Adaptive Object Management for Distributed Systems

Ken Woods

Durham University

1995

ISBN 0-000-000000-0

1995

Abstract

This thesis describes an architecture supporting the management of pluggable software components and evaluates it against the requirement for an enterprise integration platform for the manufacturing and petrochemical industries.

In a distributed environment, we need mechanisms to manage objects and their interactions. At the least, we must be able to create objects in different processes on different nodes; we must be able to link them together so that they can pass messages to each other across the network; and we must deliver their messages in a timely and reliable manner. Object based environments which support these services already exist, for example ANSAware(ANSA, 1989), DEC's Objectbroker(ACA,1992), Iona's Orbix(Orbix,1994)

Yet such environments provide limited support for composing applications from pluggable components. Pluggability is the ability to install and configure a component into an environment dynamically when the component is used, without specifying static dependencies between components when they are produced. Pluggability is supported to a degree by dynamic binding. Components may be programmed to import references to other components and to explore their interfaces at runtime, without using static type dependencies. Yet this overloads the component with the responsibility to explore bindings. What is still generally missing is an efficient general-purpose binding model for managing bindings between independently produced components.

In addition, existing environments provide no clear strategy for dealing with fine-grained objects. The overhead of runtime binding and remote messaging will severely reduce performance where there are a lot of objects with complex patterns of interaction. We need an adaptive approach to managing configurations of pluggable components according to the needs and constraints of the environment. Management is made difficult by embedding bindings in component implementations and by relying on strong typing as the only means of verifying and validating bindings.

To solve these problems we have built a set of configuration tools on top of an existing distributed support environment. Specification tools facilitate the construction of independent pluggable components. Visual composition tools facilitate the configuration of components into applications and the verification of composite behaviours. A configuration model is constructed which maintains the environmental state. Adaptive management is made possible by changing the management policy according to this state. Such policy changes affect the location of objects, their bindings, and the choice of messaging system.

Table of Contents

Part IV Appendices

Acknowledgements

This work has been part of a large scale collaborative project. I am grateful to all development staff at Prism Technologies, in particular to Steve Osselton, Andy Ridout and Keith Williams for their contribution to the development of the architecture. I have enjoyed working with them.

I would also like to thank the directors at Prism Technology for creating the opportunity for this study, Keith Steele, John Biggerstaff and in particular John Russel, the keeper of the technical vision for OpenBase.They gave me the freedom and drive to address such a challenging project.

I want to thank John Mellor for the encouragement, patience and guidance he has given me in producing the thesis and supervising my study. I am also very grateful for the time he has given in managing the extended period of study.

I must also acknowledge the efforts of the various research consortia whose ideas and experience provided a more solid foundation for this project, in particular the ANSA consortia, the ITHACA project and the REX consortia. I would like to especially thank those individuals who have taken the time to comment on or discuss the draft papers and reports that I have written, especially Eduardo Casais, Dennis Obong Nyong and David Iggulden.

Finally and most importantly I must thank my family for their tolerance, including my two children, Roslyn and Joel, born during the duration of the study and my wife, Janet. I must apologise to my children for having to spend many a night sleeping on my lap to the clunk of a keyboard. It is to my family that I must dedicate this work.

Preface - A Change in Perspective on Programming Languages

I recently read a startling scientific fact - that if you place a frog in a pan of cold water and heat it gently , then the frog lets itself be slowly boiled to death. This fact illustrates the danger of being too comfortable with continuous change. It is important to react appropriately to temperature changes - to jump when things get hot. To me, the goals of object technology are hot, yet object oriented programming is not enough of a jump. A greater degree of discontinuous change is required.

My work seeks to support component oriented development of distributed applications. Applications are constructed by composing pre-existing components. I find that current object oriented programming languages are not ideal for component oriented reuse. The main reason for this is the failure of object oriented languages to support the necessary transition in programming culture. Disciplined reuse of code components is against the very nature of programmers. To be effective, reuse needs to be made an explicitly distinct task from the normal construction of components. Many programmers treat *programming* as a creative and intangible craft. A quote from one of Apple's most creative programmers illustrates this point: " Reusing other peoples work would prove that I don't care about my work. I would no more reuse code than Hemingway would have reused other author's paragraphs." Furthermore, *programs* are perceived as being inherently complex. How can we reuse something that is so intangible and complex? Despite its support for component composition, object orientation is still a programming craft. Components are *programmed* to reuse other components. Consequently reusable components compete as untrusted alien produce with the fruits of the programmer's own labour.

A new perspective on programming

Instead of *programming tools*, we need real *engineering tools* based on *intuitive* procedures to assemble *tangible* components. We must change the very nature of programming, from a craft to an engineering discipline. Intuitive environments can be built with visual techniques that support limited well-defined assembly actions. The tangibility of components can be improved with specification tools and modelling languages. An engineering lathe is of little use for cutting components without a gauge to ensure that the components produced are within tolerable limits to fit together, wherever and whenever they may be used. Likewise the emphasis in component oriented software must be on plug compatible interfaces. A plumber does not need to understand the manufacturing process for his fittings, only their engineering specifications. Likewise application engineers should not need to be component programmers in order to fit software objects together. These engineering goals characterise the product vision behind this project.

It would be extremely difficult and commercially impractical to try to turn the complete programming task into a more intuitive discipline overnight. There is an inherent dichotomy between the desirability of intuitive formalisms that exploit the human capacity to resolve ambiguities and the necessity for precision to implement a formalism on a digital machine. If this was not the case, programming languages would be using natural language like English. Likewise intuition is at odds with the inherent universality and freedom of computational solutions. Other engineering disciplines are constrained by clear physical laws. Computing on the other hand is constrained only by the limits of our cognitive processes. To define concrete intuitive processes requires us to limit the cognitive freedom of the programmer. A program is best *engineered* using well defined programming tasks.

Consequently our system seeks only to turn component assembly and configuration into an engineering discipline. Components are still *programmed*, freely and to a precise level, in C++. Assemblies on the other hand are scripted together by following simple assembly tasks supported by engineering tools layered on top of the C++ environment.

A new perspective on language design

Not only does this product vision require a re-evaluation of how we perceive the programming process, but the technical vision that characterises the solution is also new and requires a re-evaluation of what is a programming language. In particular , the programming language needs to be opened up to integrate the specification tools.

Despite being targeted at a specific application domain, the required engineering system shares a number of fundamental and non-trivial language design problems with general-purpose programming systems. Modern programming languages live in tension between conflicting demands:

1) expressivity vs efficiency

Users need more expressive power to deal with increasingly complex applications. Complexity arises from the introduction of more system power to deal with concurrency, persistence and distribution and the introduction of more modelling power to attack larger, more dynamic, evolving applications.

Despite needing more expressivity, commercial users still want the adaptability and efficiency which they perceive as only available in low-level languages like C and C++. More expressive languages appear inefficient or highly specialised.

2) compatibility vs extensibility

Backward compatibility is required in a programming system as languages evolve. The commitment to large volumes of existing code restricts language improvements. Sideways compatibility is required to integrate other tools, and forward compatibility as external industry standards mature and new technology emerges.

Despite needing stability, there are times when variations in the programming model or extra features are entirely appropriate to satisfy domain specific requirements, making efficient programs easier to write and maintain. Yet language design and language use are traditionally seen as entirely distinct tasks and languages are built to be insular and complete, not open to integration with other languages or incremental modification and extension by programmers.

Our system must introduce new features to deal with distributed complexity like partial failures, indeterminate orderings of messages, heterogeneity, or to integrate the component programming language into the engineering system to support pluggability. Each new feature must resolve these tensions. The resulting friction is generating too much tension for a conventional language. This heat is slow-boiling our language frog.

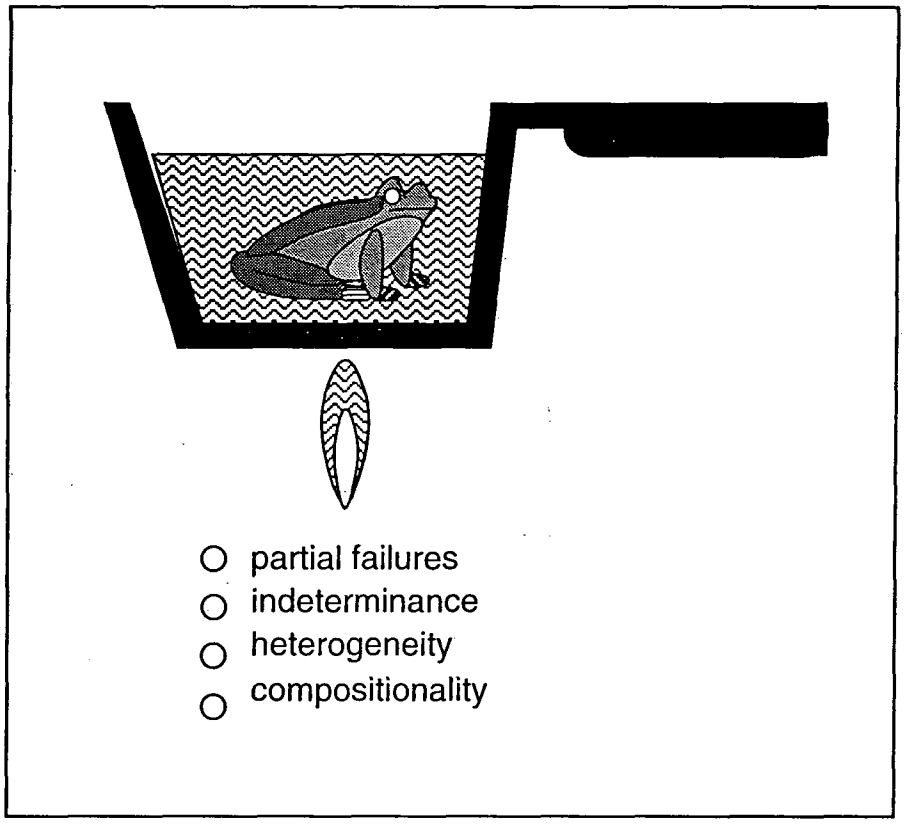


Figure 1 The slow boiling frog

A new approach is proposed in order to meet these conflicting demands, in combination, across all dimensions and to allow openness. First we rule out conventional approaches.

Conventional approaches to our problem would include :

(a) designing a high level language that supports the appropriate features.

(b) adding the features orthogonally to a commercially available language, by integrating an API or object library or a pre-processor.

We rule out a) because a single expressive high level language would prove too inefficient or too theoretical or too impractical for commercial users.

We rule out b) because of the scope of the changes required. If we added both distribution and pluggability, the programmer encumberment, the overloading of responsibilities on component programmers, and the interference between features would quickly make the programming system unmanageable.

A more drastic separation of concerns is required to deal with so much complexity. Consequently we adopt a new approach to language design, that of adding meta-object protocols. This approach requires us to represent language features at a more fundamental level.

Meta-object Protocol for Composition

If the language implementation itself is structured as an object oriented program, then the behaviour of the language can be incrementally modified by modifying the language objects (i.e. the meta-objects) . This is the basis of the meta-object protocol approach. Meta-object protocols provide interfaces to the language objects and allow them to be customised and extended by the programmer. The meta-object protocol approach allows us to satisfy all our goals in combination. The protocol may itself be an expressive high level language thus a meta-object protocol defines a mechanism for integrating an expressive language for dealing with new complexities into a more conventional language.

We add a meta-object protocol to C++ to integrate our engineering tools. It is the composition mechanism that needs an object oriented representation. The engineering tools use the meta-object protocol to manipulate the composition mechanisms to describe the structure of the system.

Modelling the compositional structure of a software configuration is useful to define both the internal architecture, i.e. the runtime support structure to deal with distribution, and the application architecture, i.e. defining bindings for pluggable components. These tools relieve the component programmers of many complexities so that they can focus on the application behaviour of components. Integration between C++ components and the rest of the engineering system is via stubs and fragments generated by a C++ pre-processor.

By opening up the composition mechanism at a fundamental level, the design is less vulnerable to surface differences between different versions or variants that use different specification and composition tools for entirely appropriate reasons. The common meta-object representation relates the different specification tools that separate programming concerns and simplifies problems of compatibility and extensibility as new features and runtime support are integrated. Whereas conventional programming systems constitute a single point in language design space, this engineering system constitutes a whole region of language design space, a region that can be made expressive, efficient, compatible and extensible.

In summary, a new style of programming technology, namely meta-object protocols, is being used to provide a new style of programming, namely visual composition. The heart of the language frog is being replaced with a set of engineering levers which can be driven by well defined mechanical processes.

Significant challenges and risks arise from introducing new and discontinuous approaches - technical, methodological and attitudinal. It is not easy to jump out of a pan. Proverbs involving fires may come to mind. Yet it is equally unsettling to live in a world of slow boiling frogs.

Part I Introduction

Chapter 1 Introduction

A large-scale, collaborative, industrial research project is underway to implement a state-of-the-art distributed platform to support the integration of applications for the process and manufacturing industries. This thesis is based on my contribution in defining the architecture for the platform and in implementing the object management system.

Previous approaches to integration used low level communication facilities to interconnect autonomous applications. This project seeks to establish a binding model that supports finer-grained integration so that individual objects rather than whole applications can be reused in different contexts. The aim is to create a graphical programming tool to compose applications from pre-fabricated component objects. Such an intuitive tool could remove the current overdependence of the process industry on system integrators, by allowing plant engineers at the process plant to configure applications themselves.

In order to graphically build flexible object oriented applications spanning the entire manufacturing enterprise, the project must integrate graphical programming technology, object technology and open distributed processing technology.

1.1 Purpose of Study

The purpose of this study is to define and evaluate an architectural approach to technology integration in order to support the project goal of enterprise-wide application integration. Technology integration is an important research topic that presents a particularly appropriate focus within the context of the broader project, for the following reasons:

- application integration across a distributed enterprise presents a number of challenges that demand a broad range of new technologies and techniques to overcome them. The key challenges include:
  - development of applications spanning different types of machine,
  - dealing with the complexities of programming distributed systems,
  - integration of applications that have been developed independently by different vendors,
  - reuse of application components in different parts of the enterprise.
- taken individually, these challenges are ambitious and demand radical solutions to solve the underlying practical problems. Yet focused research efforts may fail to integrate without an overall architectural framework.
- taken in combination, the challenges call for a significant paradigm shift that demands new types of architecture. In particular to support:
  - new development lifecycles to support new partnerships and organisations of developers,
  - new binding models for pluggable components,

  - new ways of managing components to allow them to be adapted to different contexts across the enterprise,
  - new standards to coordinate the industry and ensure pluggability.
- most work to date has been focused on the key application integration challenges within evolutionary or conventional frameworks. Yet this can be overly restrictive given the broad range of innovations and techniques available. Technology integration deserves to be a research subject in its own right in order to explore the foundations required for a more comprehensive future solution to the combined challenges.
- the growing computing industry trend away from bespoke point solutions and towards off-the-shelf products goes beyond off-the-shelf applications to off-the-shelf infrastructure products. The computing industry is in desperate need of new approaches to selecting and integrating different technologies. Infrastructure products are often selected on an ad-hoc basis and technologies do not fit together well, for example CORBA on top of DCE.
- technology integration is a difficult problem for two basic reasons:
  - the dissimilar nature of different technologies, such as messaging and remote procedure call or inheritence and delegation,
  - the inherent difficulty in relating low level programming features to high level requirements.

  Further exploratory work is important to the future development of the field and to identify practical solutions to these problems.
- because of the different nature of different integration problems, any generalisations derived from research are unlikely to have wider scope than the problem at hand. Yet application integration is a broad subject and the focus on integrative architectures has quite wide-spread applicability across different market sectors including telecommunications, finance, manufacturing, government and petrochemical industries. In addition, the basic approach to architecture design may be reapplied to other problems.
- exploratory research is best driven by the requirements of real industrial scenarios. The broader project context provides a real world setting with real requirements against which success can be measured.

1.2 Research Objectives

The thesis seeks to abstract away from the individual problems and solutions of application integration in order to rationalise an approach to technology integration. It explores the key issues and tests out the effectiveness of the approach.

One key element in the approach is to separate concerns in the architecture. This involves breaking the architecture down into a number of related subcomponents that deal with different issues and apply different techniques. It takes a purposive approach to describe the separation i.e. it describes the purpose and context of each architectural component. This identifies clearly the higher level roles and relationships between components and contrasts conventional behavioural models that describe the behaviour or function of each component.

A purposive approach (i.e modelling purpose) unifies evaluation and design activities, making it easier to:

- relate low level programming technology features to higher level programmer requirements,
- evaluate and select different technology to meet design goals,
- make orderly comparisons between design options and product options.

The main position argued by the thesis is that a separation of concerns helps integrate different techniques and can be used to construct a viable architectural foundation to meet the broader challenges of application integration.

In developing this argument, I intend to:

- identify and survey the practical problems underlying the key challenges of application integration in order to validate the hypothesis that more radical solutions are required. Individual solutions are not critical to the thesis.
- explore the key aspects that differentiates the approach supported by the broader project. These aspects in combination characterise a new paradigm for development which I call adaptive graphical objects. I will elaborate on the key components including the following:
  - a component-based lifecycle,
  - a binding model based on visual hierarchical composition,
  - adaptive object management according to the configuration state,
  - distributed object standards and domain framework standards.

  These concepts impact on high level architecture and deserve special attention in testing the hypothesis that a paradigm shift is required to meet the combined challenges.
- review any evaluation frameworks that relate technology features to their purpose or goals in order to identify a suitable unified framework for evaluation and purposive modelling.
- apply the framework to provide a high level rationale for the separation of concerns in the proposed architecture. This is in line with the proposed paradigm. I will evaluate three key hypothesis:
  - a purposive approach can result in a suitable separating of concerns.
  - a separation of concerns facilitates the integration of different techniques that solve the basic challenges of application integration. It is important that the architecture integrates well as a broad range of tools and techniques that do not fit together offers little value to users.
  - adaptive graphical objects support an appropriate paradigm shift to meet the combined challenges of application integration.
- briefly review the design decisions that were made in the lower level design of each component, highlighting issues and requirements for technology integration. This architecture is realised in a demonstrable prototype.

- use the concrete requirements of the broader project to evaluate the overall architecture. In addition, I will evaluate the hypothesis that the proposed architecture has value as an integrative architecture in different market sectors.

With the scope being a Master's degree, a number of compromises and limitations have been made. The research was limited to design, demonstration and formulative evaluation (i.e. research determining if an approach is realised). There was no time to collect a substantial amount of data, such as performance modelling and monitoring data. Consequently there is still a requirement for further experimental research. Nor was there time to provide a complete state of the art management system. This was seen as superfluous as the emphasis is on the higher level modelling of essential management mechanisms. Further development is necessary. For example, I limited the prototype to deal only with : object allocation across the network, object binding, and remote messaging.

1.3 Related Technologies

The project vision to use graphical tools for application integration mirrors a broader trend in software development that will come to the fore late in this decade where software development will move from a programming craft to a large scale manufacturing and engineering discipline. The move to software components is driven by the need for high quality software that can be configured to meet changing needs with minimal expenditures of time and cost.

This trend is enabled by the coming together of three key technologies:

- graphical programming tools for component configuration,
- distributed object management technology,
- integrative standards for plug and play components across an enterprise.

Some of the key issues and related work in each of these areas is summarised below to illustrate the full potential of an architecture resulting from their integration. This thesis will integrate a subset of the features in each technology.

1.3.1 Graphical Tools for Configuration Programming

OpenBase allows configurations of software components to be described graphically using a graphical editor. This mirrors the trend to graphical tools in the CORBA world such as IBM's VisualAge for DSOM or Oberon's Synchroworks for Sun DOE.

Plant engineers manipulate different aspects of the system in named configuration domains, called composites. These are logical groupings of application components. Configuration composites support commands for ad-hoc changes to the structure of an application e.g. instantiate, link, unlink, destroy.

Figure 2 below illustrates a simple composite that has been constructed by instantiating and linking the following classes: I/O point, maths function block, delay, alarm, and print driver.The visual metaphor uses drag and drop actions to instantiate objects from palettes and point and click actions to invoke attribute and linkage editors. This metaphor is supported by an appropriate binding model for visual composition.

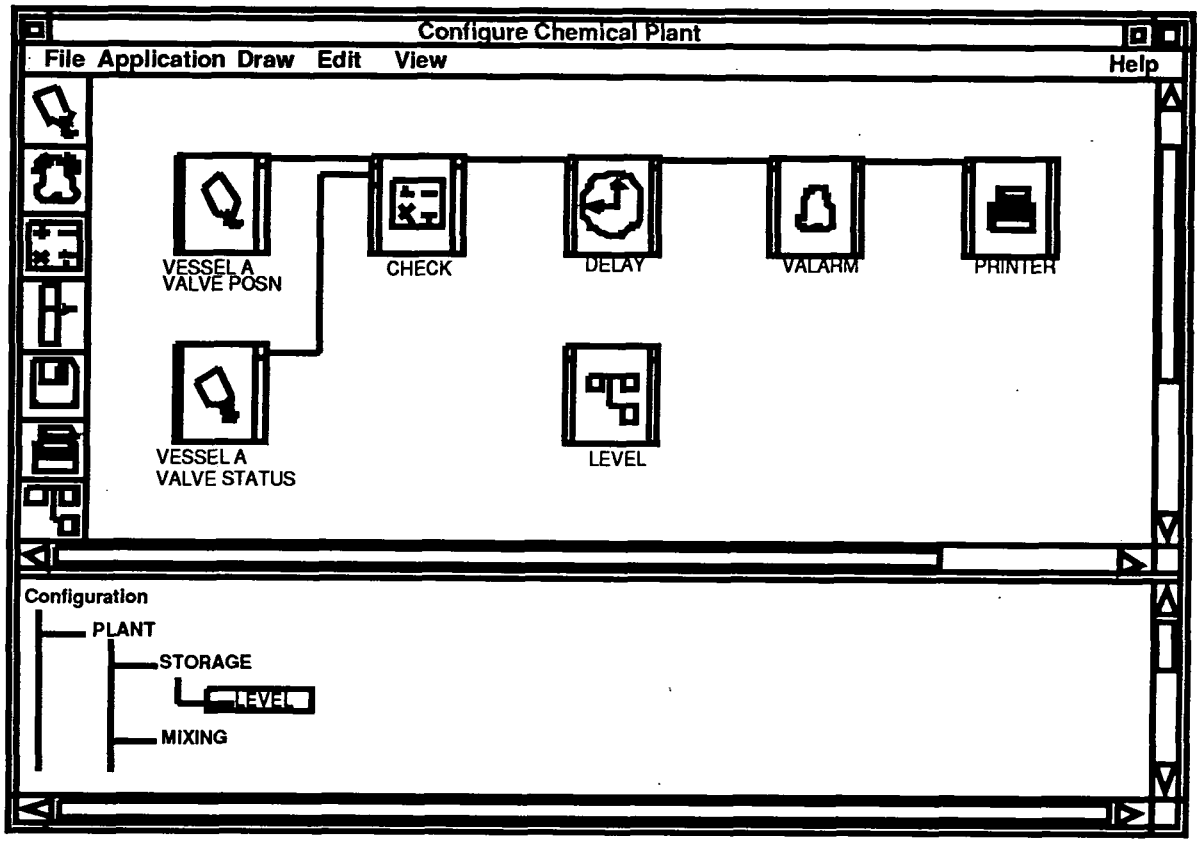


Figure 2 Example configuration composite

1.3.2 Distributed Object Technology

In a distributed environment, we need mechanisms to manage objects and their interactions. At the least, we must be able to create objects in different processes on different nodes; we must be able to link them together so that they can pass messages to each other across the network; and we must deliver their messages in a timely and reliable manner. Object based environments which support these services are already commercially available, for example DEC's Objectbroker, Iona's Orbix, IBM's DSOM.

Distributed management for object oriented systems can get complex. Program descriptions must be interpreted in a broader context where concurrent objects are allocated to disjoint address spaces, where resources partially fail, and where effects take an unpredictable time to propagate. Mechanisms to deal with these issues have been demonstrated in the last generation of research systems and are now emerging in commercial systems : Chorus/COOL (Rozier et al.,1987), Eden (Almes et al., 1985), ISIS (Birman and Joseph, 1991), Argus (Liskov, 1988), Arjuna (Shrivastava, 1991), ANSAware (ANSA, 1989) or (Herbert, 1989).

Support for distributed management can be divided into three areas: object management, resource management, and interaction management. Object management includes transaction control, security, recovery, and replication. Interaction management includes naming, binding, remote messaging and error handling. Resource management includes storage, checkpointing, allocation, and migration.

The potential of distributed objects to increase the performance and dependability of applications is great. However the thesis initially looks at only a few of these value-adding behaviours.

1.3.3 Integrative Standards

The issues of composition are broader than just the creation of reusable components that can be assembled into a target application. One needs standards that transcend any single component. Software standards are needed at the application level and at the system level. Application domain standards apply to modelling tools, to the data that can be interchanged between components, and to any broad structural patterns that are to be imposed on object interconnectivity. System level standards apply to the architectural interfaces that allow components to interoperate.

Such standards are much like the architectural standards that hardware component manufacturers establish so that they can plug their components together, for example signals for chips (enable, address lines, data lines, interrupt lines) and bus standards for boards(VME, Multibus).

Application Domain Standards

At the application level, there has been a recent surge of activity in the area of object oriented platforms for the integration of manufacturing and process applications. Standards bodies have been formed to represent various industry bodies, such as POSC for the petrochemical industries and the National Centre for Manufacturing for manufacturing software. They are defining architectural frameworks for application integration. However these bodies do not yet address many of the technical problems of distributed management.

System Standards

To manage interoperable components across a network, we need a set of standard system interfaces. Various technical standards bodies are satisfying this need in general purpose standards for open distributed computing, such as :

- the Object Management Group's (OMG) Object Management Architecture, which includes the Object Services Architecture and the Common Object Request Broker Architecture (CORBA);
- the Open Software Foundation's Distributed Computing Environment (OSF/DCE);
- the International Standards Organisation's Open Distributed Computing Reference Model (ISO ODP RM).

What is currently still generally missing from these system level standard architectures, are application level standards for pluggable components. Such environments provide limited support for composing applications from pluggable components. There is not yet an efficient general-purpose binding model for managing bindings between independently produced components and standards governing the interfaces that these components should support.

1.4 Initial Positioning of Technologies

The graphical programming tools and application level standards impact mainly at the highest level in design concepts and programming metaphors used by programmers. The object management support functionality and system standards impact mainly at the lower level in the implementation of a runtime executable.

This natural separation may be re-enforced by applying intermediate layers of representation and interpretation. The resulting architecture may be viewed as a layering of graphical programming metaphors and domain standards on top of a standards-based support environment for distributed objects.

The proposed architecture bridges the gap between these top and bottom layers with three layers as shown in Figure 3:

- tools layer - specification tools facilitate the construction of independent pluggable components. Visual composition tools facilitate the configuration of components into applications and the verification of composite behaviours;
- meta-model layer - using the tools, plant engineers construct a configuration model which tracks objects and system resources. This configuration model allows requirements for the system to be described at different levels of abstraction independently of the underlying support;
- interpretation layer - the interpretation layer adapts the configuration model and instantiates a corresponding runtime representation. The ANSAware Engineering Model was selected as our underlying management technology for the prototypes. The interpretation layer provides a stable interface that is implemented in ANSAware but can be ported to other platforms.

The broader project vision of a graphical product integration system provides a new development paradigm that impacts on all aspects of software construction:

- the technical goals of product developers;
- the development process;
- the principles employed to structure software;
- the language abstractions, programming techniques and mechanisms to support the principles;
- the architectural components implementing the functionality behind the mechanisms.

It is very difficult to talk about the role of these aspects in isolation without relating them to the whole. Consequently the notion of purpose used to describe architecture components is expressed in terms of the relationships between technical goals, abstract principles, techniques and components. The emphasis on relating abstract design principles to both diverse goals and language mechanisms is especially appropriate to programming system architecture design and helps capture the expansive vision necessary for innovation.

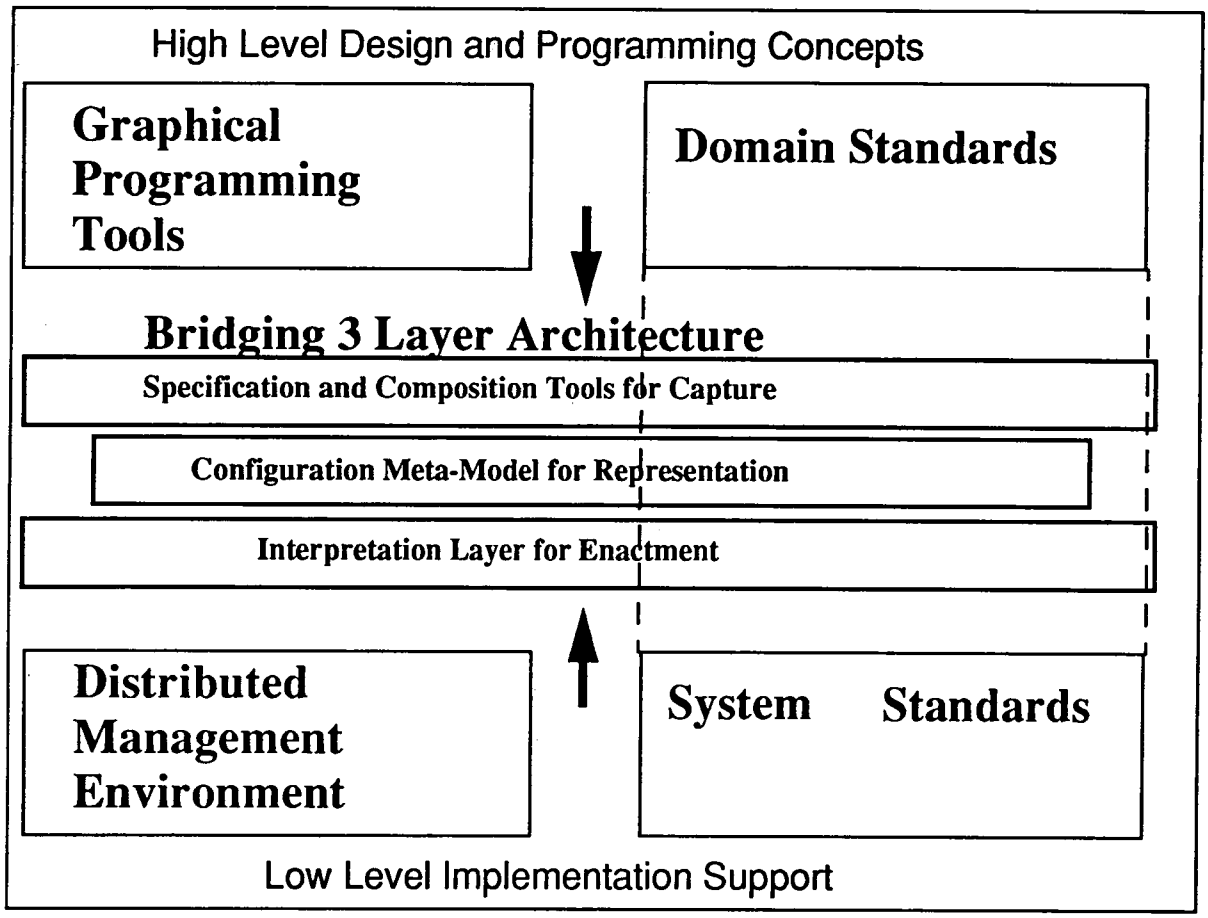


Figure 3 Bridging Architecture

1.5 Outline of Thesis Structure

The thesis breaks down into three parts.

Part I, Introduction, consists of the introductory material.

- Preface, A Change in Perspective on Programming Language, prepares the reader to reconsider the fundamentals of programming system design, as these will be challenged by new paradigms.

- Chapter 1, Introduction, summarises the research objectives and provides background information on the project.

Part II, Survey and Analysis of Techniques, is an extensive literature survey of related technologies and techniques. The survey is summarised by defining a purposive evaluation framework that is populated with classification hierarchies for technical goals, principles and techniques. Chapter 2 reviews object technology. Chapters 3 to 6 extend the work by considering the impact of distribution on object technology.

- Chapter 2, Review of Fundamentals of Object Technology, reviews object technology and introduces an existing evaluation framework. There is an emphasis on deriving a broader view of object orientation to include less familiar approaches like hierarchical composition.

- Chapter 3, Technical Goals in a Distributed Enterprise, re-evaluates the technical goals of object technology by considering new architectural goals that dominate in a distributed enterprise. These goals are used as our requirements in the evaluation.

- Chapter 4, Open Distributed System Development & Tools, describes the differing approaches to developing distributed object systems, in particular the effect of technology choices, standards and lifecycles.

- Chapter 5, Distributed Programming System Architecture, reviews principles of programming system design. This review provides an arsenal of techniques that can be used to define architecture. The approach synthesises a number of these techniques.

- Chapter 6, Evaluation Framework, consolidates the whole survey by summarising the key components in an example framework. This framework is used in part III to design and evaluate the architecture.

Part III, An Evaluation of the Architecture, describes and evaluates the architecture from the point of view of the abstract principles employed in its construction. This constitutes most of the results from this formulative evaluation. The application of an analytic framework ensures that most of the rationale behind design decisions is implicitly captured.

- Chapter 7, Overview of Problem and Approach, reviews shortcomings in existing approaches and provides the rationale behind the proposed architecture using the evaluation framework as a design framework for modelling the purpose of the key architectural components.

- Chapter 8, Interpretation Support for Adaptive Management, reviews the design decisions in the interpretation layer that uses distributed object technology services to load and manage configurations of objects. This interprets and adapts the higher level models.

- Chapter 9, Binding Model and Distribution Model, reviews the design decisions in the modelling layer that describes meta-object representation and processing support for the high level programming models. This includes the adaptive transformation of partially distributed specifications into fully distributed specifications that can be interpreted by the interpretation layer.

- Chapter 10, Specification and Visual Composition Tools, reviews the design decisions in the tools layer, describing the specification tools and illustrating their potential usage with an example.

- Chapter 11, Conclusions, reviews the contributions and conclusions of the research, summarises the results and makes recommendations for future work.

1.5 Reading Dependencies

This thesis covers a lot of ground in order to illustrate the impact of the approach on all aspects of software research, development and productisation. Consequently there are a lot of reading dependencies. It is intended that the thesis is read from cover to cover. Figure 4 summarises the most critical dependencies by defining alternative reading paths as dotted lines.

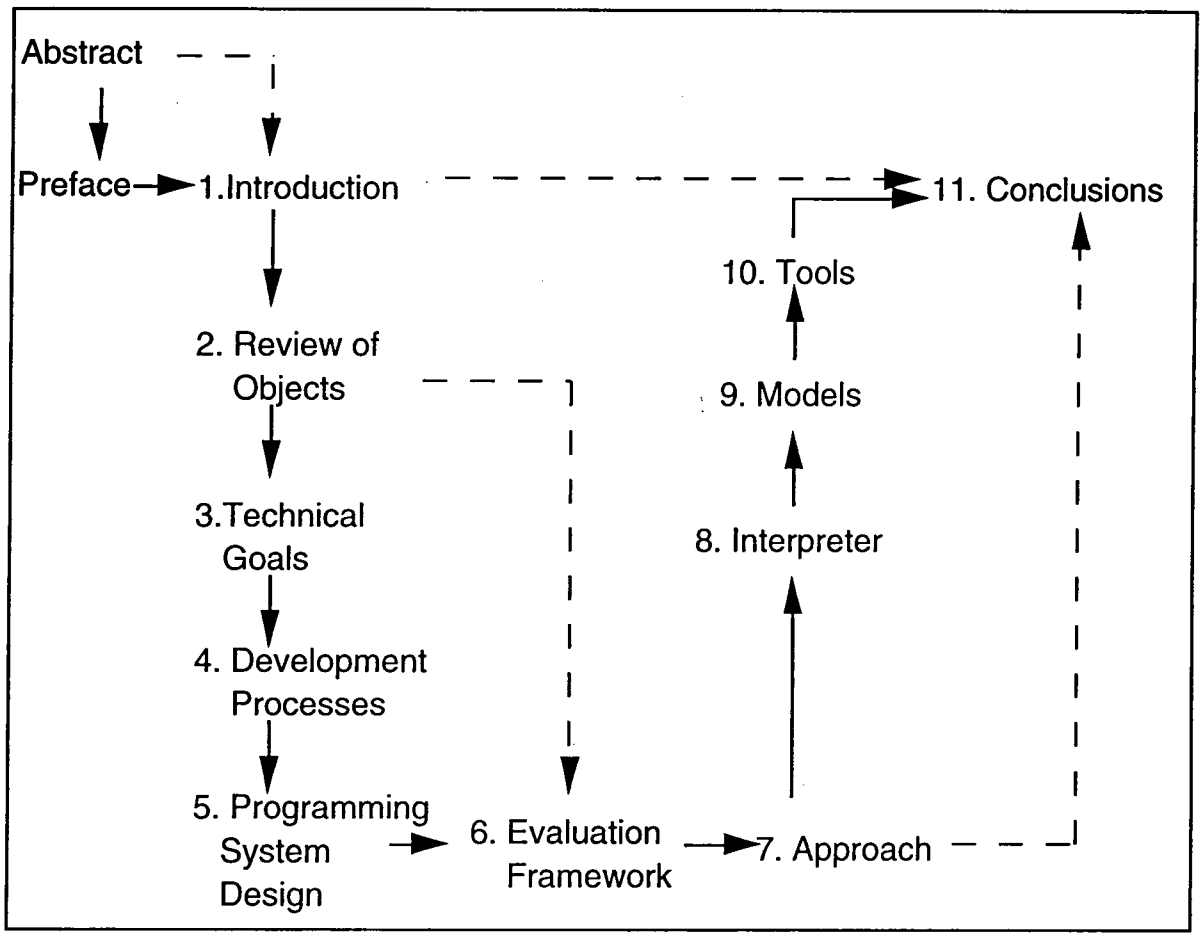


Figure 4 Suggested Reading Paths

Part II Survey and Analysis of Techniques

Chapter 2 Review of the Fundamentals of Object Technology

Information systems are getting larger, more complicated and more interconnected, and demand more cooperation between suppliers. What is needed to deal with these demands is a conceptual framework sufficiently abstract and powerful to represent uniformly and coherently the behaviour and relationships of software systems, the processes by which they are developed and distributed within human organisations, and the way they are integrated and evolve as products. The object oriented paradigm provides a technical basis for such a comprehensive framework. This section briefly describes the fundamentals of the object oriented approach and the benefits it can bring to these activities.

What are objects?

We are all familiar with things (objects) in our everyday reality. Things have measurable properties. Things can be simple or composite (made of other things). Composite things can derive hereditary properties from their components, or obtain new emergent properties of the whole that are not properties of any components.

