol implementations, and the simplification of protocols so that they can be moved out of processors into micro-processor network interface units. Simplicity enables small machines, such as personal computers, to support complete implementations of the protocols.

At the present time attention has been given to the requirements of very high bandwidth networks (e.g. 100Mbits/sec LANs) and high bandwidth, long delay networks (satellite channels). In these networks, many of the assumptions that are central to traditional protocols are being undermined. Fast networks can bombard a node with data faster than the data can be processed. By the time the situation is recognized and the processor reacts, the amount of data 'in transit' may be immense, leading to severe buffering problems, especially at intermediate gateways, and instability in congestion control algorithms. Preliminary work suggests that rate controlled protocols which avoid over-committing network nodes will be more stable

and achieve better throughput [CLARK 86]. This work has many implications that have yet to be completely explored.

4.6 Multi-media integration

Many networking technologies now have the ability to transport both isochronous forms of information (voice, video) and anisochronous information (image, graphics, text, data). Several systems have been built that provide for interaction that handles all these forms of information in an integrated fashion, for example within a conferencing application [FORSDICK 85], [AGUILAR 86].

The requirements for integration of both these forms are stressful for both communications and processing. Progress in this area is predicated upon real-time performance guarantees from networks, processors and operating systems and consequently many of the techniques described above will be essential.

4.7 Heterogeneity

Considerable attention to the problems of accomodating heterogeneous systems on a network, since the different specializations of computer science have different preferences for computers and languages. Moreover projects in different areas wish to share resources and even engage in cooperative work. The common theme has been to provide basic services to support links between 'islands of homogeneity'. For example there have been many designs of system-independent file servers [SVOBODOVA 84] and work on directory services and authentication [BIRRELL 86] based on practical experience with operating large distributed systems.

4.8 Security

Most attention in the area of security has been directed towards the use of encryption for communications security. A substantial survey is given by Voydock and Kent [VOYDOCK 83]. Encryption has been used as a means of 'sealing' data in authentication protocols [NEEDHAM 78] as well as being merely a means of achieving data integrity. Encryption has been made an integral part of RPC protocols to defend then against a wide range of network level attacks [BIRRELL 85]. The design and operation of authentication services [BIRRELL 86] has been explored in some detail. Work has also been done on the exploitation of separation in distributed systems to achieve isolation and enforcement of security policies [RUSHBY 831.

4.9 Large systems

The research community has not confined itself to laboratory-sized systems. Many of the research systems have grown to considerable size and operate in service on a daily basis. The best example of this take up is the GRAPEVINE mail system on the XEROX Internet [BIRRELL 82] [SCHROEDER 841 which provided many lessons for future systems. Several

research projects have been based on extensive wide area networks including the ARPA network in the USA and satellite systems [LESLIE 84].

A number of academic institutions are engaged in setting up large distributed computer workstation networks as an integral part of the institution's infrastructure to support teaching and research, for example Project ATHENA at MIT and the Information Technology Center at Carnegie-Mellon University [MORRIS 86].

5 STANDARDIZATION ISSUES

5.1 Introduction

Previous sections, have addressed the technical basis of ANSA. In this section, assumptions about open standardization of ANSA are listed. These assumptions imply constraints on ANSA. The importance of evolution from existing standards is emphasized.

5.2 Fundamental assumptions

ANSA is based on these fundamental assumptions for standards:

- **multi-vendor goal -** suppliers, users and regulatory authorities are generally committed to the principle that information systems should conform to international standards which facilitate the inclusion of system components from multiple independent vendors

- **standardization process -** suppliers, users and regulatory authorities are generally committed to supporting the development and promulgation of, and conformance with, the open standards necessary to achieve the above multi-vendor goal

- **standardization scope -** this standardization necessarily includes not only standards for interworking between physically separated subsystems, but also standards for relevant kinds of interfaces between physically co-located components of distributed systems

- **timeframe -** the timeframe for achieving this multi-vendor goal for ANSA is the next decade, because of the lead time for the development of the necessary additional standards

- unification - to achieve the multi-vendor goal, there is necessarily a unified kernel of generally applicable distributed processing standards, which includes a coherent basis for accommodating any necessary diversity across fields of application and across technology variations

- competitiveness - the open distributed system standards should be such that the cost and benefit of distributed information systems conforming with them is comparable with using alternative proprietary architectures

- technical feasibility - the field of distributed systems is now sufficiently mature for this standardization to be technically feasible

5.3 Evolution

It is essential that distributed system standardization builds on existing standardization work and makes effective use of existing standards and investment in them by industry and users.

It is inevitable that there will be some discontinuities from existing standards work, if only because of the different scope and field of application. But the new standards should be evolutionary for applications and equipment using existing OSI standards.

It is a fact that most networking and distributed processing today is achieved via networking architectures and products which are under proprietary control. It is certain that the importance of these various architectures will persist long into the future, if only because of the large and continuing successful investment in them by industry and users. Therefore, as a practical matter, the open standards should include appropriate provisions for co-existence and appropriate migration paths.

Advanced Networked Systems Architecture

ANSR

24 Hills Road
CAMBRIDGE
United Kingdom CB2 1JP

TELEPHONE: Cambridge (0223) 323010
INTERNATIONAL: + 44 223 323010
TELEX: 817343 BLUCAM G

ANSA Reference Manual

Part III, 1: General

Abstract:

Thisdocument is Part III, Section 1 of the ANSA Reference Manual Release 0.0.

Thissection introduces the concepts and definitions given in Part III.

Number: A0.49.00

Date: 9th January 1987 12:05 pm

Area: AO (Architectural Overview)

Status: 0 (Discussion Paper)

Classification: U (Unrestricted)

Distribution: G (General Public)

PART 3: Concepts And Definitions

Editorial history:

Issue	Notes
01	Discussion Paper status in Release 0.0 Approved AJH 9/1/87

1 GENERAL

Editorial: This introduction (indeed the whole of Part III)'focusses heavily on the computational model projection of ANSA. It will need revision and additions when the other projections of objects and frameworks are developed through work on Part IV).

1.1 Introduction

This description of concepts and definitions forms part III of the ANSA Reference ***Manual*** and provides the technical basis necessary to describe and specify ANSA in detail.

This part is developed in terms of the distributed system concepts of separation, transparency and quality discussed in part II.