The view of objects as models of physical things builds on the origins of object oriented programming in simulation (Simula, 1967). Software objects are like 'real' things. They represent information, process and behaviour. They package useful abstractions that form the building blocks that can be assembled in a multitude of configurations to construct the overall properties of software systems.

Despite its long history, object oriented programming is still lacking a profound theoretical understanding. Whilst logic and functional programming languages are based on well understood concepts like equations, relations, predicates, etc, there is no single agreed mathematical theory or model for object oriented programming. Instead it is more often defined by specific programming language constructs, like inheritance. Attempts have been made to give it a more formal theoretical basis, but none have been universally adopted. For example, work on the Beta language (Blair, 1991, chapter 12) is founded in a formal theory for physical information models that is expressed in terms of phenomena, concepts, objects, actions and patterns. (Wegner, 1987) has also worked on more formal theories for object oriented programming.

Recently object oriented ideas have found their way into a diverse range of fields, including operating systems, distributed systems, methodologies, formal methods, and databases. This divergence has made it difficult to define precisely what constitutes an object oriented system. The rest of this chapter presents a series of fundamental views of object technology in order to extract the essence of the technology and the benefits it can bring.

2.1 Conceptual Frameworks of Object Technology

Several conceptual frameworks based on different themes have been put forward to describe the essential concepts, principles and practices that constitute the object oriented approach. These vary in content, generality, formality, terminology and depth. Different frameworks base their definitions on different perspectives:

- on the components of object models - for example the OMG Core Object Model (OMG, 1990);
- on specific mechanisms or techniques (Wegner, 1987);
- on the principles behind pure object modelling techniques (Beta, 1987);
- on the goals and benefits (Blair, 1991);
- on the nature of the development process (Booch, 1991; Rumbaugh et al., 1991).

This section provides a synthesis of definitions from these different perspectives, briefly describing the most fundamental and essential aspects of the object model from different viewpoints. It serves three main purposes.

1) Different models overload different words and attach different meanings to the same word. It defines a consistent vocabulary that will be used in the thesis.

2) Different models are limiting in scope or make assumptions about the exact nature of mechanisms. It presents an overview of object technology that is sufficiently fundamental to challenge the experienced reader to reconsider what is essential, and sufficiently broad to introduce inexperienced readers to less familiar aspects of the technology, in particular, aspects relevant to distributed object technology.

3) A fundamental problem to overcome in understanding object oriented systems is to be able to relate abstract principles and goals to concrete techniques and mechanisms. This section concludes by presenting a framework to map concepts between the different viewpoints such as from principles to mechanisms and this approach is used in later chapters to elaborate the design space for our system and to present the rationale for decisions. The missing framework is illustrated in Figure 5. The components of the framework will be described in this chapter. This sort of framework is based on the work of (Blair et al., 1991) but will be extended in later chapters by the author.

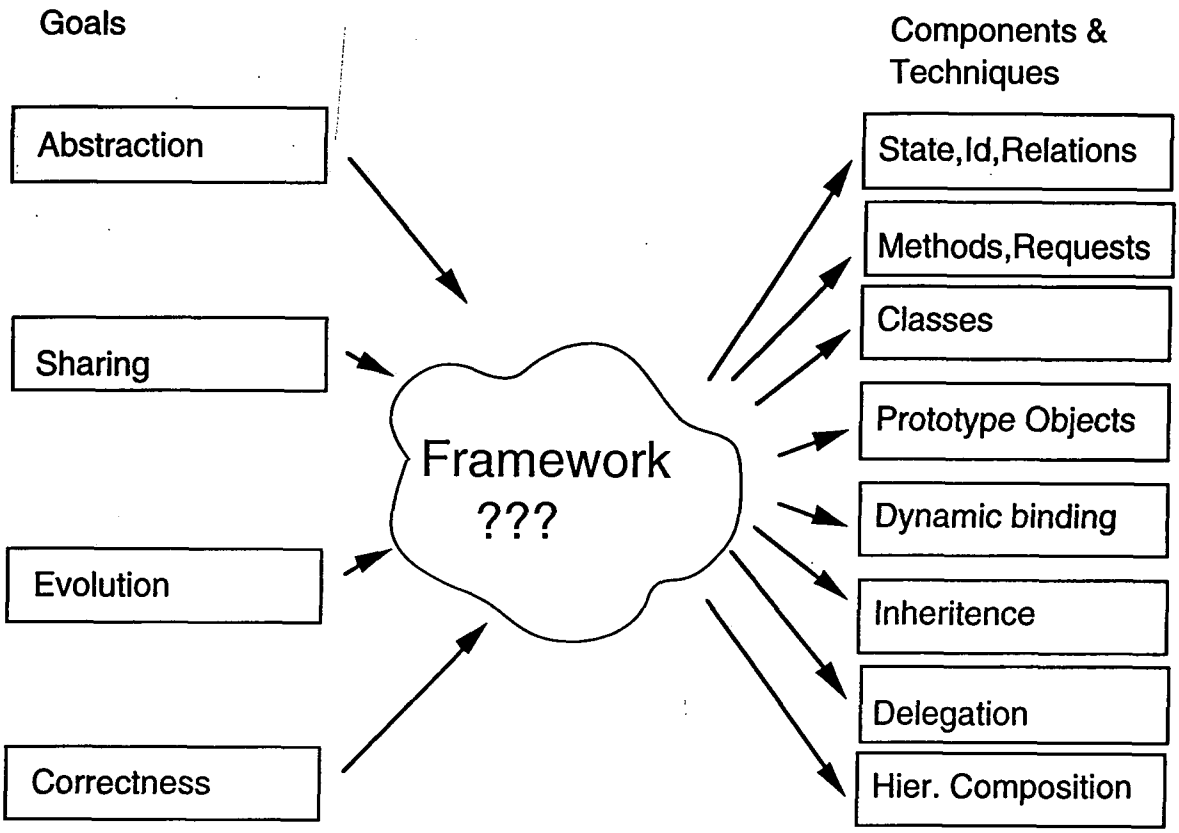


Figure 5 Missing framework to map between perspectives

The objective is not to seek a better object oriented paradigm, but rather to synthesise a variant more appropriate to our specific goals.

2.1.1 An Essential Framework of Concepts and Components

The object model may be defined in terms of objects. Objects can be characterised by static properties, such as structure, and dynamic properties, such as behaviour. This framework describes basic aspects of these properties without committing to concrete realisations.

Static Properties

Object State

The state of an object is represented by a number of variables, called instance variables or slots, that may change over time.

It is usually desirable to distinguish the internal representation of the state, from the way the state is presented and accessed by users. For example, an object representing a 2D point might represent its state internally either in Cartesian or Polar co-ordinates, but provide an interface for users to access it either way without being aware of which representation is actually used. This separation between concrete representation and external interface is called data abstraction.

Object Identity and Name

Each object has a distinct identity which identifies it uniquely. Two objects with the same state, attributes and behaviour might still have different identities.

Languages often mix the concepts of identity and address, using addresses as identity. This causes problems in distributed systems spanning different address spaces.

The representation of identity is particularly important in a persistent, distributed environment, so that objects can reference other objects consistently to access their interfaces. References to other objects are held in global variables or instance variables. Access is by de-referencing these variables. This may involve remote invocation or activation of a passive object from a persistent store.

Object Relationships and Class Relationships

Instance variables that hold references provide one way to represent a relationship between objects. Objects may participate in many types of relationship, some are fundamental to the object model and are not represented by instance variables.

In particular, there are certain relationships concerned with object similarity, which can be used to share implementations or interfaces across sets of objects.

Sets of similar objects are usually grouped together into **classes**, the objects in this set being known as object instances of the class and share the same concrete representation. Classes themselves can be related to other classes using 'is-kind-of' relationships and 'whole-part' or aggregation relationships.

Not all systems base their set concept on classes. Sets of objects may share a common prototypical object. This is the often used to capture weaker for of commonality such as 'is-like-a' relationships. The is-like-a relationship is weaker than is-kind-of, but is useful to form sets of objects by incremental modification of shared prototypical objects.

Relationships are also important to manage distributed or persistent objects. For example, objects may be clustered by an aggregate object to bound :

- the extents of queries.
- the unit of collocation in a shared process.
- the unit of mobility in migrating objects between processes.
- the unit of atomic activation and passivation from persistent store.
- the unit of recovery.

Clustering semantics can be unified with the semantics of object relationships. Object clustering may occur across different relationships - whole-part, is-a , or using. Furthermore, not all extents are defined on object boundaries, for example transactions are defined on activity boundaries.

Object Dynamics

Object behaviour

There are several categories of behaviour that an object can exhibit: State-related behaviour , i.e. accessing and altering the internal state of an object and its parts; functional behaviour, i.e.using an object as a machine to calculate a new object; and operational behaviour , i.e. managing an object implementation in its operating environment.

Active objects are autonomous and concurrent with their own threads of activity, passive objects simply respond to invocations and are dynamically allocated the thread of the caller

Methods

Object behaviour is implemented by pieces of code called methods, normally associated with classes. Different kinds of method implement different kinds of behaviour. There are methods dealing with accessing the state of an object, methods implementing algorithms which provide the functional behaviour, methods which implement operational behaviour such as spawning processes, and methods which route invocations through a composite structure.

Some methods may return a result to the caller while others do not. Some may execute sequentially whilst others execute asynchronously to the caller.

Different kinds of methods may be distinguished by the programming system, for example C++ supports runtime binding selectively for methods declared 'virtual' (Stroustrup and Ellis, 1990). Other languages use one kind of method for all purposes, for example Smalltalk (Smalltalk, 1983). Pattern based languages even use the same kind of abstraction for methods and classes, for example Beta (Blair, 1991, chapter 12) or (Beta, 1987).

Methods are structured in different ways. Some methods belong to classes of objects, particularly those implementing state-related behaviour. Other methods belong to larger units of organisation, particularly those concerned with interactions among several classes. Other methods really belong to the programming system, particularly those implementing operational behaviour. Many object oriented languages insist on all methods belonging to a single class. Some systems introduce meta classes for each class to hold certain class methods and system methods.

In CLOS (CLOS, 1989), methods do not belong to objects but instead they belong to generic functions, which are collections of methods with the same name.

Method Invocation

Different languages provide different forms of expression for invoking object behaviour. The expressions identify one or more objects and the method to be invoked upon them. Arguments may be passed. For example, Simula, Eiffel, Smalltalk and C++ use a postfix invocation form equivalent to o.f(x,y,z) which means 'invoke method f in object o with parameters x,y and z'. Hierarchical composition systems do not name the target object directly but use a local port name to denote the target i.e. port1.f(x,y,z)' means ' invoke method f in the object that is connected to port1'.

In a distributed environment, a richer invocation form may be used to reflect additional failure semantics, for example o.f(x,y,z) abort g(x,y,z) retry (10) timeout (200) means 'repeat message ten times with a timeout of 200 milli-seconds before aborting and calling function g(x,y,z)'.

Fundamentals of Runtime Binding

The method to be invoked is usually determined by the identity of the object or objects in the invocation expression at the time the invocation takes place. A single invocation expression can bind to methods of more than one class and the exact message definition to invoke is not known till runtime. This is called runtime binding. This is often seen as a precept of the object model. It brings a runtime overhead in navigating the class hierarchy to find the appropriate method.

Runtime binding is sometimes mixed with static binding, by distinguishing runtime bound methods, called virtual methods in C++, from statically bound methods or by including the class in the invocation form whenever a virtual method is invoked in a known class i.e. 'c::o.f(x,y,z)' means 'invoke method f of class c on object o'.

Note that static type checking does not prevent runtime binding. A static type conformance check when assigning objects to typed object identities or references, does not tell us which subclass the object actually belongs to, only that it conforms to the type of the reference. Therefore the system must still find the right method for the appropriate subclass at runtime.

Fundamentals of Polymorphism

Polymorphism is the general ability of a software component to fit into many contexts. There are several types of polymorphism, distinguished by the detailed properties of the type system. In essence, a polymorphic object is one which can be referred to by more than one type. A polymorphic function is one which can operate on many different argument types.

Polymorphism and runtime binding give object oriented programming much of its flexibility. To understand why this is so, it is useful to think about a system in terms of producers and consumers of behaviour. Objects collaborate with other objects to implement the system behaviour. This collaboration is expressed by objects invoking the methods of other objects. The designer of a new object is the consumer, and the objects whose methods he invokes are the producers. The consumer makes certain assumptions about the services he requires from other objects, for example that they respond in a certain way. As long as the assumptions hold true, any producer object will do. Polymorphism makes it possible to express these assumptions efficiently, to restrict the degree of flexibility appropriately. Runtime binding enables the dynamic substitution of new objects without any need to inform the consumer.

Programmers can align themselves as producers and consumers of each others product in different ways. New consumer objects may be implemented to use services of an existing library of producers. Alternatively producers can be developed in parallel with consumers to implement an entirely new system from scratch as a bespoke system. At the other extreme, general purpose producers and consumers can both be developed in isolation, and a third external party resolve the producer-consumer binding. The last example calls for statements not only about the services provided, but also the services required and is not supported by mainstream object oriented languages. It also calls for standard sets of services to which both parties conform.

Different mechanisms can be used to express the assumptions a consumer makes about a producer. Flexibility is constrained by type conformance along type hierarchies. Flexibility is therefore maximised if the producer and consumer work with highly generalised primitive types. The extreme is when the types map to typed tuples for the methods , for example void(int,int,float), yet this carries the minimum semantic content. This is fine if an external party is responsible for verification on combining producers and consumers. A tighter specification is safer and generalisation-specialisation or is-a hierarchies are popular ways to express semantic constraints by choosing types at different levels of abstraction up the hierarchy. Even tighter specifications can be provided by using formal models (Stepney et al., 1992) or algebraic/axiomatic methods, including Eiffel style pre and post conditions, assertions and invariants (Meyer, 1992, ref [1]).

Another issue is when the environment can be altered. Runtime binding allows object instance substitutions to occur dynamically at runtime whether or not there is static type checking. But new types can only be added dynamically if the system also supports runtime type checking. Static type checking requires recompilation and re-linking of the type hierarchy for changed classes, and introduces strong dependencies between classes which inhibits an incremental, or prototypical style of development. The need for recompilation can be minimised if the type hierarchy is shallow with stable roots, as for standardised sets of primitive types.

Summary of Fundamental Framework

A fundamental definition of the key components of the object model has been given. This is useful to establish the conceptual framework necessary to look at specific mechanisms and techniques that characterise object oriented systems. The fundamental framework has deliberately been kept broad so as not to restrict our definition of an object oriented system.

2.1.2 A Framework of Specific Mechanisms and Techniques

The object model can also be characterised by looking at specific techniques and mechanisms that typify object oriented systems.

Structuring Mechanisms - Classes, Types, Conformance, Inheritance ,Delegation and Hierarchical Composition

Much of the power of object oriented programming stems from its ability to structure software efficiently and coherently, in particular the derivation mechanisms which allow the sharing of abstractions, behaviour and representations between components. The exact nature of the derivation mechanism varies between systems that use inheritance, conformance, delegation and hierarchical composition.

Classes

A class acts as a template to create objects with the same implementation. A set of objects which share their implementation are said to be instances of the class. In some systems, Eiffel and C++, classes are purely a compile time construct. In other systems, such as Smalltalk and CLOS, classes are represented in the running systems as objects. Classes themselves have classes, called meta-classes.

Types

A type characterises a set of values, that are said to belong to that type. Values include object instances. A single type may characterise instances of more than one class. Likewise a single object may be referred to by more than one type, i.e. polymorphism. A system of types can be used practically to check method invocations at compile time and to optimise code.

The type of an object is often viewed as a definition of its external behaviour, whilst the class describes its external behaviour plus implementation. Clients of an object do not need to know every detail of the implementation of a class ; they need only know about the kind of abstraction that is implemented. For example an abstraction of a stack, can be defined by an interface providing push and pop methods, yet it may be implemented using either arrays or lists. In this case, the abstract interface a stack object presents to clients is quite different from the array or list templates used to create it.

Conformance

If an object presents a certain interface to clients, we say that it conforms to that type. The ability of different implementations of objects to share an interface is known as conformance or subtyping. This is different to the sharing of both interfaces and implementations, known as subclassing or inheritance.

If we view a type as a predicate, we see that subtyping defines stronger predicates.

Inheritance

One class of objects may be somewhat like another, whilst not being identical. Inheritance relationships allow classes to be defined in terms of their differences. If class A inherits from class B, class A can be thought of as an extension of B. Instances of A automatically possess all the attributes that instances of B possess Likewise if type A is a subtype of type B, A conforms to the interface of B. Subtyping and subclassing may be controlled by encapsulation protocols that determine which attributes are visible in the interface of A, which are not inherited in the implementation of A, and which can or must be overridden. For example the C++ protocol uses the virtual, public, protected and private keywords (Stroustrup and Ellis, 1990) Inheritance facilitates differential, incremental programming and code sharing between classes. Most systems, notably C++ and Eiffel (Meyer, 1992, ref [1]) do not distinguish types and classes or subtyping and subclassing but use classes to identify types. This unification is simpler but forces modifications to generate new subtypes to extend implementations or use multiple inheritance to express conformance to multiple interfaces. In a distributed system there may be merit in providing multiple management interfaces to a single object for different distribution mechanisms.

In the above example, Class B is said to be the superclass of A and class A is said to be the subclass of B. Some systems only support a single superclass and are said to be single inheritance systems. Other systems can have any number of superclasses and are said to be multiple inheritance systems. Multiple inheritance may occur naturally in a classification hierarchy. For example, waterfowl would normally belong to the class of creatures-that-swim as well as the class of creatures-that-fly. Multiple inheritance can also be used to add orthogonal behaviours such as distributed mechanism behaviours. Base classes used in this way are often called mix-ins.

Multiple inheritance gives rise to the possibility of ambiguity if attributes share names in different superclasses or superclasses themselves share a common superclass. Virtual inheritance ensures subclasses never derive more than one copy of attributes, even if multiple paths of inheritance exist. Conflicting names can be resolved by imposing an order on the priorities of superclasses or by requiring explicit renaming.

Delegation

Delegation is an alternative to inheritance for code-sharing and differential programming. An object will delegate to another object, acting as a superobject, any calls it can't handle itself. Any object may be a superobject, if it can act as a template for other objects to delegate to. Delegation extends object behaviour across the well understood object messaging interface, rather than relying on new relationships between classes. It's emphasis on objects and independence from any notion of class has particular advantages in certain contexts:

- in user-interface contexts, where objects have concrete visual realisation,

- in runtime typing contexts such as parsers and marshalling code, where the type of an object can change over time or where it is not defined till runtime, making a static association of class restrictive, for example a complex number may behave as a real number when the imaginary part is zero;

- in concurrent systems where a protocol must exist to synchronise superobjects with subobjects. Clearer concurrency semantics can be defined with a simpler model for object composition.

Hierarchical Composition

Hierarchical composition is a useful alternative to inheritance especially when an external party takes responsibility for specification/verification of relationships between objects. Objects can then be developed in isolation and can specify weak constraints on plug compatibility.

The component interface identifies the services required as well as the services provided as typed entry and exit ports through which messages are received and sent. The types may be primitive, general purpose types, resulting in a shallow wide type hierarchy and a high degree of compatibility. An object will typically support many port types, often one for each service provided. This maximises the contextual independence of components.

Object are instantiated from component definitions and bound together to form composites using commands of a configuration language. The paradigm is hierarchical in the sense that composites can themselves export unbound typed ports from nested components and act as component definitions to be instantiated and bound in a higher level composite. In this way composite types can be defined from instances of base components. The mechanism is primitive. Both internal bindings and exported ports must be specified at the method level using the port names. These method level relationships contrast with the class level relationships of inheritance and the object level relationships of delegation.

Modularity

Unfortunately the object-oriented view of modules is often restricted to considering compilation units. This overlooks the coarse granularity that is an important design aspect. Closely related objects form cohesive centres of activity and interest. Classes are one way to partition and structure an application for reuse. But designs may consist of higher level structures than just class instances. Subsystems (Wirfs-Brock and Wilkerson, 1989) are groups of classes that collaborate to fulfil a common set of responsibilities. Composite components may play a similar role in systems supporting hierarchical composition. Contracts(Wirfs-Brock and Wilkerson, 1989; Helm et al., 1990) are higher level groupings of messages. Frameworks (Krasner and Pope, 1988; Profock at al., 1989; Nierstrasz and Papathomas, 1990) are abstract patterns of interaction.

The different ways of structuring code are summarised in Figure 6.

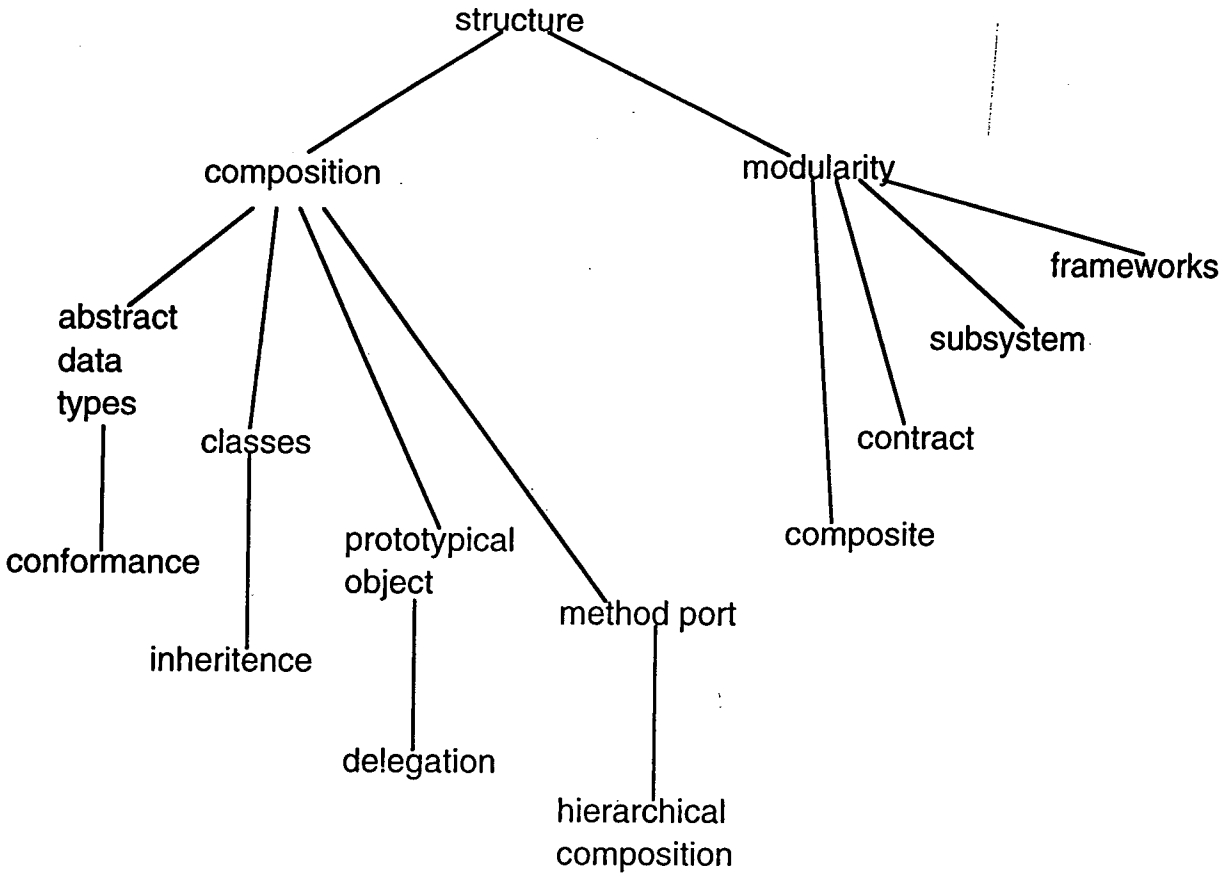


Figure 6 Types of structure

Object Binding

Direct and Indirect Naming

With direct naming a client will identify a server object whose methods he wishes to call using a name or reference that is bound to the server object , e.g. 'name.method ()'.

Not all names directly identify an object. It is often useful to introduce a degree of indirection, for example to allow different objects to be substituted without affecting the client. Configuration languages exploit indirect naming to define the interface between the programming language used to program components and the configuration language used to bind components together (Kramer et al, 1992). For example, as well as being typed, input and entry points may be named as inports and outports in a component interface, for example declaring 'outport outport1 (int)'. The component invokes methods in another component using the outport names in his own interface, for example 'outport1->methodA(5)'. The target object is identified by binding statements in the configuration language outside of the component definition, for example a configuration definition for a composite may declare a binding such as 'bind component1.outport1 to component2.inport1'. Outport1 is thus an indirect name and it is resolved when evaluating the configuration script not when compiling the component definition. The client component is unaware of the actual target component nor even its exact type, only that it conforms to a method that takes an integer parameter.

Types of Polymorphism

A broad interpretation of polymorphism is the principle of making weaker statements about the compatibility of objects in any binding. This includes bindings of object references used in invocations or to access public instance variables. Weaker statements gives the binding mechanisms flexibility to substitute different objects. Type constraints and name matching (direct naming only) restricts the degree of flexibility to those objects which include a member with a compatible member in their interface. Polymorphism is intimately connected to the type system and to the naming scheme.

(Cardelli and Wegner, 1985) define two general kinds of polymorphism: *universal polymorphism* which works on an infinite number of types; and *ad-hoc polymorphism* which supports flexible bindings to a limited, specified set of types. Universal polymorphism includes *coercions*, explicit mappings between types supported by a compiler, and *overloading*, the ability to provide different definitions of an operation using the same name. Ad-hoc polymorphism is more subtle and is best defined using set relations. The concept of a set is similar to the concept of type in that an item shares a type with other items that are instances of that type. If sets intersect, an element of polymorphism is introduced, i.e. entities can have more than one type through set inclusion and be freely substituted over bindings of any of those types. This is called *inclusion polymorphism*.. Blair, in (Blair et al., 1989), distinguishes *explicit inclusion polymorphism* where the sharing is explicitly described from *implicit inclusion polymorphism* where the sharing is a direct result from inclusive relationships between types such as inheritance. One other type of ad-hoc polymorphism is called *parametric polymorphism* where behaviour is defined once but can be interpreted for a number of types, the type being parameterised in the definition. Figure 7 shows a classification hierarchy for polymorphism.

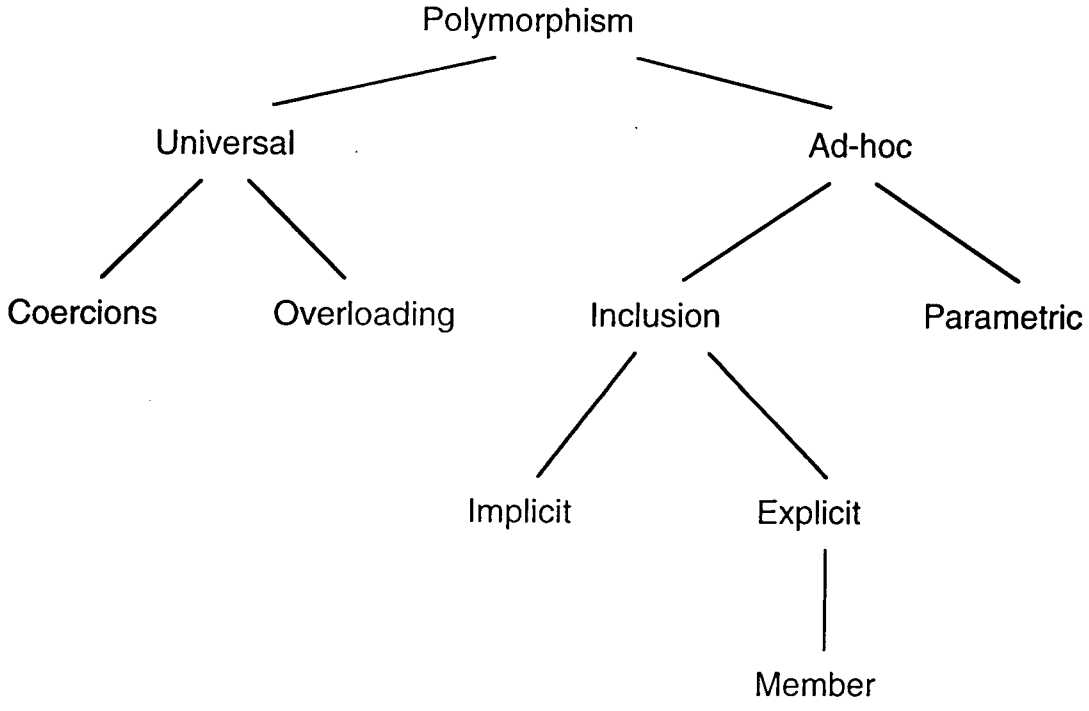


Figure 7 Types of Polymorphism

Implicit inclusive polymorphism exploits the mechanisms used to compose and extend interface types, like subtyping and delegation, by allowing the restriction to be defined as the minimum extension of an interface . If this interface includes a compatible member, any interfaces that derive this interface by subtyping or delegation also include a compatible member. Such a mechanism must resolve any conflict between multiple implementations of compatible methods. For example, C++ virtual methods give precedence to redefinitions of a method in an extension of the type.

Object interfaces may be viewed as sets of typed and named members. Inclusive polymorphism is based on the inclusive properties of sets. However the inclusive relationship used for subtyping and delegation is stronger than is needed. Subtyping only allows conformance if there is an intersection across the entire set but all that is required for safety is an intersection across the members actually used. The minimal inclusive relationship is expressed at the level of individual members. We can call this *member polymorphism*. This is one form of explicit inclusion polymorphism.

Member polymorphism can be supported when bindings are also specified at the member level, as with hierarchical composition. Individual invocations and method definitions are treated as typed exit and entry points. The type system checks exit point types are compatible with entry point types when bindings are resolved. Name conflicts are avoided by indirect naming. Classes and object instances are effectively untyped composite structures of typed entry and exit points. These structures may be extended to create a composite which is the union of the entry and exit points of its components, sharing interfaces and implementations of its components. This scheme offers as much flexibility or polymorphism as untyped object oriented languages and as much safety and efficiency as statically typed languages, but makes binding an explicit programming task. Not only do object instances need to be identified and substituted by the programmer but also methods must be selected or substituted from among compatible implementations, including any shared methods across other relationships.

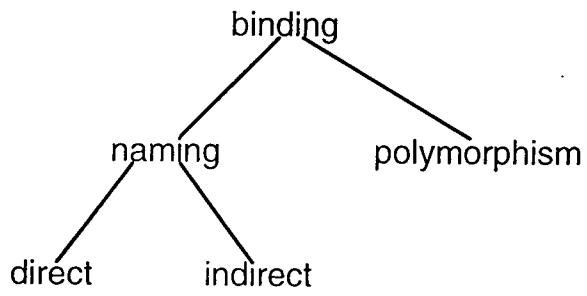


Figure 8 Types of binding

Object Creation

Instantiation from Classes

Classes normally embody the ability to create new objects, thus behaving as templates for their instances. In C++, classes are not objects but are used by the compiler to generate constructor functions that create new instances. Constructors normally also initialise the instance variables. The programmer may provide his own implementations of constructors to initialise the instance variables in different ways. In Smalltalk and CLOS, the constructors exist at runtime as methods of the object representing the meta-class. Meta-class objects can be simulated in C++ environments by generating an extra class with methods which call the compiler generated constructors.

In a distributed environment, not only do we need the constructor capability to be supported uniformly in different processes but we also need some way to identify the node and the process into which the instance will be created and some way to notify the initialisation code in that process. This may also involve process creation.

Cloning from Prototypes

A new object can be created by making a copy of another object. This is a natural way to extend a prototypical object. Prototyping is a valid alternative to classification and generalisation / specialisation for software reuse and incremental development.

Instantiation from Named Composites

Hierarchical composition can be used to define new composite classes from instances of component classes by exporting entry and exit points and renaming them as members of the composite class. Like any class, these composites can themselves be instantiated and bound as components.

Configuration languages make the exporting of entry and exit points trivial. There are three simple steps to defining a composite interface using the configuration language:

1) declare an exit point in the composite interface, e.g. 'outport export_1 (int)'

2) embed an object of class 'classA' in the composite, e.g. 'inst classA objA'

3) export its port to the exit point in the enclosing composite, e.g. 'bind objA.outport1 -> export_1'.

With inheritance, the sharing relationships are defined for classes and methods are shared automatically. With hierarchical composition, sharing is specified at the method level. The entry and exit points need to be explicitly selected from the union of both classes and bound to new exported names in the interface of the composite class, i.e step 1 and 3.

We should clearly distinguish the process of exporting the internal interfaces of nested objects to the composite interface from the internal binding of nested objects to each other. Exp;orting of ports may be called *class binding* since it uses a binding model to define a new class, the composite class, with an interface defined by the exported ports. Binding of entry ports to exit ports can be called *object binding* since it similar to the binding of object instances to references to form an aggregate object.

We can compare class binding to inheritance as a mechanism to specify sharing relationships between classes. However there is no runtime overhead in finding method implementations because the exact binding is always explicit at some higher level in the hierarchy. Hierarchical composition is more primitive, with no sophisticated encapsulation protocols or conflict resolution policies across sharing relationships. This gives flexibility and simpler semantics at the cost of a weaker abstraction of class relationships.

The primitive nature of the hierarchical composition mechanism makes it an ideal candidate for meta-object representations for composition. It conveniently unifies mechanisms for class sharing and object assembly. In theory multiple mechanisms could be mixed. For example, a hybrid system may use multiple notions of class ***compositeclass*** supporting class binding to define a composite class and internal object binding of nested components to define an aggregate object; ***componentclass***, supporting both class binding to export its interface to the enclosing ***compositeclass*** and object binding to resolve its own bindings; and ***subclass*** supporting binding as for componentclass but also supporting inheritance relationship to another ***subclass*** to compose interfaces at the class level.

Exemplars (Blair et al. , 1991) are a hybrid system that mixes object delegation and class inheritance.

Object Destruction

Storage Management

Some systems provide automatic storage management. The built-in ability to periodically check to discover which objects are no longer referred to from anywhere in the system, and reclaim their storage, is known as garbage collection. Schemes vary in efficiency and the degree to which they interrupt the application. Techniques include reference counting , periodic stop-mark-sweep or generation scavenging (Ungar, 1984).

Other systems leave it to the programmer to explicitly deallocate objects when they are not needed. C++ provides destructors which are called automatically when an object goes out of scope or is explicitly deallocated by the delete function. It is up to the programmer to call delete on all dynamically allocated objects. Programmer errors cause memory leaks or attempts to access deallocated objects.

Configuration languages not only make memory management explicit but make object allocation and deallocation a responsibility of the configuration language not the programming language. Hybrid solutions may mix hierarchical components with local dynamic objects.

Distribution makes garbage collection more difficult. For detailed discussion of the issues refer to (D.Tsichritzis, 1989).

Finalization

Finalization occurs when an object is destroyed and is normally concerned with releasing any resources and recursively deleting any embedded objects that are not automatically scoped to that object. It may be carried out by explicit memory management code or by providing the garbage collector with a call-back routine to use on destruction.

Summary of Mechanisms Framework

This section has presented a framework of specific mechanisms. Many of these mechanisms represent distinct styles of object system. In order to provide a more unified view of object technology, the next perspective presents a framework of goals that pervade all systems regardless of mechanism.

2.1.3 A Framework of Technical Goals

This section defines the key underlying goals behind object technology, to establish a basis for rationalising what we really mean by object technology. This is based on the work of (Blair et al.,1991).

Abstraction Goals

The task of understanding can be simplified by partitioning the design and by suppressing unnecessary or confusing detail.

In an object oriented system, abstraction is concerned with both *representation* and *stratification*. For representation, object orientation focuses on *data abstraction*. A set of operations collectively define the behaviour of the data entity. A user is not necessarily aware of internal representation or implementation. Control over access to an entity facilitates maintenance and testing. For stratification, object orientation normally focuses on *object aggregation* , the whole-part hierarchy, and *generalisation/ specialisation*, the kind-of hierarchy. (Also a is-like-a hierarchy often results from prototyping). Stratification is primarily concerned with constructing hierarchies of classes or objects which appropriately structure data and function to break up a complex problem and to package components for extensibility and reuse.

Other types of programming system use different forms of abstraction: for example logic languages use an abstraction for problem solving logic and functional languages use abstractions for mathematical functions. It is the particular approach to abstraction that characterises object oriented systems.

(Blair et al., 1991) differentiates between the use of abstraction to aid understanding of complex issues from the use of abstraction to solve complex problems.

To aid understanding, generalisation has been applied extensively, for example to biological phenomena. Detailed classification hierarchies have been worked out by a process of observation over a long period of time, for example the animal kingdom is understood by using terms like vertebrate, invertebrate, carnivore, herbivore to abstract over various attributes and concepts of a given animal.

To solve complex problems, abstraction has also proven itself invaluable to break problems down into subproblems, for example the design of a car can be broken down into the design of a fuel system, the design of an ignition, the design of suspension and so on.

In computing, abstraction allows us to solve a problem at a series of levels. Abstract requirements are typically mapped to a high level model of the solution. These abstractions are then mapped to abstractions provided by tools and languages. These in turn abstract over the technological detail of the underlying computing system.

The notion of abstraction levels sounds simple in theory. Yet there is conflict between the need to provide good general purpose problem solving abstractions and the need to map them to machine code that runs efficiently on a computer system. Traditional languages and tools have been tailored to particular machine architectures, in particular procedural languages for von Neuman architectures. Languages with higher level problem solving capabilities such as logic and functional languages suffer from efficiency and from their inappropriateness for certain types of application.

In contrast, the abstractions provided by object oriented languages, i.e. data abstraction and stratification, support of a wide range of problem solving areas and there have been efficient implementations. Stratification and data abstraction allows us to structure high level models around real-world and application concepts. The direct representation of real-world concepts has significant benefits. A close conceptual and structural correspondence with the problem domain leads to coherence and involatility in mapping requirements to the model, minimising redundancy, facilitating iterative refinement rather than structural transformation, and permitting changes without destroying the conceptual clarity of the software. Simple modelling of real world entities has been done in the database field and in systems analysis for many years, but previous programming languages have not been expressive enough to carry the concepts across into implementation. The structure clash that traditionally occurs between analysis results and implementation is known as impedance.

Flexible sharing goals

An important aspect of the efficiency of expression is the ability to share concepts, behaviours and structures. Data abstraction helps us to express the notion of behaviour i.e. entities can be defined in terms of their external interface. Stratification helps us to express the notion of hierarchical structure i.e. entities can be characterised by relationships with other entities. Flexible sharing takes these notions further to allow many entities to have the same interface or the same relationships.