1.2 Scope

This part of the **ANSA** ***Reference Manual*** begins by explaining the object-oriented computational model adopted for ANSA. The model is introduced in three stages:

- a description of typing and encapsulation concepts that provide a static object model for describing the components of a system and their configuration
- a description of naming, binding and invocation concepts that enable objects to interact with one another and achieve communication; this is a dynamic or behavioural model for objects
- a description of abstract machine concepts which mechanize the dynamic behaviour of objects

The model is correlated with the real world of hardware, software and users through concepts relating to three environments:

- a physical environment of real processors, storage, I/O devices and communications networks
- a logical environment of software concepts for use within the ANSA nucleus
- a user environment for interaction between human users and external systems with ANSA-conforming systems

The remainder of this part builds upon the basic object model to define concepts relating to:

- distribution transparency
- security
- stored data
- systems management

1.3 Format

The definitions of the computational model are grouped into self-consistent sets, with each set relating to a single major concept. The concepts are progressively developed throughout part III as a whole.

Each concept is described under three headings:

- **definitions**
  which provides definitions for technical concepts
- **usage**
  which explains how or where the concept in used in ANSA
- **discussion**
  which explains the principles and relevance of the concept to ANSA

Italics are used in definitions to indicate the use of technical terms defined later on in the part.

Advanced Networked Systems Architecture ANSR

24 Hills Road
CAMBRIDGE
United Kingdom CB2 1JP

TELEPHONE: Cambridge (0223) 323010
INTERNATIONAL: + 44 223 323010
TELEX: 817343 BLUCAM G

ANSA Reference Manual

Part III, 2: Types and Objects

Abstract:

This document is Part III, Section 2 of the ANSA Reference Manual Release 0.0.

This section presents concepts and definitions for a static computational model of objects.

Number: A0.50.00

Date: 9th January 1987 11:54 am

Area: AO. (Architectural Overview)

Status: 0 (Open Document)

Classification: U (Unrestricted)

Distribution: G (General)

Editorial history:

Issue	Notes
01	Open Document status for Release 0.0. approved AJH 9/1/87

2 TYPES AND OBJECTS

2.1 Introduction

This section introduces the static aspects of the ANSA object model. It is concerned with the use of objects as a system structuring and classification methodology.

Objects are explained starting from the concept of 'datatype' that originated in programming languages. The concept is widened to cover all sorts of digitally encodable information (video, image, voice, text and graphics) as well as data. The term 'information type' is used in ANSA to emphasize this extension.

Types are general classification concept based for sets of things have similar characteristics. Type theory has used outside of ANSA, indeed many abstract nouns - red pens, fast cars - are examples of types.

Objects focus on an operational view of systems - what systems do, rather than on how they go about it. Therefore objects are primarily characterized in terms of operations, or the jobs they do in a system.

Objects therefore provide a modularity concept for systems in which implementation detail is hidden by abstraction in terms of operations.

Designs for objects are necessary in order to be able to build systems; designs tell the implementor how to build modules that implement the operations of an object. Systems using these designs are based on abstract machines that enforce the encapsulation necessary to treat modules objects and that mechanism operations on objects via the modules they encapsulate.

2.2 Type concept

Type = {characteristic1, charac2, ... ch

2.2.1 Definitions

Type: a set of characteristics that distinguish any particular identifiable set of items.

ANSA type set: the set of all types defined in ANSA. $TS^{ANSA} = \{type1, type2, \ldots type$

Instance: an element of a set of defined type. $x \in TS^{ANSA} = \{type \mid type$ is defined

Immutable type: a ~~types~~ whose instances have fixed values. (≡ constant?)

Mutable type: a type whose instances may have variable values within the bounds permitted by the type characteristics.

Type specification: the formal description of the characteristics that distinguish a type.

Type completeness: assurance that all items in a domain of discourse are items of known types.

Specification completeness: assurance that a type is fully determined by its specification.

Type safety: assurance that instances are always consistent with their type specification.

Static type safety analysis: the process of assuring type safety through static analysis of type specifications.

2.2.2 Usage

Particular typing characterizations are used for different aspects of the ANSA object model. For example information types are characterized by structure; objects are characterized by operations. Use of the term "type" without these distinctions may be misleading, and should be avoided. Collectively these characterizations comprize the ANSA type set.

In succeeding sections definitions will be given in terms of types rather than instances.

If two items have identical characterizations, they belong to the same type (since types comprize a set).

By similar argument, there cannot be duplicate instances of a type. Distinct items must differ in at least one characteristic.

Interaction between objects in ANSA satisfies type completeness, enables type safety and is optimized towards static type safety analysis.

The type specifications used in ANSA are machine processable.

2.2.3 Discussion

A comprehensive discussion of type concepts and related programming paradigms is given by [CARDELLI 85].

There is an inexorable movement to richly typed systems. At the leading edge of this is the object-oriented approach to system design. Advanced computer languages are strongly typed. Document architectures [ISO 8613] are another kind of richly typed structure. The most successful type systems are those which exhibit type completeness.

The use of type concepts has now reached a transition point which is profoundly important, both philosophically and practically:

- **previous view**
  the world is populated by values, and that types are an explanatory mechanism introduced for the purpose of classifying and managing values
- **subsequent view**
  that types are the basic conceptual entities, and that values exist only in so far as they are constructable from some type

The latter view is appropriate to the timeframe for industrial exploitation of ANSA. Therefore ANSA specifications are at the type level rather than at the instance (ie value) level.

Type safety is an important assurance in system design. The question arises of where in the design and development cycle should type safety be analysed?

In ANSA, static type consistency is emphasized for three reasons:

- **correctness**
  it is important to be able to confirm the correctness of specifications and implementations before a system goes into operation, and this requires automation for complex systems. Compiler and verification technology is oriented towards static analysis. Moreover, dynamic checks can be inserted into systems where static analysis is not possible
- **efficiency**
  static analysis occurs no overheads during operation; the costs of assuring type safety only occur during development
- **productivity**
  static consistency depends largely on declarative structures. This facilitates the use of higher level languages for specification and programming, and automated tools, such as program generators, for the design, construction, maintenance and validation of systems

2.3 Information type concept

2.3.1 Definitions

Abstract syntax: the formal grammar used for describing the type of an information item without determination of its representation or value.

Encoding rules: a series of rules for determining the representation of an information item as a finite string of binary digits from its abstract syntax and value.

Information type: a characterization of a set of values in terms of their abstract syntax and associated encoding rules.

Primitive information **type:** an information type whose information type is a terminal symbol of the abstract syntax.

Constructed information **type:** an information type whose abstract syntax is a non-terminal symbol of the abstract syntax.

2.3.2 Usage

These concepts are applicable to all information items within the scope of ANSA.

Abstract Syntax Notation One (ASN.l) [ISO 8824] is the basis of the abstract syntax used in ANSA with encodings defined by [ISO 88251.