There are two typical approaches to sharing: classification and taxonomies. Entities may be classified according to their common behaviour and thus share this common behaviour. For example, a list may be classified as an entity supporting add and remove operations. These operations are shared by all instances of the list. Taxonomies are a refinement of classification that allows one class to form inclusion relationships with another classification. For example an array classification may be included in the list classification and share the add and remove operations. Yet an array may also have additional behaviour such as addAt and removeAt operations.

As has already been discussed, it is possible to have more selective forms of sharing which are not based on classification such as delegation and hierarchical composition. It is also useful to separate the sharing of interface specification, subtyping, from the sharing of specification and implementation, subclassing. Despite their diversity, all these techniques are based on some notion of set and set relationship and all realise the same sharing goal. This goal is important to characterise an approach as object oriented.

Evolution goals

Object technology supports evolution as a fundamental aspect of the computational model. There are two general types of evolution :

1. Maintainability and requirements evolution

If the structure of the software matches the structure of the problem domain, changes in the requirements can be fairly directly mapped onto simple additions to the software, without significant structural changes. Data abstraction isolates volatility to individual problem domain components which exist behind standard interfaces. The ideal is when the relation between changes in requirements and resulting modifications in the software is continuous : small changes necessitating small modifications. In a conventional program, a small change in requirement, such as a simple representation change in postcode or currency, can ripple through a software system and cause potentially unbounded changes. Taxonomies must support evolution and allows new classifications to be added to reflect new requirements. Polymorphism allows these new classes to be substituted without affecting bindings between classes.

2. Solution by evolution

It is often the case that requirements simply aren't known in advance, possibly due to the complexity of raw functionality and implicit non-functional requirements demanded, or due to the impedance mismatch causing misunderstandings between technologists and users, or simply because the introduction of the system is bound to change requirements anyway. There is a growing trend in applying object technology not just to support an existing task but to re-engineer user processes altogether, in particular in designing user interfaces and information dissemination systems for group working, or sharing objects over networks. The only sensible strategy is to prototype, introduce it to users, and to iterate in a design lead manner. Key to this strategy is the ability to extend systems incrementally and not to commit to premature decisions.

The object oriented philosophy is to provide a single approach which encompasses both aspects of evolution, using techniques like data abstraction, set relations and polymorphism to give stability and flexibility in the software architecture.

Correctness goals

Traditionally object oriented systems provided little support for developing correct applications. This allowed much of their flexibility. However for general purpose programming systems intended for more demanding applications, correctness is an important design goal.

(Blair et al., 1991) defines correctness in object oriented systems as being the term used to describe determinant behaviour i.e. whether there is guaranteed to be a valid interpretation for a given request. A determinant system will never fail due to an inability to respond to an operation. Systems which do not support type checking can result in a runtime error when a message is sent to an object that does not support the requested operation. They are non-determinant. Even with type checking it is difficult to make assertions about what the interpretation might be as the interpretation may change over time due to polymorphism. Thus determinance is concerned with abstract levels of specification and is intimately tied to the type system, polymorphism and any algebraic or formal techniques.

2.1.4 A Framework of Abstract Principles

This section takes the abstract goals and tries to generalise some principles from the mechanisms to support these goals. This is also based on Blair's framework (Blair et al., 1991).

Encapsulation

Encapsulation is used as a general term for techniques which provide data abstraction. This includes both modularity and information hiding. The designers attention can be focused on manageable portions of the program if data and processing are bound together into a single modular entity called an object. Furthermore clients should be restricted to accessing the object only through well defined, external, operational interfaces. Objects that hide the information structure behind explicit behavioural interfaces, have weaker dependencies with other objects. An object may provide many different interfaces: to descendants, to client objects and to itself.

If enforced, encapsulation can be basis for security by building security into all servers. Security is not a difficult problem, it is just that it needs to be done everywhere.

Classification or Inclusive Sets

Some form of set abstraction is useful to reason about the behaviour of an object based on its inclusion in a group of related objects. This grouping may be a union of sets intersecting on a shared behaviour or an inclusive extension of subsets sharing all behaviour. All entities in the problem domain can be thought of as belonging to sets that abstract out the common properties and behaviours of the members. This implies there is some mechanism for sharing behaviour. It is possible for a single entity to belong to more than one set through intersection or subsets. In addition sharing across the group may include either specifications only or sharing of both specifications and implementations.

Polymorphism

In the broadest sense, polymorphism is the search for greater compositional flexibility in languages. It may be used as a general term for techniques that support component evolution, in particular techniques that limit the impact of a change in one component on other related components. It thus draws on techniques for specifying assumptions that components make about other related components.

More specific practical definitions may be provided such as the ability of functions to operate on more than one type. However this may prove limiting.

Dynamic Interpretation Techniques

Given the flexible and evolutionary nature of behaviour sharing, it is necessary to provide techniques to resolve the precise interpretation of an item of behaviour. There are a variety of techniques to do this. The interpretation may even be broken into multiple steps: steps that check that a valid interpretation exists; and steps that search for the actual interpretation. Techniques vary in the number of steps, the rigour of the checks and the time when the steps execute for example compile time, link time, load time, runtime or re-configuration time. The later the interpretation, the greater the flexibility. The earlier the interpretation, the more efficient and determinant the system. A compile time check and runtime search gives the best of both worlds but introduces static type dependencies.

2.2 Summary and Conclusions

This section has presented a series of distinct views of different aspects of object technology: the fundamental components; the specific mechanisms; the abstract principles; and the underlying goals. Figure 9 shows how a mapping between these views can be used to characterise and position different approaches to programming system design.

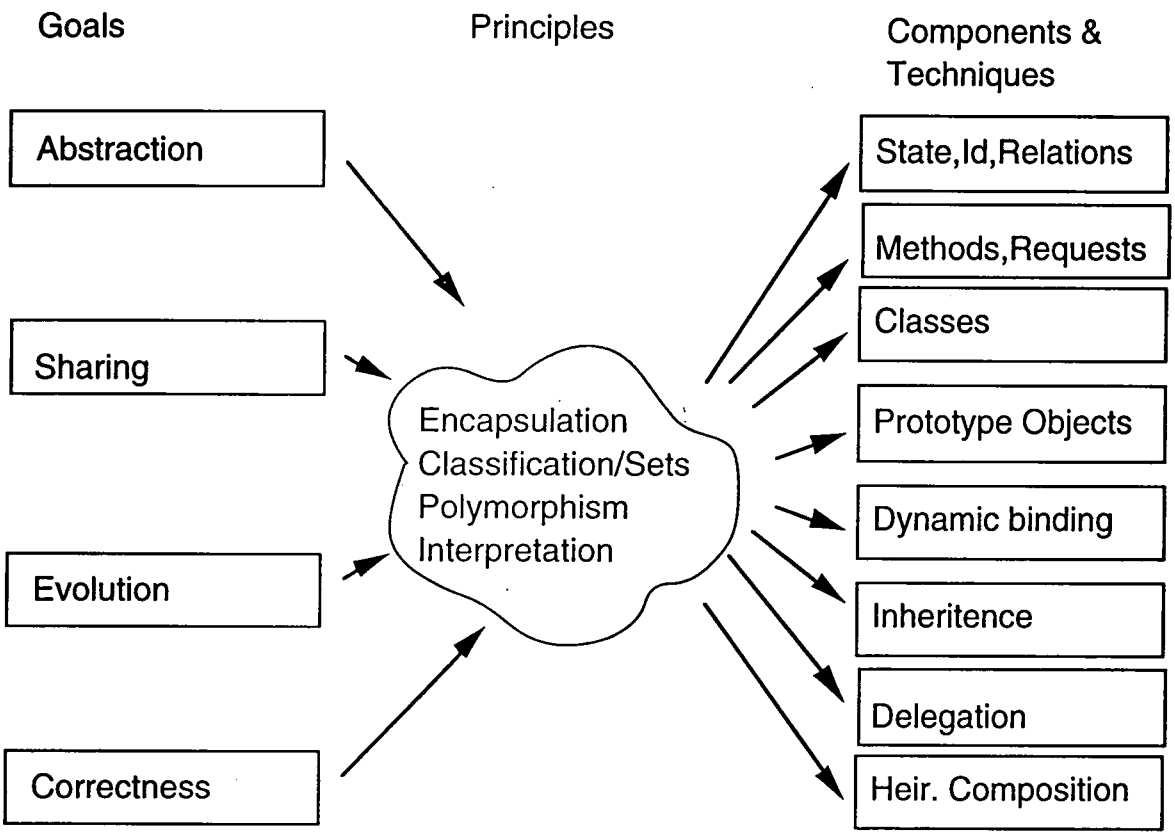


Figure 9 Mapping between Frameworks

By defining profiles of items from this framework, we can discuss the value or design implications of any specific item. In particular, a given project may have a certain goal profile that drives decisions in other aspects of the framework. Different profiles can characterise different styles of object oriented system.

Object technology is not a Mecca that every community is merging towards, rather it is a common vision of a Holy Grail that is inspiring a divergence to satisfy a wide range of mixed personal goals. These goals take subtly different forms in the different communities adopting the technology: distributed systems, operating systems, AI, formal methods, software engineering methods, object oriented databases. The appropriateness of the object paradigm for many styles of system stems from the extreme generality of its constructs and principles and its appropriateness to overcoming the complexities of modern software construction, that characterise what is frequently called the software crisis. Object technology is sometimes seen as the panacea to these problems.

Unfortunately object techniques are not yet well enough understood. Their simplicity and unification into simple mechanisms like inheritance gives them much of their power. However these simple mechanisms are easily undone, especially when dealing with distribution and concurrency. For this reason this section has taken a fundamental look at object technology and included related approaches like hierarchical composition that would not be included in mainstream object technology research.

The growth of object oriented programming is closely related to the rapidly increasing connectivity of computer networks. This introduces rapid technological change. The increased sharing and collaboration, coupled with more intuitive user interfaces, allows software to play a more central role in business processes across an organisation. Often software is changing processes. This in turn increases the exploratory nature of application development, and the demand for software that can withstand changing requirements. As more and more systems become connected, the need for standards and principles of integration and experimentation grows.

The view of object technology presented in this section is essentially one of a single system. A more sophisticated architectural framework of tools and infrastructure support is needed to prevent an overloading of programming concerns to meet the combined goals of distributed systems. The next 4 sections will refine the perspective presented here by considering the impact of distribution on our framework. Object technology is really part of the larger picture that includes distributed systems.

Chapter 3 Technical Goals in a Distributed Enterprise

This chapter describes a framework of technical goals that can be used to position and evaluate the support offered by different infrastructures.

3.1 Introduction

The age of the stand-alone computer system is dying, along with the monolithic applications that run on it. The future of computing is distributed. Corporations are recognising that information is a valuable asset, an asset that must be globally accessible across organisations. Distributed computing technology provides the information highway on which a corporation is built and evolves.

Enterprise-wide information technology solutions are enabled by developments in two parallel areas :

- distributed object infrastructures are breaking down traditional application boundaries by taking business systems beyond client/server applications into the realm of interchangeable components that can be integrated across an enterprise.
- object oriented modelling is transforming business models from technology dependent static abstractions to dynamic well-factored simulations that in combination present a virtual model of the entire business that can be shared across the enterprise.

Object technology is driving these developments due to it's support for both componentisation, the ability to view software as assemblies of independent, pluggable components and virtualisation, the ability to provide tangible modelling formalisms that abstract away from technology concerns.

Componentisation is supported by providing:

- cleaner encapsulation of business logic and data,
- greater flexibility to incrementally evolve the model through polymorphic bindings between components,
- more uniform integration protocols based on object messaging,
- more tangible visual tools and assembly processes arising from the naturalness of object models.

Virtualisation is supported by :

- greater abstraction through classification or set abstractions,
- more declarative control of components through rich interface specifications,
- correct semantic interpretation through rich object type checking techniques,

- greater insulation from technology complexities and other contextual dependencies through selective transparency wrappers around objects.

The next four chapters derive a framework that relates the high level goal of componentisation to the principles of encapsulation, polymorphism, protocol unification, and assembly process visualisation; and the high level goal of virtualisation to principles of classification, declarative properties, interpretation and insulation. Each chapter expands on different parts of this framework.

This chapter expands on the two high level goals by breaking them down into more specific goals that include the object technology goals discussed in chapter 2 :

Componentisation covers the following technical goals:

- component abstraction, to suppress detail and focus on manageable portions of a design.
- sharing and reuse, to reuse component behaviours, implementations or patterns and structures.
- evolution, to evolve or maintain a system incrementally with localised effects from changes.
- platform commoditisation, to integrate or interchange infrastructure components freely.
- application interworkability, to integrate application components freely so that they can talk to each other across a network.
- application re-engineering, to integrate and migrate legacy system components.

Virtualisation covers the following goals:

- correctness, to maintain valid, efficient and determinant interpretations of high level abstractions.
- dependability, to support properties such as performance, reliability, availability etc. in programming abstractions.
- federation, to allow technological diversity by supporting mappings of abstractions across different technology domains.
- group working, to support the notion of a team of users working together across multiple machines.
- enterprise modelling, to integrate and share information across an enterprise and efficiently co-ordinate the scheduling of tasks.
- infrastructure abstraction, to hide the complexities of a diverse, heterogeneous distributed infrastructure.

These technical goals are summarised in Figure 10.

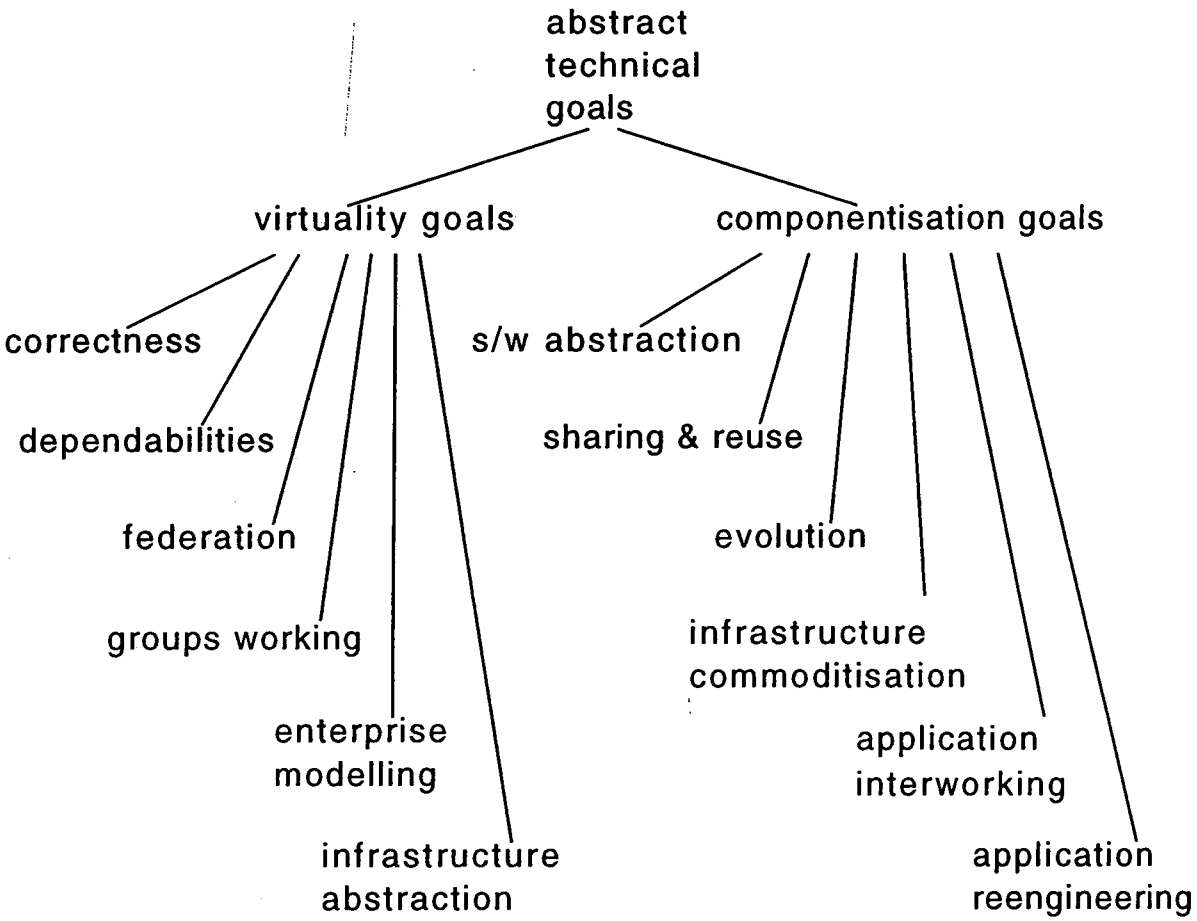


Figure 10 Key technical goals in a distributed enterprise

This framework of goals are the goals that prevail in a distributed enterprise. Failure to take into account the full range of goals limits the applicability of an infrastructure to all the needs of an enterprise. This is true across domains. A software integration platform to support computer integrated manufacturing (CIM) shares many of the technical goals with a software integration platform to support financial applications. Different domains will emphasise different goals to different extents.

This rest of this introduction describes what we mean by a distributed system and a distributed enterprise. Section 3.2 provides a more detailed description of each goal and discusses their implications on technology development. It does not analyse specific business goals in detail. It is envisaged that specific business goals can easily be mapped to these technical goals.

The framework is a synthesis of technology concerns from a number of related fields that refine the view of object technology: enterprise modelling, group working, distributed systems, open systems, object methods, databases. These areas themselves represent a synthesis of disciplines and other technologies that aren't covered by this review: AI, HCI, GUI design, ethnology. The emphasis here is the technology developers viewpoint and the architectural implications of different goals.

3.1.1 What is a distributed system ?

At the hardware level, there is a spectrum of types of distributed system, characterised by their interconnecting network : vector computers, dataflow / reduction machines, multiprocessors, multicomputers, minicomputer networks.

For our application domain, we are only interested in multiprocessors and minicomputer networks. There is a conceptual similarity anyway between systems based on processes, shared memory and messaging concepts. A more useful definition of a distributed system is based on the software architecture.

Distributing a system is a way to organise an application so that parts of the application can be transparently run in separate memory address spaces across a network. Conventional client-server applications can be thought of as having three basic functions : presentation, application logic and data management, that can be separated to different processes. Other applications have less rigid application partitionings in a peer-to-peer style. Processes communicate using a common communication protocol using such structures as sessions and remote procedure calls (RPCs). Types are used to classify data structures to make it easier to identify common operations and characteristics.

Because we are looking to support pluggable software components, we are interested in object based distributed systems. A low level definition of distributed object computing is found in the intersection of the definition of the architectural concepts that constitute distributed systems and object oriented systems. There are several overlapping concepts which map the two together.

- **Communication Protocol vs Interface**—An object's interface defines the protocol required to speak to it. Interfaces can be expressed at both low-level and application level.

- **Process vs Object**—The view of an object as a process is attractive for specification. Objects conveniently tie together data and processing. An active object with its own thread is equivalent to a running process. Object-orientation adds data and function encapsulation and provides a unified communication model. This unification overcomes one of the main criticisms of distributed systems which is that the programmer has to know many programming features to get two entities talking.

- **Session vs Message**—Messages are high-level users of sessions, RPCs or sockets. Using object messaging to control sessions, provides the developer with a more transparent view of the network.

- **Function vs Method**—An object-oriented method is like a function. In the object-oriented world, the method is associated with the data it manipulates to create objects.

- **Type vs Class**— Syntactic and semantic consistency of interfaces is guaranteed in distributed systems using abstract data types and static type checking. Types describe interfaces. Interfaces can be shared by subtyping or conformance. Types do not define implementations. Several different implementations may have the same type. In contrast, a class is defined by the interface and the implementation. Classification results in advanced software construction techniques. Implementations as well as interfaces can be shared by subclassing or inheritance.

- **Clients & servers vs Roles**—Most current distributed systems are based on client/server models. Concepts such as server and client become roles that communicating entities can play. Object models may be any network, one of which is client/server.

The impact of distribution on the object orientation goes beyond specific architectural concepts. A more general definition of distributed object systems it obtained by characterising the reason for distributing.

3.1.2 Why distribute?

There are usually good reasons for using a distributed system. These reasons bring a new emphasis to the goal based framework of object technology. Distribution impacts on the technical goals that are important. This makes direct comparison between a stand alone object implementation and a distributed one a rather fruitless activity. Instead we need to understand these new goals and new types of application. There are six main reasons for distributing an application :

1) Speedup

Parallelism can improve performance.

2) Fault tolerance

Availability can increase with the number of machines and their geographical independence.

3) GUI front-ends

The lifetime and cost-effectiveness of mainframe applications can be extended by front-ending them with new presentation layers that take advantage of user friendly workstations or desktops.

4) Specialisation

Heterogeneity is often treasured. One problem with large systems is that different users have quite different needs and want a degree of autonomy. At the hardware level, different tasks often require different technologies: vector processors for CPU bound tasks, disk arrays for database server tasks etc. At the system level, different users want to use different naming schemes and employ different system management policies. At the data management level, federated databases can be used to avoid performance bottlenecks on overloaded servers. At the functional level, different users may want shared objects and applications to have quite different behaviour. At the application level, different users want to have different versions of shared software and migrate at different times.

5) Platform Flexibility/Commercial Viability

Many users still see hardware as their biggest investment. Network solutions are scaleable, reconfigurable and can evolve incrementally with changing user needs. This flexibility makes them commercially viable. The extra complexity is often overlooked. Users can control the cost of change more easily, taking advantage of changing prices, changing suppliers, changing requirements, changing techniques and changing business policies such as out-sourcing. Cheap workstations are cost effective and open systems are seen to protect the users investment in networks of workstations.

6) Inherent distribution

Many businesses are trying to remove vertical interdepartmental barriers as well as seeking to get closer to customers. This has lead to growing demand for more connectivity in their supporting software systems. Such enterprise wide solutions are inherently distributed with people and manufacturing plants.

What is enterprise integration?

There are a number of fundamental changes going on in businesses that has lead to a reevaluation of IT strategy and goals and resulted in a synthesis of the above reasons that can collectively be called enterprise integration.

Business process re-engineering is changing the focus away from departmental structures to business processes. As companies become flatter in response to competitive and economic pressures, the old departmental barriers are breaking down. The business is moving towards a collaborative work model with more involvement across operating units in decision making.

There are three new pressures on IT departments:

- the business pressure to support more business processes and therefore manage their interdependencies and changing practices. This is driven with a "more-for-less-sooner" mentality.

- the technical pressure to integrate products from more vendors, to make more inter-departmental systems work together, and to address more complex applications.

- the commercial pressure to reduce IT investment to an ongoing operational cost that can be justified at the operational level through the incremental purchase of networked workstation technology and off-the-shelf software rather than the single strategic investment in bespoke mainframe solutions.

There are real tensions in mixing the "off the shelf" approach with existing legacy systems and existing "project-centric" cultures and methodologies. The former demands openness the latter encourages monolithic applications.

These pressures are leading to a re-evaluation of IT goals and processes

Many key business players are beginning to view IT technology less of a competitive weapon than a cost. This is leading to a more open attitude to IT development and driving the commoditisation of not just applications but whole business models. What is technically driving this trend for cost-sharing is the ability of object models to capture general business models that have applicability across enterprises and between companies. POSC for example is a consortium of petrochemical companies that are trying to pool costs in defining common models for their industry. Likewise a consortia of Canadian Gas companies are amortising the cost of developing their next generation customer information systems. Approaches to business modelling are leading to a clearer separation between the business model and the applications and often a clear recognition that the real value is in the model, not the applications.

Instead of demanding monolithic applications that crunch data, business professionals expect business systems to provide a foundation on which the business can grow, a foundation that is defined by changing business opportunities, not simply a high-tech file cabinet defined by preconceived systems designs.

The rest of this section evaluates and defines different technical goals that may be combined into profiles to characterise different strategies for defining a distributed enterprise. In chapter 7, we will characterise the profile emphasised by OpenBase and use this to evaluate it's limitations as a general purpose enterprise wide infrastructure.

3.2 Goals of Distributed System Development

3.2.1 Object-Oriented Goals in the Distributed Enterprise

Object-oriented goals of abstraction, classification, flexible sharing and evolution have been described in chapter 2. They have a particular interpretation in a distributed enterprise.

Previously, information models were static blueprints of business data and function. Using methods like information engineering, their construction was a long drawn out process. During the two-to-three years of modelling effort, the business had usually changed enough to obsolete the work. Nothing in the approach or underlying technology made systems change any easier to manage. For example there may be a corporate data entity for a customer, yet on a separate hierarchical decomposition model, one would find the business functions that use customer data. Nowhere would a complete picture of customer be maintained. All references to the customer would be hard wired to the customer entity. Information Engineering models can't be constructed piece-wise because representations are split across separate modelling paradigms - corporate data models and hierarchical function models.

Object models treat the business concepts like customers as active players making it closer to a simulation than a blueprint. Abstraction and classification help organise and structure the business model more appropriately. Flexible sharing and evolution support are critical to manage the model through its life-cycle. Because objects can be built incrementally and different representations of objects be freely substituted, the simulation can evolve with the changing demands of the business.

3.2.2 Dependability and Performance Goals

The first two reasons for distribution listed in the last section emphasise two specific dependability goals, reliability and performance (speedup). Dependability can be defined more generally as a property of a system that allows reliance to be justifiably placed on a service. Distributed systems offer significant new opportunities for dependable service in terms of the seven properties of :

- performance (throughput & responsiveness),
- reliability (correctness in presence of failures),
- availability (readiness),
- integrity (data consistency),
- safety (avoidance of catastrophic events),
- security (no unauthorised disclosures),
- robustness (continuity of service).

There is a requirement to specify and manage these properties to meet non-functional requirements such as coping with fluctuating demands, predictable deadlines, ordering guarantees etc.. Many systems have introduced the concept of quality of service as the basis for specification and management of dependability. This lends itself well to explicit binding models that substitute different implementations of a service to make trade-offs since quality of service can form the basis of bind time negotiations. Other systems introduce new language constructs such as transactions or ISIS process groups that provide all-or-nothing dependability properties. The infrastructure requires new internal mechanisms to support these new constructs.

Whatever syntactic form the programming interface takes, there is a heavy emphasis on ensuring that the virtual machine supports the right sort of programming abstractions. Dependability considerations impact at many levels: new concepts are required such as different types of failures, new programming models are required such as transactions, new engineering mechanisms are required to support these models such as rollback recovery services and new operating systems such as multithreaded, pre-emptive kernels.

Ultimately the virtual machine on which a programmer develops an application is a combination of the features of the programming language, the operating system and the hardware architecture.

The distinction between programming language runtime support and operating system is frequently blurred. Indeed many language features have been migrated successfully into operating systems where global resource management is easier. Likewise low level system calls have been replaced by higher level language abstractions. What is more important than the form of the interface, is the degree to which the programmer is masked from the complexities of distributed computing by virtual machine abstractions.

Dependability considerations inevitably lead to issues of uniformity right across an enterprise. It is unlikely that the same level of dependability is needed everywhere. Different functionality must be provided for different applications. Thus this raises the question of how to integrate heterogeneous islands of different levels of dependability, such as real-time or fault-tolerance, into a larger system ocean. The problem boils down to how to mix several different virtual systems and is related to the federation goal described next.

3.2.3 Federation Goals

For greater specialisation/ decentralisation, we require more flexible technology. Whilst the trend is to integrate business functions, it is also desirable to protect the diversity, to preserve autonomy and adapt technology to best fit local needs. It is important to match the technical style within each part of an enterprise.

This also applies between organisations. Doing business together should not require a merger.

The business should drive the system not the other way round. Enterprise boundaries must be recognised right down to the management engine. This requires the infrastructure to support some notion of domains and gateways between domains. Gateways convert from one set of protocols or management policies to another. Domains may also have hierarchy to provide different levels of unification. These architectural constructs are at least as important as infrastructure functionality but are often overlooked by developers and authors.

3.2.4 Groupworking Goals

Personal access to computers is no longer unusual. Personal computers have penetrated large segments of traditional work practices. However most of these systems are too "personal". They have been considered in isolation of other users, or groups. Yet most work practices rely on the co-operative activities of groups of people. The result of the design philosophy of isolation is that group of users must communicate around the computer not via the computer.

The growth of networking technology has enabled the development of a range of different co-operative applications or groupware. These applications directly support the work of groups by encouraging and supporting co-operation between users. The computer becomes the window for communication between the group. Analysis shifts from supporting individuals working on fragmented tasks to supporting teams as complex structures in their own right with distributed knowledge, skills and roles.

Groupware makes the user aware that he is part of a group rather than hiding and protecting him from other users. Group awareness has implicit significance in work and technology support. A compromise must be achieved between transparency between users for simplicity and awareness to facilitate co-ordination of a team.

The majority of groupware systems build on existing and proven computer technologies. There are two main techniques: systems that support information exchange such as electronic mail and workflow and systems that support information sharing such as electronic white-board and decision support systems. These approaches may also be combined to create a more comprehensive co-operative system, for example electronic meeting systems which combine information sharing with informal group interaction.

Advances in groupware are dependent on the facilities offered by distributed technologies. There are a number of technology development principles and design philosophies that must be re-examined in the context of implementing a group working application:

- Protection/Security - Conventional architectures deliberately control the actions of users in order to prevent damage or interference with other users. This inhibits collaboration. Servers and resources must be shared by groups and access synchronised, rather than servers being instantiated for each user and access serialised.

- Synchronisation - same-time sharing requires interactions to be synchronised. Interaction using multimedia also bring severe timeliness constraints such as the lip-synch problem for video and voice channels.

- Group Interaction Protocols - Group interaction is not point to point but involves many to many interactions between multiple users.

- Decision Making Support - Users don't just share information but they also make decisions, hence AI capabilities may be important. Group decision making increasingly requires the computer to understand how to compare and combine results, not just combine the visual representation on a shared whiteboard.

- Information Retrieval - The focus is also shifting to specifying information requirements and information flows, locating and routing information effectively and presenting it to users and tasks rather than manipulating data shared in a common database.

3.2.5 Enterprise Modelling Goals

On a broader scale, the flattening of organisational structures and increasing importance of product quality, flexibility, time to market and improved customer service has lead to the need to share more information across an enterprise and coherently schedule tasks to avoid bottlenecks, meet deadlines and allocate resources most effectively. This co-ordination of information and tasks between distributed agents in an enterprise is an important aspect of enterprise modelling. An enterprise is any collaboration between business units whether within or between companies or along a value adding supplier/production chain.

Most technology developers work in this area has so far been conceptual, primarily delivering simple workflow analysis tools, methodologies and descriptive standards rather than prescriptive standards or architectural support.

Perhaps the most important aspect of enterprise modelling is information integration and dissemination across an enterprise. The increasing global availability of corporate information is changing the types of applications being supported. Information is being recognised as an important asset not only in supporting operational tasks but also for tactical and strategic decision making. There are a number of changing technology requirements on the information management infrastructure:

- whilst operational tasks typically use internally generated information, tactical and strategic managers also require more external information. Information is becoming more of a commodity that can be bought-in.

- managers are already overloaded with information hence considerable value can be leveraged by improving selection and presentation facilities. Information must be relevant, timely and high quality. Multimedia network technology has huge potential for business as well as taking the enterprise to the home.

- there is considerable opportunities for intelligent agents to aid information dissemination and decision making, hence AI techniques are being combined with information retrieval technology (Bock, 1994).

- to increase the context sensitivity of the information, there is greater focus on the semantics of the information requirement - including levels of abstraction and aggregation.

- there is a greater need for ad-hoc access to information or for rapid application development to support rapidly changing decision making requirements. This is not supported by infrastructures based on static control structures like procedure calls. Interactive querying, visual development tools and RAD tools are proving useful here.

3.2.6 Infrastructure Abstraction

In a distributed enterprise, information technology provides a foundation on which the business is built. The term that best describes this foundation is infrastructure. Infrastructure connects together all parts of the business, provides any mechanisms for sharing and co-ordination to enable the business to operate as a coherent whole despite cross-vendor, cross-administration and cross-culture divides. Sharing includes objects (e.g. customers), products (e.g. off the shelf software) , or management policies (e.g. naming conventions).

In the past infrastructure to an IT professional meant only technology. It was defined in how terms: hierarchical or relational databases; SNA or TCP/IP networks. The implementation of the infrastructure could not be separated from the applications because the interface between them was articulated in terms of how computers worked not what services were provided. The technology foundation therefore shaped and constrained the system, which in turn defined and constrained the business by providing support structures that reflect the technology not the business itself. Object technology allows a broader definition of infrastructure. The goal is to raise the level of abstraction, redefining technology infrastructure in terms of what technology services are provided not how the services are implemented. The result is a service -oriented architecture that abstracts away from the implementation technology and specific procedures.

The level of abstraction can even be raised to support design and analysis tasks by integrating CASE into the infrastructure. This breaks down into specification goals, the development of appropriate notations that are expressive, relevant and construct models that are efficient to implement, and toolset goals, to provide an integrated framework or tool architecture into which support for the specification goals can be developed and co-ordinated.

3.2.7 Infrastructure Commoditisation

Previously IT costs were centralised as a strategic once-off corporate investment. There is a growing trend for users to make price/performance trade-offs at the level of operational units. This is enabled by flexible platform architectures that allow different infrastructure products to be freely substituted and integrated. This flexibility also allows users to try out innovative immature technologies without the degree of risk usually associated with technology choices. Contingency planning is easy if there are alternative products that can be substituted. This allows users to take a shorter term view and evolve the system at their own pace.

Platform commoditisation support is primarily concerned with standards for portability , configurability and manageability:

- Portability is important if software products are to be integrated independently of hardware strategies.
- Configurability is important if changes are to be made with minimum effort and minimum disruption to service.
- Manageability is important since you need to know what's out there and who's using it to manage changes effectively and measure success.

Platform flexibility allows low-risk, rapid, and incremental adoption of new technology and new products. No-one wants to be first, no one can risk being last.

Portability goals currently demand high volume product solutions, since custom solutions can not justify the investment to take maximum advantage of each target platform. The introduction of standards such as POSIX, unifying the interface to different UNIX systems, SNMP, CMIP, OSF/DME and ISO Managed Objects (Kramer, 1993) making it easier to manage networks, and CORBA (OMG, 1991) making it easier for applications to talk to each other irrespective of location or operating system, yet these standards have not stabilised sufficiently.

Unfortunately standards tend to be targeted towards specific IT cultures. For example, the ODP community behind CORBA have been proving their ideas primarily in the UNIX/TCPIP world. ICL have found that there is still some work to do to implement ODP architectures in other environments such as OSI, VME etc. On DOS machines the size of the runtime library becomes important.

The development of portability standards, portable operating systems, and generation tools like CORBA IDL make it easier to develop cross platform products.

The development of polymorphic object oriented languages, configuration languages and configuration tools, makes it easier to re-configure with minimum effort and minimum disruption to service.

The development of system and network management products, hardware description languages, and resource managers makes it easier to manage networks to accommodate changes.

3.2.8 Interworkability Goals

Interworkability goals are related to the infrastructure commoditisation and group working goals. All are concerned with collaboration. Group working is about collaboration between users. Infrastructure commoditisation is about collaboration between infrastructure components. Interworkability is about collaboration between applications. Inherently distributed enterprise solutions demand increased co-operation, openness and connectivity across technology and application domain boundaries. Their growth signals the death of the monolithic application in a proprietary environment. Standard based distributed objects provide the only sensible way forward for improving on the ridiculously low level of communication and co-operation that currently occurs between applications.

The traditional approach to communication between business applications is via file formats. However every application defining a format requires every other application and every new release of the same application to include code to support it. Each new program needs additional code and old programs can't communicate with new ones.

Corporations frequently communicate by sharing a common corporate data base. However once again there are problems maintaining the data model across the enterprise and schema or application changes can be costly to manage.

Another approach to communication between applications is using network communication protocols. Like file formats, communication protocols deal with standardisation at the wrong level of abstraction for large scale integration. They provide very low level services, and do little to standardise application interfaces, let alone extensible ones.

The inability of concrete file formats or network protocols or corporate databases to cope with evolution has lead standardisation efforts to consider abstract types of object instead. It is important to understand the implications of object technology on the extent to which applications can interwork. With existing technology, even something as simple as a name and address can only be communicated between applications as a lump of text. Once transferred it is treated as a lump of text. It can't be asked to print itself on an address label, without cutting and pasting. On the other hand, with object technology, a shared object modelling an address could be asked to print itself on a label in a remote application simply by calling the print label method. Objects allow richer semantic information to be communicated. Objects can communicate behaviour as well as state. New objects can provide interfaces which conform to old standard types, providing a restricted view of the new data. Thus change does not always lead to obsoletion.

If shared objects provide the behaviour in this manner, then you might ask where is the application. Shared objects free applications to focus on ways of managing them, finding them, putting them together into composites. The application is the framework in which to co-ordinate the sharing. Future object solutions to interworkability goals will make applications appear more like operating systems. The boundaries are going to change, as well as what people call them and how they understand them. The Penpoint operating system from GO already provides such an application-less environment of objects.

It is important to understand the cultural constraints that inhibit the interworkability of applications. Issues of trust, different concepts and terminology, skills mismatches and commercial considerations all inhibit interworkability. There are three main application cultures that must be brought together: the data culture, managing the corporate information base and its integrity; the control culture, providing processing-centric applications using dependable messaging and online transaction processing (OLTP); and the PC culture, empowering users with document-centric tools. Each has done a good job of defining its own native low level standards such as SQL, RPC & OLTP and OLE . However the disparate needs of these cultures mean that they are unlikely to grow into each other easily. This has already lead to legacy-style problems with new systems. What is needed is a generic abstract view of these access interfaces as services that are independent of the technology. RPC and objects won't displace the cultures, abstraction might and so interworkability goals share much in common with the goals of infrastructure abstraction and component abstraction.

3.2.9 Application Re-engineering

Application re-engineering is the process of migrating an application onto a new architecture. This should not be confused with process re-engineering which is about business processes.

Commoditisation and interworkability can be met in future systems by imposing some open architecture on new developments. The goal of meeting portability and interworkability in existing systems requires a different approach. Legacy systems must be migrated into the new architecture and the new architecture must support on-going evolutionary change.

The monolithic mainframe is a myth. Software engineers have been building modular software for over twenty years. The problem is gaining access to the modularity. This can be overcome with object wrappers that add future flexibility to legacy systems. This sort of flexibility is important to respond to changing supplier prices, technologies, customer requirements and management policies - to allow control over when and how much of a legacy system will be upgraded at any time.

There are a number of different styles of wrapper architecture: data download, remote terminal, query server, procedural API etc. These styles of architecture are again as important to developers as infrastructure functionality, yet aren't given the same emphasise in literature.