2.3.3 **Discussion**

Information type is a generalization of the concept of data type found in may programming languages. The term information is adopted in ANSA to emphasize the orientation towards all kinds of digitally encoded information items.

An information instance may be something complex, such as a multi-media document, a segment of real-time digitally encoded voice, the content of database, a computer program, or something as simple as binary digit. All these are examples of information types.

[redacted]

2.4 Operation concept

2.4.1 Definitions

System: a set of *objects* (§?.?) bound together in order to interact with other entities.

System states: a finite set of conditions within which a system may exist.

Event: an occurrence which causes a change in system state.

Action: a sampling of system state or a forced a change in system state, or both.

Operation: an interruptible action performed taking place within one *object* requested by some other *object.*

Operation specification: a formal specification of all possible effects of an operation.

2.4.2 Usage

Operations are the only means of interaction between *objects* that exist in ANSA.

2.4.3 Discussion

An operation is elementary in the sense that it is a smallest unit of activity that is complete in itself. But its functional purpose may be arbitrarily complex, and the encodings of its parameters and results may be arbitrarily large (but finite).

An operation is abstract in the sense that the effects of the operation are defined in terms of effects, which may be independent of the details of how the operation is implemented. ANSA describes possible implementations for operations in terms of designs for *modules* (§?.?) and *abstract machines* (§?.?). Implementors may chose to follow alternative strategies in actual systems (e.g. via non object-oriented systems), provided that the implementation conforms to the operation specification.

2.5 Operational abstraction

2.5.1 Definition

Operational abstraction: the characterization of an information type in terms of the operations applicable to the information type rather than by its abstract syntax or encoding.

2.5.2 Discussion

The term "operational abstraction" as used here is generalized from the term "data abstraction" found in the literature to emphasize the extension of the concept in ANSA to all forms of information.

For example the information abstraction of a dictionary might be via the operations **insert** and **lookup.** This behaviour-oriented operational view is independent of the data structure used to implement of the dictionary (eg sorted list, tree or hash table).

The operational abstraction concept is very important since it enables the designer to discuss **what** something is, separately from the issue of how to implement it.

2.6 Object concept

2.6.1 **Definitions**

Procedure type: an array of *instruction type* (§?.?).

Module type: a constructed information type which includes one or more procedure types.

Object specification: a set of operation specifications that include the **initializeObject** and **finalizeObject** operations.

Object type: a module type which includes procedure types that correspond in behaviour when executed by an *abstract machine* (§?.?) to a given object specification.

2.6.2 Usage

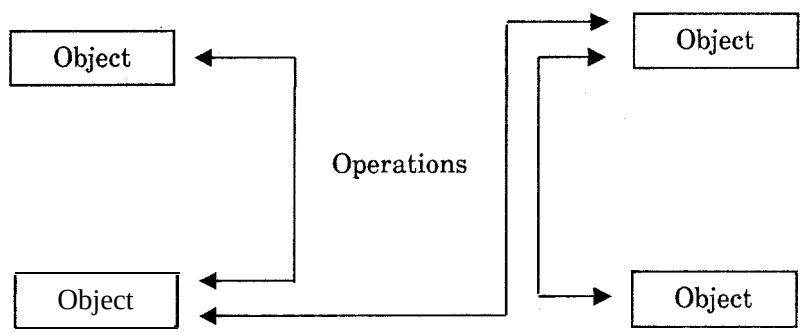
The term **object** is used for object instance where no confusion will result.

Modules are the units of implementation which permit independent relocation and replacement of **function** in a system.

The values of the module instances in a system are a partition of the system states.

Object types are the unit of encapsulation which permit the re-implementation of function in a system. This encapsulation concept of is illustrated in figure **2.1.**

Figure 2.1: Objects and external operations



To maximize the advantages of encapsulation, the exposure of implementation detail is minimized in ANSA object type specifications.

Object types are the 'black box' view of module types, that is, as entities able to interact with other entities via operations.

Module types are the 'white box' view of object types as a structure of procedures and other information instances bound together in order to respond to operations.

2.6.3 **Discussion**

This encapsulation concept is a packaging, for the purposes of ANSA, of the concept of "object" used in object-oriented computational models.

There are other packagings of the concept of "object" used in ANSA based on other models of systems, which are explained in ?.

Ediorial: The other packagings are the dimensions of part 4.

The general applicability of object-oriented concepts is discussed in part I, 92.5.

Encapsulation minimizes interdependencies between separately defined objects by restricting all interaction to operations. The operations of an object serve as a contract between the object and those objects it interacts with, and thus between the designer of the object and other designers.

If the uses of an object depend only on the operations, the object can be re-implemented without affecting any uses of it, provided that the object continues to meet its object specification.

The encapsulation of modules into objects and the execution of *instructions* (§?.?) from a modules procedures is the function of an *abstract machine* (§?.?).

The **initializeObject** and **finalizeObject** operations are used to provide a means by which an abstract machine can break open the encapsulation of an object to execute procedures within its module. They are described in §?.?.

2.7 Subtype concept

2.7.1 Definitions

Subtype: a type *A* is a subtype of a type *B* if an item a of type *A* can be used in a context where for an item of type *B* is expected.

Object type composition: the specification of an object type as a subtype of other object types.

Generic object type: an object type which has subtypes in the ANSA type set.

Canonical object type: an object type which has no subtypes in the ANSA type set.

2.7.2 Usage

In ANSA, an object trype is allowed to be a sub-type of more than one parent object stype.

In ANSA, object type composition is governed by strict rules so that the outcome of type composition is predictable and practical.

ANSA-conforming systems are constrained to be made up from canonical object. Generic objects are merely an aid to reducing the burden of specifying complex canonical objects.

2.7.3 Discussion

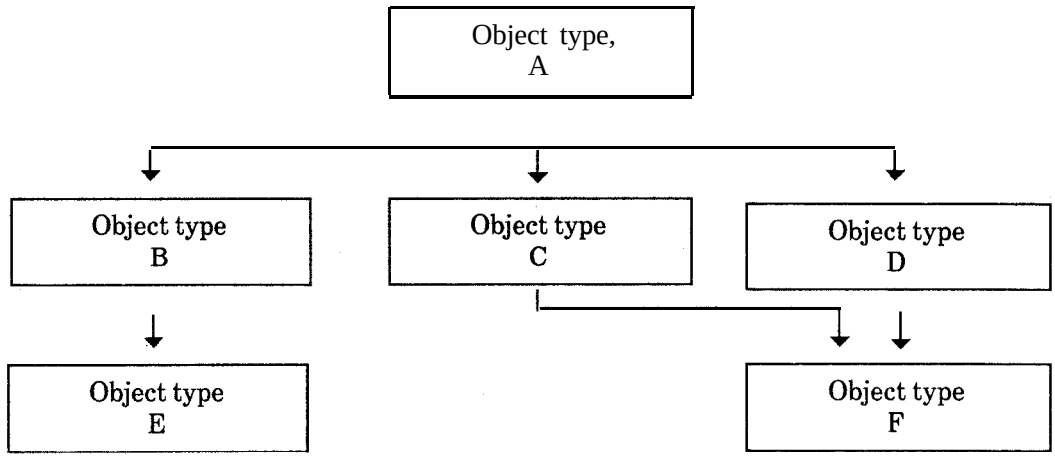
Generic object types are used in ANSA to describe sets of operations that can be combined with canonical objects via object type composition to enhance the canonical object, for example by adding replication and voting for dependability.