The re-engineering requirement is not a one-off transitional need.Today's new systems will be tomorrow's legacy systems. Future flexibility should be architected into today's systems. Re-engineering goals will persist and re-engineering goals for the last generation of systems will remain subtly distinct from the interworkability and portability goals of the next generation.

3.2.10 Large-Scale Reuse Goals

Reusability is widely believed to be a key in improving software development productivity and quality. The reuse of software components amplifies the software developers capabilities. It allows him or her to write fewer total symbols in developing a system, and to spend less time in the process of organising those symbols.

However whilst reusability is a strategy of great promise, it is one whose promise has largely been unfulfilled. Reuse is often too narrowly defined. For example, conventional object technology offers powerful mechanisms for code reuse, yet source code induces a high degree of specificity on the reusable components and code oriented reuse has so far been limited to small and simple components, such as linked lists, GUI objects, and well-understood numerical functions. Small components leave a lot of work in building the architectural superstructure that binds the components into the whole system. Such narrow views of code reuse have not yet met the huge potential of broader reuse strategies.

Distributed systems bring technical diversity and administrative divisions that make reuse more challenging. Not only have we to reuse across platforms but also we must reuse across cultures and across application domains where the level of trust in reused code may be small. To overcome these barriers we need a more expansive view of reuse that goes beyond small scale reuse of code. The expansive view is not limited to technology issues. There are other important viewpoints to reuse, in particular cultural and component management, that are not given the attention they deserve.

Reuse is the re-application of a variety of kinds of knowledge of one system to another. This is not just code, but includes artefacts of domain knowledge, development experience, software architectures, technology constraints, as well as analysis results, designs, documentation and so forth. Representations must allow a broader range of information to be specified than source code can accomplish. This should include requirements, design structures, dependability properties, resource usage characteristics, behavioural abstractions and roles played by components.

Objects provide a common conceptual framework that carries across from analysis results to code. One can easily conceive of an integrated development environment that formalises the representations of design and code and allows large-scale reuse of designs as well as code. However few tool environments are truly that integrated.

It is unfortunate that despite suggesting that designers should reuse the results of other applications in the same domain, most popular object methods (Coad-Yourdon, 1991, Rumbaugh et al., 1992) emphasise construction activities rather than adapting existing designs. Some environments that have taken large-scale reuse seriously such as ITHACA are discussed later.

Not all reuse is based on component composition. Chapter 4 also discusses generative tools that reapply transformations instead of reapplying components.

3.3 Summary of Chapter 3

This chapter has presented a description of a series of technical goals that in combination characterise what has come to be called enterprise integration. Enterprise integration is enabled by the growth of workstations and networking technologies. It is driven by the requirement for more collaboration and for more flexibility, to integrate more functions across a business and to evolve the information technology foundation of a business to be responsive to rapidly changing demands

Enterprise integration is relevant to OpenBase which seeks to integrate applications across a manufacturing enterprise, including: process control, materials requirement planning (MRP), shop-scheduling, CADCAM, simulation.

These goals may be combined into different profiles. The profile of goals that are emphasised in any infrastructure limits its general suitability. In particular failure to take into account recent trends in the types of applications being developed, such as groupware, could very quickly make an infrastructure obsolete. This framework of goals will be used in part III to evaluate OpenBase.

Enterprise integration is a broad field. This report provides a technological view of enterprise integration only, since it is evaluating an architecture. The more conventional view is user-centric. Enterprise integration demands a synthesis of many disciplines: business modelling, HCI, AI, organisation design, ethnography. The framework does not touch on the full range of user benefits. Consequently the scope of this framework is limited to architectural definition not application development.

This chapter discussed the significant new requirements that these goals place on infrastructure architecture. The impact of the goals on the technology is significant and demands a reconsideration of what we really mean by a distributed system. It is not enough to merely look at the physical architecture.

We can characterise a distributed object system according to the relevance of the different goals. These include:

- component abstraction
- flexible sharing
- correctness
- software evolution
- dependability
- federation
- group working
- enterprise modelling
- infrastructure abstraction

- infrastructure commoditisation
- application interworking
- application re-engineering
- large scale reuse

The next four chapters will survey the techniques that can be used to realise these goals.

It should be remembered that distributed usually means multi-vendor, multi-process and multi-user. It is not surprising that the resulting global issues impact right across applications, languages and operating systems:

- the multi-vendor nature results in more co-ordination issues that are best dealt with in the application
- the multi-process nature results in more system and resource management issues surfacing in languages.
- the multi-user nature results in more access and sharing issues that are better dealt with in the operating system.

In discussing the architectural implications of the above goals, notions of application, languages and operating system have already been merged and interchanged. The conventional separation into operating system issues, language issues and application development issues is not very useful. This observation has affected the structure of the survey. The term programming system is used to denote both the programming language and the operating system, so as to avoid the need to distinguish language features from operating system features.

Rather than surveying languages and operating systems, chapters 4, 5 and 6 survey the processes and techniques that can be employed in programming system design.

Chapter 4 Open Distributed System Development and Tools

How do we develop open distributed systems?

Large distributed object systems are hard to specify, develop, maintain and sometimes to use. These systems are complex to develop using conventional methods because they are made up of a large number of interrelated parts and the relationships between the parts are implicit. Efforts to control this complexity range from life-cycle models and methodologies to high level communication technologies and development environments. This chapter elaborates on the discussion of object oriented development in chapter 2 to reflect some of the approaches taken for open distributed object oriented systems.

Distribution has a considerable impact on the process of software development, in particular there are two significant effects:

- the need to allocate objects to the physical architecture makes it useful to distinguish different types of objects that vary in their granularity and in the method by which they are identified and related to other objects. This is because the granularity of objects and degree of coupling is critical to allocation decisions;

- the need to co-ordinate components across a network makes the choice of communication technology particularly important. This is made more difficult by the disjoint nature of different communication technologies.

Openness is interpreted in the broadest sense as meaning that the software architecture consists of substitutable and portable commodity components from different vendors that are reused in different contexts, across the network and in different areas of application. This also has two significant effects:

- component reuse demands a standards-based component-oriented life cycle so that off the shelf components can be selected and fitted together despite being developed in isolation of each other. Application standardisation and component reuse are not part of the traditional top-down life-cycle models.

- large scale reuse across a heterogeneous network demands the development of new integrated tools to select, manage, migrate and customise components. This results in new tool-based reuse systems.

This chapter elaborates on the discussion of object oriented development to reflect some of the approaches taken for distributed systems. This chapter is organised in the following sections that address these four concerns:

Product development method, this section discusses two key aspects of development methods:

- *Life-cycles*, discusses the appropriateness of different life cycle models, in particular the impact of rapid innovation in tools, standardisation and reuse on the product life cycle.

- *Types of Object*, this defines different notions of an object according to the method used to identify and represent the object. This provides a conceptual distinction between different styles of distributed object system. The immaturity of the field is highlighted by the inadequacies of any one approach taken in isolation.

Choice of technology, this section discusses two key choices that need to be made:

- *Communication Technology*, provides an approach to select the most appropriate communications technology.
- *Reuse Tools*, describes approaches to develop tools that support large-scale reuse across platforms.

4.1 Product Development Method

A development method consists of a number of components as described in (MacLean, 1992) :

- concepts, for example the notion of an aggregate classes.
- overall strategy, such as a top-down waterfall life cycle.
- notation, for example OMT's diagrammatic notation for objects (Rumbaugh et al., 1991).
- procedures, for example use X-notation to create model-Y.
- heuristics/metrics, i.e. rules to guide decision making such as when multiple inheritance is allowed.
- tactics, i.e. concrete suggestions such as underlining nouns to find objects.
- tools, supporting the method, such as diagrammatic tools.

The conceptual and strategic components of a method are the most critical. Good notation, tactics, tools and heuristics will not compensate for a method founded on inappropriate concepts or an inappropriate overall strategy. Notations, tactics and tools can come after the basic concepts and strategy

The importance of the concepts and strategy is also true of open distributed system development. Yet it seems that it is tools that are coming first. Most literature talks at the level of specific functionality and communication tools. As a result, the conceptual and strategic implications of distribution and reuse are not well understood.

The confusion over basic concepts and strategies is a symptom of an immature, innovative field. Distributed object technology brings a whole new set of concepts yet there is relatively little experience of applying these concepts. The relative immaturity of methods, standards and tools both for distribution and reuse, means we can not rely on the level of support and process maturity that we are used to in a stand alone environment.

This section focuses on strategy and concepts. It first considers the impact of innovation, standards, tools and reuse on the strategy or life cycle. It then defines several variations on the concept of an object that characterise different methods. A full discussion of methods is outside the scope of this survey.

4.1.1 Life-cycles

Life-cycle models define steps in the development of a software system such as requirements, design, implementation and maintenance. Life cycle models give structure to the software development process, for example by ensuring that requirement are defined before the system is built. Sometimes life-cycles are too restrictive and force project managers to take short-cuts. This is a symptom of the need for more practical life-cycles that truly reflect the software development process.

With distributed system we may be interested in different types of life cycle. The technology developers that productise infrastructure tools have a different life cycle from application developers that build system for end users.

Innovation Life cycle

Distributed systems are usually innovative in some respect: be it in the communications technology, the software architecture or the way other systems are integrated. Distributed object technology represents a significant investment in new skills. Skills and experience must be built-up quickly. Users wishing to move to distributed systems are required to be innovative in their choice of tools and methods. Technology developers and vendors are required to be innovative to overcome the complex problems that must be addressed for distributed computing.

How we manage the innovation process is probably the most important aspect of distributed system development. Innovation in software technologies should be driven by application users. Application-led solutions need quicker feedback from users into the innovative process than traditional hardware technologies.

The traditional linear flow from research to technical development to product engineering to production and subsequent feedback to research may well be appropriate to base technologies like semiconductor technology. However this process is too slow for soft technologies like middleware that progress at a much faster rate. A better structure is to fund early applications. They can proceed in parallel with technical development as soon as ideas have been proven in research. Early applications provide rapid feedback throughout the whole process of technical development, product engineering and production:-

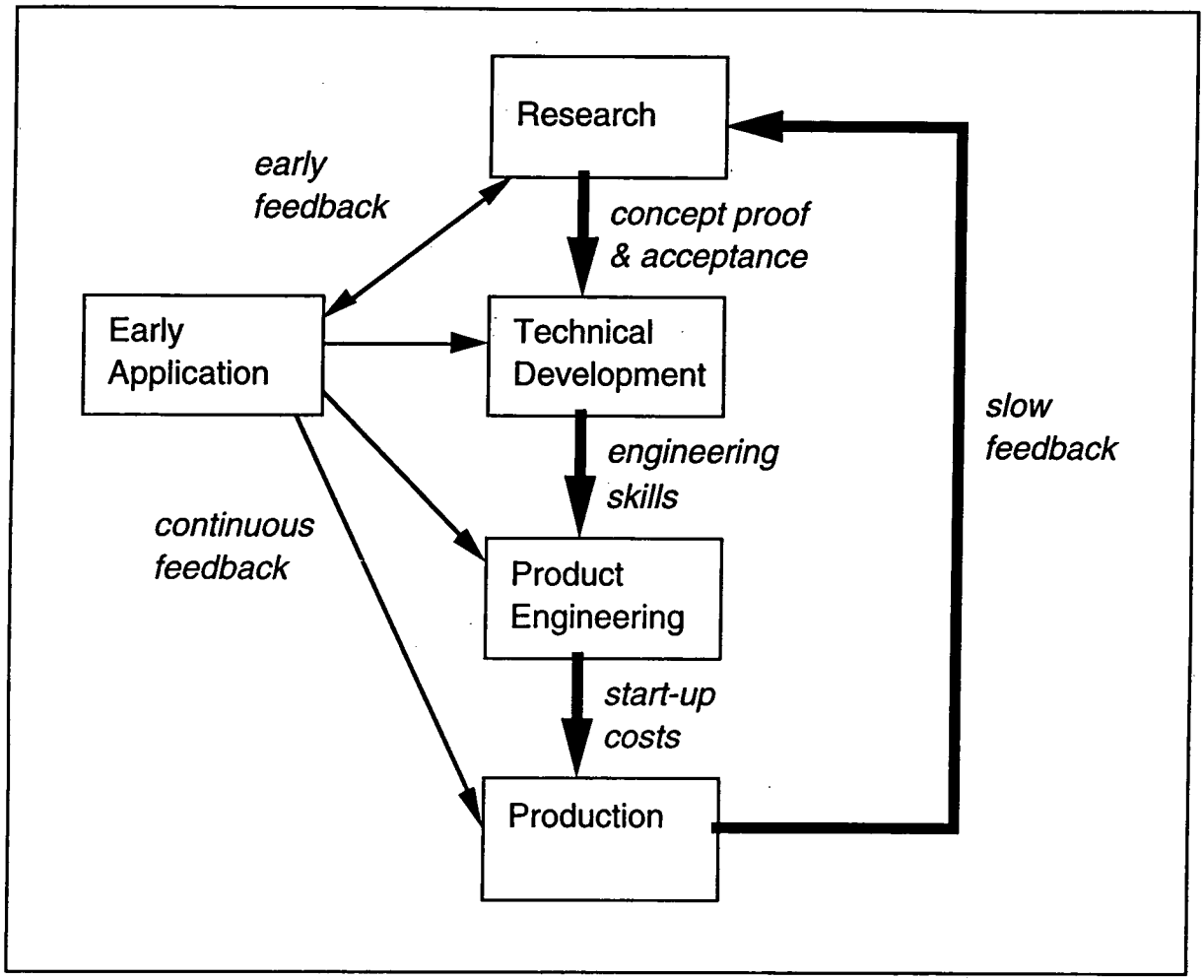


Figure 11 Innovative product life cycle

Such an innovation process needs: new supplier relationships (early application demonstrator), new processes (rapid development and prototyping), new technologies (GUIs to give ideas more visibility), new architectural support (gateways to isolate and control the integration of early experiments into an infrastructure), and new organisations (collaborative funding structures that involve end users in the innovation process, industry structures that achieve early consensus on de-facto standards).

ANSA has been managed as a collaborative research body that promotes these goals. It has proven its research in a demonstrable infrastructure ANSAware. This may be viewed as a prototyping tool to provide an early application, although at version 4.1, it is more mature than the recent surge of Object Managemnet Group CORBA compliant products. Gradually ANSA concepts and components can be re-implemented as the product matures.

There are now several products that began their life in this way including ICL's DAIS which began in the OASIS project of Scottish Hydro, a utility company.

Consortiums and Standardisation Impact on Life-cycles

To facilitate reuse there must be some agreement as to how the parts that are to be reused will fit together. To facilitate distribution across a heterogeneous, system there needs to be agreement as to how interactions are to be encoded and transmitted across the network. There are several standardisation efforts that are attempting to establish common models for reuse and interoperability.

The basic problem is that it is difficult to get agreement between vendors. After all they are competitors in one of the fiercest industries. The historical definition of a standard has a coincidental relevance to the reality of modern high-tech standards.

definition standard n. Battle insignia or tribal totem

The distributed computing community has recognised the need to form a body to standardise high-level application interworking. This body is called the Object Management Group (OMG).

The OMG is a group of organisations and individuals formed in 1989. Today the OMG's members number 330. This includes most if not all the major industry players such as IBM, DEC, HP, Microsoft, Borland as well as end-users and individuals. All these people are trying to work together to develop specifications to maximise the portability, reusability and interoperability of commercial software. Obviously it is not an easy job to get consensus between these organisations.

In October 1991, the OMG announced its adoption of the CORBA 1.1 specification. This was a major achievement as it headed off a schism within the object community between static and dynamic binding, that could have proved as divisive as the UNIX wars between UNIX International and OSF. CORBA endorses both approaches to binding.

The OMG are different from other national and international standards bodies in that they do not create the standards. The ISO and CCITT (now called ITU-T) are working on the de jure standard in open distributed processing, the ISO/ITU ODP reference model. De jure standards (i.e. official standards approved by recognised bodies) are slow to emerge since they tend to consolidate experiences. The OMG have positioned themselves as the fast-track to open distributed computing. De facto standards (i.e. where the market adopts a product) are quicker to emerge but slow to achieve consensus. One goal of the OMG is to catch distributed object technology before it becomes entrenched in market politics and customer preferences. Industry consortia like the OMG provide a balance between the de facto and the de jure. They give more control to the customer since end-users can demand standards from their vendors.

Different industrial consortia operate in slightly different ways. The OMG merely organise committees and task-forces to choose specifications. They adopt their members ideas and publish these as standards. This has the effect that products exist when standards are announced. In contrast, until recently the OSF solicited technology contributions and centrally stitched it together into vendor distributions. The OSF are now changing the style of funding to external development teams (probably due to the growing complexity of products like DCE). UNIX International, the body behind the distributed architecture called Atlas, was another consortia but it no longer exists.

The other type of standards body are user organisations like X/Open that act as standard-blessing organisation. X/Open have already adopted OSF/DCE. X/Open may provide CORBA compliancy tests by the end of 94.

There are a wide range of domain specific standards usually initiated by user consortia. In general standardisation efforts in enterprise integration remain descriptive and conceptual rather than prescriptive and concrete.

Standards from other user consortium in specific industries include:

- CIMOSA (Macintosh, 1994), is an ESPRIT funded standardisation effort for manufacturing.

- POSC, the Petrotechnical Open Software Corporation (POSC, 1992) is a consortium of oil and gas companies that are developing a common business model for exploration and production. The purpose of this effort is to amortise the costs of defining a single model for the exploration and production business. This represents a new pattern of collaboration, between competitors that no longer see IT as a competitive weapon, who would focus on core businesses.

  Business integration is key to the petrochemical industry because geologists, drillers and production engineers must work together. They need a common model to share information. POSC not only defines infrastructure but also database entity models. An object oriented approach was found necessary primarily to protect data integrity. The STEP EXPRESS modelling language was adopted to incorporate object concepts into the data model and APIs. POSC invited model submissions and selected the best to define a root model and full function API.

  Sample POSC implementations have been built by UniSQL and HP. POSC are co-operatively developing the model for their industry so that third party vendors can build plug-and-play applications to the industry-standard business model.

- NCMS, the US National Centre for Manufacturing Sciences is defining standard-based architecture for integrating manufacturing applications.

The Suppliers Working Group SWG provide a conceptual architecture and reference taxonomy covering the three perspectives of capture (enterprise model standards), representation (APIs, applications and tool standards) and enactment (execution services and platform standards). The SWG Reference Taxonomy can be used to position technology and standards (Macintosh, 1994).

The growing importance of application-level standards has a significant impact on software development. The availability of plug-and-play components that conform to standards puts more emphasis on component selection and less on traditional analysis and design activities. A balance must be achieved between bottom up standards-driven conformance and top-down specification-driven design.

Application development life-cycles

Conventional methods for application development most frequently adopt a variant of the waterfall model. The waterfall model (Royce, 1970) is a top-down single-product life cycle model based on a sequence of steps: requirements analysis, design, design module decomposition, coding, integration, maintenance. It delivers documentation at each step that is the input to the next step, thus providing a way to track progress and notice when it is off schedule. However constantly changing requirements or open ended solutions are not well supported since there is no feedback or incremental extension of the model under development.

Distributed object technology demands radical surgery in project development life-cycles. In fact to gain the strategic advantages of speed and flexibility, the whole notion of a well-defined life cycle must go.

There are three forces behind these changes:

- object technology lends itself well to rapid prototyping and incremental evolution,
- open systems demand a bottom up approach based on reusable commodity components,
- rapid technological and business change requires more complex innovative solutions.

Consequently the focus is on proving and refining design concepts, integrating commodity products and experimentation.

These forces impact in different ways:

- the life cycle may be iterative, involving throw-away and evolutionary prototypes. Throw-away prototypes are rapid partial iterations of the development process designed to give user feedback that modifies the requirement specification. Evolutionary prototypes are partial implementations that meet known requirements to which additional functionality is slowly added by successive iterations in the process.
- re-use is not top-down. Traditional top-down design decisions are being replaced by bottom-up engineering trade-offs between existing components.
- in an innovative environment, there are more unknowns such as capabilities, limitations, risks, costs and time-scales that must be explicitly addressed.

An infrastructure can provide new forms of support for these processes such as incremental compilers and debuggers for efficient iteration, trading services for commoditisation and general-purpose system description and planning tools to manage an innovative environment. This places significant new requirements on development tools.

Spiral Life-Cycles

The spiral model (Boehm, 1988) seeks to break away from the relatively restrictive conventional waterfall models to allow a more flexible approach that takes into account the uncertainties surrounding development. The spiral model is a risk-driven evolutionary lifecycle where each cycle begins by identifying the objectives of a particular cycle, the implementation options and the constraints on the implementation. The alternatives are evaluated against the constraints and objectives and strategies for reducing risk are proposed. These strategies may include different options: rapid throw away prototyping, incremental development, evolving prototypes, top-down subsystem, bottom up components.

Object-Oriented Life-Cycles

The object oriented approach provides a consistent view of the system as groups of communicating objects between analysis, design, coding and maintenance, which lets developers go back and forth in the life cycle, applying a common notation at all levels. This supports an iterative, incremental, evolutionary life cycle that better reflects the way developers actually work.

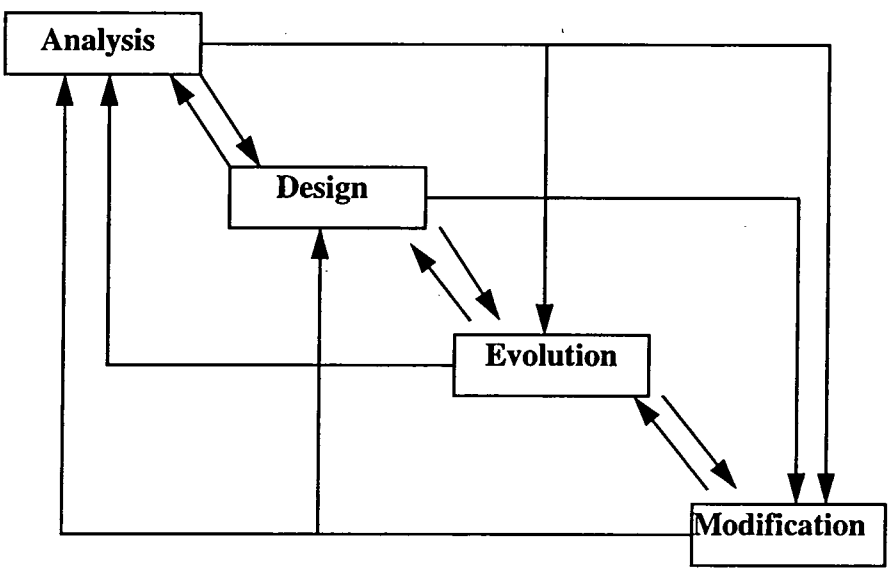


Figure 12 Object oriented lifecycle

Component-oriented life-cycles

Complex systems may be built using off-the-shelf, standard source components as building blocks. This demands more investment in the development of reusable component sets and in tools to streamline the development activity as much as possible. Components are not limited to code but include domain knowledge and development experience. Viewing applications as "families" allows for collections of component frameworks to be supplied and slightly modified to produce specific applications. This assumes sufficient domain knowledge has been gathered from specific applications to make it possible to abstract and package general knowledge. The key characteristic of this life cycle is the two distinct activities of component development and component configuration, as shown in Figure 13:

- component development, is the activity of abstracting the domain knowledge and encapsulating this knowledge in frameworks of reusable components.
- application configuration, is the activity of integrating and customising the general component frameworks to construct a system that meets a particular application requirement.

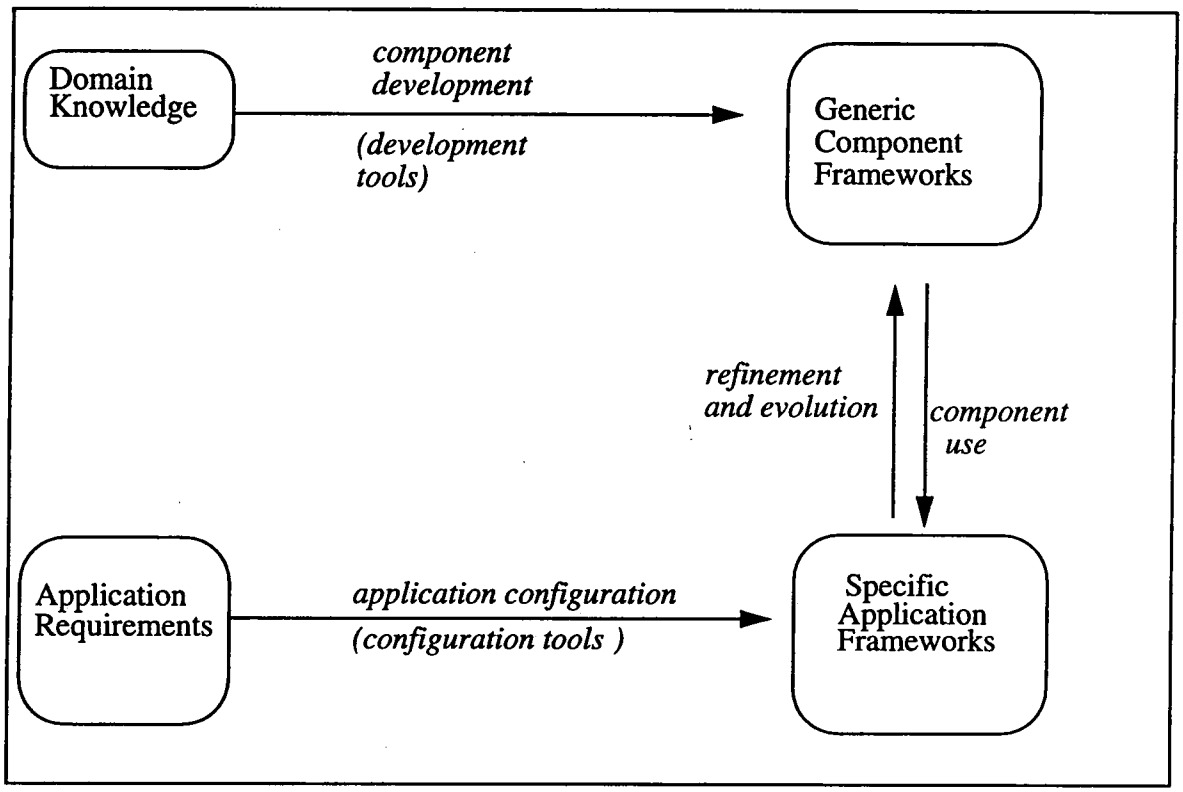


Figure 13 The component-oriented lifecycle

Component development and application configuration are not necessarily orthogonal tasks. It is likely that they are concurrent to some degree, with feedback from the configuration task into the development task. A single person may even perform both roles, but should not combine the tasks. Component development is an experts role as it demands both domain knowledge and knowledge of reuse mechanisms.

Hybrid life-cycles

Another important consideration is the relationship between life-cycles and conceptual models. Narrow definitions of object technology have a distinct implementation flavour and are consequently too prescriptive to be universally applicable i.e. they enforce a particular approach to the entire life cycle. This may be the undoing of object technology. For example, inheritance is essentially viewed by developers as a code sharing mechanism and as such is not directly applicable at more abstract stages of specification. What is applicable is the notion of classification taxonomies. (Blair et al., 1991) suggests that there are two common misconceptions about life-cycles that must be revised for distributed objects:

- that object oriented techniques and mechanisms are equally applicable at all stages of development.

- that object orientation should have the same interpretation at all stages.

These issues become clearer at a higher level of abstraction where object technology mechanisms such as inheritance and messaging are seen as vehicles to achieve data abstraction and behaviour sharing. Object mechanisms are not a panacea to all problems. Object oriented techniques may be useful in certain phases of the life cycle and not in others or in certain applications and not others. Other techniques like functional decomposition may be more appropriate to large distributed systems. A more pragmatic approach of mixing techniques may be more powerful. For distributed systems, object techniques must be mixed with distributed system techniques. Rather than whole-heartedly committing to narrowly defined method, it may be more effective to formulate a hybrid approach and concentrate on providing traceability between various phases and components.

In order to manage the complexity of a distributed system, it is useful to define different perspectives of the system. A broader view of object technology may be necessary to accommodate several distinct object models in different perspectives. This may be formally defined viewpoints as in ISO ODP(ANSA, 1993) or simply a separation of object models: logical from physical and structural from behavioural.

Note that in the iterative style of development, the developer visits all perspectives at all levels of abstraction in no fixed sequence.

Conclusions to Life-cycles

A single well defined life cycle model for distributed object technology is a long way off. Consequently a developer is left with an informal approach that will be difficult to manage on large or complex developments.

The most important aspect is not rigid adherence to object oriented mechanisms but rather to concentrate on maintaining consistency and traceability between appropriate diverse models and processes that make up a hybrid solution. Traceability is important because it provides lines of accountability between different phases or models and can be important in determining the rationale for a particular approach or for analysing the effects of change. Consistency is important for valid and verifiable interpretation of the specification.

4.1.2 Types of Object

This section defines different notions of an object according to the method used to identify and represent the object. This provides a conceptual distinction between different styles of distributed object system. The main implication of these differences are on the granularity of objects, resulting in problems of allocation , performance and reuse.

In a conventional system the choice of method is usually determined by the expected major source of complexity: for functional complexity, flowcharts, decisions trees, structured design and its derivatives (DeMarco, 1978); for complex data, entity-relationships and its derivatives (Chen, 1976) or for complex time sequencing as in real-time systems, statecharts such as (Harel, 1987).

Object models mix these views by orienting them around a common conceptual framework of objects. However the relationships between these different views of the system are still poorly understood.Furthermore methods currently fail to explicitly address design-with-reuse, focusing more on design-for-reuse.

Distributed systems are often designed in a much less formal, ad-hoc manner, being technology driven, for example using a simple client-server enabler tool. The absence of any formality in the development process will severly limit the applicability of the technology to solve complex problems. An enterprise integration architecture should be built to support the highest level of application complexity in the enterprise.

Object-oriented methods (in-the-small), focusing on reuse, and enterprise integration architectures (in-the-large), fosuing on integration mechanisms, are increasingly being applied to solve more and more complex problems and frequently must consider several complexities simultaneously. Technically complex applications frequently arise in areas like defence, space, telecommunications, investment banking, process control. To address these applications, there is a need for a method that provides an integrated view of these different perspectives of design, including views that focus on functionality, data, time sequencing, reuse in-the-small and distributed technology. A taxonomy to characterise the different existing approaches and the required integrated approach is outlined in Figure 14.

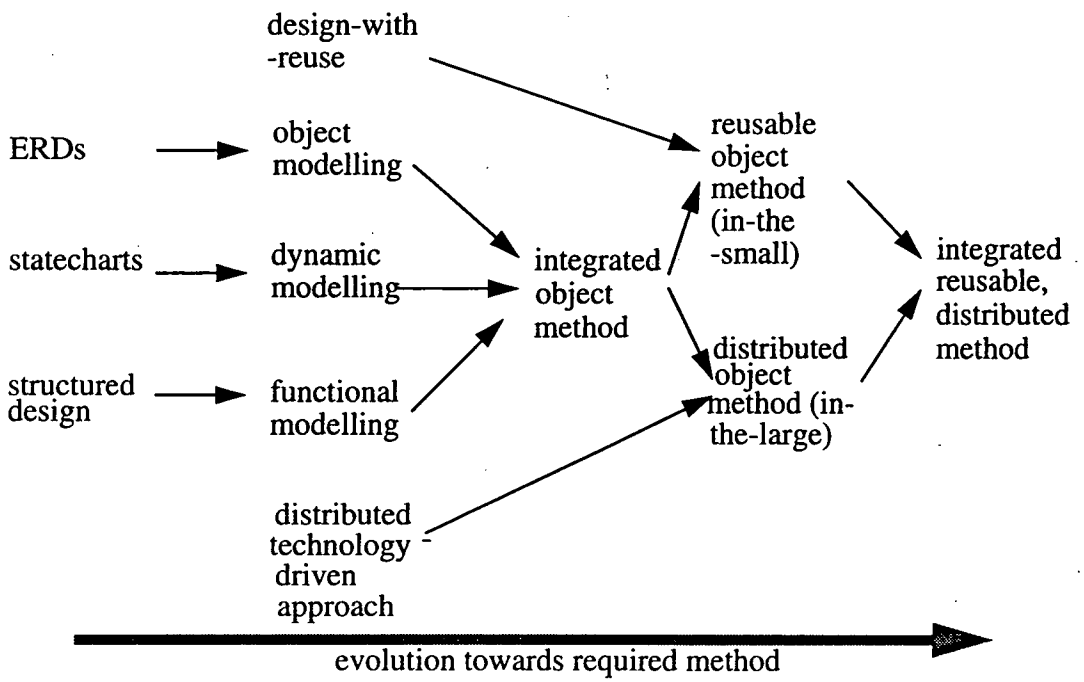


Figure 14 Requirements for an integrated method

A distributed object system will typically have a distinct flavour according to the method roots, the developer experience and the relative emphasise placed: on different types of formalism; on enabling technologies; and on reuse. These flavours of object system will be elaborated in this section by discussing approaches that distinguish different branches of the above taxomomy.

a) Object Architectures-in-the-Large

The pure object-oriented enthiusiast may challenge the use of the term object to describe a wrapper for a whole application or a group of functions that make up a server process in a client server architecture. However the physical structure of an enterprise both in terms of users and computers, is inherently an architecture-in-the-large and objects-in-the-large can be very useful to structure this architecture.

Process Objects

The view of large grained objects as processes is attractive for specification of dynamic behaviour. Objects conveniently tie together processing, state and functionality. Processes are powerful for modelling asynchronous parallelism, and operational requirements naturally arise as constraints on asynchronous parallelism, between the system and the environment or between system components. Process-oriented specification languages are based on this principle. (Zave, 1986).

Process-oriented specification languages provide a formalism to describe the process architecture, i.e. the allocation of functionality to processes. Less formal, simple, static process architectures are described using conventional client-server architectures.

Conventional client-server systems use processes to provide a simple partitioning of presentation, application and data management functions There are number of different ways functionality can be split between the client and server that lead to different styles of system, as shown in Figure 15. Often different styles are required for different parts of a single application, for example a sales order processing application may use distributed database for disseminating pricing catalogues and remote presentation for warehouse stock enquiries.

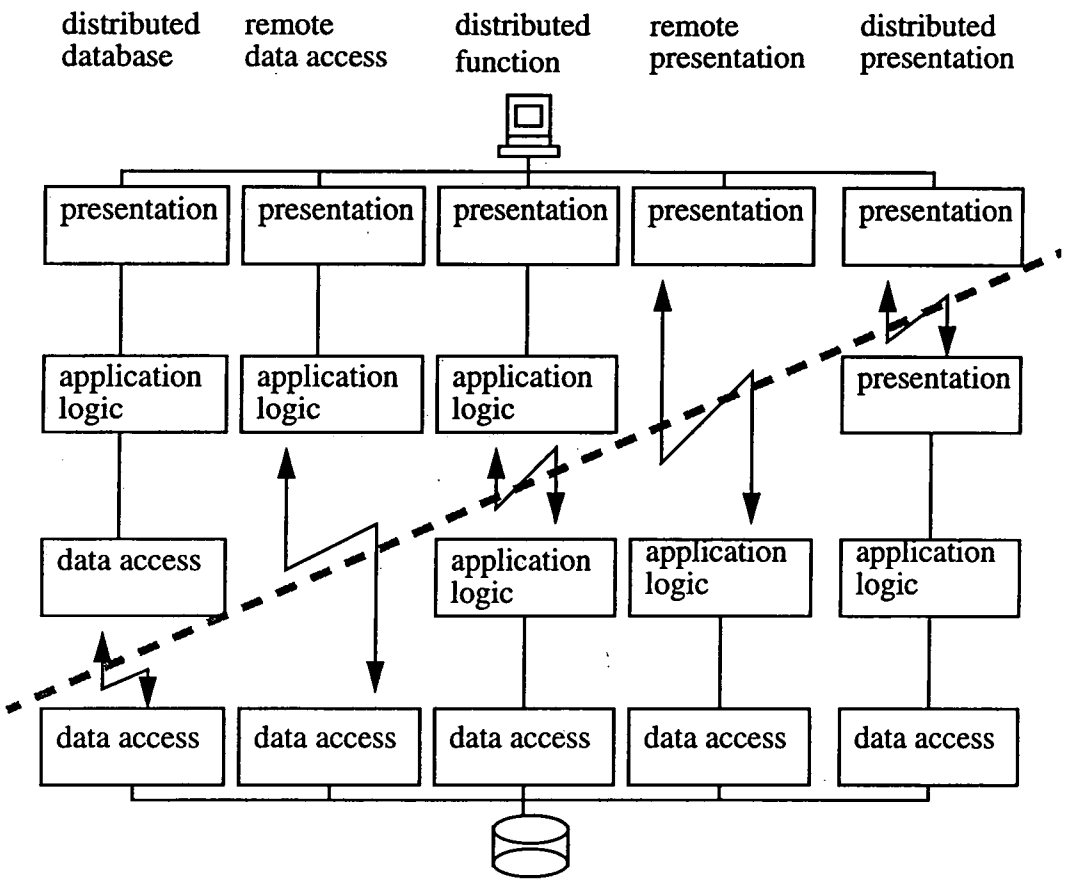


Figure 15 Gartner's five segmented models of client-server computing

Where objects make synchronous calls we need to track nested calls. Complex nesting is not normally a problem for coarse-grained systems. Since they tend to either use asynchronous calls or have very simple process architectures as in client-server systems.

Application Objects

A large-grained object architecture can also be driven by wrapping applications as objects, for example applications may be wrapped as CORBA objects. This is usually done to achieve application integration. The application objects may consist internally of local component objects. Such an architecture is very much driven by the inherent division of users or I/O devices and tasks.

This view of CORBA objects is one vision of how distributed objects could be used for application integration in HP's Distributed Smalltalk.

Coarse-grained integration is required to integrate current software. Applications are moving to finer-grained integration, but current systems, that are coarse-grained, must still be supported while the move takes place. Object-oriented Tool Integration Services, abbreviated OOTIS (Harrison et al., 1992), is an approach that supports both coarse-grained and fine-grained application integration.

Distributed Programming System Objects

Many distributed programming systems support different notions of object as part of the programming language. In surveying different distributed object oriented programming systems, (Chin and Chanson, 1991) define four types of object that are supported in the various programming systems: large-grained, medium grained, small grained and fine-grained.