Subtyping for objects is derived from operation specifications If objects of type A meet the external operation specifications for objects of type *B* then *A* is a subtype of B. Object subtyping is illustrated in Figure 2.2.

The operations of object type A are common to all the other object types (B, C, *D*, E, F). Therefore, assuming the content of A to be non-trivial, the complete set of specifications have the coherence of all possessing the operations of *A* (i.e. the same operational abstraction). Conversely, without subtyping it is difficult to identify what is mutually consistent. The economies are evident in this example. Any operations high up in a subtyping lattice may be re-used many times over. The largest economies occur lower down in the structure (eg *E* and F), and where there is multiple subtyping (eg F).

A common way of achieving subtyping in programming systems is via 'inheritance', where object type composition rules describe syntatic manipulations of the parent module type to arrive at the module type for the

Figure 2.2: Object subtyping example



child. The type composition rules guarantee that there will be no inconsistencies between the constructed child module type and the object type of the child. Thus with inheritance, any child is guaranteed to be a subtype of all its parents object types.

Inheritance has a weakness in that any change to an ancestor module compromizes encapsulation since it has consequential effects on all of its children. The designer can no longer safely re-implement the module type of the ancestor without the risk of adversely affecting descendent object types. Thus permitting direct access to inherited information types and procedures weakens the freedom of the designer.

For these reasons inheritance is not used in ANSA. Instead the object type composition rules dictate how to derive object subtype operation specifications in terms of the operation specifications of the parent object types. This obliges an object type to meet the behaviour of its parent without inheriting the implementation of its parent, thereby maintaining encapsulation. Thus an object subtype can implement the external operations of its parent without inheriting the module type (i.e. implementation) of its parent.

The ANSA type composition rules are based on syntatic matching of specifications, to enable static checking, which imposes greater constraints on sub-typing than would be necessary if the checking were to be based on the formal semantic specifications of operations.

The declaration of a self-consistent object subtype hierarchy helps to achieve understandable and stable system designs. A successive refinement approach is used in ANSA (i.e. the increment of function in each object subtype is small).

Advanced Networked Systems Architecture

A N S R

24 Hills Road
CAMBRIDGE
United Kingdom CB2 1JP

TELEPHONE : **Cambridge (0223) 323010**
INTERNATIONAL: **+ 44 223 323010**
TELEX: **817343 BLUCAM G**

ANSA Reference Manual

Part III, 3: Interfaces

Abstract:

This document is Part III, Section 3 of the ANSA Reference Manual Release 0.0.

This section presents concepts and definitions that explain the way in which objects can invoke operations upon one another.

Number: A0.51.00

Date: 8th January 1987 12:48 pm

Area: AO (Architectural Overview)

Status: **0** (Open Document)

Classification: U (Unrestricted)

Distribution: G (General Public)

Editorial history:

Issue	Notes
01	Open Document status for Release 0.0 Approved AJH 9/1/87

3 INTERFACES

3.1 Introduction

This section describes the way in which objects discover each other's existence and perform operations.

The concept of an interface as a collection of self-consistent operations is introduced as a way of partitioning and managing the availability of operations.

Interface trading is introduced as the mechanism by which objects find out about interfaces and set up the necessary communications path to be able to be able to perform operations.

Information about the availability of objects to support particular interfaces has to be managed. This is done via a trading object. A trading object is an object into which details about objects and interfaces can be published for subsequent reference.

Editorial: this introduction needs extension to cover later subsections.

3.2 Operational interface concept

3.2.1 Definitions

Operational interface: a self-consistent partitioning of an object's operation specifications.

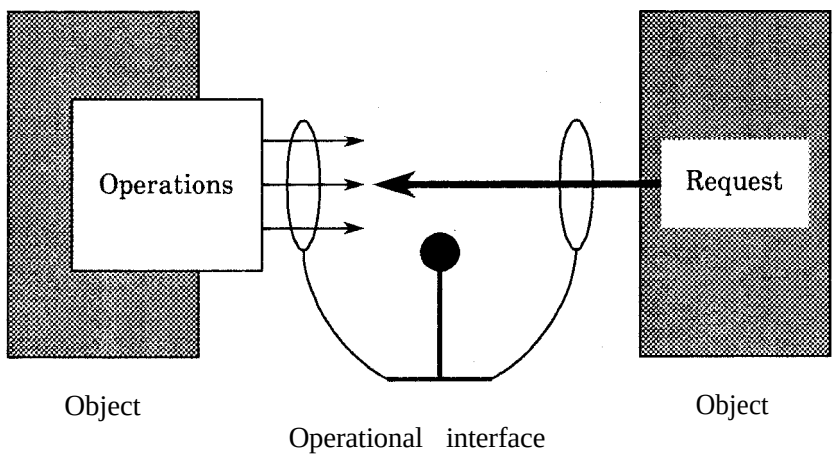
3.2.2 **Usage**

Support for interactions via these operational interfaces is the essence of communications within ANSA.

3.2.3 Discussion

For each object type in a system there is some set of operations which specifies all possible behaviour that is valid-and visible. Self-consistent subsets of operations are packaged together and made accessible as operational interfaces. Figure 3.1 illustrates an interface between objects.

Figure **3.1: An operational interface between objects**



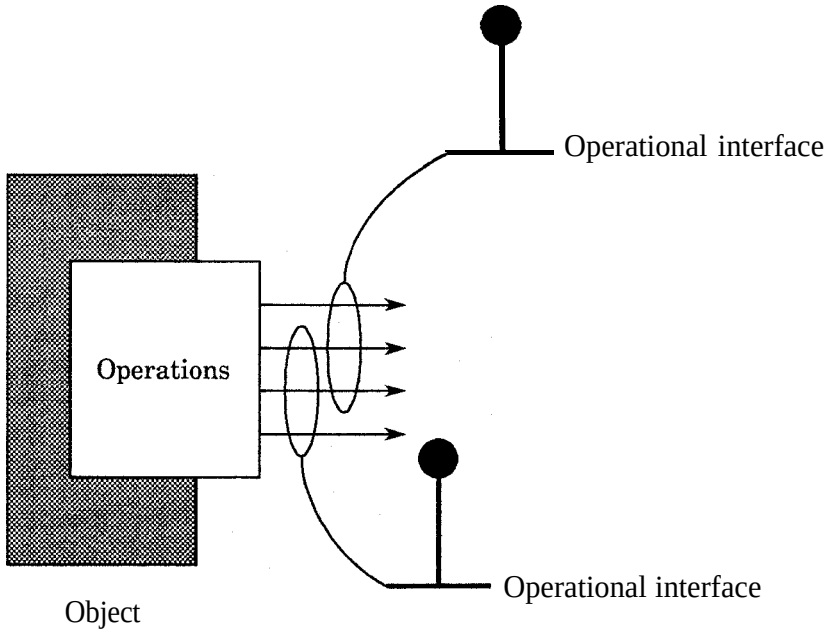
Such interfaces define the views that objects have onto each other. The operational interface is a contract between the object requesting an operation and the object the performs the operation. The contract provides the *abstract machine* (§?.?) mechanizing the operation with sufficient information to go about its task.

An object may adhere to multiple different operational interfaces which allow it to be viewed in different ways. This is illustrated in figure 3.2.

An operational interface is a concept which has a separate existence from the objects on to which it provides a view. This is directly analogous to the database concept of schemas being abstractions with a separate existence from the underlying storage onto which they provide views.

This notion of the separate (although not wholly independent) existence of the operational interfaces and the objects which adhere to them is

Figure 3.2: Operational interfaces providing multiple views onto the same object



fundamental to the ANSA object model. It is also fundamental to programming-in-the-large, as pioneered by modular programming languages such as Modula-2 [WIRTH 83].

3.3 Indirect binding concept

3.3.1 Definitions

export: an *instruction* (§?.?) that is used by an object to inform an *abstract machine* (§?.?) that the object is prepared to support the to operations in a particular operational interface.

Trader name: a *name* (§?.?) that is used as a parameter of the **import** and **export** instructions to indicate the *trader object* (§?.?) where interface type and instance names are to be resolved.

Interface trading name: a *name* (§?.?) that is used as a parameter of the **import** and **export** instructions to designate an operational interface known to a trader.

Interface type name: an *name* (§?.?) that is used as a parameter of the **import** and **export** instructions to assure type safety.

Interface instance name: an *name* (§?.?) that is used as a parameter of the **import** and **export** instructions to assure the integrity of a trader.

import: an *instruction* (§?.?)that is used by an object to request access to the object supporting a particular operational interface.

Indirect binding: the relationship that an importer has with an exporter.

Association type: an information type returned by a successful **import** *instruction* (§?.?).

3.3.2 Usage

Trading via **import** and **export** instructions is the way in which objects discover one another's existence as a prelude to interaction.

Objects export names for the operational interfaces through which they want to be viewed. The **export** instruction indicates, to an abstract machine, the accessibility of an object to respond to the operations in the referenced operational interface and an obligation to continue to respond to to such invocations. This obligation is only terminated by certain kinds of failure, or by explicit withdrawal of the **export.**

An abstract machine matches the trader, interface type and interface instance names in an import instruction with a previously executed export instruction (usually within some other object) using the same names. The abstract machine forms an indirect binding from the importing object to the exporting object. The importer is returned an association instance for this binding by the abstract machine. The importer is then able to request operations in the referenced operation interface of the exporter via **announce** and interrogate instructions (§?.?). These requests will lead to the execution

of a ***procedure*** in the exporter object's module to achieve the requested operation.

An object can be both an importer and an exporter. It can have arbitarily many bindings to many objects (including itself, which is useful for testing). There can be multiple indirect bindings between the same objects, and in both directions.

3.3.3 Discussion

Exports and imports can be envisaged as the trading of operational interfaces in a market place. If offers and bids match, trades may occur.

Trading is organized by ***abstract machines*** (§?.?) and ***trader objects*** (§?.?). Trader objects provide management controls over the accessibility of interfaces. A abstract machine may support several traders and therefore **import** and **export** instructions must nominate which trader is to be used.

A trader may possess knowledge different kinds of operational interfaces. For example a trader may know about file servers, print servers and authentication servers.

There may be several objects offering the same operational interface known to a single trader. For example a trader might know of three file servers with identical operational interfaces.

An **import** instruction will only succeed if a corresponding **export** instruction has occurred. Typically exports are activated when systems are configured or initialized (e.g. at the start of day, or after crashes), and imports are activated on demand.

The terms "import" and "export" are used in essentially the same way as in modular programming languages such as Modula-2 [WIRTH 83]. These terms are well-established for programming-in-the-large.

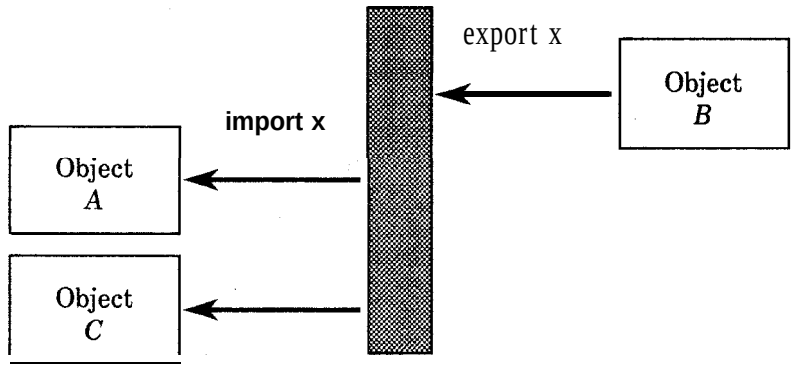
Figure 3.1 illustrates a simple example.

In this example, object ***A*** obtains a view onto object B, and object C obtains a view onto object B. Whether or not A and C could both access *B* at the same time depends on the operational interface and the irnplementation of *B* and the abstract machine.

The importing objects ***(A, C)*** have the initiative, and object *B* does not necessarily gain access to A or C (and it does not know of any relationship with them until they subsequently attempt to access it via their implicit bindings).

There is no binding between importers. Thus the objects ***A*** and C do not learn of the existence of each other or get views onto each other.

Figure 3.3: Indirect bindings



Trading is the culmination of a binding process that begins at module design time.

Definition ***phase***

The designer of the exporting module publishes an operational interface specification which determines the external operations that will be made available.