For example, Argus(Liskov, 1988) supports large grained objects called Guardians that control remote access to small-grained objects that exist within Guardians.

b) Architectures in-the-small

The above approaches often result in heterogeneous object models consisting of large grained distributed objects and fine-grained non-distributed objects. The degree of interaction and sharing is limited by this extra layer of encapsulation and modelling. A pure object oriented approach would result in a finer grained model with no clear mapping between fine grained objects and processes. This pure model is often compromised to achieve meaningful finer-grained architectures.

There are six main ways to depart from such primitive coarse-grained models. All provide richer notations or mechanisms to model object structure at a finer granularity than a process:

- component-oriented composition that allows configurations of object components to be organised in a understandable way using hierarchical composition and frameworks.
- structured object-oriented analysis and design methods that effectively provide specification-driven functional decompositions that are structured as objects.
- object-oriented analysis and design methods that focus on identifying and structuring applications using classification hierarchies of reusable objects.
- reactive programming, that builds an object-oriented system prototypically.
- enterprise modelling that mix different business modelling formalisms with object identification.
- algorithmic approaches that automate object allocations.

Each of these approaches are discussed in turn.

Component-Oriented Objects

Component-oriented software development (Nierstrasz, 1992) is an approach to software development that shifts the emphasis in the software life cycle from the development of individual applications to the engineering of reusable frameworks of software components. This implies there are framework builders and framework users that perform distinct tasks of component construction and component usage. Component-oriented objects are objects that have been developed as components of such reusable frameworks without a specific user project in mind. This includes objects developed for a visual composition tool or configuration programming system as defined in section 4.2.2 or in (Kramer, 1990).

Component-oriented development does not replace analysis and design but encourages the results of analysis to be sets of reusable components and design to be the act of organising and specifying these components

There are four fundamental tasks that need to be supported to make component oriented methods more productive than conventional bespoke development (see section 4.2.2 for a fuller description of the tools that support this) :

- finding components, for example using browsers, query services, selection tools, categories and catalogues.
- understanding components, for example using specification languages, formal methods and structured descriptions of knowledge.
- modifying components, for example using configuration attributes OSF/ DCE ACF(Shirley, 1992), specialisation, transformation techniques.
- composing components, for example using visual composition, configuration languages, object-oriented frameworks.

Structured Objects

There is a dichotomy between those methods that focus on the whole, and break the whole down into parts, and those methods that focus on reusable parts and structure them into patterns that satisfy the whole. The former includes conventional structured methods like functional decomposition and to an extent the newer object methods that have their roots in structured design like (HOOD,1991) and (Schlaer and Mellor, 1988). The latter includes purer object methods like OMT(Rumbaugh et al., 1991) and (Coad and Yourdon, 1991).

The use of a decomposition model to identify objects restricts the reusability of the resulting parts because they are defined with a particular facility in mind. Coarse facilities are not very reusable between applications. This is particularly true if the decomposition is based on functions. For example, consider a plumber fitting water pipes. He does not think "What (functional) steps do I take to make pipes to connect this outflow to this inflow". Rather he thinks " I need a short pipe, a spanner, some sealant and some compression joints." It is these small parts that are reused in different jobs not the functions of cutting the pipe to length and sealing its ends.

Much of the poor reusability resulting from decompositions can be alleviated by viewing a subsystem as an organisation of objects and specifying the subsystem in more abstract terms than functions, for example using roles, responsibilities and services as in (Wirfs-Brock and Wilkerson, 1989). An essential aspect of these concepts is the notion of purpose and context. However note that in these approaches the identification of objects is very informal and is not based on a strict process of decomposition as in functional decomposition. (Kramer, 1992) observes that the identification of reusable components is always ad-hoc and this is one of the limitations of process centric approaches like functional decomposition. Note that despite using terms like subsystem, the dichotomy is not brocken, because the focus of these approaches is really still on identifying reusable parts. Roles and responsibilities can be useful to generalise away from a specific context and aid mapping subsystems to reusable objects. There is an added burden of identifying and allocating responsibilities and roles. In constrast, the more structured object methods apply a more process-centric decomposition to derive objects, focusing on the subsystems and this inhibits reuse.

A decomposition approach has clear benefits in a distributed environment. Decomposition results in a coarser partitioning that makes it easier to allocate across the network. Structured methods give precise guidance on the partitioning of the application to subsystems. This makes the overall structure manageable at a coarse level and results in less coupling between subsystems.

Purer object methods do not have a clear mapping to subsystems and there tend to be strong couplings between objects. A reusable object will typically play a role in many facilities. Support for partitioning designs is weak. For example Coad introduces components to separate user interface behaviour from problem behaviour from data management behaviour. Yet this separation is not as coarse-grained as it sounds since individual objects are often responsible for displaying themselves, for collaborating with other objects to solve problems and for storing themselves.

Mainstream object-oriented objects

There are a number of object oriented methods that have abandoned the notion of functional decomposition all together. They focus on identifying reusable objects and provide rich notations to describe classification and messaging hierarchies. These methods include OMT (Rumbaugh et al., 1991), (Coad and Yourdon, 1991), CRC (Wirfs-Brock et al., 1990), (Booch, 1991). There are also a second generation of methods that mix the best from the above in a single method such as Fusion(Jeremaes and Coleman, 1993), Syntropy(Cook and Daniels, 1994).

The general approach in these methods is to consider three types of model. Classes of objects are identified and structured into a class model that is a static class hierarchy, using generalisation-specialisation and whole-part relationships. The dynamic behaviour of objects is captured in a dynamic model using notations like state transition diagrams (Harel, 1987), object life-cycle diagrams. The user tasks are identified and traced through the objects to define a processing model using notations like use cases(Jacobson, 1992), messaging diagrams (Coad and Yourdon, 1991) . The relative emphasis to place on the different models should depend on whether the complexity is in the information structure, the dynamic behaviour of objects or the collaborations between objects. However most methods emphasise the class model and the relationships between the models are generally poorly explained. Overall these methods are strong in notation yet weak in process. Most are overly dependent on good requirement specification and design without detailing the method to be used.

These methods still generally assume there is a specific user project. Despite focusing on finding reusable objects, actually reusing these objects is still ad-hoc. They generally provide no rules or tactics for how reuse should be done and there are no formal constraints that require or enforce reuse. In particular, there is nothing that constrains a programmer to use only previously defined interfaces. It is easy to extend interfaces through inheritance and define diverse protocols.

More recently some efforts have been addressing the absence of reuse rules and interface constraints, including work on frameworks (Deutsch, 1989), behavioural contracts (Helm et al., 1990), design by contract (Meyer, 1992, ref [2]), methodology heuristics and laws, such as the Law of Demeter (Lieberherr et al., 1989) or Law Governed Systems (Minsky et Rozenshtein, 1987).

The mainstream methods also say little about distributed process architectures. In fact they even give little guidance on providing coarse partitionings of a large design. Efforts to address partitioning shortcomings include CRC subsystems, Coad component and subject areas.

Reactive objects

There is a wide range of software systems that build on organisations visions but have few concrete requirements that can be precisely defined at the outset or implemented without incorporating rapidly changing high-technology solutions. Conventional systems analysis which views design and implementation as consecutive steps, is not able to handle these systems. The requirements of users and abilities of technologies are so likely to change that design decisions must be made throughout the development. The solution is to admit that design and implementation are concurrent and iterative activities. Object technology offers a flexible and dynamic programming environment that allows models of proposed solutions to be built, with small commitment and cost, simply to try out ideas, capture the users opinions and articulate concrete design concepts. We call this reactive development. Smalltalk offers the most popular reactive programming system.

There are inherent problems in supporting the same degree of responsiveness in a distributed programming environment, due to issues of ownership, trust and locality of code that arise when we introduce multiple users and multiple processes. This is discussed in (Bennett, 1987; Tonks, 1994). The most obvious constraint is that the overhead of navigating distributed class hierarchies at runtime makes inheritance across processors impractical unless class definitions migrate or are replicated across the network. However replication introduces dependencies.

Enterprise Objects

Enterprise modelling formalisms drive a distributed system architecture from business concerns. This provides a promising arena for specifying peer-to-peer business object architectures that are finer grained than processes or applications. However enterprise modelling seems to mean different things to different communities and tends to suffer from the legacy of information modelling techniques for relational databases.

- The AI community are doing considerable work on knowledge representation and knowledge sharing across an enterprise, for example DARPA work on specifying domain ontologies (shared vocabulary) as applied by (ATOS, 1994). This work may deliver rich modelling semantics and methodologies for distributed AI but it does not say much about general purpose software architecture used to implement an enterprise wide system.

- The database community are providing information modelling tools that can drive a federated distributed database design suitable for enterprise wide information services. Most of this work is more appropriate to remote data access systems.

- The object-oriented community are dressing their methodologies up as enterprise modelling methods. Object-oriented analysis can reduce the impedance mismatch of traditional analysis thus their objects seem appealing as tangible business objects, however most methods say little about infrastructure.

All approaches to business enterprise modelling depends on open-systems based solutions to deliver a distributed object computing platform on which the business model will run.

Existing enterprise modelling practictionerss tend to adopt a rather ad-hoc pragmatic synthesis of conventional database design and object oriented methods. There are three distinct types of approach varying in the mechanism used to find objects :

- build one or more applications and bubble up business objects from any fundamental business concepts that have general utility. The generalisation of these concepts into enterprise classes is usually not formalised in this approach.

- develop an entity-relationship data model and as a totally separate activity "objectify" it by passing it to a team of object experts, usually with an application development focus, who would figure out what methods were appropriate. This has an application focus in the identification of objects.

- develop information engineering or entity-relationship data models and discover objects by attaching business processes to entities. This is usually accompanied by a clear concept of enterprise architecture and more formality in the discovery and design of business objects.

One factor is the capability of the model to factor application logic from business logic so that it can accommodate change. This is an obvious advantage that enterprise architectures have over conventional client server architectures. In the former it is generally recognised that there is a clear separation of the business model from the applications. In the latter, the business logic is often intertwined with the presentation layer as in tools like Easel and PowerBuilder.

In effect the business objects establish a common set of behaviours that are shared across the enterprise. This is an extension of the use of entity-relationship data models to establish a standard for data sharing across an enterprise. In the distributed AI community, ontologies are used in a similar way to establish a standard for knowledge sharing. The standardisation of these models across an industry sector allows even broader integration. Vendors can develop plug and play components.

There are additional interoperability and legacy system problems. Businesses simply don't want to rewrite all their applications to use a single corporate database model or single ontology or single set of business objects. Wrappers allow you to put a new face on an existing application hence giving more leverage in environments that are rich in legacy systems. Wrappers are supported by CORBA and DCE and intelligent agents technologies.

CASE tool support is weak for enterprise modelling. CASE vendors and structured method gurus are frustrated because they have no prior experience with object technology, distributed object computing or business object modelling. Advertising varies from misleading to downright lies. The buyer must often be his own integrator.

The ad-hoc nature of the approach masks an implicit problem that is inadaquately addressed. It is inherently difficult to find enterprise wide objects because different parts of an enterprise do different things and demand different semantics for the same concepts. Furthermore it is often difficult to understand object behaviour. As the process becomes more formal it must address these practical concerns.

Conclusions - Types of object

The critical issue is partitioning an application at a coarse enough level to make it manageable in a distributed environment without compromising the reusability of its constituent objects. Existing analytic approaches to identify objects taken in isolation are inadequate in a distributed system. An integrated method based on a hybrid of approaches or richer unifying concepts, like roles, is required to solve this dichotomy and support reuse explicitly.

4.2 Choice of technology

Building a distributed system necessarily requires more technology choices than a stand-alone system. Networks of computers are inherently more complex and developers need supporting technologies to overcome these complexities. This requires new skills such as product evaluation and new development methods that are oriented around managing infrastructure product procurement and integration.

This section discusses two key choices that need to be made:

- the most appropriate choice of communications technology.
- the choice of tool technology to support large-scale reuse across platforms.

Despite the importance of infrastructure technology, most user organisations have purchased this technology as a secondary feature of a database server or mail system or diagrammatic CASE tool or other high level service. Yet infrastructure technology can provide more than the high level service. It can become the foundation for a variety of applications and the basis of enterprise integration, interoperability and reuse. As such, an evaluation of middleware should be the focus not an afterthought.

4.2.1 Selection Method for Communications Technology

A choice of standard communication architecture to serve an entire enterprise: promotes consistency and leverage among applications, focuses developer expertise on a smaller set of technologies, and reduces the complexity of the environment, by reducing redundancy and diversity.

However the choice of technology is a difficult one, in particular due to the lack of experience and rapid rate of change in the technology. Architecture is best defined in the context of specific projects that provide the proving ground for architectural alternatives.

This section describes a method for selecting infrastructure technology based on work from the Patricia Seybold Group (Seybold, 1994). This is a useful method because it can be generally applied and provides a useful introduction to the sort of rationale and evaluation frameworks that are employed by practitioners in industry.

6 Steps towards selection of middleware

The Seybold Group provide a generic architecture for middleware that consists of the following components as shown in Figure 16:

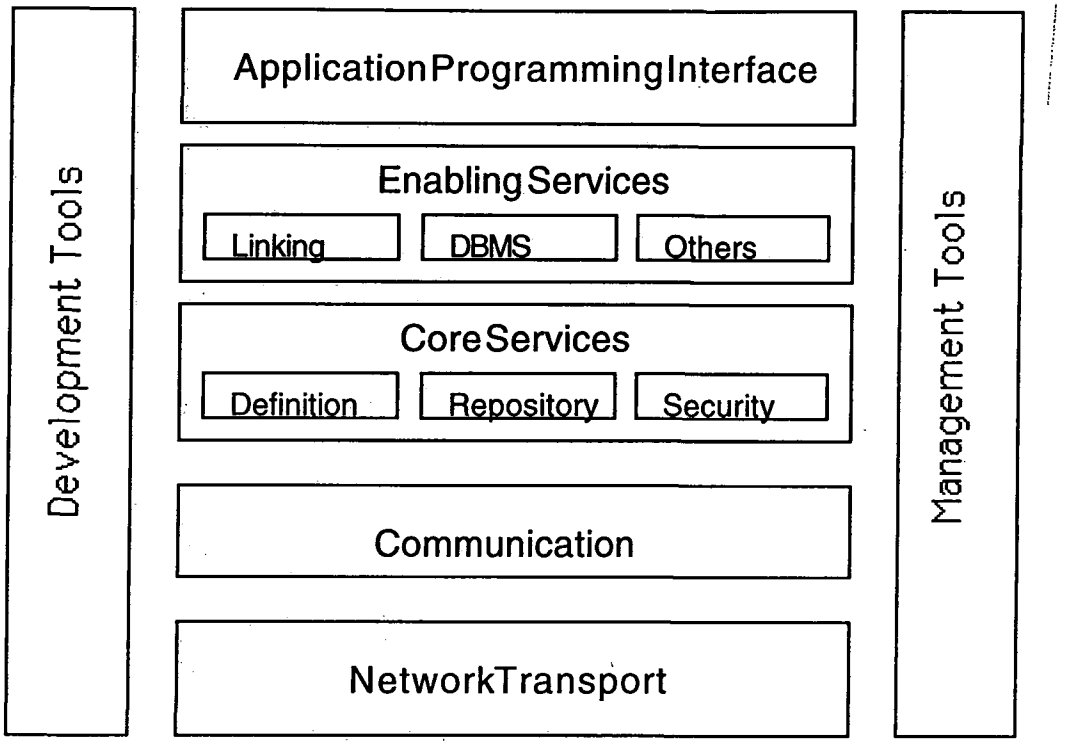


Figure 16 Generic Architecture for all Technologies

- network transport - such as TCP/IP or SNA, usually provided by the operating system.
- communication back-plane - such as SQL, CORBA, e-mail, providing high level protocols that use the network transport to support communication between application components.
- definition facility - such as IDL or database schema generators to define and name the application entities, so that names and functions of entities can be known to other entities.
- repository services - such as directory services and service registries, to map the names and service properties of entities to physical locations for matching clients requests.
- security services - such as authentication and authorisation to control access to servers.
- enabling services - such as document linking, object binding, message routing, database management, to provide application level services other than point-to-point communication and security.
- programming interfaces - to provide access to the services.
- management tools - to manage, monitor, configure and control the infrastructure.
- development tools - such as browsers, debuggers, modelling and CASE tools to provide a complete development environment.

Having defined the critical elements of an infrastructure product, the six steps are as follows:

1) Select a transport protocol

2) Select the most critical class of application and a pilot within that class

3) Select a type of communication technology for that class of application.

4) Select a middleware environment that matches that type of technology.

5) Select application development tools for the environment

6) Select management tools for the environment

The network transport protocol will depend on the predominant hardware and network technology in the enterprise. IBM shops may tend towards SNA, some PC LAN environments may lean towards IPX as used by Novell and most other environments , especially the UNIX world will use TCP/IP. Integration between these cultures remains extremely difficult, despite the growth of interoperability standards. Limiting the environment to one culture will simplify things considerably.

A pilot application should be representative of the most important class of applications that the business will need to develop. It would help if this application is self contained and can be isolated so that global commitments are not enforced too prematurely during the proving phase. The pilot is used to generate a set of requirements that can be used to evaluate the architecture.

The following discussion elaborates the Seybold method by defining three key classes of distributed application:

- *information dissemination*, including mail, text retrieval, data presentation.
- *data processing*, including transaction processing systems and workflow applications.
- *distributed control*, breaks into several subclasses of applications including simulation, process control, communications, business intelligence, command and control.

Information Dissemination Architectures

Where personal access to relatively static global information is important, an information dissemination architecture may be used. Where this is between companies, commoditisation issues like charging may be important. The data structure is generally less complex than for online corporate data and serial text or binary formats may be used. Control flow is also less important. There are five main types of information retrieval technology:

- Text Retrieval - using catalogues and abstracts for searching based on indexes. Indexes may be inverted files of key terms such as author, title and subjects.
- Hypertext - supporting cross-referencing and browsing between topics typically by clicking on hotspots in the document.

- Structured Data - supported by DBMS. Other technologies may be represented at too low a level of abstraction for effective personal access. A number of even higher level access methods are being researched: for example corporate information servers that hide the physical and logical structure of enterprise data models; AI techniques to allow higher level searches; Geographical Information Systems (GIS) systems using novel spatial indexing techniques

- Multimedia - multimedia is used in very user oriented communication. The emphasis is on data transfer problems such as lip-synch rather than data processing.

- Publish-subscribe - information services may use local or global networks to disseminate information - using news-groups etc.

Data Processing Architectures

Where data must be manipulated as well as accessed, database management functionality is more important. This is usually the case for systems supporting operational tasks such as concurrent updates to customer information. The data is usually internal to the company. Commodisation issues like charging are generally less important than transactional access, data structuring and data semantics. The increased complexity can be managed by structuring the data using a schema definition language and accessing it using a data manipulation language. This style of application demands structured data technology like a relational database or object oriented database.

Distributed Control Architectures

Where application functionality as well as data must be distributed or accessed remotely, we need to manage both data flow and control flow. Not only do we need to define the data structure but also the control structure of the program. The extra complexity can be managed by structuring the data and behaviour statically using a schema of abstract data types or interfaces. Many definition facilities are based on abstract data types with procedural interfaces. Examples of this technology include distributed functions and distributed objects.

Management functionality may be built into the computational model and hidden by the definition facility. For example CORBA IDL hides object management interfaces, see Appendix B:. Such architectures may mix or combine distributed databases with distributed functions and distributed objects. The definition facility may be unified with the data definition facility such as a embedded data manipulation language or both may be used orthogonally, such as CORBA and ODMG.

Type of technology

Having defined three classes of application, we can relate the choice of type of client server technology directly to the class of application.

Client-server architectures are widely applied in distributed systems. In the client server model, an active process (the client process) on one node invokes a service on the server (process) which may be on a different node. Typically the invocation consists of a request message sent to the server and a result message which is returned. The form of request depends on the type of technology.

There are six categories of technology that reflect different types of client server system. Each type will be discussed in turn. These types include the following (Note: product literature can be obtained direct from vendors and is not included in the references):

- Distributed File, including PC LAN products (e.g. Novell Netware's File Service, Microsoft Window's NT Advanced Server File Service) ; UNIX file services (e.g. Sun NFS, OSF/DCE Distributed File Service); and document management products (e.g. OpenDoc, OLE and products from Verity, Saros, Fulcrum).

- Distributed Display, including terminal services (e.g. Easel Easel, Wall Data Rumba) and windows standards (e.g. X-windows)

- Distributed Database/OLTP, including relational database access products (e.g. Information builders EDA/SQL, Sybase OmniSQL), and other product from the main vendors (e.g. Oracle, Informix, Ingress); distributed relational database products (e.g. Oracle*Star, Informix *Star and Ingres*Star); on-line transaction processing (OLTP) products (e.g. IBM CICS, Transarc Encina, Novell Tuxedo, AT&T Topend); and object oriented database (e.g. Object Designers Objectstore, Servio's GemStone).

- Messaging, including e-mail (e.g. HP OpenMail, Microsoft's Enterprise Mail Server), intelligent agent technologies (e.g. General Magic Telescript, IBM Network Communications Manager); and workflow products (e.g. Action Technologies Workflow Builder, AT&T GIS's ProcessIT, Fujitsu's Regatta).

- Distributed function, including RPC-based systems (e.g. OSF DCE, Sun ONC, Netwise RPC Tool); message-oriented systems that support functions (e.g. Message Express, IBM MQ Series, DEC DECmessageQ); and stored procedure data access systems (e.g. Oracle's SQL*Net, Sybase NetLibrary/DBLibrary and Transact-SQL tools).

- Distributed objects, including CORBA products (e.g. IBM DSOM, Iona Orbix, DEC Objectbroker, Expersoft's XShell).

Distributed file systems, support access to files stored on remote processors. Operating system calls to the local file system are typically intercepted and redirected to the servers local file system. File servers like Novell Netware give visible access to remote disks. Distributed file systems like SUN Network File System (Collinson, 1992) give transparent access either using a directory service or using broadcasts to redirect access. Enabling services include mounting services, remote file transfer services, transaction control for remote file access, file replication and document management e.g. OLE 2.0 (Rymer et al., 1994), which includes linking and embedding, version control, and searching.

Distributed display makes presentation functions of a host available to remote clients. Conventional user interfaces provided by mainframe terminals and terminal emulation software, are character based and enforce sequential navigation through screens revealing data and commands. Modern GUIs like X-windows use graphic displays and event loops that allow more flexible user interaction and give more control to the user. Enablers include session and connection management between a client and a host, format conversions and event loops.

Distributed database middleware allows a client to read and update remote databases by providing a transport service for data access requests. Distributed databases vary in the way management functionality is split between the client and the server process and in the data access mechanism. Relational *database servers* generally perform most of the integrity checking, concurrency control, query processing , etc. on the server. Access is by queries. This is the traditional *database server* architecture. OLTP (Johnson and Hudson, 1993) extends this model so that data access requests can be structured into stored procedures on the server that are invoked from the client. Object oriented databases generally support access that is closely coupled with the object-oriented language. They may be classified database servers, page servers and true object servers. *Page servers* like Object Design Inc.'s ObjectStore (ObjectStore, 1995) pass memory page segments across the network to cache clusters of objects in the clients process. Page faults on using pointers results in page activation. Query processing is performed on the local cache and locks can even be cached. True object servers like Servio's Gemstone allow similar distributions of query processing and concurrency control but at the level of objects. Object-oriented database queries differ from relational queries in that they tend to be based on extents or sets defined by aggregate objects or collections whereas relational queries tend to be based on schematic knowledge such as tables and key attributes. Enabling services include concurrency control, transaction control (2-phase for distributed updates, nested transactions) and data access (stored procedures, large database).

Messaging middleware allows messages and task instructions to be transported across a network. This includes mail based systems (e.g. Rodden and Sommerville, 1989), intelligent messaging systems (Bock, 1994; Brost and Malone, 1986), electronic news and interest groups (Moran, 1992), workflow systems (Marshak, 1993 ; Marshak, 1994). Messages are more general than other requests and may include files, text, graphics and even executables, for example Telescript (Seybold, 1994, ref [2]). Enabling services include a store-and-forward facility, message filtering services, workflow rules and monitoring. Messaging may be built on distributed file or distributed database technology.

Distributed function allows a client to invoke a function on a remote server. This is usually synchronous, i.e. the client blocks until the server has completed the request and returned the reply. The most common form is the remote procedure call RPC (Birell and Nelson, 1984) which allows a client to invoke a designated remote function by name. Remote procedure call is popular because it uses familiar procedural techniques. Among the earliest commercial releases are Sun Open Network Computing ONC, Xerox Courier and Apollo Network Computer Architecture NCA, as described in (Stevens, 1990, chapter 18). Normally the client-server relationships are known at compile time and support for transporting the call and returning the reply is encoded in stubs which are linked into the application. The makes the model static. Enabling services in this style include binding services, transaction control, selection services. Many mesaging middleware products also support functions, such as IBM MQ and Message Express (Rymer, 1992; Kramer,1994).

Distributed objects support interactions among objects on different nodes in the network, with varying degrees of transparency. This allows flexible distributions of behaviour - fat clients or fat servers. This adapts the client server model such that the client and server are objects not processes. Furthermore calls can be nested such that another server may be invoked by the first server. Interactions need not be to a specific object. The infrastructure may automatically select the object best able to complete the task. Enabling services include those of distributed functions plus many more to manage objects such as replication, persistence, life cycle management, migration, concurrency control.

The relationship between the class of application and the type of technology is shown in Figure 17.

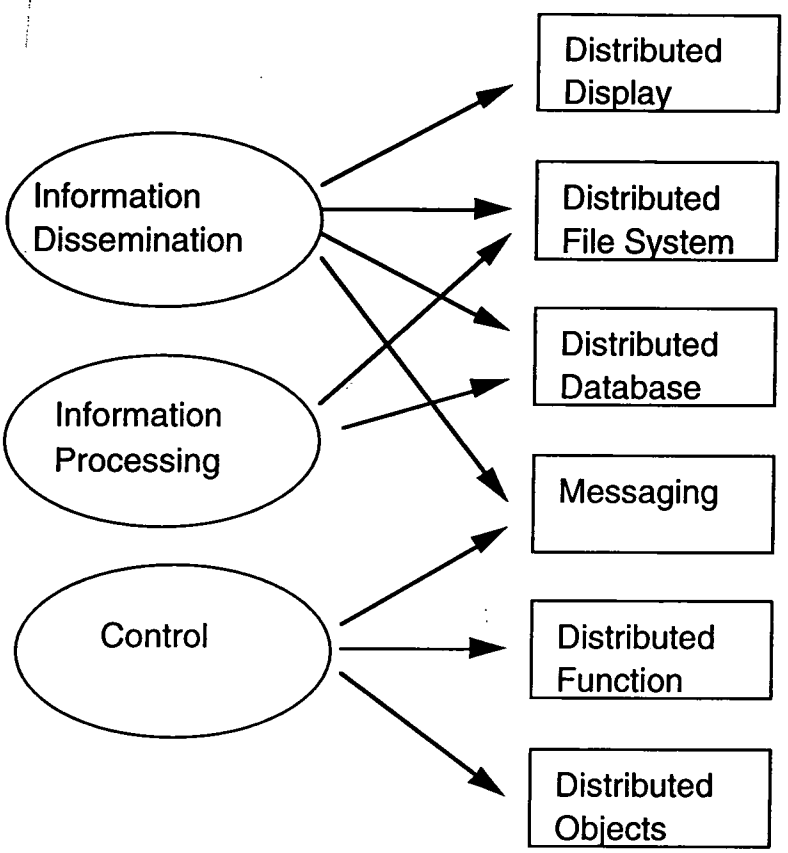


Figure 17 Mapping between application classes and technology types

Once the type of technology is decided, the best product in that technology type can be selected. This should consider the quality of the repository, definition facilities and other services, in order to give the organisation the scalability, robustness, flexibility and platform support that it needs. The relationship between technology types and product mechanisms is shown in Figure 18.

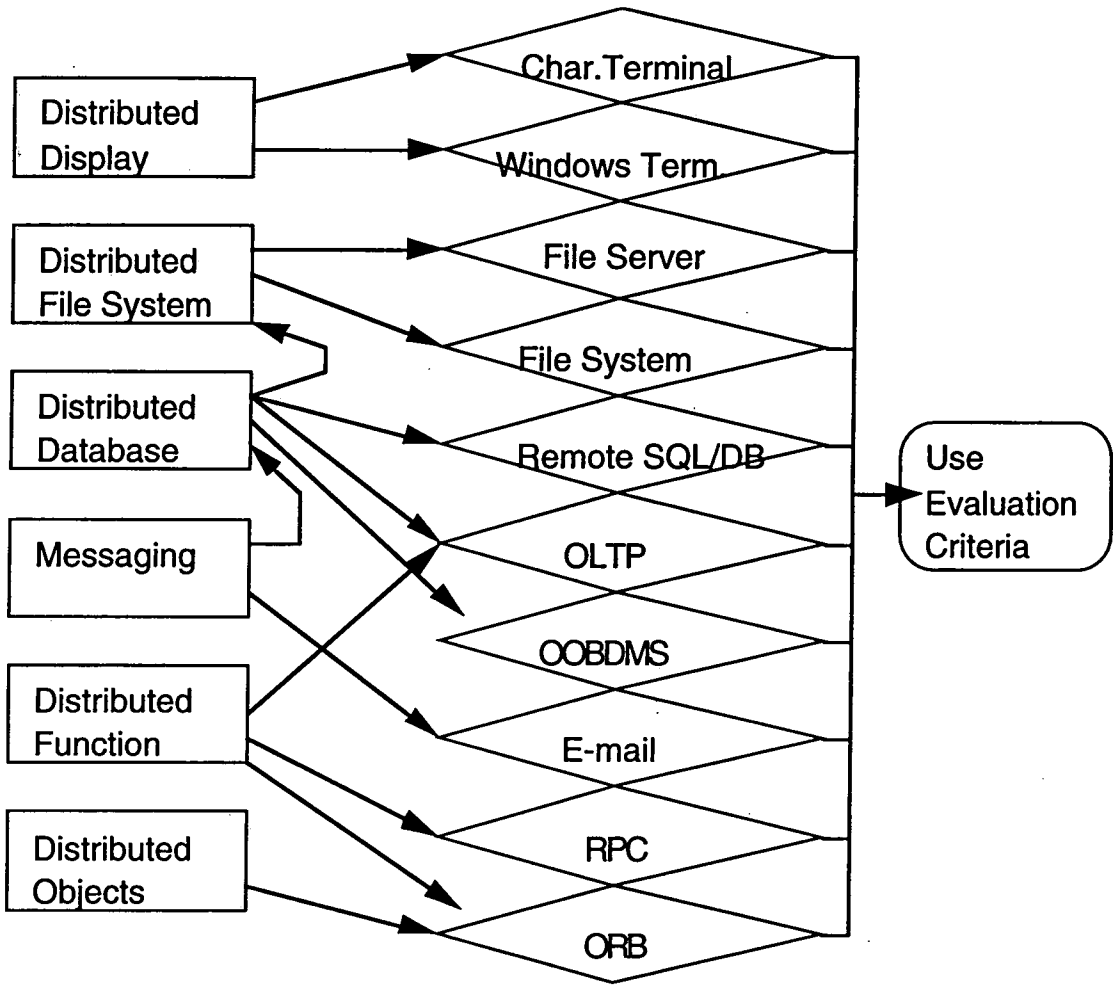


Figure 18 Mapping between technology types and products

The key evaluation criteria need to be defined, such as:

- Platform support, in particular UNIX products may be difficult to port to single-tasking desktops or IBM mainframes and gateways to Windows and Macs are frequently slow and unreliable.
- Reliability and performance, in particular an important factor for each is the ability to recover from network failures and the ability to process interactions in parallel, asynchronously. Performance will also decrease by a factor of the granularity of components and the coupling between them. Also the number of translation points such as gateways will reduce performance.
- Completeness, which may be defined as providing all the components in the generic architecture.
- Scalability, ability to support both small and large scale applications. Features such as hierarchical name spaces, server groups and flexible application partitioning, for example fat clients and thin servers, are invaluable.
- Openness, portability and interoperability. A crucial issue here is understanding the vendors interpretation and strategy for adopting standards, since most standards are partial specifications and continuously evolving.

- Cost, especially set-up, familiarisation and configuration costs which can be high in a distributed environment

- Development tools, in particular the effort and complexity of the definition facility, the integration of the tools with programming environments like C++, and the completeness for design, testing and debugging and maintenance.

- Management tools, this requires unification of system management tools and network management tools to gather and manage information about the environment, install and re configure the system, administer users and security and provide problem analysis tools.

Conclusions of Communication Technology Selection

This section has presented a simple method for selecting infrastructure technology that involves first evaluating the most critical class of application then the type of technology before evaluating products against general evaluation criteria and the specific needs of a specific pilot application in the most critical class of application. This will be used to assess the technology choices made by OpenBase.

There are a diverse number of technologies that can be used in client server systems. The commonest to date are distributed files system using network operating systems like Netware and relational database products like SQL*Net. The limiting factors in the growth of client server are the ease with which complex distributed applications can be developed and the restrictive scalability and openness of the first generation of client server systems.

Standards-based object oriented solutions offer more flexible dynamic partitionings of applications for scalability and performance, higher levels of abstraction to simplify development to tackle more complex applications and open standards for portability and interoperability.

4.2.2 Tools for Large-scale Reuse

Large scale reuse demands an up-front capital investment in tools and high quality components. This section provides a taxonomy for the tools that can be applied to large-scale reuse across a distributed enterprise.

Tools for software development started as programming environments supporting editing, compiling and versioning at the code level. Next came design tools like CASE that support parts of a software development life cycle or methodology. However these tools were expensive to develop and demanded a high degree of customisation. As a result, the concept of "tool-building" tools or meta-CASE tools arose.

Rapid change in higher level programming languages has since blurred the distinction between design tools and programming tools. This has led to the concept of integrated-CASE where the design tool is integrated with the programming environment and generates source code. With the growth of object oriented programming and object oriented development and its ability to carry concepts from design to code, both meta-CASE and programming languages are not only integrated, but they are now addressing the same reuse problems and applying the same object oriented principles and modelling concepts.

Even more recently tools are emerging that support new life cycle models that lessen the notion of a distinction between design and programming steps. This includes application generators, transformational systems, 4GLs, rapid application development tools and component assembly tools.

The distinctions between these different categories of tools are less important than the technologies that are applied by the tools and the impact they have on the development process.

The technologies that can be applied to the reusability problem can be divided into two major groups depending on the nature of the components being reused. New programs can be derived from building blocks by composition technologies. New programs can also be derived from patterns of code or transformation patterns woven into generation technologies. The former reuse parts of the product of software engineering. The latter reuse parts of the process or intermediate results of software engineering. Product centric reuse and process centric reuse are subtly different in practicality, generality, formality and flexibility.

The first section distinguishes these two basic approaches to software development. The next two sections look at technologies and tools that support each of these distinct approaches.

Two Basic Approaches to Software Development

There are two fundamentally different approaches to software development:

Specification-driven system generation

Traditionally, software engineering has been process centric. There has been considerable work on descriptions of the software development process and new and powerful tools to support it. These efforts have placed emphasis on the important role of formal specification as the key to validation of requirements and the generation of the system. Evolution is not conducted on the system itself but on the specification with new versions of the system being re-generated from the modified specification. Reuse is almost entirely process oriented by re-application of the generation process. This is the "specification-driven" approach.

The specification driven approach attempts to formalise the decomposition process. However it suffers because for non-trivial systems, no one specification technique seems to be adequate. This leads to a proliferation of views and formalisms that are difficult to integrate. Verification and generation has proved extremely difficult in practice. Declarative languages, such as functional and logic languages, that do have theoretically, sound transformations, fall short as practical specification tools. They are in fact programming languages.

There is also one inherent flaw in attempts to formalise decomposition. The underlying process of component identification remains informal as it requires design information not usually included in specifications.

Composition-based Approach

In contrast, the "constructive approach" views system specification as the composition of component specifications that may be identified informally but integrated formally. The validation process is the construction of the overall system specification from component specifications. Evolution is supported by component addition or replacement. A description of software components and their interconnection patterns provides a clear and concise level at which to formalise specifications. The system can be generated from such specifications by construction tools. A representation of system structure can act as an unifying framework upon which to hang and integrate specifications, design, construction, evolution and reuse of systems.

Generation Technologies and Tools

This section describes the technologies and tools that apply the generation approach to software reuse.

With generation technologies, reuse is less a matter of composition of components, but of execution or transformation of components. The programs that result from a common generation pattern often bear little resemblance to each other or to the patterns that generated them. The effects of processes can be more diffuse than the effects of building blocks.

Reusable patterns take at least two forms: patterns of code that are held in the generator program itself and patterns of transformation rules that are implemented by a transformation system.

- 4th Generation languages, usually integrated with a database and provide screen and report generation capabilities.
- Formal languages, like Z allow a high level unambiguous specification, from which applications can be generated automatically.
- Application Generators, take a high-level description of the task using structured editors, graphical notations or application-oriented language like a 4th generation language and produce the application.
- Meta-level application generators provide customisable application generators , for example Gandalf (Habermann & Notkin, 1986) tries to semi-automatically generate families of development environments that combine notions of programming and development.
- Cross-Platform Generation, can be used to abstract away from diverse machine architectures to provide a unified virtual interface. For example CORBA IDL (OMG, 1991) generates the appropriate stub for any platform to link to an application object.
- Transformational techniques, can be used to convert a program suitable for one architecture into a program suitable for another architecture.

Composition Technologies and Tools

This section describes the technologies and tools that apply the compositional approach to reuse.

In composition, the building blocks are largely atomic (all-or-nothing), relatively immutable (unchanged by reuse), and are passive elements operated upon by an external agent. Examples of such items are code templates as in C++ templates, subroutines and objects.

A few well defined organisational principles can be applied to component composition. :

Component Integration Systems

The issues of component composition include those of component interface designs and component integration. Composition systems may be layered on top of integration systems. Component integration occurs at two levels:

Coarse-grained integration occurs between entire applications or tools for example using UNIX mechanisms like files, I/O redirection, pipes, shell programming or Sun's ToolTalk (Julienne and Russell, 1993) for inter-application interaction.

Fine-grained integration includes efforts like OMG CORBA for communication between objects, Microsoft's Object Linking and Embedding (OLE) to allow seamless integration of objects with different formats.

Component Repository Services

Component repositories should address the problems of finding components, understanding components, and modifying components. Composition systems may be layered on different types of repository service.

Browsers and query services are the conventional way to find components. The issues of finding and understanding components are being addressed at a more abstract level by repositories that store and select components using requirement and design knowledge, for example the ITHACA software information base (Bellinzona et al.,1993), and repositories that support catalogues and component classification schemes, such as the faceted classification scheme (Prieto-Diaz, 1991). The issues of modifying components are being addressed by reusability systems like the Esprit project REBOOT (Morel & Faget, 1993).