The designer of the importing module reads the published interface to determine which external operations he will invoke. The designer then includes a mutable association instance in the modules's state.

The designers of both the importing and exporting modules agree the interface trading name (or the process by which it can be discovered) that will be used in the execution phase and this name is made part of each module's information instance.

Construction phase

Templates for importing and the exporting modules are made. For software this may be via a compiler.

The construction process for the exporting module chooses an interface type name for operational interface that can be exported by the module. These names are stored in the module and recorded externally.

The construction process for the importing module stores the recorded interface type name of the exporter within the importing module.

Configuration ***phase***

A configuration of importer and exporter modules is planned. For software this may be via a linker.

The configuration process results in the assignment of an interface instance name to each planned instance of the module. These names are stored in the corresponding modules and externally recorded.

A choice is made of which exporter instance should be used by a particular importer instance and the operational interface instance name is stored in the importer module.

Execution phase

The exporting object is introduced to the abstract distributed machine and initialized. For software this may be via a loader. This process generates an (§?.?) to distinguish the newly created object from other object known to the same abstract machine. Its module performs an **export** to the abstract machine to announce its availability.

The abstract machine records the interface type name, interface instance name and the exporter's object name as the value of the interface trading name

The importing object is introduced to the abstract distributed machine and initialized. The importing object performs an **import** instruction.

The abstract machine resolves the interface trading name and finds the interface type name and interface instance name recorded by the exporting module. These are checked against the interface type name and interface instance name offered with the **import** instruction. If all of the names match, the abstract machine sets up an indirect binding between the two modules (since the name of the particular exporter to perform operations on behalf of the particlar importer has been decided). An association instance to the importing module so that it may refer to this indirect binding.. With this association instance the importing module is able to execute instructions to perform operations (§?.?).

The comparison of interface type names ensures consistency with static type analysis of the two modules against the operational interface specification.

The comparison of interface instance names ensures consistency with the configuration plan.

The management of the interface type name space and the interface instance name space is outside the scope of ANSA. This allows for binding decisions to be taken at a variety of times.

For example, if all interface type and instance names are unique, the interface trading name is redundant since the abstract machine can search

for the object name using type **name** and instance name. This is an example of **early binding,** which would be appropriate in a system with a static configuration.

Alternatively, the interface name may be ambiguous, in which case the resolution of the interface trading name by the abstract machine must choose a particular instance. This is an example of **late binding,** which might be appropriate in a system with a dynamic configuration.

3.4 Trader concept

3.4.1 Definitions

Trader: an object type which possesses the **registerExport** and **findExport** operations.

Import register: part of an *abstract machine* (§?.?) which records the results of **import** instructions.

Export register: part of an *abstract machine* (§?.?) which records the results of **export** instructions.

Export handle type: an information type returned by a successful **export** *instruction* (§?.?).

withdrawnExport: an *instruction* (§?.?) to undo the results of a previous **export** instruction.

discardImport: an *instruction* (§?.?) to undo a previous **import** instruction.

3.4.2 Usage

Traders provide management controls over the accessibility of interfaces.

3.4.3 Discussion

The trader object is typically implemented via the directory object.

The export register is an array of records with six fields:

- interface trading name
- interface type name
- interface instance name
- exporting object name
- export handle instance
- reaction procedure name

When an abstract machine *restarts* (§?.?) the object name fields in the register are all unset.

If an **export** occurs for an operational interface which is not recorded in the export register, the register is updated and the trader informed via the **registerExport** operation (the abstract machine does this by simulating the effect of an **interrogate** instruction - see §?.?). The information sent to the trader is the interface type name, interface instance name, the object name of the exporter and the reaction procedure reference (reaction procedures are explained in §?.?).

An exception is generated if an **export** occurs for an operational interface which is already recorded in the register.

As part of the export, the abstract machine assigns a export handle instance to the export register entry and returns it as the result of the **export** instruction. This handle can be subsequently used in an **withdrawExport** instruction to remove an export table entry. Distinct handles are assigned to each export register entry belonging to the same object.

The import register is an array of records with six fields:

- importing object name
- association instance
- interface type name
- interface instance name
- exporting object name
- reaction procedure name

If an import occurs for an operational interface which is not recorded in the import register, details of the interface are requested from the trader via the **findExport** operation. The returned result is the object name of the exporter.

As part of the import, the abstract machine assigns an association instance to the export register entry and returns it as the result of the **export** instruction. This association can be subsequently used in an **discardImport** instruction to remove an import table entry or in **announce** and **interrogate** instructions (§?.?). Distinct associations are assigned to each import register entry belonging to the same object.

3.4 Invocation concept

3.4.1 Definitions

Reaction: the execution of a procedure within an module as the response to the execution of an **announce or an interrogate** *instruction* (§?.?).

listen: an *instruction* (§?.?) available during a reaction, which completes the current reaction by retains the executing *thread* (§?.?) until another request from the same indirect binding arrives.

interrogate: an *instruction* (§?.?) which stimulates a reaction and obtains its results.

announce: an *instruction* (§?.?) which stimulates a reaction, but does not obtain its results.

break: an *instruction* (§?.?) which signals an *exception* (§?.?) to a *thread* (§?.?).

donateThread: an *instruction* (§?.?) by which an object can commit a thread to execute reactions.

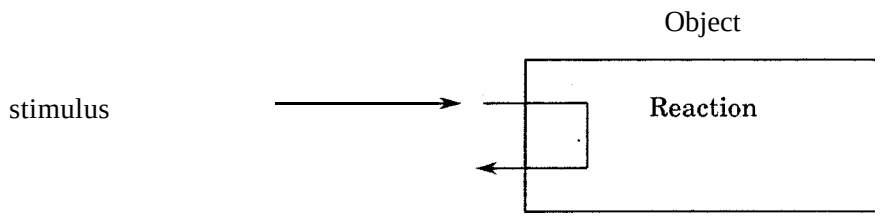
3.4.2 Usage

No remarks.

3.4.3 Discussion

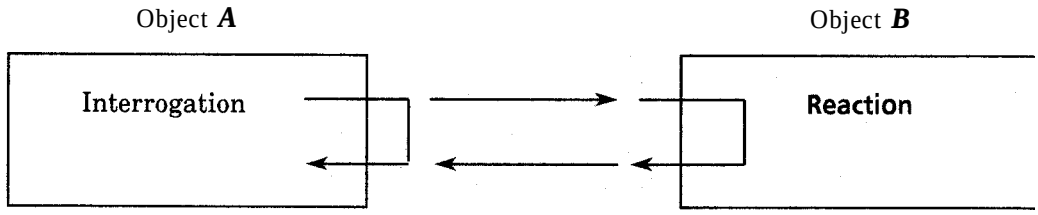
The structure of a reaction is illustrated in Figure **3.4.** The stimulus may be any of several kinds of ANSA abstract machine events including the execution of **announce, interrogate** or **break** instructions. The stimulus causes the abstract machine to *schedule* execution of a *procedure (see* §?.?) within the target module.