Generalisation-Specialisation Composition

Composition based on generalisation-specialisation is powerful since generalisation emphasises techniques that look for similarities (hence finding opportunities for reusing tools and models) and specialisation emphasises techniques to accommodate differences to support wider scale reuse (by specialising tools and models that are similar). Generalisation-Specialisation practitioners are not in the business of building one point solutions. Unfortunately it is a capital-intensive approach demanding an up-front investment in capturing and packaging generic models and tools.

There are problems applying inheritance to reuse across a distributed environment. Widening the domain of sharing has implications on the development dependencies and the loss of understanding and trust:

- reactive inheritance, where programmers evolve the inheritance hierarchy incrementally, as in the Smalltalk style, is unproductive, since the class hierarchy must be rebuilt everywhere to avoid the overhead of navigating distributed inheritance hierarchies at runtime to support runtime binding. The alternative is extreme inefficiency. See (Bennet, 1987) for a critical evaluation of Distributed Smalltalk.

- inheritance introduces strong dependencies between components that will only be manageable on large distributed projects if the classes are sufficiently stable and easily understood. This demands a higher up-front investment in development and limits the use of inheritance to well-packaged class libraries.

- inheritance right across an enterprise demands careful management to bridge departmental barriers, for example to achieve the necessary programming skill levels and establish trust between departments.

- wider reuse leads to broader demands on reused objects. Different domains will require different semantics and behaviours from shared classes and it will be difficult to aggree commonality.

As a result of these problems, generation techniques and object composition techniques are more useful in a distributed environment than ad-hoc use of inheritance. Inheritance is useful to develop quality class libraries for distribution across the enterprise. Inheritance also plays a part in the representation of frameworks of collaborating objects.

Role Modelling & Object Frameworks

Role modelling is more appropriate to derive frameworks than generalisation-specialisation.

The *type* describes *what* an object does; the *class* describes *how* it does it. OORASS (Reenskaug et al., 1992) extends these two aspects of objects by adding the notion of purpose. The *role* of an object describes *why* it does it, in terms of the responsibility of the object within the organised structure of collaborating objects. Role modelling and responsibility-driven design (Wirfs-Brock et al., 1990) are two powerful developments in object oriented design . Sharing can occur at the role level. An object can play many roles, known as role *synthesis*. Methods that focus on role models, derive abstract patterns of behaviour that can be shared across tasks by specialising each role for the task. OORASS provides tool support for role modelling and synthesis activities.

The re-use of whole designs is potentially more powerful than the re-use of individual classes, yet this is also overlooked. A *framework* (Wirfs-Brock et al, 1990; Deutsch, 1989; Nierstrasz and Papathomas, 1990) is a high level design, consisting of a set of collaborating classes that are specifically designed to be refined and reused as a group. One of the goals of the ITHACA project is the formalisation of frameworks within a programming environment (Profrock et al., 1989). The steps taken when developing an ITHACA application are as follows, taken iteratively:

1. Select a generic application framework from the repository, called the software information base (SIB), using some very general application requirement such as the required application domain. The generic application framework encapsulates domain knowledge, requirements models, generic design and generic components in the framework.

2. Specify the specific requirements guided by the requirements model and requirement collection tools of the general framework and find and refine relevant generic designs.

3. Visually compose concrete reusable components according to the generic design to form a concrete running application.

4. Verify and modify the application.

5. Refine the general application framework using knowledge gained.

ITHACA provides tool support for: organising and finding components (querying and browsing) in the repository; for visually composing applications from components; and for requirements and design representation and selection. The SIB provides structured descriptions of reusable knowledge and components, organised as a semantic network mixing object relationships and hypertext, with a meta-model defining the basic components used in definitions. The requirements specification tools extends objects with the notion of role and allows model descriptions at different levels of detail with transformations between levels. Various requirement mapping strategies are supported for selecting designs.

Programming in the Large or Configuration Programming

Structuring a collection of objects to form an integrated system is a different intellectual activity from the construction of the individual objects. Configuration-based approaches to integration describe a system at two separate levels: a programming level and a configuration level. At the programming level, objects providing the program's functionality are constructed using an object oriented language such as C++. At the configuration level, objects are interconnected and encapsulated into composites using a configuration language. Configuration programming combines the efficiency and flexibility of a low level interconnection approach with the safety and simplicity of a high level language environment. This has been demonstrated in a number of systems: CONIC (Magee et al., 1989), RCMS (Coatta and Neufeld, 1992), DARWIN (Magee et al., 1992), GEREL(Endler and Wei, 1992). This approach has also been called programming in the large (DeRemer and Kron, 1976) or processor-memory-switch-level programming .

Distributed processes communicating by message passing is the underlying model most commonly used to implement distributed systems. Models consisting of loosely coupled and communicating, processing components can even be applied quite generally to none distributed software systems. Configuration programming advocates the use of this underlying interconnected-component model throughout the software process not just implementation. Systems are conveniently described and managed in terms of their software structure.

Structured design methods like Yourdon and Jackson Structured Design base designs on descriptions of configuration structure, but fail to carry notions across to implementation. Environments based on structured design or specification tend to be strong on notation yet weak in generating implementations and in evolution and reuse.

There are four key principles to configuration language design that also apply to specification:

- component specifications should specify the visible behaviour at the component interface.

- a distinct configuration language is used for structural specification and this is separate from the specification language used to specify component behaviour

- complex specifications should be definable as a composition of component specifications using the configuration language.

- changes should be expressed as changes of the constituent component specifications or their interactions at the configuration level.

These specification principles map onto detailed language design principles that are described with example code in section 5.2.3 . Conic (Magee at al., 1989) is an example of a configuration language and includes facilities for hierarchical definition of complex component types, parameterisation, graphical representation of configuration structure and conditional configurations with guards.

The ideas demonstrated in Conic are being extended to all phases of development in the Esprit II project REX on Re-configurable and Extensible Parallel and Distributed Systems (Kramer et al., 1992).

Structural specification provides a framework on which to compose specifications and generate and evolve specifications. One can associate optional specification attributes with actual components to include specifications for aspects such as functional behaviour, timing, resource requirements An analysis tool could be provided for each type of attribute to verify specifications across composition and support diverse formalisms for each type of specification. Users can select a formalism most appropriate to the requirements. Generally generation techniques are weak at system evolution except by re-application of transforms and the generation approach could benefit from structural specification. Composition rules can be used to derive composite group behaviour from constituent components, to identify where changes can be applied and to predict the affect of change. In addition, there are techniques to support compositional reasoning such as (Milner, 1980).

The process advocated for configuration language systems follows the following steps, which is a variant on the component-oriented life cycle:

- identify component types, identify processing components and produce structural description including main dataflows.
- produce specifications of interfaces, introduce control and intercommunication to refine component interfaces and descriptions into a precise enough description to permit isolated component design and testing.
- elaborate components, by hierarchical decomposition of components into sub-components or by detailed functional specification to identify further component types.
- construct configuration by instantiation and interconnection to form composite components at a suitable granularity for distribution and allocate these to nodes.
- evolve and maintain, by component replacement or reconnection.

In addition the process needs to account for the following:

- need to support distributed development by teams of developers,
- selective means of viewing different requirements, e.g CORE viewpoints(Mullery, 1979)
- support for duality between graphical and textual configuration descriptions

- the need to support multi-paradigm interconnection, for example by grouping and ordering primitive ports into protocol units that map onto different protocols for synchronisation in different languages and constrain inter-paradigm connectivity.

The key problem with programming in the large is capturing the dynamic structural change. It may be reasonably trivial to capture the static structure of an application in a configuration language, independently of the component implementations. However it is not so easy to describe dynamic object lifetimes.

The simplest approach to dynamic change is to depart from a purely declarative configuration language. Pre-planned structural change may be expressed using change scripts that may be executed as a method, for example a recovery method to explicitly describe recovery behaviour as object migration and relinking. Typically change scripts will destroy objects, instantiate new objects and relink them. Methods affecting the dynamic object structure must be separated from other method behaviour. The structural change belongs to the configuration language, other method behaviour to the object programming language.

Ad-hoc change is even more tricky and requires that transactional control and consistency constraints can be imposed on changes to ensure consistency.

Configuration programming may use visual composition to present a graphical configuration programming interface (Kramer et al., 1989). Visual composition is described next.

Visual Composition

Visual composition assumes that applications are made up of reusable components that can be composed using a visual metaphor. The visual representation of component structure gives an application developer a consistent conceptual model of the application under development. A component is made up of three parts, a presentation, a behaviour and a composition interface. The presentation is how it looks on the screen. This may be the actual user interface the component will have in the final application or some other visualisation. The behaviour is what the component has been designed to do. The composition interface binds a component to a given context. It consists of entry and exit ports that may have names, types and properties.

Components are assembled using a binding model which declares the rules for component compatibility and provides the composition paradigm. Responsibility for checking that bindings are compatible is divided between the binding model and the compositional interface. It would be possible to build a visual tool that supported this binding model and allowed visual program composition.

The UNIX pipe mechanism is a good example of a simple specific binding model. Constructing more complex programs out of simpler ones requires the connection of one programs output to another programs inputs. Input and output are read serially across the pipe. However this is not a visual binding model.

Different styles of application will require more sophisticated binding models than UNIX pipes. In particular bindings will usually be typed and exploit polymorphic features of the programming system. Dataflow composition models are popular such as Fabrik (Ingalls, 1988), which supports composites through gateways. Framework-oriented models are also becoming common such as Apple's ATG Component Construction Kit (Smith & Susser, 1992) providing a framework into which components can be dragged from a palette and plugged. Many commercial vendors are now adopting drag and drop metaphors for application development such as IBM VisualAge (VisualAge, 1995) and Servio Geode, Oberon Synchroworks and Serius (Rowan, 1992). The design choices for binding models are described in more detail for visual scripting languages in section 5.2.3.

Visual Programming

Visual composition is not the same as visual programming, which covers many areas ranging from program visualisation to visual languages. A programming language defines a set of primitives that are designed with a specific syntax and semantics. With visual programming these primitives are visually represented and composed. Visual composition uses application components as primitives and these can be reused using different syntax and semantics. Visual programming systems suffer because they lack constructs to express complex algorithms and data structures. Visual composition does not suffer in this respect because it does not try to represent language constructs. Instead the component programmer must express data structures and algorithms and the application engineer uses composition models to describe complex organisations of those components.

Most visual languages use a single visual representation. Garden (Reiss, 1987) allows the programmer to work with a variety of graphical and textual languages to provide multiple views of the system. A user can define and implement new languages. No one visual language is considered satisfactory for all users, rather the optimum system provides a mixture of textual and graphical views of a single program.

Conclusions to Reuse Tools - The Need for a Hybrid Reusability System

Large scale re-use is not an easy problem to solve and may require a combination of multiple generation and composition techniques and tools that reuse different development results and knowledge. In particular generation techniques applied to components before they are reused by composition can be particularly powerful.

One central problem in reusability systems is how to make the component representations or generation transformations sufficiently general to allow reuse in a broad range of contexts. The application context may affect the software in a number of ways: the precise role played by a component; the performance and dependability requirements that must be satisfied; the constraints imposed by the specific technology used to implement and execute the software. Distributed technology in particular imposes significant constraints on software.

Reuse strategies have been successful in systems where these factors don't vary greatly. However such systems have limited scope. This is typical of application generators, which generate software that performs the same role, such as reporting, on a limited set of target platforms.

A more adaptable reusability system is needed where the context does vary, as in enterprise integration systems that integrate different types of application. Such systems must eliminate some of the specificity introduced into the source code by contextual variables. Source code requires a more manipulative representation, so that the programmer can incrementally control the way a specific context is accommodated, yet at the same time specify precisely the permanent invariant aspects that are reused.

Composition techniques are limited by the precision to which executable components must be defined if they are to run on a digital machine. Computers do not tolerate ambiguous executables. This introduces specificity in the components. Generation techniques are limited by the freedom of general-purpose software development processes. This makes it difficult to formally capture and automate sufficiently general processes. In combination generation and composition can be the antidote for the problems of the other. Generation techniques may be easily defined to transform partially specified components and partial specification relaxes the level of specificity to which reusable components must be represented.

Partial specification of components could be supported entirely by composition systems, for example by specifying an abstract base class whose interface must be implemented in all classes that derive it. However the development of a new specialisation of a class is a process-centric activity. This is better supported by generation techniques. In contrast, the use of the specialised component is a product-centric activity, that is best supported by composition techniques.

For reuse across a distributed system, a partial specification may be a virtual specification that abstracts away from machine dependencies. Generation techniques are particularly powerful for migrating software across platforms.

4.3 Summary and Conclusions to Chapter 4

This chapter has discussed the appropriateness of different development processes for open distributed computing. There are four main conclusions that can be made.

Traditional top-down life-cycles do not support component reuse nor do they allow for application conformance to evolving standards for open distributed computing.

There are different styles of objects and these different styles impact reusability and distributability. Some have coarse-granularities, making them easy to distribute but difficult to reuse. Large things introduce a high degree of specificity. Those that do identify truly reusable objects, lead to fine-grained architectures that are difficult to manage in a distributed environment. In order to solve the dichotomy, more sophisticated structuring is required either using a hybrid of multiple styles or a compositional approach that explicitly manages the structure of the application. This is currently being done implicitly or on an ad-hoc basis for simple applications. Very few mainstream methods say much about distributed architectures and have no formal constraints that require or enforce reuse. This limits the applicability of the technology to more complex problems.

There are a number of different technologies that can be used to co-ordinate components of a distributed application across hard process and machine boundaries, for example distributed files systems, remote queries, remote procedure calls. The choice of technology is as important as other design decisions. Infrastructure commercial-off-the-shelf COTS products must be evaluated. This emphasises new skills. Development is as much driven by bottom-up COTS selection as top-down specification-driven design. The need to choose technology has a considerable impact on the life cycle. Technology choices also effect choices of standards.

Choices of CASE and programming tool technologies are also important. Many tools have now evolved to be more than design and implementation tools, in particular reusability tools. There are two basic approaches to develop reuse tools, generation-based tools and composition-based tools. The development of reuse tools in integrated programming environments has resulted in new styles of development, tool-based development such as RAD, visual composition, programming-in-the-large.

The large scale reuse problem is not an easy one to solve and may require a hybrid reusability system. Systems mixing generation and composition offer the greatest flexibility, in particular generative mechanisms to adapt the reusable representation of a component to a specific context before it is reused by composition mechanisms. This is facilitated by well factored transformations that deal with different variables independently. The system becomes more precisely specified as it is adapted. The client should have some control over the generality or precision of components at any time. This necessitates representations that support partial specification. Partial specification allows the programmer to leave many of the smaller decisions uncommitted and accommodate independent variables incrementally. Different trade-offs can be made, depending on the current adapted state of the partial specification.

The next section looks at the techniques that can be employed to design and implement a programming system architecture supporting open distributed computing and the integration of different techniques.

Chapter 5 Distributed Programming System Architecture

This chapter reviews some of the techniques and design principles that can be used to define a distributed programming system architecture. Detailed runtime support functionality is described in Chapter 8 and Appendix B:.

5.1 Overview

What is a distributed programming system architecture?

An architecture is a description or specification of the structural organisation of a system. It is only when a system is placed in an environment and used by its clients that its structural organisation becomes really important. In this context the operating constraints of the environment become important, such as timeliness and space limitations. In addition an architecture must be responsive to the changing needs of its clients. It must encapsulate appropriate abstractions given the problem domain, and maximise the applicability and minimise the volatility of these abstractions as the system develops.

We can view an architecture abstractly as consisting of a set of discrete model components linked by some co-ordination mechanism. The co-ordination mechanism may be implemented as a simple communication mechanism. Most dynamically, this may consist of a messaging channel where interpretation of messages is up to the receiving component, ensuring maximum flexibility and extensibility in the architecture. More tightly, it may consist of the invocation of procedural or functional abstractions. This increases the predictability and safety of the system. Alternatively, the co-ordination mechanism may consist of a simple composition mechanism. Most statically, it may be interpreted as code composition such as inheritance or included libraries. More abstractly, it may be interpreted as a transformation or process, such as the application of a textual processor on a source component to generate/manipulate a target component. A programming language is itself a generic architecture in the sense that it can be instantiated by compilation or interpretation to provide particular execution structures.

The term programming system is used to signify both programming language and operating system. As discussed in Chapter 3, the distinction between language runtime support and operating system is frequently blurred, especially in an object oriented system. From an architectural viewpoint, a programming system is the combination of features of the runtime support of the operating system and the runtime support of the programming language.

In summary, a programming system architecture is both a layered implementation structure and a commitment to a stable organisation of interfaces and runtime support services for the benefit of its clients.

What are the most appropriate architectural concepts?

Distributed systems are structured by processes at the most basic level, and the most appropriate concepts build on process communication channels.

Portable systems must ensure the system behaves consistently over changes to the operating or device environment. This can be achieved by localising such changes in back-end components while preserving the abstract behaviour in independent higher-level components. Heterogeneous architectures naturally form layered stacks that abstract away from encoding or representation biases up the stack. Architectural concepts that deal with separation and transformation between layers of abstraction are important.

Open integration systems must ensure the substitutability and interworkability of application components. This goes well beyond process communication. Architectures for integration provide a high level rationalisation of the overall design of the integrated system in terms of its structure and component applications. The software architecture defines the glue that binds the components together and provides tools to support the procurement, assembly and distribution of components. Architectural concepts concerned with component compatibility and composition are important.

What are the key architectural goals and principles?

The basic goal in the design of an architecture is a separation and factoring of concerns and the basic organising principles in its construction are standard software engineering strategies: modularity, levels of independence, abstraction, extensibility and component reuse.

The rest of this section describes some concepts and techniques that support these goals and principles.

5.2 Architectural Concepts and Organising Principles

This chapter surveys several organising principles and concepts for architecture design. These principles use a variety of mechanisms to co-ordinate components of the architecture, including : messaging, invocation, inheritance, pre-processing and transformation. There are three sections that map to the most appropriate types of concept, identified above:

- the first sub-section overviews concepts used for process communication.
- the second sub-section describes concepts used to provide abstract layers in the programming system design.
- the third sub-section looks at concepts that aid reasoning about the composition and structure of the software architecture.

5.2.1 Implementing Process Communication Concepts

This section describes a number of different communication mechanisms for components on different hard processes or on different nodes in the network. Detailed design alternatives and runtime support functionality are described in Chapter 8.

Unstructured data streams

Unstructured data streams are useful in a number of situations, for example for continuous media in multimedia systems. A complete general-purpose system should provide both a stream interface as well as higher level mechanisms such as RPC. CORBA 2.0 is investigating stream protocols.

Stream protocols for multimedia are not trivial. There are classic problems such as the lip-synch problem in mixing video and voice. Research is underway to enhance stream type systems to support quality of service properties for dependable data transfer and composite types for mixed media streams

Unidirectional Messaging

One-way message passing gives clients explicit control. In particular, clients are free to interleave activities, submit multiple requests in parallel and direct replies to arbitrary addresses. However higher level protocols must be explicitly programmed. Servers are obliged to keep track of any reply addresses, and clients and servers must co-operate to match requests to replies. A protocol in which requests implicitly entail replies eliminates this problem.

Network Programming Channels

There are a number of network programming interfaces that have widespread availability in the UNIX world, in particular Berkeley sockets, UNIX pipes, the System V Transport Layer Interface and System V Streams (Stevens, 1990). By using TCPIP or UDP transport protocols, these interfaces support interoperability across UNIX systems.

These programming interfaces can be used both for unidirectional and bi-directional messaging, both using connection-oriented and connectionless policies. They provide system calls to allow network endpoints to be created, bound using network addresses, establish connections and accessed to read and write messages. Name services are used to map network addresses to textual names and to pass addresses across the network.

Remote Procedure Call

Remote procedure call systems are now mature with commercial systems including Netwise RPC, SUN Open Network Computing, Xerox Courier, Apollo Network Computing Architecture, OSF/Distributed Computing Environment, DEC NIDL. For three examples see (Stevens, 1990, chapter 18).

With strict RPC the calling thread is blocked until the server accepts the request, performs the service and returns the reply. Procedure call semantics make it trivial to obtain replies and return results. This makes RPC easier to use than unidirectional messaging or network programming interfaces where requests and replies are explicitly read and written. However it is not so trivial to interleave activities, submit requests in parallel, or re-specify reply addresses.

RPC systems typically have three levels in their architecture:

- in the application level, the client makes an RPC call on a remote server and the server provides a definition for the procedure called.

- in the stub level, a clients call to a remote server is intercepted by a **stub** that supports the servers interface. The stub encodes the call and the arguments into a message which it passes to the transport layer. At the other end the **skeleton** receives a message from its transport layer and decodes the call and arguments and performs the call on the server.The skeleton then receives return values and encodes them into a reply message. The stub receives the reply and decodes the return values and returns control to the client.

- in the transport level, the source receives request messages from the stubs and sends them over the network to the destination. The destination waits for incoming requests and forwards them to the appropriate skeletons. On return the destination receives the reply message from the skeleton and transmits it across the network. After transmitting a request message, the source waits for a reply message and forwards it to the stub.

This behaviour is shown in Figure 19.

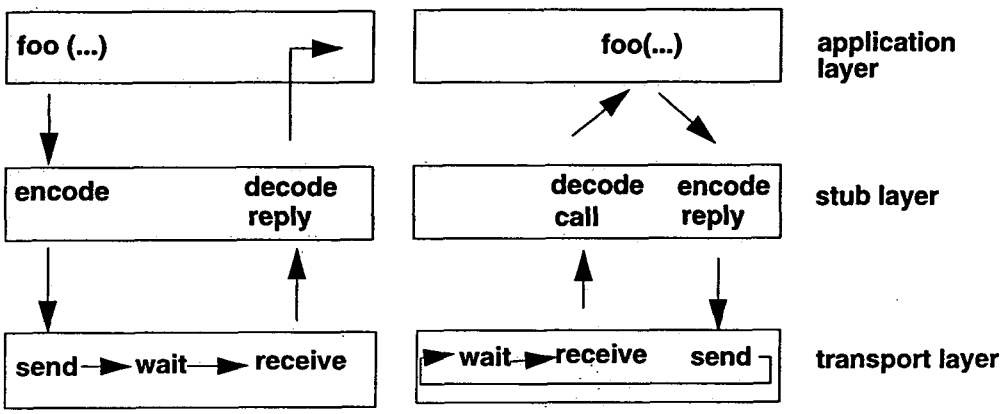


Figure 19 Implementing RPC

The following discussion gives detail that is hidden by the implementation of an RPC system and is not usually required by users. However an awareness of the underlying mechanisms is necessary for a complete conceptual understanding of an RPC system, especially to understand the design of an interface definition language and to rationalise performance.

The application layer is programmed in a procedural programming language to define the behaviour of the client and the server.

The stub layer provides the presentation protocol for the remote server. It is type-specific since specific typed procedure calls are invoked: on stubs, and by skeletons. Stub/skeleton pairs must be linked with the client and server for each server type supported. The stub is responsible for encoding/ decoding arguments, called marshalling and is also involved in binding.

Stubs are involved in managing dynamic binding information.Some systems will create a stub and skeleton for each binding between a caller and a callee. In this case, the stub encapsulates the binding information and acts as a proxy (i.e. a local representation) for the remote object. In other systems stubs may be shared by all objects of the type.

Marshalling may use such techniques as outlined in ASN.1 (Steedman, 1990) for basic data types to deal with issues like byte order and encodings. When references to objects are passed as arguments the system must also generate bindings in a stub/skeleton pair for the object passed. If the system manages garbage collection then reference counts also need to be incremented or decremented so an object is not deleted whilst there are active remote clients. When complex composite data types are passed such as unions and sequences, the stub must also send additional metadata for the argument such as an enumerator discriminating the type of the value in the union, or a value for the size of a sequence or string.

The transport level is responsible :

- for managing connections, for example using tcp sockets or named pipes. Some systems write over a single active connection on each change in connection, others may set timers to control the lifetime of a connection. The binding information, e.g. the name of a named pipe or the socket address, will be maintained in the stub or be embedded in the object reference in the application, so reconnection or connectionless protocols can also be used without making calls to a name service to rebind on each call.

- for multiplexing and demultiplexing messages between multiple stub/skeleton pairs or multiple callers/callees pairs when there is more than one caller or callee in the same process. A single endpoint for a process or a single connection between processes is usually shared by all callers and callees in a process, i.e. transport layer connections are between processes not objects. The socket address or pipe name is usually used to bind to a remote process. An extra identifier needs to be introduced to resolve which stub or skeleton the message is intended for. If stubs and skeletons are shared between calls rather than being instantiated for each call, the identifier also needs to identify the caller or callee. The transport layer must maintain some form of session table mapping identifiers to memory addresses. These identifiers must also be passed in the messages on binding and procedure invocation. The session table may also be cleared using timers, requiring reallocation of identifiers and remapping of session tables.

- for request scheduling at the destination and reply scheduling at the source. Requests must be dispatched to server threads. The server may spawn new threads or allocate threads from a fixed pool. Likewise a multithreaded client may carry on processing other activities whilst one activity is blocked waiting for a reply. If the interface from the stub to the transport layer has separate downcalls for both *send* (non-blocking) and *receive* (blocking), it is feasible to depart from synchronous RPC semantics and allow a client to carry on processing after issuing *send* and to later block by explicitly calling *receive*. Likewise a server could be processing and determine when it wants to block to *receive* messages explicitly as in a Ada rendezvous using *accept* and it could carry on processing after calling *send* for the reply.

- for fragmenting large messages and retransmitting lost packets. Typically ordering information must be embedded in each packet. One issue here is whether to use flow control or windowing protocols to ensure receivers don't get too far behind.

- for detecting lost or corrupted or out of sequence messages and retransmitting them. The protocol can be optimised for normal RPC processing by using replies to acknowledge sends rather than using an enq-ack protocol. Probing with an enq-ack protocol after a timeout is used to detect failures in the rare case of an exceptionally long service or when communications fail. Sequence numbers embedded in each message can be used to detect lost messages.

- for notifying and tidying up abortions on partial failures such as long term communication failure or node failure. This can be detected by having a limit on the number of retransmits. On aborting a client can leave orphan servers processing a request and any orphan processes must be detected and terminated. Incarnation numbers can be included with binding information passed in the message to ensure that a binding is invalidated if a server is restarted.

Rather than hard coding stubs for each procedure, a stub generator can be used to generate the stubs from a higher level description of the procedures. An **interface definition language** is one way that procedures can be described and this has been adopted in both CORBA (OMG, 1991) and DCE (Shirley, 1992). The interface definition language compiler generates the stubs. This hides the interface between the application and the stubs. The transport layer itself is hidden by the stubs. Consequently both DCE and CORBA can be used without any knowledge of the stub and transport behaviours described above. An interface definition language consists of four main parts: a specification of the composition of the interface and relationships with other interfaces; data specifications; operation specifications; and property specifications, as described in the section on interface specification in section 5.2.3.

RPCs have been found useful internally to structure systems architectures. They have been used in microkernel operating system design and distributed system management and control.

Remote Method Invocation

RPC systems tend to have quite a low level binding interface requiring several calls to bind server functions and data (contexts). This makes them inappropriate for general purpose object invocation. Instead they tend to be used for models based on client and server processes where functions are distributed not objects and services.

RPC systems are not object oriented. They generally do not support interface inheritance or polymorphism.

A remote method invocation system seeks to make RPC more appropriate for objects. To do this RMI systems extend RPC in a number of ways:

- all introduce object references and higher level binding mechanisms. This may be as simple as providing a bind method to return a pointer which can then be used in a normal method invocation.

- others support subtyping (interface sharing) and polymorphism in the binding mechanisms, where a server of a different subtype can be substituted. The simplest are statically typed and the client must include the type definitions of the server interfaces it is using. (Many C++ based systems require a client to include the subtype definition even when only using a base reference.)

- others depart from strict procedure call semantics, for example by allowing asynchronous messaging or dynamic invocation where the client can compose an invocation request using enumerated type codes without including the static type definition. This increases the contextual independence of clients. Invocations can be constructed on the fly without advance knowledge of interface definitions.

- others also support subclassing (implementation sharing) where an invocation may be directed to a superobject component of an object. In some schemes, abstract interface classes are separated from implementation classes across inheritance and subclassing requires multiple inheritance from both the interface class and the implementation class

There are three main approaches to build a remote method invocation system:

- An object based style can be superimposed on RPC by application developers, for example by parameterising method and object identities. This compromises inheritance, polymorphism, interoperability with any external software that uses a different approach, and access transparency (where local and remote object invocations look the same)

- A remote method invocation system can be engineered on top of an RPC system, hiding the RPC system internally. However failure to support object references and inheritance is likely to reduce the RPC to being used as a generic interprocess transport mechanism only. There are problems in trying to layer the two:

    - two IDLs are difficult to unify.

    - extra naming, data marshalling and location services are required for managing location-transparent object references.

    - RPCs are static, extra support is required for dynamic invocation.

    - RPCs are synchronous, a thread dispatcher is required for asynchronous calls or deferred synchronous calls where the client issues a request asynchronously and blocks later to collect the results.

- A remote method invocation system may be engineered on top of network programming primitives of the operating system, such as sockets.

RPC and RMI Tools

RPC and RMI systems may provide a number of tools to deal with user logins, security, system start-up and to manage directory services.

Indirect Interaction

Not all high level messaging services build on direct point-to-point delivery or requests and replies. Store-and-forward capabilities allow a variety of interaction policies. These systems vary in the way they name, allocate and access the storage units into which messages are added, read or removed by clients. Alternatives include:

- a single logical, globally accessible repository, for example a tuple space (Gelernter, 1985).

- a federated network of message servers, for example news servers and interest groups (Moran, 1992).

- dedicated message queues that are allocated to and read by individual components, for example mailboxes and mailtrays (Rodden and Sommerville, 1989).

Indirect interaction is inherently flexible due to the indirection. Servers need not be named, for example they may be identified dynamically by the content of the message, content-based routing or on configuring the system. This makes indirect interaction powerful for configurable systems. The store-and-forward facility can maintain a mapping between messages and servers using different abstractions of their properties, for example mappings may be resolved by matching service requirements and service properties, news and interests, resource needs and imposed system loading, group names and group membership. Indirection also enables active concurrent recipients and message filtering and scheduling.

Indirect interaction can be particularly powerful for developing parallel distributed applications, especially when both communication and synchronisation are combined into a single interaction primitive. For example, Linda (Carriero and Gelertner, 1986) is based on the notion of a tuple space. Programs can throw tuples of typed data into the tuple space without knowledge of the recipient using the *out* command that takes a tuple of data as argument. Likewise they can selectively consume or copy tuples out of the tuple space using the *in* or *read* commands respectively. Tuples can be selected by value or type according to the argument to *in* or *read*. Linda supports parallel programs in many styles: live data (e.g. future variables (Yonezawa et al., 1987)); co-operating networks of specialised workers; or bags of tasks executed by homogeneous workers. Object oriented languages using tuple spaces have been designed to research parallelism and configurable systems, for example Kitara (Guffick and Blair, 1992).

The main problem with indirect interaction is the runtime overheads that it introduces. Indirect naming and the transparent selection of servers can be supported in the component programming language without incurring these runtime overheads, for example by resolving the indirection on configuring the system as in a configuration programming approach.

Triggered Interaction

Triggered communication is based on the notions of a call-back address or event handler. A server registers the call-back or handler and the client triggers it. This basic scheme varies in a number of dimensions:

- in the way signals are named and raised. Options include signalling named events, raising exceptions, publishing news items, requesting service descriptors or changing a data item that has attached data triggers;

- in the degree of indirection, for example an independent server may co-ordinate registration and triggering, alternatively signals may be automatically propagated backwards like exceptions from server to client to outer client.

Publish-subscribe interaction allows a server to register an interest in a subject group and be triggered if another component publishes an item of news in that group (Moran, 1992).

5.2.2 Architecture Layering and Interface Design Concepts

This section describes some techniques that can be used to organise the interfaces between layers in the architecture or to define the relationships between runtime support and higher level programming abstractions.

Pre-processors and Generation

Pre-processors may automatically generate low level runtime support structures. In this way, the runtime support is hidden from the programmer. This masks the programmer from engineering detail and can be the basis of an abstract programming interface that supports portability across different mechanisms and transparency to the complexities of the runtime support mechanisms.

For example, the IDL compiler is a valuable gift that the OMG has given the computing industry. It is used to describe interfaces and generates the necessary support to hide the complexities of interfacing to the communications system behind higher level binding and invocation mechanisms. Specifically, the IDL compiler builds the appropriate code to manage proxies for remote objects, dispatch incoming requests in a server, and manage any underlying object services.

Generation tools may also generate installation scripts, for example a tools may create make files and generate loaders to build and load the software automatically. This helps manage the diverse component based environment by masking component programmers from dependencies of the environment and complex build dependencies.

System APIs

Conventional operating systems provide system libraries of services that can be called to access resources and services of the operating system. These libraries can be selectively included and linked with application software. However they tend to support low level services that can be represented procedurally and do not mask complexities.

System Frameworks

Derivation mechanisms like inheritance provide a more abstract and flexible way to standardise messaging interfaces than conventional APIs. A client need only know that a component conforms to a given base interface or type, not the complete set of types provided by the component nor which implementation of the given type is provided. A supertype interface can standardise the interface to a set of object implementations. Frameworks use abstract base classes to capture interfaces between components .

Libraries of standard component classes usually must make some assumptions about other classes with whom they collaborate to provide a behaviour. Frameworks allow these assumptions to be articulated by specifying supertypes and messaging patterns between these supertypes. Any number of concrete implementations can be derived from these abstract frameworks. This allows behaviour to be selectively composed by choosing different specialisations of the abstract component.

Frameworks are well established in the user interface community, such as MacApp (Schumucker, 1986) and the Smalltalk-80 Model-View-Controller (Krasner et al., 1988). They typically provide the building blocks of a user interface such as buttons, scroll-bars, windows and so on as well as the protocols and conventions by which they work together.

Object oriented frameworks can also be used to interface to the system to provide object management services. This is different from a conventional API in a number of ways:

- inheritance can be used to mix-in system behaviour to application objects, for example a persistence framework may provide a class called Persistent_Object from which any persistent objects derive the persistence capability. Such classes are called *mix-ins*. Mix-ins are base classes that are included to support the implementation of a class and are not valid problem domain generalisations for the application class.

- the ability to mix-in behaviour means that the API may easily involve both downcalls and upcalls, i.e. don't call the system, the system calls you.

- shared classes can be built to provide the interface required by a framework and reused across a domain, providing default behaviours everywhere they are used. To illustrate this consider an example of a data exchange framework. Data exchange between objects would usually require either naming conventions or naming mappings. On the other hand, a data exchange framework may build exchangeability into all primitive classes of the domain. Classes built from these primitives , through aggregation, inheritance etc. , could exchange data automatically without name mappings or conventions. An example usage of this framework would be to allow a customer business card to be dragged between applications for example from an customer account statement onto a customer loan application form.

- frameworks are extensible and customisable. A framework consist of a pattern of messaging between abstract classes. These abstract classes can be specialised to implement different custom behaviours.

The best known examples of system frameworks are Choices (Campbell and Islam), Taligent's application , domain and support frameworks , NCR Co-operative Frameworks , IBM's DSOM Persistence, Replication and Emitter Frameworks (Rymer, 1993).

Taligent's support frameworks alone include frameworks for: distributed data access, store and forward for e-mail and workflow, transport-independent networking, portable I/O and microkernel extensions and runtime object services.

Despite the importance of frameworks there is little support for their construction and customisation in programming environments. The ITHACA project (Profrock et al., 1989) seeks to formalise binding models for constructing frameworks within a programming environment. There is no one universally applicable protocol for plugging collaborating components together. Nor is there a universally accepted model for capturing abstract patterns of interactions as frameworks. Instead ITHACA seeks a range of binding models for different problem domains that each provide different ways of configuring objects as applications or as reusable frameworks.

Framework-oriented programming necessarily requires a methodology that emphasises capturing abstract patterns of behaviour, such as role-modelling techniques.

Proxies

The sending of messages and the receipt of replies can be delegated to external proxy objects. Proxies may act as local representatives for remote objects. Proxies work by manipulating the reply address. Proxies introduce indirection in the messaging path. This provides a simple means to abandon a strict RPC protocol. For example, they can support caching strategies or local call semantics.

Proxies are sometimes used to disassociate the sending or receiving of messages from the current thread of control, in order to add external concurrency to an object with restrictive internal concurrency. The benefits of simple procedure call semantics can be preserved if the proxy provides a blocking request for the original client to obtain the reply. For example, on asynchronous sends, a proxy may provide a token for the client to redeem later when collecting the results.

Proxies maintain the flexibility of message passing systems, without the difficulty in matching requests to replies. They may be generated automatically by an IDL compiler or stub generator and created and deleted on the fly by object factory services, object constructors and destructors or data marshalling code used to pass object references.

A proxy need not be an object, data arguments can use proxies. A future variable (Yonezawa et al., 1987) is a proxy for a live data item that is being computed in parallel by another server. Access to a future will cause a user to block until the data item has been computed. Futures are similar to the redeemable token except that they allow control at the level of individual data arguments. Furthermore futures are not object references and cannot be passed onwards as arguments.

Meta-object Protocols and Reflective Architectures

If the language implementation itself is structured as an object oriented program, then the behaviour of the language can be incrementally modified by modifying the language objects (i.e. the meta-objects) . This is the basis of the meta-object protocol approach. Meta-object protocols provide interfaces to the language objects and allow them to be customised and extended by the programmer, for example by specialising a meta-object class.

Proxies may be viewed as meta-objects for remote references. Meta-classes of (Smalltalk, 1983) can be viewed as meta-objects for classes. (CLOS, 1989) supports a much richer set of meta-object protocols as detailed in (Kiczales et al., 1991), including meta-objects for methods, generic functions, objects and classes. Meta-object protocol techniques are widely used, for example in C++ in advanced object management (Bijnens, 1994) and in extensible language design (Chiba, 1993).

By opening up the language mechanisms, the programmer can himself resolve the tension between the conflicting general goals of expressivity and efficiently and between extensibility and backward compatibility. Different applications can use different versions or variants of the mechanisms for entirely appropriate reasons. The conventional distinction between programmer and language designer is blurred by meta-object protocols.

Conventional programming systems constitute a single point in language design space. A language supporting meta-object protocol constitutes a whole region of language design space, a region that can be made expressive, efficient, compatible and extensible.

Transparencies

Transparencies is a property of a system such that engineering detail is hidden from the user of the system. For a programming system the user is the programmer. The ANSA architecture (ANSA, 1989) defines seven types of transparency:

Access transparency - call semantics are identical for local and remote calls

Location transparency - user unaware of physical locations

Migration transparency - user unaware of object movements

Concurrency transparency - user unaware of other users

Replication transparency - effects of multiple copies are hidden

Failure transparency - problems of partial failure are masked.