Figure **3.4:** A reaction



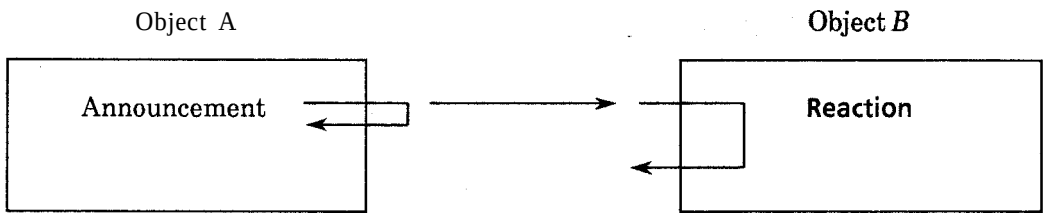
The structure of an interrogation is illustrated in figure 3.5. If successful it stimulates reaction and confirms its termination. Note that since interrogate is an instruction, the *thread* begins executing in the requesting object is *blocked* while the reactiontakes place (although other threads in the object are able to continue).

Figure 3.5: An interrogation



The structure of an announcement is illustrated in figure 3.6. If successful it stimulates a reaction.

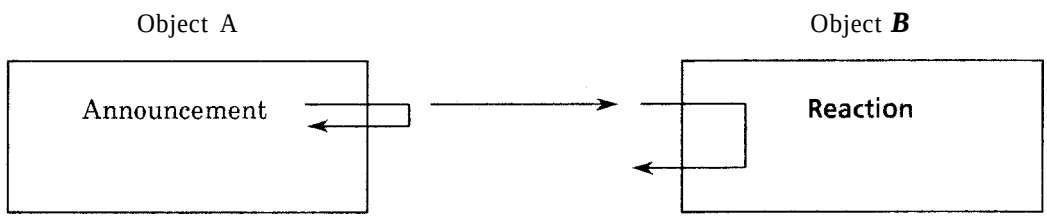
Figure 3.6: An announcement



Editorial: something must be said here to explain unsuccessful interrogations and announcements. Basically we have to address what happens when an object is removed (i.e. object name becomes invalid). The registers will contain incorrect information. The present description shows the registers as absolutes. They should be hints so that if an interface moves between objects we can track it down.

During the reaction to an interrogation a special association is set up in the import register. This association gives the reacting module the ability to stimulate the interrogating object. The reaction to this reverse stimulation is executed by the thread blocked on completion of the interrogation. Some examples are shown in Figures 3.7 and 3.8. The **listen** instruction allows similar structures to be used with one way announcements as shown in figures 3.9 and 3.10. All these structures are 'two-way alternate' (i.e. there is an implicit token passing back and forth).

Figure 3.7: ?



Editorial: the following pictures are not the intended ones!

Figure 3.8: ?

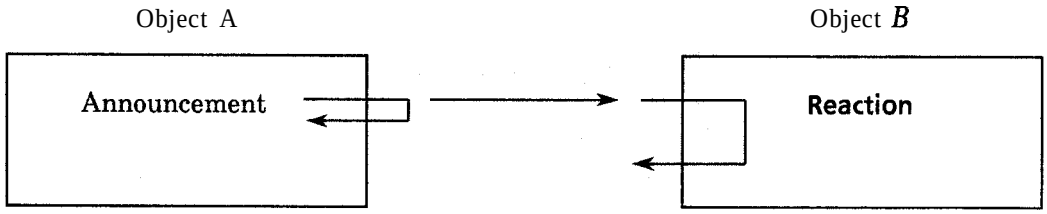


Figure 3.9: ?

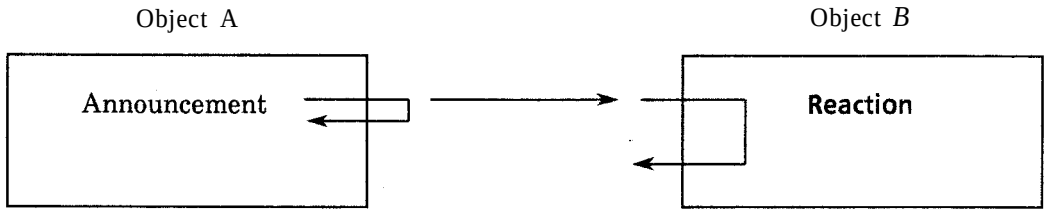
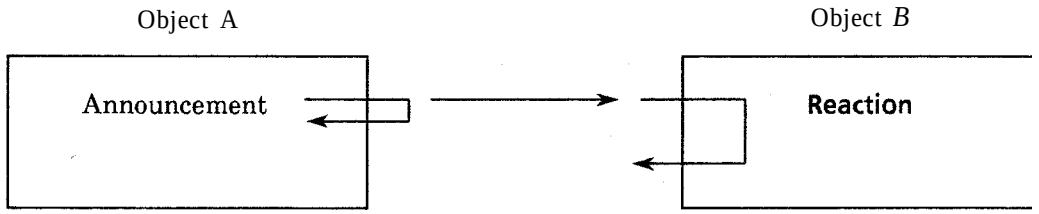


Figure 3.10: ?



The **listen** instruction has the effect of returning a response to the request and reserving the reacting thread to process the next request from the requesting thread.

An object informs the abstract machine of the procedure to for reactions to requests via particular interfaces through a parameter of the **export** operation. This information is kept in the abstract machines' export register.

The necessary communications to transfer the operation instance to reaction procedure are internal to abstract machine.