Despite the existence of transparencies, there are still two fundamental assumptions that cannot be overlooked: that effects are not instantaneous, and that the right resources are required to do anything. These assumptions manifest themselves in the interface to the programmer and should not be ignored. For example, call semantics must assume remote access and allow for communication latency and extra failures. Resources and objects must actually be deployed and allocated over the network. Concurrency awareness is important to avoid deadlock or excessive synchronisation delays. Replication policies must be selected according to the availability requirements and service semantics.

Transparency in C++ has been studied in the context of the ARJUNA project (Parrington, 1992). It has become apparent that transparencies are most useful if they can be applied selectively, i.e. selective transparency, for an example see (McCue, 1992). Sometimes a programmer wants explicit control of a property, for example for critical systems that must be deterministic. Selective transparency allows him to chose which properties are explicit and which are transparent.

Viewpoints or Projections

In the purest object architecture, all components of the architecture are objects. Runtime support for high level objects is internally object oriented. This does not mean that all co-ordination between programming objects and runtime support need be implemented as object oriented relationships such as inheritance and messaging. Projections allow us to define different object models at different levels of abstraction and explicate the transformations between an object at one level of abstraction and the objects at the next level of abstraction that implement it.

An example of such a transformation may be between a replicated object and the several replicas that implement it. A computation view is of a single object, messages are sent to a single object and a single reply is returned. Yet the engineering view is of several objects, a message is broadcast between the replicas and replies are collated to form a single return.

The transformation between projections may be viewed as a runtime support relationship between the high-level model and the low level model that implements it. This relationship may be an object oriented relationship e.g. using frameworks, delegation to proxies, meta-objects. Alternatively a model expressed in some object oriented modelling language may be transformed into a more sophisticated concrete object representation by generation using a language processor.

It is important to separate or distinguish runtime support relationships from other model relationships. This separation of concerns provides a layering in the architecture that provides a framework for rationalising language design and applying techniques like selective transparencies and transformation. The modelling tool at any one level need not deal with all the operational complexities of distributed systems, for example persistence, synchronisation, recovery, late binding and replication. Projections avoid semantic overload at any one level of abstraction by separating concerns and dealing with distributed properties more declaratively at higher layers.

The notion of layers and different views of a system is widely adopted. The ODP architecture defines five projections each with its own modelling language as described in (ANSA, 1989 and ANSA, 1993):

- The enterprise projection models what the system is to do and who for. It explains the role of the computer system within an organisation
- The information projection models the information structure, flow, interpretation, value, timeliness etc. It focuses on the location of information and the description of information processes.
- The computation projection defines the programmers view of IT services including languages, APIs, application protocol stacks.
- The engineering projection describes the execution services for processing, memory and communications functions. It enables designers to reason about performance & dependability of systems and trade-offs between mechanisms.
- The technology projection defines the physical components in terms of hardware and software.

The notion of projections or viewpoints has more widespread usefulness than language design. For example, the architectural services department of the Inland Revenue have adopted the ANSA projections as an organisational framework for analysis, procurement and technical research. This encompasses: the selection of technology products and interfaces; determining the application topology; systems analysis; business modelling; and advising on capabilities, cost, risk and time-scale. The framework gives them: traceability from requirement to provision to impact; complete and consistent coverage of technical issues; and a component based approach to architecture design and reuse

Other frameworks of viewpoints are popular for analytic activities rather than language design, especially the framework of (Zachman, 1987) and of CORE (Mullery, 1979). The Zachman framework provides a number of different angles from which to view the software system and its environment. These angles are tied to roles in a development project and to aspects that each role should consider: data, function, people, time. These angles are considered at different levels of abstraction. The result is a method that describes different aspects of the system from different perspectives. Applying such a framework results in a comprehensive set of well-bounded and well-integrated models.

5.2.3 Concepts for Defining Program Composition and Structure

This section describes some design approaches for programming systems that can be used to analyse and define the structure of a distributed program. It begins with mechanisms that operate on flat structures of components where the overall structure is implicit in the sum of the relationships between individual components. It then considers approaches that deal with global structure explicitly.

Narrowing and Trading

Two mechanisms that seem to have gained a degree of consensus and popularity are trading and narrowing.

Narrowing is the process of exploring a component's interface dynamically at runtime to find what one can do to it. Narrowing provides a flexible way to select servers when the interface required is not known in advance. It is also useful for safe casting of a polymorphic reference to a specific subclass. In statically typed environments, narrowing may be associated with dynamic invocation mechanisms where the client composes an invocation on the fly using enumerated type data rather than using statically typed references. This avoids the need to include the type definitions for used classes and reduces dependencies between components.

Narrowing support requires the enumeration of typing information and the definition of interfaces to access and use this information. The enumeration of typing information is especially useful to support generic object management tools for browsers, configuration managers, debuggers, routers. Many of these tools can be implemented by writing generic narrowing code thus avoiding the need to write extra code for each type of object manipulated. A type repository can be used to find interfaces and narrowing used to find appropriate methods in these interfaces. CORBA provides a type repository and type exploration methods to support this (OMG, 1991).

Trading is a less flexible mechanisms than narrowing, since the interface is normally know in advance. Trading is used to select a server that conforms to a given interface. An offer of service is exported to the trader and clients import references from the trader. An offer may include the type of the server, the server name and various properties of the server. A request optionally includes a service name, type and properties required of the service. Type conformance and hierarchical names can be used to restrict the selection. Protocols are then required to resolve conflict between multiple servers. These may use server properties. ANSA provides a query mechanism for matching properties(ANSA, 1989). Conventional directory services such as DCE Directory Services (Shirley, 1992) may select servers randomly.

Trading can occur at several levels. In large system there will be many traders, optimised to different needs, with different naming and trading policies and interconnected into a trading federation with context relative naming and interception at boundaries.

Interface Specification and Adaptive Management

One convenient characteristic of objects is the amount of semantic information that can easily be made visible in interface definitions. An adaptive system can exploit this information. This facilitates a declarative approach to object management. The programmer can specify the properties required of objects, using an interface definition language. It is up to the programming system to interpret these properties correctly.

Correct interpretation requires us to develop concrete mechanisms for requirement representation and implementation. Techniques such as meta-object protocols, viewpoints, configuration programming, workflow models, all facilitate the design of an adaptive system by breaking down the problem with well defined boundaries. One simple approach is to reduce interpretation to the level of trading where the interpreter is trading for different system services based on the semantic information provided in interfaces.

Such systems are dependent on the amount of information that can be specified in an interface. Generically an interface can be thought of as consisting of four parts, for an example consider a CORBA IDL interface (OMG, 1991):

- a specification of the composition of the interface, including interface inheritance, code dependencies and scoping constructs like contract groups of services (Wirfs-Brock et al. , 1991) and CORBA modules (OMG, 1991). An object may have multiple interfaces, for example a management interface, an application interface and a customisation interface. Likewise an interface can have many implementations.

- a specification of the data, including declarations for attributes, data structures for exceptions, and constructed data types like sequences, nested interfaces, unions. Data types may require extra meta-data such as the length of a dynamic sequence to allow distributed memory management.

- a specification of the signatures for the operations or methods provided by the interface and optionally for operations required in other interfaces. This can be extended with extra contextual and error handling constructs, extra qualifiers for properties such as argument direction, operation synchronisation, operation idemnipotency, and timeliness constraints.

- a specification of object properties, for selecting servers based on semantic properties or for verification of suitability to meet requirements. This can include quality of services properties or dependability properties of the implementation like performance, efficiency, latency, resource usage, availability, robustness, reliability and can be used to analyse requirements or to resolve engineering trade-offs between different competing implementations.

Quality of service management seems to be the most promising approach to adaptive management. This involves both the specification of quality of service properties and the definition of end-to-end engineering support for managing qualities of service and monitoring services levels, for example to avoid overloaded servers. Communication filters have been used to support different quality of service properties like latency, bandwidth, throughput, error rates for multimedia traffic over broad-band networks.

Visual Scripting Tools and Languages

Visual tools and scripting languages are used to support visual composition and visual programming as described in section 4.2.2.

Visual scripting tools have been used extensively in windows development environments for many years. They are useful to associate windows events with object messages. For example, in Digitalks Parts a user can click on a source window and target window to connect an event in the source window with a message to the target window. Supported events and methods are listed in dialogues and can easily be selected. The links that have been set up in this way are shown on the screen so the programmer can get a better view of the behaviour. Methods and events names are shown as labels. NeXT's Interface Builder provides similar facilities to connect graphic objects to underlying application objects which have been programmed separately. It provides a graphical editor to define the static layouts of the interface using standard interface objects. The configuration is stored as a file and can then be built and executed. Other systems allow interactive editing and dynamic demonstrations of the interface response without requiring system builds.

Digitalk Parts and NeXt's Interface Builder provide examples of specific binding models used for interface building. Binding models can be provided for different programming tasks other than interface building. Furthermore there are a number of ways that a binding model can be supported in an object programming language. This can impact on the naming model and binding mechanisms:

- binding capability can be derived through inheritance from scriptable base classes. These base classes may support methods and event lists that the binding tools use to add events, connect events to methods and lookup the appropriate methods.

- a weaker variant is to derive the interface only from an abstract base class, forcing the class programmer to implement the capability.

- binding capability can be embedded in proxy objects to which messages are delegated. Proxies may be RPC stubs, meta-objects or communication agents that provide indirection.

- binding capability can be generated by an interface definition language processor or configuration programming language, see below.

One limitation of event based systems is that event based programming is extremely difficult and therefore inappropriate for general applications. This limits its practical use to areas like GUI development. Different binding models and visual scripting tools can be used more generally to enhance other programming styles, including synchonous method calls:

- Dataflow programming tools use a binding model based on the separation of control flows and data flows. DEC's real-time integrator (literature available from DEC) uses visual composition to build testing software for real time instrumentation. CONDOR (Kass, 1992) is a constraint-based data flow programming environment that has a graphical interface in which functions are represented as boxes that can be connected along vector or scalar inputs and outputs to compose functions

- Distributed systems may use a binding model that supports transparent distribution of the objects across a network. This masks problems of location and protocol from the component programmer. For example, CORBA IDL generates the binding support and the association service allows tools to keep a handle on bindings.

- Scripting tools and process description languages are currently widely used in workflow management and document management systems where the processing and synchronisation problems are simplified by a high degree of user control. A more complete survey of these tools is described in (Macintosh, 1994, section 7 & 8).

- Visual scripting is used in multimedia applications, for example ICL's IntelliPAD (literature available from ICL) to combine medias and TEMPO (Fiume et al., 1987) a temporal scripting tool for animation

- Visual scripting is used in image processing to interactively build networks of modules to do data input, filtering and reformatting, mapping to geometry and rendering, for example Advanced Visual Systems /Express.

More sophisticated binding models may be supported:

- ff the interface to an object not only defines the services provided but also the services required using an indirect name, it is possible to separate component implementations from bindings to other components. In this case the visual scripting language may act as a language for programming-in-the-large or a graphical configuration programming language (Kramer et al., 1989), removing all responsibility for binding from the components.

- integrated language processors like an interface definition language or configuration programming language provides a convenient vehicle to extend the programming language, for example IDL may specify properties and constraints that are used to select components and make engineering trade-offs between different protocols.

- an visual scripting tool may populate a repository that is accessed by other integrated development tools, such as browsers, configuration managers, traders and design or reuse tools. This facilitates the design of integrated CASE environments.

Configuration Programming System Design

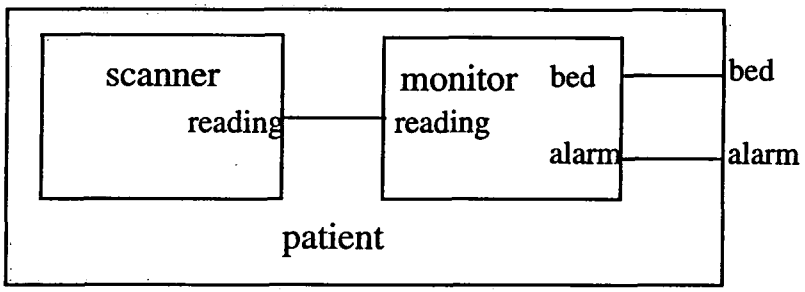
Configuration programming is defined in section 4.2.2. The key feature in configuration programming is to use a distinct language to define program structure as explicit bindings between sets of object components. Visual scripting languages may be used as graphical configuration languages. Other configuration languages are purely textual. Many configuration programming systems provide both textual and visual languages.

Configuration programming adheres to the following four main principles, described in (Kramer, 1990). These principles have been outlined briefly in section 4.2.2 at the design level. A code example, also taken from (Kramer, 1990), is used below to illustrate how these principles may be supported at the programming level:

1) Use a Distinct Declarative Configuration Language

We can abstract away from programming concerns by using a distinct configuration language for structural specification. A separation of structural description from basic component programming facilitates comprehension, interpretation and manipulation of the system in terms of its structure. In order to make such specifications more amenable to analysis, interpretation and manipulation, the configuration language should be declarative, describing what the structure is, not how it is constructed.

The example code in Figure 20 uses the CONIC configuration language to define a patient module as a composite of two components. It is assumed that the two components, the scanner and the monitor, are defined in the programming langage. They form part of a patient monitoring system that periodically reads sensors attached to a patient and sends alarm messages if readings exceed thresholds.




```
groupmodule patient ;
    use monmsg: bedtype, alarmstype ;
    use scanner , monitor ;
exitport alarm:alarmstype ;
entryport bed:signaltype reply bedtype ;
create
    scanner ;
    monitor ;
link
    scanner.reading to monitor.reading ;
    bed to monitor.request ;
    monitor.alarm to alarm ;
end.
```

Figure 20 Configuration script for a patient

The configuration language script is interpreted by a configuration manager which downloads code and instantiates processes. By being declarative, the actual order of interpretation operations is determined by the configuration manager.

2) Define Context Independent Types using Indirect Naming

Context independence means that the component makes no direct reference to any non-local entities. This requires that components access only local data and use locally named ports as indirect names for other components. Ports are bound to connected components transparently to the components themselves.

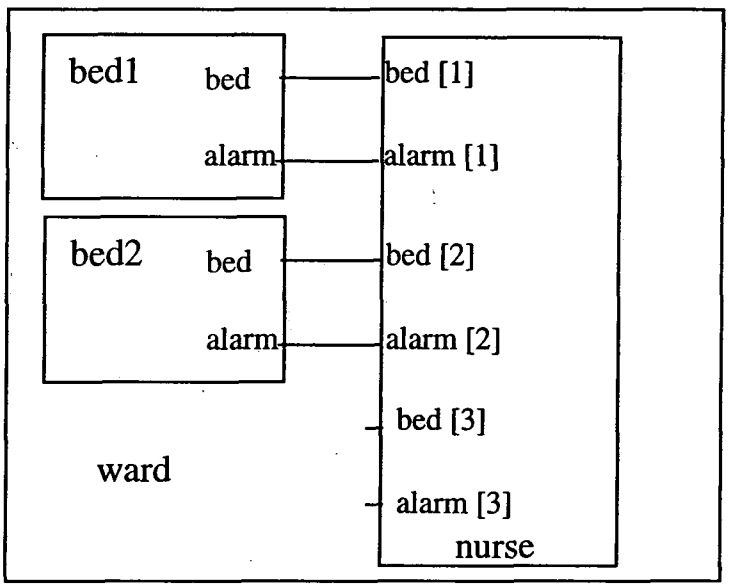
Defining components as types permits reuse by instantiation in different contexts. True context independence would imply that components can be integrated without any refinement or recompilation. Component types require well defined interfaces describing interaction points sufficiently to permit validation of interconnections.

In the above example, messages are sent via the *exitports* and received via the *entryports*. The type definitions are imported from the definition modules by the *use* clause. Internal ports may be linked together as in *reading*. Alternatively internal ports may be linked to external ports as in *bed* that are resolved at a higher level in the composition hierarchy.

3) Hierarchical Composition

The configuration language should permit complex component types to be defined as a composition of interconnected instances of more basic types. Such instance hierarchies explicate the instance structure, making interpretation and distribution clearer and simpler than with other systems based on type hierarchies. Hierarchical composition is an viable alternative to type extension mechanisms like inheritance or subtyping.

A ward consists of a nurse and a series of patients that are named by their bed number. The nurse component is assumed to have been defined as a group module in a similar way to the patient as above. Figure 21 shows the definition of the ward as a higher level composite involving components that are themselves composites i.e. patients and nurses.




```
system ward ;
    use patient, nurse ;
create
    bed1:patient at node1 ;
    bed2:patient at node2 ;
    nurse:nurse at node3;
link
    bed1.alarm to nurse.alarm[1] ;
    nurse.bed[1] to bed1.bed ;
    bed2.alarm to nurse.alarm[2] ;
    nurse.bed[2] to bed2.bed ;
end.
```

Figure 21 Configuration Script for Hierarchical Composite

The configuration language also describes the physical allocation of group module components across the network as is illustrated by the *at* clauses.

Components at the bottom of the hierarchy like monitor and scanner are sequential tasks, implemented in the programming language. In CONIC the programming language is Pascal extended with message passing. A runtime executive is allocated to a logical node which is identified as a group module like the patient and nurse. Although the indirection across links is removed at runtime and the structure is flattened, the logical node still acts as the unit of distribution, failure and change and links between logical nodes may incur the overhead of interprocess communication.

Rex (Kramer et al., 1992) is investigating another hierarchical structure called a domain which is used to group components to which a management policy is applied.

4) Programming Change

Changes should be expressed and managed. Configuration languages allow changes to be expressed as structural changes to the component instances and their interconnections. Capturing and managing change allows evolution and reuse. Changes can be planned or ad-hoc.

CONIC allows dynamic changes to a running system to be programmed as an incremental edit rather than the traditional approach of regenerating the build. For example a patient may be added to a ward by the following change script that may be invokable at runtime:

```
change newbed;
        create
                bed:patient at node2 ;
        link
                bed.alarm to nurse.alarm[n] ;
                nurse.bed[n] to bed.bed ;
end.
```

Figure 22 Change script for adding a patient

It is important to prevent interference between concurrent changes and ongoing interactions. Invocation of change scripts may be restricted to enclosing composites at that level which are responsible for serialising changes. For ad-hoc changes we need constraints to prevent interference. Application consistency may be preserved by change rules to derive boundaries for change transactions and provide the ordering and control of changes. Affected parts of the system will need to be identified and put into a quiescent state so as not to cause partial effects on interrupted activities. Changes that remove components must support finalization activities to tidy the environment. Changes that add components may need to support initialisation activities

Tools Requirements for Configuration System

In order to support a configuration system, a minimum set of tools must be provided: for off-line editing, compilation, analysis, verification, and for runtime support for constructing, reconfiguration, loading, execution and monitoring.

In a configuration system, objects, links, interaction entry and exit ports and composite aggregates are represented explicitly. These primitive components offer considerable advantages for representing various structural abstractions. In particular, tools could easily be provided to capture abstract frameworks for partitioning services between objects; to compose and decompose object designs as subsystems; and to explicitly define and dynamically manipulate the program structure. Requirement capture tools may also structure requirements.

There are two key approaches to implement these integrated tools:

- A single repository and representation may be shared by all the tools.
- Translators and transformation tools may be used to make different representations available to other tools.

Workflow Models

The term workflow comes out of the enterprise modelling community. Workflows model the route that any particular piece of work takes to ensure it reaches each of the people who must process it in an appropriate order. To make good use of a workflow the process and structure of an organisation must be understood. Most workflow products provide a process description language PDL and scripting tools for this purpose.

Existing work flow tools have limited synchronisation and tool integration capabilities. They are usually built on top of e-mail, provide simple client server APIs or are combined with Document Image processing DIP systems. This makes them most appropriate to people-centric tasks such as a document review and approval. Typically workflow modelling tools manage work routing, queuing and alarming to the relevant people. Further information on products can be found in (Macintosh, 1994, chapter 8; Marshak, 1993 ; Marshak, 1994).

Workflow modelling potentially has more general applicability for automation of complex processing tasks. Automated workflows could route work to objects that performed the processing. This would require a mapping between workflow and actions or transactions at a design level. For example, workflows may become data flows and control flows in the design. Workflow formalisms might also be useful in behavioural modelling and in capturing non-functional requirements such as dependabilities, perhaps even driving engineering trade-offs between different architectural components. Unfortunately this level of integrated support and automation is not provided by existing workflow products that focus on user-level processing rather than complex data processing.

5.3 Summary of Chapter

This chapter reviewed some of the techniques and design principles that can be used to define a distributed programming system architecture. The term programming system is used to signify both programming language and operating system. A programming system architecture is concerned with the relationships between the runtime support and the programming interfaces and APIs.

Distributed systems are structured by processes at the most basic level, and process communication is important. Portable systems must ensure the system behaves consistently over changes to the operating or device environment. Interface design and abstraction layers are important to isolate applications from backend volatility and encoding biases. Open integration systems must ensure the substitutability and interworkability of components. It is therefore important to provide a high level rationalisation of the overall design of the integrated system and for the glue that binds components together. These important areas of process communication; layering and interfacing; and structuring and binding were discussed in detail in three sections:

The first section described a number of different communication mechanisms for components on different hard processes or on different nodes in the network.This includes unstructured data streams, unidirectional messaging, network programming channels, remote procedure call, remote method invocation, indirect interaction, and triggered interaction.

The second section described some techniques that can be used to organise the interfaces between layers in the architecture or to define the relationships between runtime support and higher level programming abstractions. This included pre-processors and generation, system APIs, system frameworks, proxies, meta-object protocols and reflective architectures, transparencies, and viewpoints or projections.

The last section described some design approaches that can be applied by programming systems to analyse and define the structure of a distributed program. It began with mechanisms that operate on flat structures of components where the overall structure is implicit in the sum of the relationships between individual components. This includes narrowing and trading, interface specification and adaptive management. It then considered approaches that deal with global structure explicitly.like visual scripting , configuration programming, workflow modelling.

The next chapter summarises the survey of techniques and concepts and defines an evaluation framework to be used to position and evaluate the OpenBase architecture.

Chapter 6 Analytic Framework

During the survey of distributed objects, it has become clear that distributed systems techniques and products do not relate to each other as parts of a total system. There is not yet any single grand architecture, nor is it likely that there will ever be one. Rather there are a number of styles of distributed system that use quite distinct techniques and architectures. The scope required of a general purpose analytic approach to support all these styles is too large to be unified by a single formalism. Likewise defining a general-purpose architecture design as a set of interrelated tools would have limited usefulness to different styles of system. Consequently it is difficult to rationalise a choice of style and a choice of architecture for integration across the enterprise.

The only way to tackle this dichotomy is to present a separation of concerns. A set of related frameworks can be defined to portray the architecture from a number of distinct abstract perspectives. Such a separation of concerns facilitates the orderly comparison of different solutions at different levels of abstraction. General purpose frameworks to support development of distributed systems include the ANSA frameworks (ANSA, 1993) and Zachman frameworks (Zachman, 1987). This section provides a simpler set of frameworks based loosely on the approach of Blair (Blair et al., 1991).

6.1 Overview

Distributed object systems differ in many ways. It is possible to compare them in a number of different perspectives. One simple way to capture differences, is to explicitly define key variables in each perspective.

The first part of this section introduces a framework of four significant perspectives and defines key variables in these perspectives. This summarises the broad survey of distributed systems in a series of classification hierarchies whose elements are the instantiations of the variables. This defines the problem space using the elements in Figure 23.

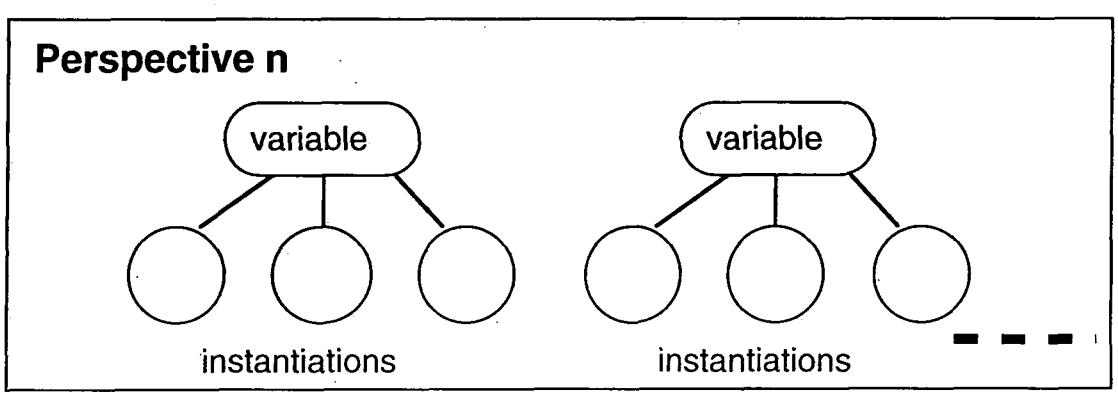


Figure 23 elements defining the problem space

The next subsection takes each perspective in turn and defines the feature that is most important to optimise to facilitate advanced enterprise wide computing. It relates each feature to the key problem or impediment that needs to be overcome in order to achieve the optimum solution. The impediments are used to position four significant new design principles for enterprise-wide computing. This derives an abstract solution space using the elements shown in Figure 24.

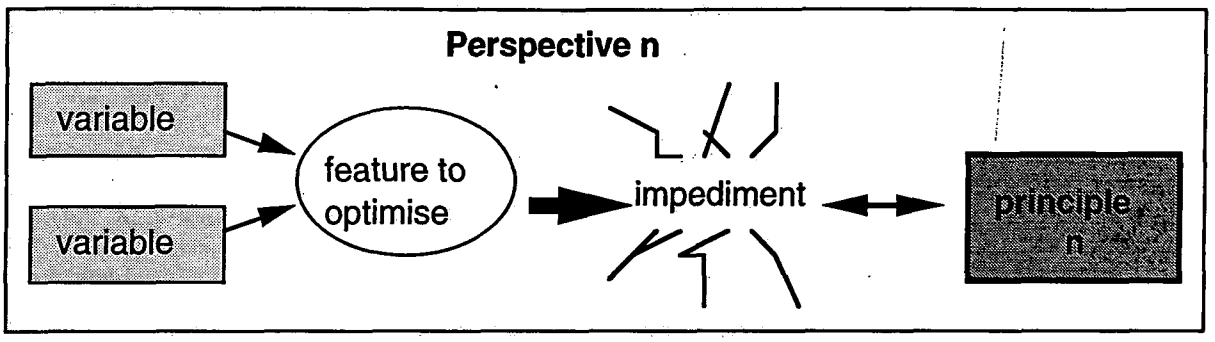


Figure 24 elements used to derive the abstract solution space

The last section merges these new principles with Blair's principles and perspectives of object technology, defined in chapter 2. The combined result is summarised in a unified classification hierarchy of design principles. This hierarchy is related back to the goal framework of chapter 3 to define a generalised design space for enterprise objects, as shown in Figure 25. This framework maps goals to principles to specific instantiations of the design variables and thus captures an abstract rationale for making decisions between otherwise disjoint low level solutions.

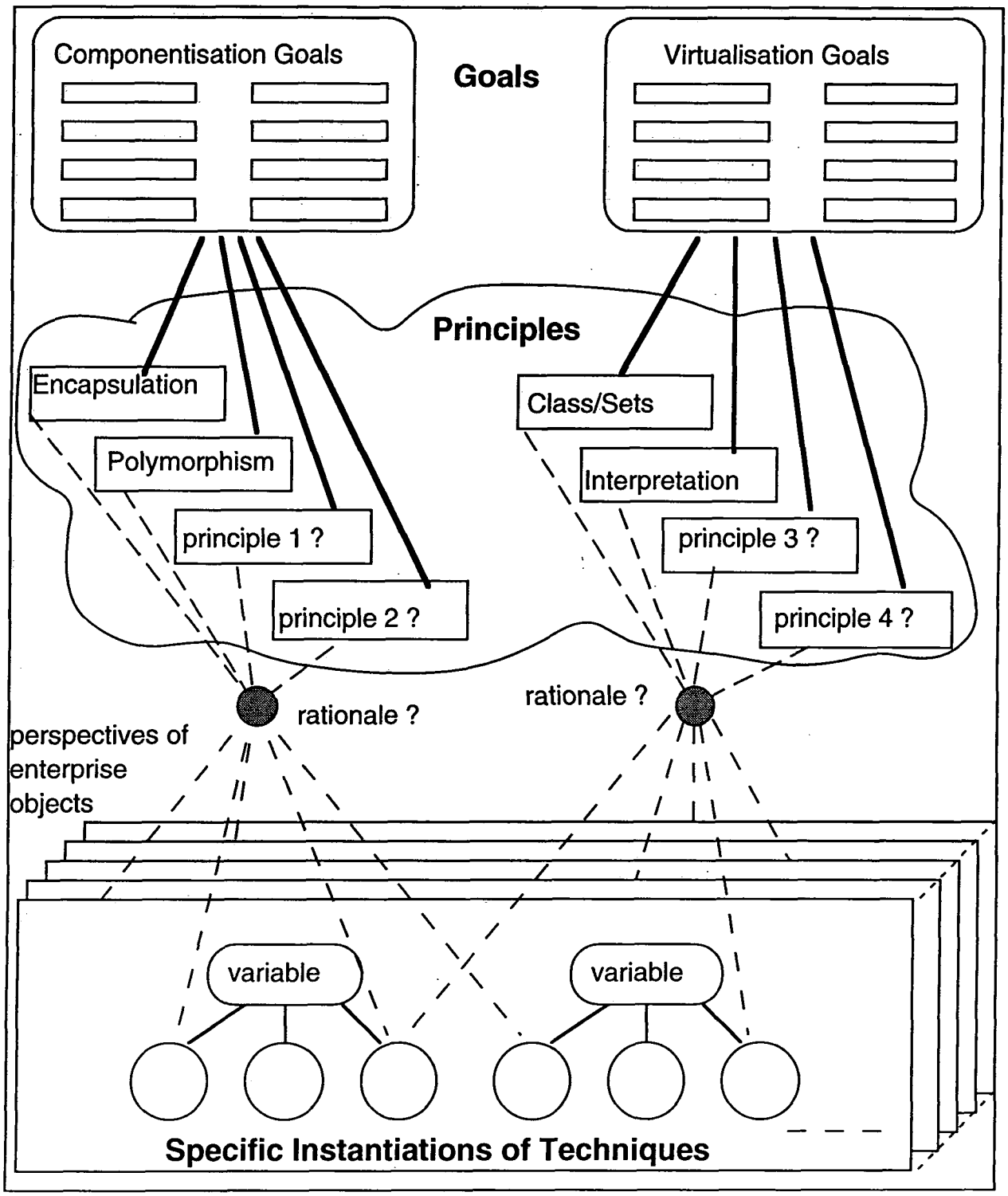


Figure 25 generalised design space for enterprise objects

This unified design space provides an analytic framework for rationalising design decisions in OpenBase and is used as an evaluation framework in the rest of the thesis. By providing an abstract structure for the design space, it is also found useful to structure an overview of the architecture.

6.1.1 The importance of an evaluation framework

An evaluation framework is important for the following reasons :

- A major problem in understanding object oriented and distributed system technologies is to bridge the considerable gap between abstract principles and concrete techniques and mechanisms. It is difficult to relate techniques back to principles. Because of this difficulty, techniques tend to be evaluated and compared in an ad-hoc manner.

- Many of the business requirements for an enterprise system are non-functional, for example the requirement for openness or reusability. It is also difficult to relate these ambiguous requirements to specific principles and approaches. Consequently relevance and traceability to requirements is difficult to establish.

- As IT is becoming more commoditised, the rate at which commercial-off-the-shelf products (COTS products) are appearing is increasing exponentially. This invalidates conventional specification-driven approaches. The emphasis must shift from bespoke development to COTS selection and from specification to COTS integration. Conventional frameworks of analytic processes must give way to frameworks that support selection and technology integration.

- Distributed object technology is too often defined by specific low level programming mechanisms like inheritance and RPC. This leads to narrow-minded definitions of solutions. A fundamental reevaluation of the underlying principles and goals is required to regain a merit-based, expansive vision that doesn't exclude viable alternatives.

- These mechanisms represent disjoint technology choices yet they have overlapping goals, for example messaging and remote procedure call both provide high level interaction across a network. The disjoint yet overlapping nature makes it particularly important to be able to rationalise choices.

Consequently something is needed to help elucidate the essential fundamental design principles and to map requirements to these abstract principles and from principles to specific techniques, to capture design rationale and support both traceability and orderly selections of COTS.

6.2 The Problem Space

Perspectives on Distributed System Architecture

The following perspectives have been defined based on the survey of distributed systems:

- the ***motivational*** perspective, covering the scope and intent of the system. This characterises the system according to typical profiles of technical goals, as defined in chapter 2 and 3.

- the ***constituency*** perspective, covering the physical environment, i.e. the network architecture: it's technology, scale and proprietaryness, and the audience of the system: its various users, developers and owners. This characterises a system according to it's inherent physical, application domain and cultural diversity, as discussed briefly for OpenBase in chapter 1.

- the ***capability*** perspective, covering instrumentation and utility. This perspective characterises a distributed programming system according to the efficacy of it's coordination/communication mechanisms, and the potency of it's programming interfaces, as described in chapters 4 and 5.

- the ***cognitive*** perspective, covering the cognitive processes of system development and the concepts around which software systems are organised. This characterises the approach to specification and the partitioning concept used as a building block for large scale systems, as described in chapter 4.

6.2.1 Motivational Perspective

The first perspective to consider with respect to positioning third party technologies and tools is the motivational perspective i.e. the business needs and goals that the tools are geared to satisfy. This has already been discussed in more detail in Chapter 3. We consider three profiles of goals here that summarise the ways goals are typically mixed.

- workstation front-ending - this is essentially concerned with adding a remote presentation layer to conventional mainframe applications that exploits the power and user friendliness of workstations. As such it is primarily concerned with GUI aspects of abstraction and evolution goals, as well as platform commoditisation, and application reengineering goals.

- distributed application - some distributed applications seek to exploit the extra potential that a distributed solution offers to meet performance and dependability goals, such as fault tolerant applications.

- enterprise model - enterprise modelling support seeks to establish an open distributed computing backbone to an entire organisation as the basis for interworking and sharing of results. In addition to interworking and sharing, such systems may also emphasise a range of technical goals: federation, application re-engineering, group-working, modelling and large-scale reuse of business models.

The breakdown of this perspective into instantiations of key variables is shown in Figure 26.

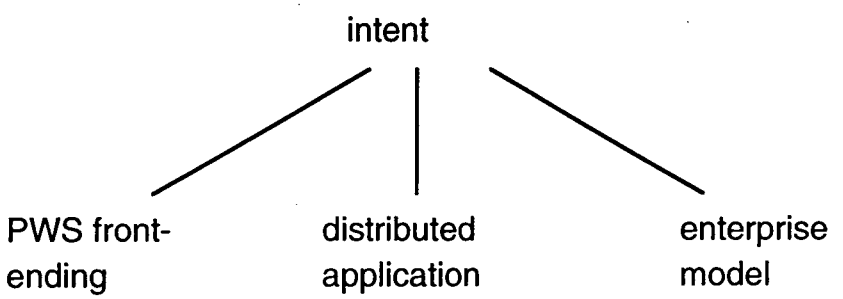


Figure 26 design variables for motivational perspective

Distributed development tools can be positioned according to these profiles, despite deceptive marketing literature. Failure to take into account the full scope of goals, limits the applicability of products that are really targeted at the first two strategies. Yet products suitable for enterprise integration are often immature, partial solutions from small vendors.

The current generation of programmable workstation front-ending tools includes products like Easel and Powerbuilder. With these products it is quick and easy to build a user interface. However many tools in this category have grown in a world of outmoded form-based manual business processes and merely allow the designer to blend existing application logic with better presentation facilities. Their poor business modelling capability results in poor flexibility and maintainability, especially when tools embed proprietary scripts in a user interface screen or hardwire data items to data values in a relational database. Their scaleability is often poor, with limited support for multiple data sources, distributed databases and high transaction volumes.

Distributed application development tools generally build on specific distributed programming mechanisms such as distributed transaction monitors like Encina and Tuxedo and reliable multicast protocols like ISIS. Whilst these mechanisms facilitate fault tolerance, high availability, data integrity, parallelism etc., they are often limited in scope and too proprietary for general-purpose use since they are based on specific technologies.

Distributed enterprise modelling tools support a broader range of technologies. This includes new areas of application like business process re-engineering and co-operative working. They offer improved commercial viability and flexibility through open systems, downsizing, distributed objects etc. This category includes products supporting integrative standards like CORBA and DCE.

6.2.2 Constituency Perspective

The second perspective in which to position technologies and approaches is according to the constraints imposed by the target environment. This is important to any infrastructure development. Two particular aspects of this perspective are considered: the physical diversity of the computing network and the development cultures that surround them. This perspective must consider legacy system and learning curve factors.

Protocol diversity

We can classify environments crudely according to the degree of physical diversity:

- proprietary environments, based on a single platform type.
- homogeneous network operating system, where a single type of network operating system integrates clients and servers on different machines, for example OS/2 LAN server or novell.
- heterogeneous network, with multiple operating systems integrated using open systems technology under a single administration.
- federated heterogeneous network, spanning different technology and administrative domains that may use different protocols.

The diversity of a distributed system in terms of platforms, protocols, administrative domains is an important constraint. Unification and integration requires higher levels of infrastructure abstraction to map across representation/encoding biases. Capacities, timescales and resource management issues may also affect design.

Objects support both infrastructure abstraction and resource management:

- Objects communicate using an unified interaction model that hides diversity of mechanism. They also lend themselves to declarative interface definition mechanisms that can define unified encodings across a heterogeneous system.

- Resource management can be integrated with object management protocols. Objects not only encapsulate functionality and data but they also encapsulate what can be managed by the system. Objects can become the unit of storage, the unit of mobility, the unit of replication, and the unit of failure. Strong resource constraints favour finer grained object management architectures to minimise redundancy.

Developer cultures

There are three dominant developer cultures:

- PC or Mac, the desktop world is a world of high volume general purpose products, traditionally personal productivity tools like word processors, spreadsheets. The growth of networked desktops has resulted in an explosion of new styles of application including file servers, mail and remote data acquisition. This culture has established their own de facto standards like OLE, ODBC.

- Database, the impact of relational database technology and 4GL tools has led to a specialisation in skills between database experts providing corporate information services and other programmers writing bespoke applications. Database applications support operational and decision making activities of a business, including enterprise database integration. This culture have established their own standards like SQL.