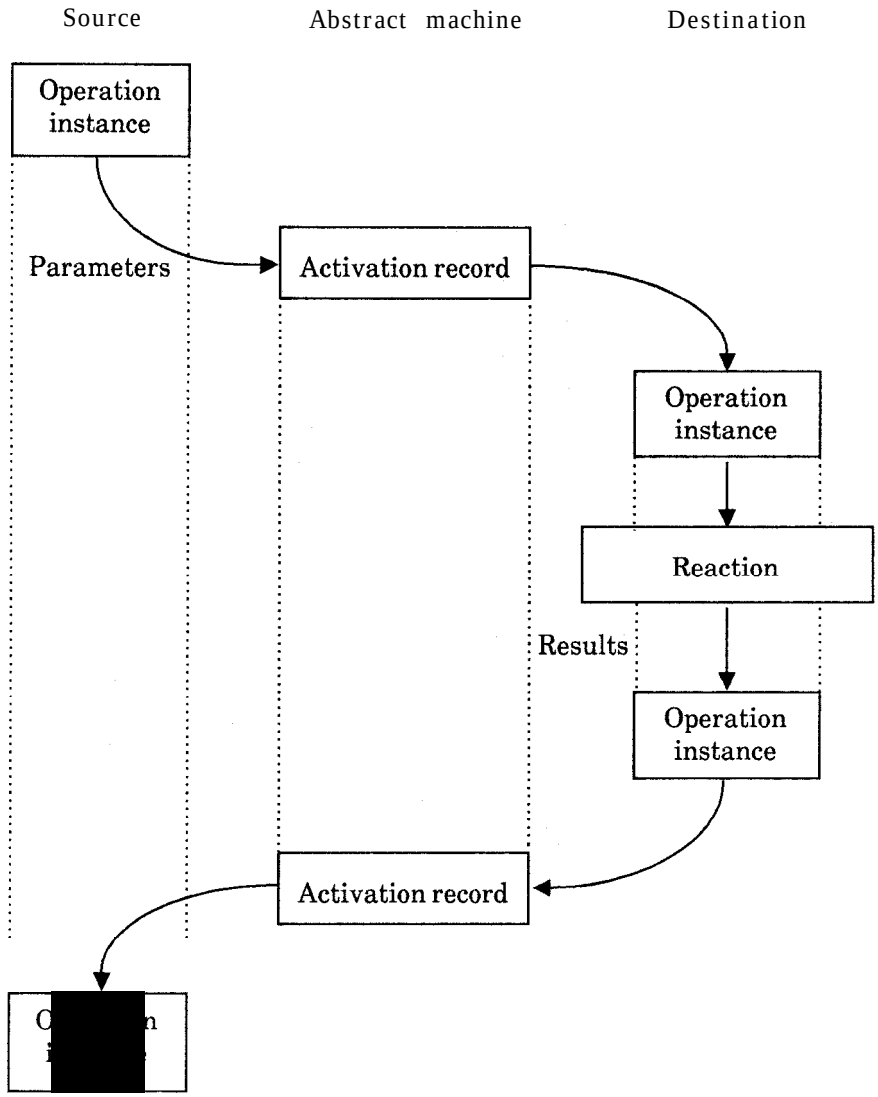
An object must commit ***threads*** (§?.?) to execute reactions via the **donateThread** operation.

When a request arrives at the target object, the abstract will select one of the donated threads to execute the appropriate reaction procedure. If there are no donated ***threads*** (§?.?) left, the request is blocked until one becomes free.

Editorial: explain here how the abstract machine copies operation parameters from A to B and takes back the results. The following text is part of an earlier draft for this.

An internal operation is invoked from an object source abstract machine and processed at a destination abstract machine. For invocation to be possible, there must be a communications path between source and destination abstract machines. The abstract machines transfer an activation record back and forth while the invocation is processed. This is shown in Figure ?.

Figure ?: Operations



Editorial: activation record contains an operation type and source thread name and target thread name.

Editorial: the following material is a residue that needs major edits and moving out to section 6.

All ANSA instructions return results - there is no asynchrony within an abstract machine, although there will be a high degree of parallelism. This strategy is intended to simplify resource allocation problems within the abstract machine by only requiring resources for the duration of an

instruction . Since primitives are synchronous, the abstract machine can 'borrow' the resources of the source to effect the primitive.

This rule imposes a top-down view of instructions: the abstract machine effects the primitive and returns its result. The rule imposes a layering based according to a principle which is traditionally defined as 'using' or 'depending upon the correct operation of'. This is an important organizational principle and is the basis of the OSI Reference Model [ISO ??] and many computer system architectures.

However, especially in a distributed environment, the flow of control is not always naturally downward. For example in communications processing, many of the actions are initiated not by the user from above, but by the network from below. The natural flow of control is thus upward and not downward.

In ANSA this problem is solved using the 'upcall' methodology of [CLARK 85].
Editorial: reactions are the ANSA word for upcalls

Upward control flow is implemented via operation invocation from an abstract machine. Prior to such an upcall taking place, the abstract machine must be provided with the resources to use during the upcall. This is done via enabling instructions.

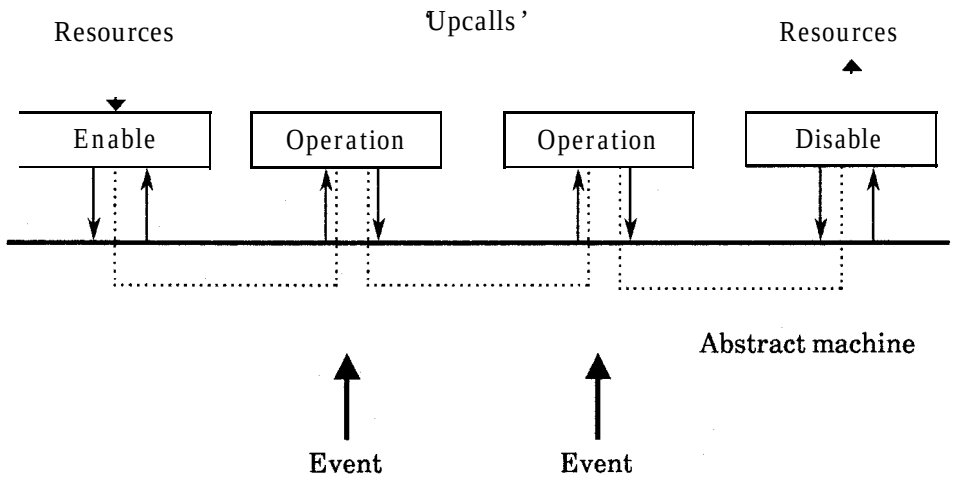
Editorial: donateThreads / listen

Resources are reclaimed from the abstract machine either by an explicit disable primitive, or as a side effect of an upcall (in which case further upcalls are implicitly disabled. This is illustrated in Figure 3.?.

Editorial: hence no interrupts, but what happens if an event and no resources? Answer abstract machine must fix up, probably by bouncing the caller.

Editorial: Talk about multi-process modules.

Figure 3.?: Upcalls



3.6 Interaction mode concept

3.6.1 Definitions

3.6.2 Usage

3.6.3 Discussion

3.7 Interface coupling concept

3.7.1 Definitions

3.7.2 Usage

3.7.3 Discussion