- Control, the control programmers culture is itself divided, mainly between the mainframe world of COBOL, ABACUS etc. and the UNIX world. Application styles range from transaction monitors to complex technical applications like defence and telecommunications. This culture have also defined their own standards like OSF/DCE.

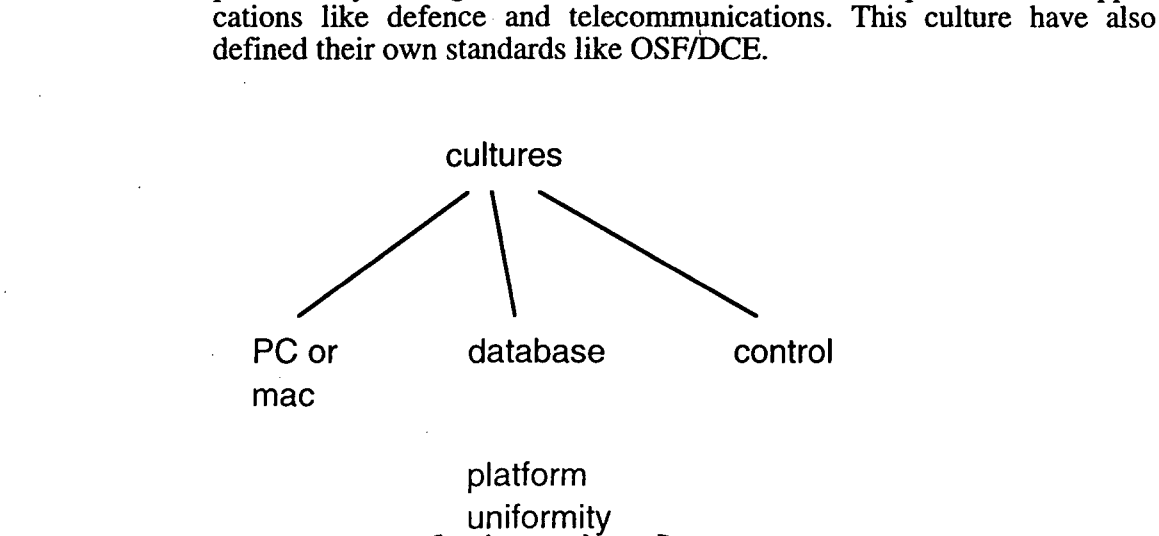


Figure 27 key variables in the constituency perspectives

6.2.3 Capability Perspective

An important perspective influencing the suitability of technologies is the capability required of the programming system. This includes the efficacy of the mechanisms used to coordinate components and the potency of the programming interface.

The type of facility required by applications is important in determining both of these requirements. For example, systems that are merely concerned with disseminating information are less complex than those that require information to be updated consistently and effectively. Their simplicity means they can use simple coordination schemes and less sophisticated management engines.

Design variables are chosen to reflect different types of facility. This includes :

- the level of coordination between components
- the degree of abstraction in the programming interface.

The level of coordination

The level of coordination typically increases with both the complexity and the predictability of the application.

Application complexity affects the need for structure. Less complex applications do not require high levels of coordination between components. The simplest schemes share unstructured data such as documents. As the information becomes more complex, structured data becomes important. Likewise as the processing becomes more complex, control structures become more important.

Unpredictability limits the usefulness of rich static typing systems. Synchronous, prefabricated RPC-based interfaces provide rich semantics. However it is difficult to maintain such rich semantics for an unpredictable interaction, such as an ad-hoc query on a database entity. Flexibility can be built into a system by developing more sophisticated programming features such as polymorphism, schema evolution, dynamic requests, concurrency. A simpler solution is to adopt weaker forms of coordination, such as untyped asynchronous messages or interpreted queries.

Asynchronous event driven architectures with multitasking user interfaces effectively take control away from the programmer and give it to the user. Interactive ad-hoc queries on a relational database do likewise.

The following instantiations of types of facility are intended to reflect increasing levels of coordination both structurally and semantically:

- information dissemination - unstructured serial data, for example mail, text retrieval systems.
- document processing - This includes linked documents and group working systems that support workflows.
- data processing - This includes transactional systems with simple flows of control between users, the application and the database server to perform query requests.
- data distribution - This includes replicated objects, distributed databases or batch transfer programs that download data.

- distributed presentation - This includes event-driven windows GUIs.
- distributed function - Using functional abstraction with remote procedure call structures and typed arguments. This includes applications where the functionality is separated to presentation layers, logic layers & data management layers that are allocated accordingly.
- distributed objects (business objects) - uses semantically rich class and object structures with method invocation or messaging.

These categories map cleanly onto the technology choices of section 4.2.1, also shown in Figure 28, including:

- distributed file system - For example NFS.
- document architectures - For example OpenDoc and OLE.
- remote data acquisition - This typically involves querying using a vendor neutral language, for example SQL. Relational queries support interactive ad-hoc access and application specific views. They may also support procedural SQL using stored procedures, precompiled SQL, and SQL looping and branching constructs.
- distributed database - This includes distributed relational and distributed OODBMS. Distributed relational databases are an extension of RDA to support distributed transaction processing using transaction monitors and 2 phase commit protocols.
- events - This includes X-windows events. Event driven architectures place control in the hands of the user or the system. This facilitates scheduling of activities to exploit the inherent parallelism of the system more fully.
- store & forward messaging and queuing - this includes e-mail, asynchronous messaging and publish-subscribe interaction protocols. Abstract semantics can be defined as standard protocols using formatted messages.
- remote evaluation - this includes telescript, where a message is interpreted and executed at the server. This allows the server to exhibit ad-hoc behaviour that is encoded in some language as a script in the message.
- remote procedure call/ remote method invocation (RPC/RMI) - This includes DCE or CORBA. RPC provides more abstract semantics that any other type of communication using rich abstract data types. This can give better semantic integrity through type specific constraints, more elaborate distributions of functionality and data, thus avoiding bottlenecks, and easier federation using federated object managers.

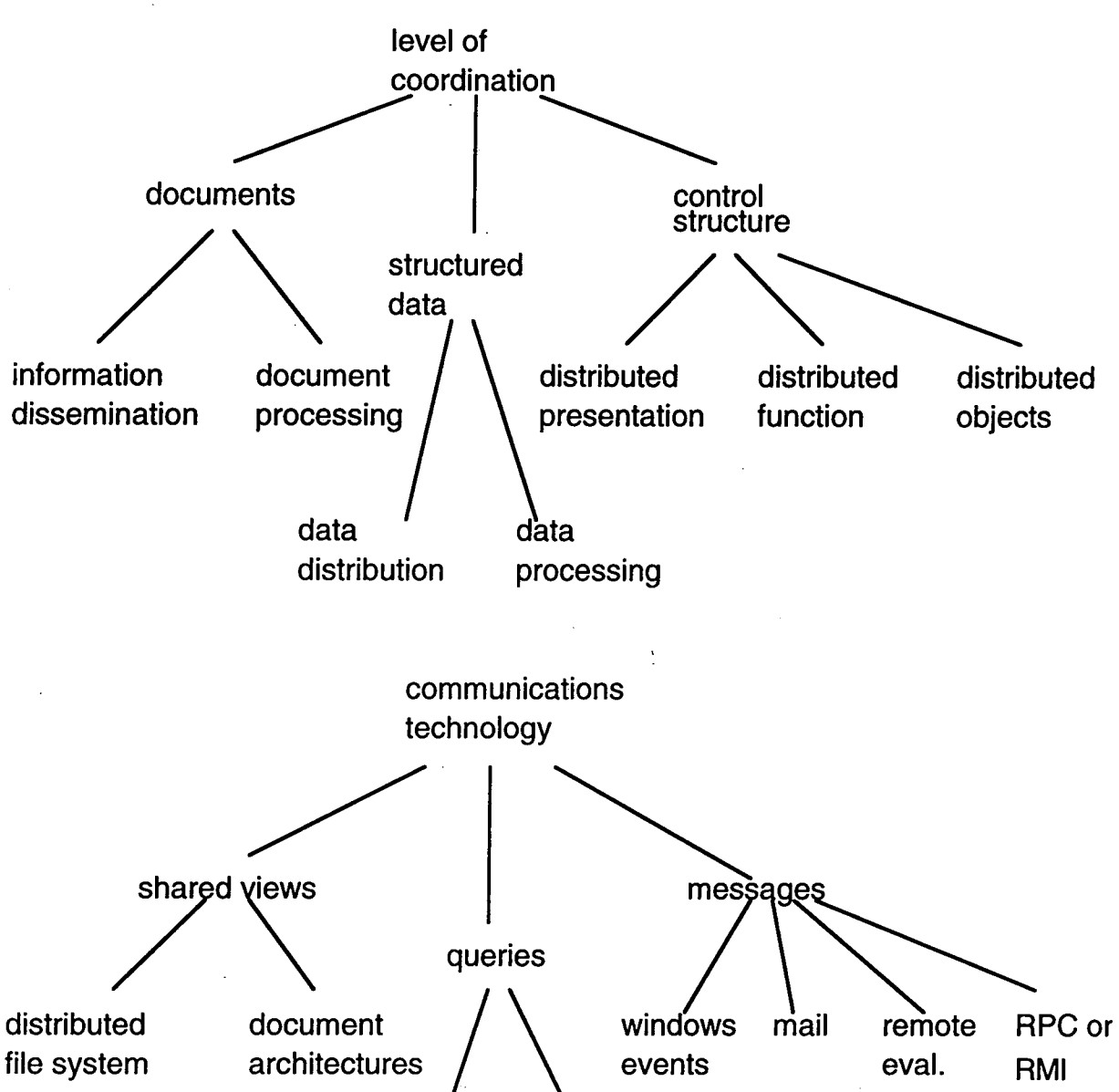


Figure 28 levels of coordination and corresponding technologies

Level of programming interface abstraction

The level of abstraction in the programming interface to the system depends on the nature of the interface and breaks down into the interface types defined in section 5.2.2, as follows:

- orthogonal low level system libraries - a programming interface consisting of a system API as for conventional operating systems. This represents a low level DIY (do-it-yourself) approach. Example: UNIX sockets (Stevens, 1990).

- orthogonal system class library - a programming interface defined as system classes that can be called as for an API or inherited. This represents a high level DIY approach. Example: C++ class libraries for POSIX.
- reflective customisation - a meta-level programming interface is provided by meta-objects that define or are defined by the computational model of the programming language. This represents a meta-level DIY approach. Example: proxy classes. and meta-classes.
- distributed programming model - tying together object management, interaction management and resource management in the computational model itself. This a generative approach. Example: CORBA IDL.
- viewpoints - as for the programming model, but where there are several distinct but related object models i.e. views of the objects. These integrated views deal with management concerns at different levels of abstraction. This is a generative/transformational approach. Example: ANSA projections (ANSA, 1993) or ISO ODP viewpoints.

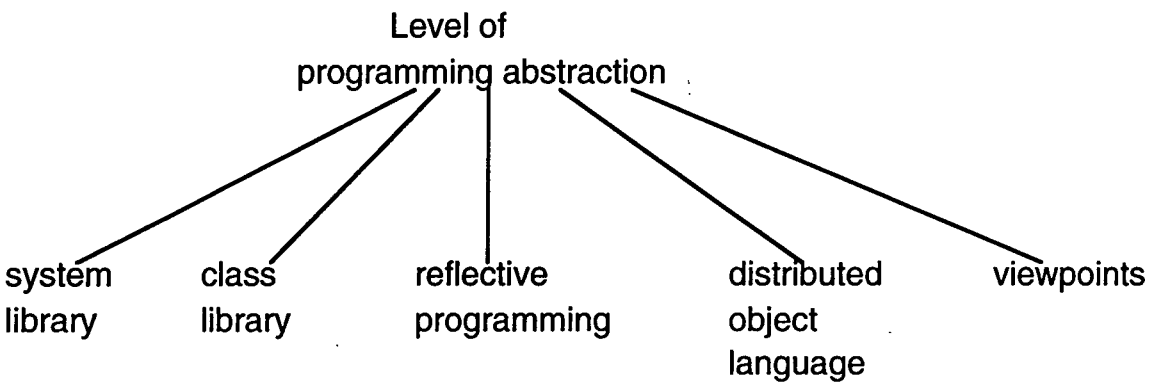


Figure 29 level of programming system abstraction

6.2.4 Cognitive Perspective

A development method may be characterised by the cognitive processes employed to develop a system, for example processes of analysis and interpretation , synthesis and insight, model generation , hypothesising and making implementation decisions. These cognitive processes in combination constitute the development process. The development process may be articulated as a methodology. This framework is about the method and concepts used to develop a system.

There are many approaches to building distributed systems. These vary in the granularity and type of organisational concepts used as a building block for the system and in the way the system is specified and verified.

- The organisational concept varies between methods that focus on functional subsystems, on processes, on applications, or on fine-grained objects.
- The specification approach varies according to whether the approach is specification-driven or composition-driven, in the relative emphasise placed on functional decomposition and composition techniques, and whether it is iterative or generated by transformation.

The breakdown of this perspective into instantiations of key variables is shown in Figure 30.

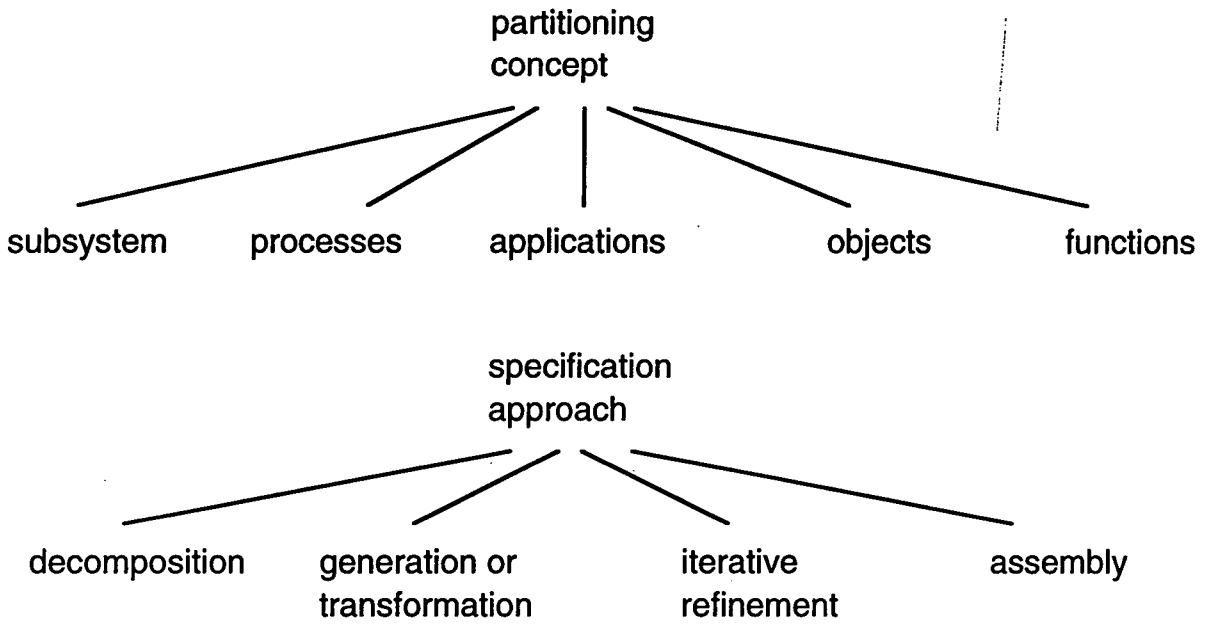


Figure 30 variables in the cognitive perspective

6.3 The Solution Space

Definition of Key Features, Impediments and Design Principles

6.3.1 Motivational perspective

The key feature that is emphasised in an enterprise wide system is the degree of sharing and reuse across the enterprise. The scope of the system is no longer limited to a single application on a single computing platform. Component reuse right across the enterprise demands a greater investment in the components. As we adopt higher degrees of distribution we need to move away from monolithic proprietary world into a well defined world of standard plug and play components.

Conventional approaches to software development, such as structured methods, tend to focus on the whole solution. The finer parts are then defined in the context of the larger parts, for example by hierarchical decomposition. This necessarily results in more contextual dependencies between the parts and the whole which limits their reusability. The overall solution is not reusable only components of the solution.

Likewise development that is targeted at a specific platform generally results in proprietary solutions that can not be ported across the enterprise. System dependencies also limit the reusability of the parts.

The key impediment in this perspective is therefore the contextual and system dependencies. Source code introduces a high degree of specificity that makes large scale reuse unlikely. The general principle that characterises the solution space is the principle of component insulation both insulation from contextual dependencies between components and insulation from system dependencies. This principle generalises those techniques that achieve contextual independence like indirect naming, runtime type exploration, partial specification, transformational generators, abstraction layers, virtual interfaces and object wrappers. This is shown in Figure 31.

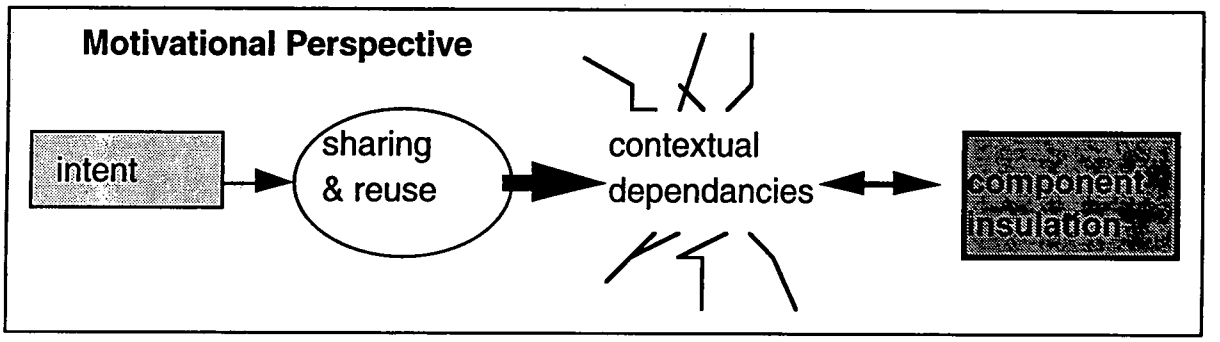


Figure 31 solution space in motivational perspective

The basic problem is one of context isolation, context representation and interpretation. There are two basic approaches: generation based and composition based.

- Specification-driven generation techniques build contextual independence into the tools and transformations themselves allowing tools to be reused to interpret specifications and generate application in different contexts.
- Composition allows contextually independent components to be configured in different patterns in different contexts. Contextually independent components have limited knowledge of bindings to other components. This is facilitated by indirect naming and various types of polymorphism such as subtyping or generics (representations with parameterised contexts).

6.3.2 Constituency perspective

The key feature in this perspective is the diversity of the system at three levels:

- different types of user or developer
- the different types of application
- different types of technology.

Whilst it is desirable to support diversity to match the style of the infrastructure to the diverse local needs, an enterprise system needs to provide some unified control to facilitate overall coordination and seamless integration.

The main problem is achieving a balance between unification and diversification across these levels. The dichotomy between the two can be broken through the use of standardised layered protocols.

Protocol standardisation makes the scope of the diversity explicit and manageable. Protocol layering allows the diversity in one layer or level to be unified in another layer. Traditional communications stacks such as the OSI stack, support diversity in lower layers whilst unifying the upper layers. This principle may also be applied at the application level. For example, application engineers need a unified protocol for plugging components together. A high level of plug compatibility is unlikely unless the components themselves conform to protocols governing their interfaces. This does not stop these components from having diverse implementations and behaviour or using diverse mechanisms internally.

Protocol should govern the application interfaces (for pluggability), the system interfaces (for portability) and the management interfaces (for configurability and administration).

The granularity at which we require protocol is rapidly getting smaller as the IT world evolves from the vertical proprietary world to the horizontal open systems world to the grid-like world that is the real commodity world. We can not expect a single open system standard to permeate the entire enterprise for all time, hence we need to evolve and adapt standards both horizontally and vertically. This is illustrated in Figure 32. Standard protocols are required at every node in the grid. Given this granularity, is not surprising that standards are adopting standard object models at all levels in the architecture.

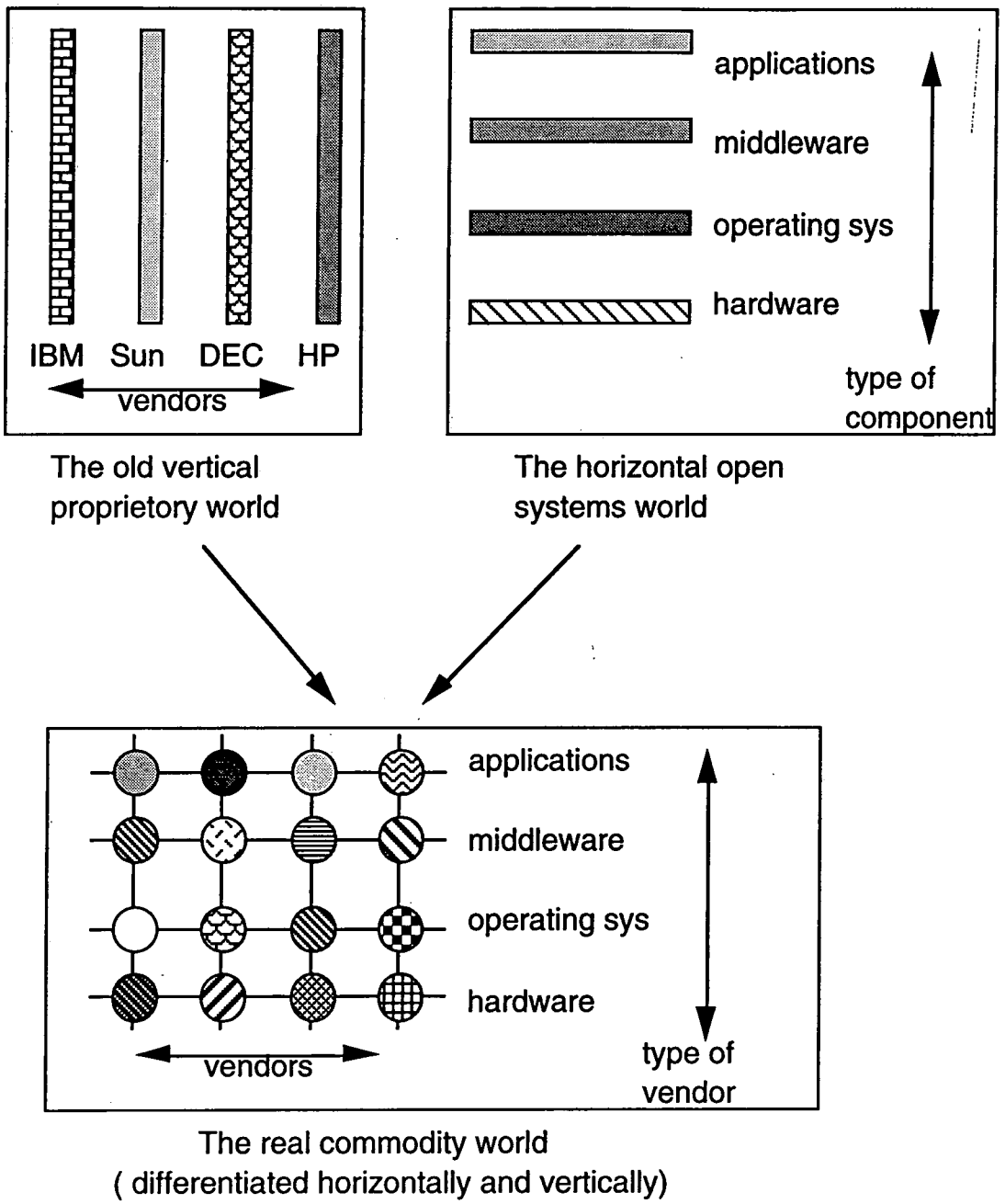


Figure 32 the evolution of the commodity world

The principle of standard protocols generalises a number of different techniques that can be used to specify or restrict development. This includes communications standards, operating system standards, object frameworks, inheritance mixins, APIs, message formats, interface definition languages, specification languages, visual binding models.

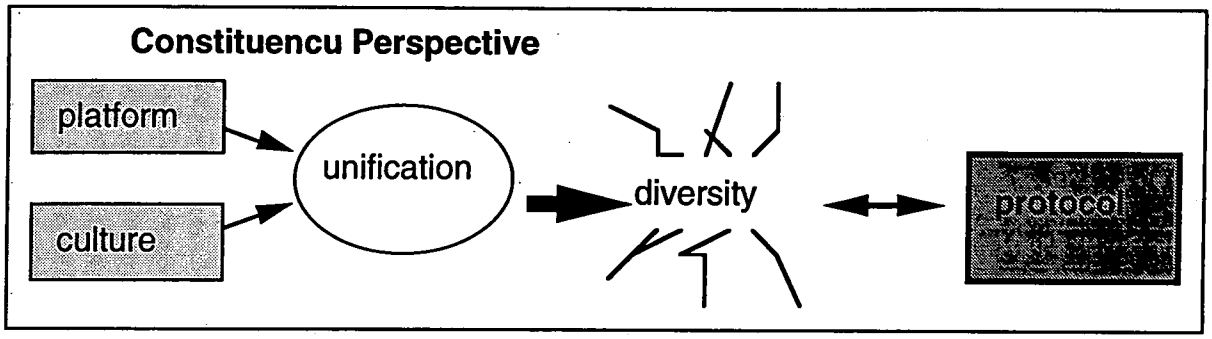


Figure 33 the solution space in the constituency perspective

6.3.3 Capability perspective

The key design variables for infrastructure development introduced in this perspective are the level of coordination and the level of programming system abstraction. The level of coordination and the level of abstraction are both concerned with the power of the programming system. This is particularly important in a distributed enterprise to avoid encumbering the programmer with too much complexity.

Distribution brings new complexities such as extra failure modes, indeterminent communication, concurrency, insecure communication, costly communication. Enterprise systems also bring the added complexity of heterogeneous federated systems and the need to map across representational/encoding biases. It is important to mask as much of this complexity as possible, without compromising on efficiency.

The key feature to optimise is the expressivity of the programming system to allow the programmer to efficiently coordinate components so that the nett system behaviour achieves the required effect. Two interacting components may constructively cooperate. Two non-interacting components may destructively interfere with each other. Coordination is essential and is facilitated by high level abstractions: for managing interactions, i.e. control and data flows; for managing resources, i.e. scheduling and distributing components; and for composing behaviours across components. The ultimate programming goal is to support both declarative specification, where the programmer describes the required system properties without needing to consider actual mechanisms, and compositionality, where a composite component derives the properties of its components.

The main impediment in providing more powerful interfaces is that of efficiency. Complexity can be masked by high level abstractions and sophisticated automated management mechanisms. However it is difficult to do this efficiently. Abstraction usually introduces an extra overhead in the runtime support. This is a problem that has plagued 4GLs. Too many management overheads can cause serious deterioration of system performance.

There are usually trade-offs to be made in selecting management mechanisms. Therefore, an essential capability is to have control over the management mechanisms. Different parts of the system will have different dependability requirements and require different trade-offs.

The principle that best sums up solutions to these impediments is the principle of providing ***selective properties***, where declarative properties can be selectively specified at two levels:

- a component can selectively choose the system properties that it requires
- a composite component can selectively derive the properties of its components.

These properties range from low-level service properties of the communication system such as security properties of the messaging system to high level dependability properties such as fault-tolerance, timeliness, availability.

The principle of selective property specification generalises three types of technique:

- techniques that provide declarative properties, such as viewpoints, selective transparencies and specification languages,
- techniques that facilitate derivation of abstract properties, such as service classification taxonomies, delegation and cluster managers,
- techniques that support customisation, such as system frameworks, meta-object protocols, protocol negotiation, policy parameterisation, stub/proxy subtyping.

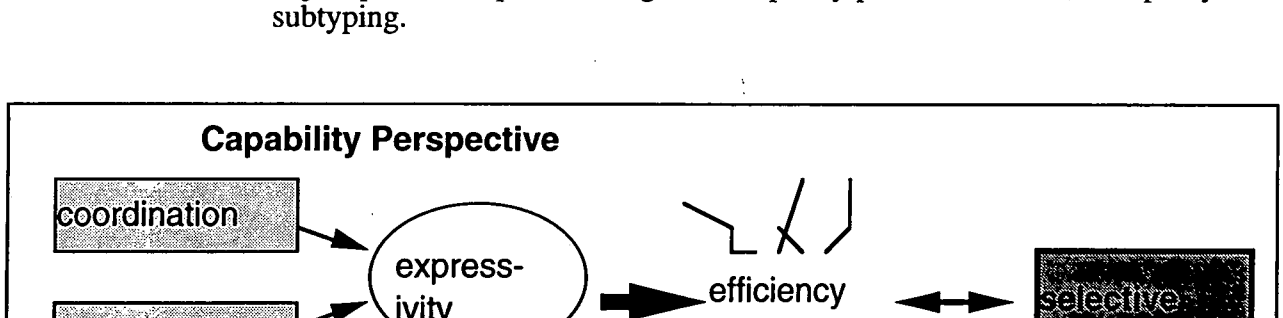


Figure 34 solution space in capability perspective

6.3.4 Cognitive perspective

An enterprise-wide system must allow the seamless integration of software components across an enterprise and maintain the flexibility to evolve the system. Conventional process-centric software development approaches are not suitable because they are geared towards specific problems. This clear project-centric focus leads to bespoke point solutions that are difficult to integrate. There is little investment in finding general-purpose solutions that cross application boundaries. Consequently these point solutions do not fit together easily nor evolve easily with changing problems. What is required is a product-centric approach that focuses on packaging generally useful solutions for reuse across problems.

The key feature in this perspective is therefore the degree of commoditisation i.e. product-orientation as opposed to process-orientation, of the development approach. Necessarily a product must be defined clearly. Commoditisation is the market result of the customer being more concerned with the service or function (the what) than the implementation (the how). Commoditisation arises when competition moves to the quality and cost of implementation and away from defining the characteristics of the product itself. A reuse strategy is flawed if the effort to understand a product is equal to the effort to produce a new one from scratch.

In order to allow programmers to organise themselves as producers and consumers of each others product, we must increase the tangibility of the software engineering processes and products. At the one extreme, hierarchical decomposition leads to very specialised software components and uses a complex process of transformation and verification. At the other extreme, systems are emerging that use intuitive visual tools to assemble prefabricated components using visual iconified representation. The key impediment is dealing with the inherent intangibility of software. Terms like impedance are frequently used to signal the conceptual gaps that exist between software models at different levels of abstraction in a transformational system. A product centric system must remove this cognitive impedance.

Product oriented approaches typically place more capital up front in production tools that facilitate the development, packaging and exploitation of product - to reduce latter volume usage expenditures. Capital-intensiveness is related to the degree to which solution-construction activities need to be amplified and simplified.

The principle that best captures the need to make software development more tangible and more productive is that of substantiation. We can define this as the principle of giving software more precise form or substance. This generalises techniques like visualisation, visual scripting, iconisation, metaphor-based programming, modelling of real-world objects and workflows and formal specification.

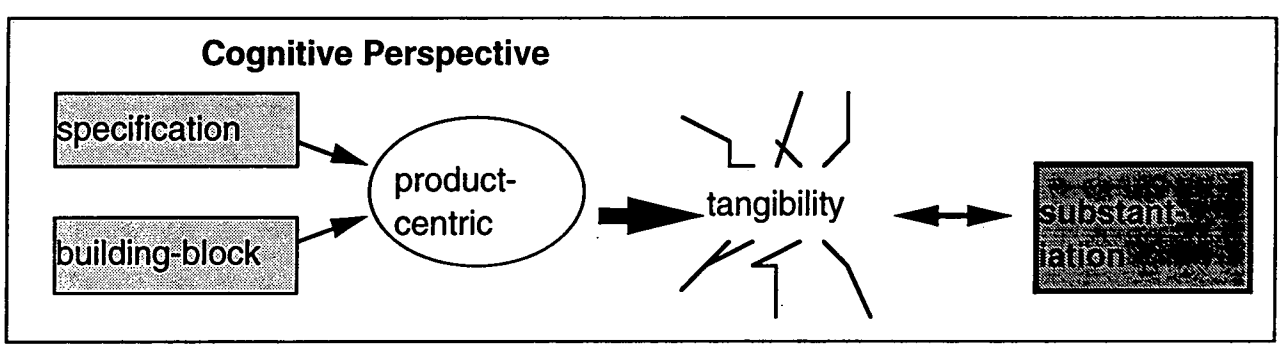


Figure 35 solution space for cognitive perspective.

6.4 The Design Space

A unified framework for distributed object technology

This section provides a summary of the analytic framework that can be used to relate goals to general principles of architecture design to specific techniques and implementation decisions. It shows how a problem space can be mapped to a design space by examining the underlying principles being applied and relating them to the key problems.

In chapter 2 we presented a simple framework for evaluating object technology principles This consisted of the four principles:

- encapsulation, the binding together of different computational aspects such as data and processing, with control over access to data through well-defined, operational interfaces.

- classification/set, some form of set abstraction that allows reasoning about behaviour based on inclusion in a group of related objects.

- polymorphism, compositional flexibility of a language, facilitating substitutions by allowing components to express general assumptions about related components.

- interpretation, the ability to efficiently and safely resolve the right interpretation of an abstract item of behaviour.

In the last subsection, we presented a simple framework for evaluating distributed and reuse technology principles. This consisted of the four principles:

- selective properties, the capability to select the behavioural properties of components, composites of components and the runtime support of the programming environment without reworking low level mechanisms.

- universal protocol, an agreement to conform to a finite set of interfaces and composition methods at each level in the software architecture.

- substantiation, giving precise form or substance to software processes & products, typically using mathematical formalisms, simulation or visual metaphors.

- component insulation, isolating components from contextual dependencies with other components and system dependencies with the infrastructure.

These eight principles reflect a fairly comprehensive spread of concerns and provide a useful vehicle for structuring evaluations of such a diverse field. This is in fact a classification hierarchy for principles and techniques. Classification hierarchies are useful to summarise a range of concepts. Organising the summary as a hierarchy helps deal with the mapping between specific problems and solutions at an abstract level. By applying these classifications to specific techniques and design options we can rationalise our design decisions more clearly.

We can unify these eight principles at an even higher level of abstraction as eight aspects of the two highest level goals which I introduced in chapter 3 as virtuality and componentisation.

By virtuality we mean the characteristics of a system that limit the impact of technology artefacts on the presentation of the system to users. A computing system is virtual if it appearance differs from its strict implementation on a digital machine. Virtuality allows developers to reflect application domain or business concepts in representation schemes. Classification is important as an organising and packaging principle. Interpretation is important to ensure safety and efficiently in mapping to concrete realisations. What is still missing is the ability to ignore all the engineering detail and contextual dependencies that source code introduces, especially in a distributed system. Selective properties are important to specify and manage practical engineering properties and trade-offs that must be made between distributed mechanisms. Insulation is important to allow contextual independence and system independence and to define manipulable, well factored components and tools.

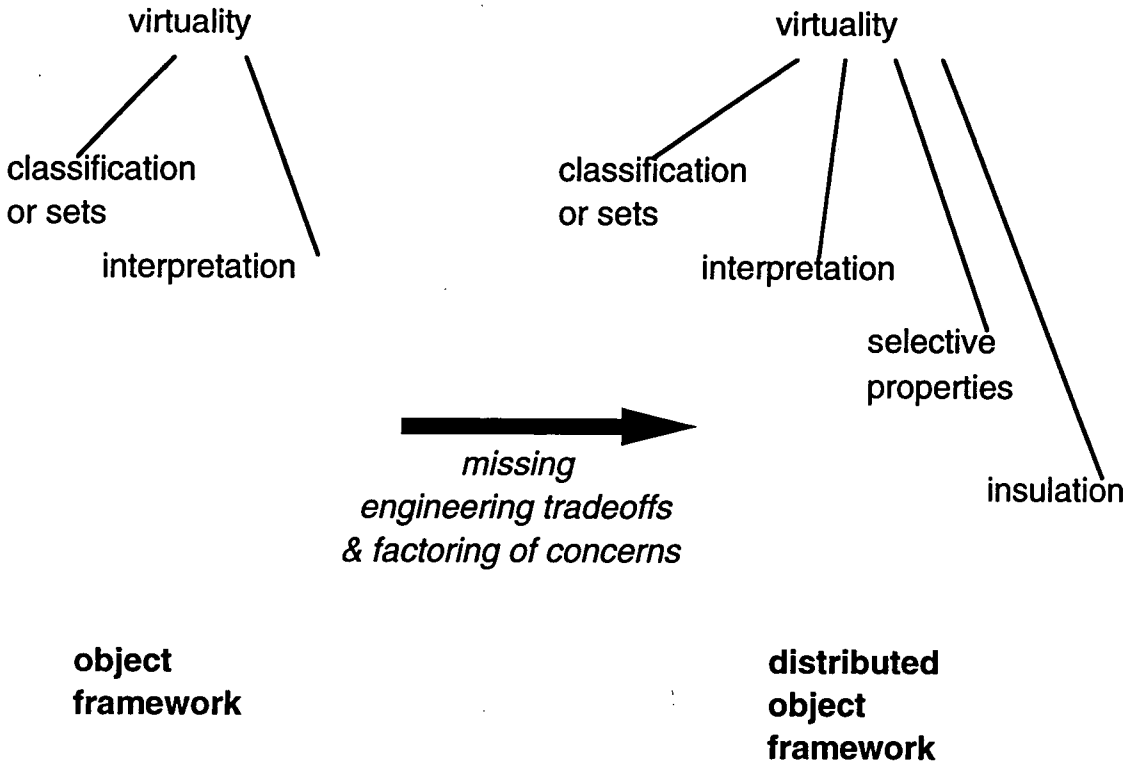


Figure 36 principles of virtualisation

By componentisation, we mean the capability to build software from components that may be integrated across multi-vendor product lines. On the technical side, object technology offers powerful techniques for abstraction, separating the how from the what, and generalisation-specialisation, allowing technologists to break problems down sensibly. Here polymorphism is important to allow components to be substituted easily. Encapsulation is important to isolate the interface from the internal implementation. What is missing is an agreed binding model for plugging components together and specification tools and trading tools to make the process of reusing components more efficient and easier than component fabrication from first principles. A comprehensive protocol is important to ensure application interfaces are widely agreed for plugging multi-vendor components together and for managing the configuration process. Techniques to increase the tangibility of development processes and products are important to overcome some of the cultural barriers to large-scale component reuse.

Componentisation is the key to giving more programming power to naive users. The reuse of prefabricated software systems has the potential to be a simpler task than the construction of software from scratch. More ambiguous, intuitive formalisms can be used when the components are already implemented using a precise programming language.

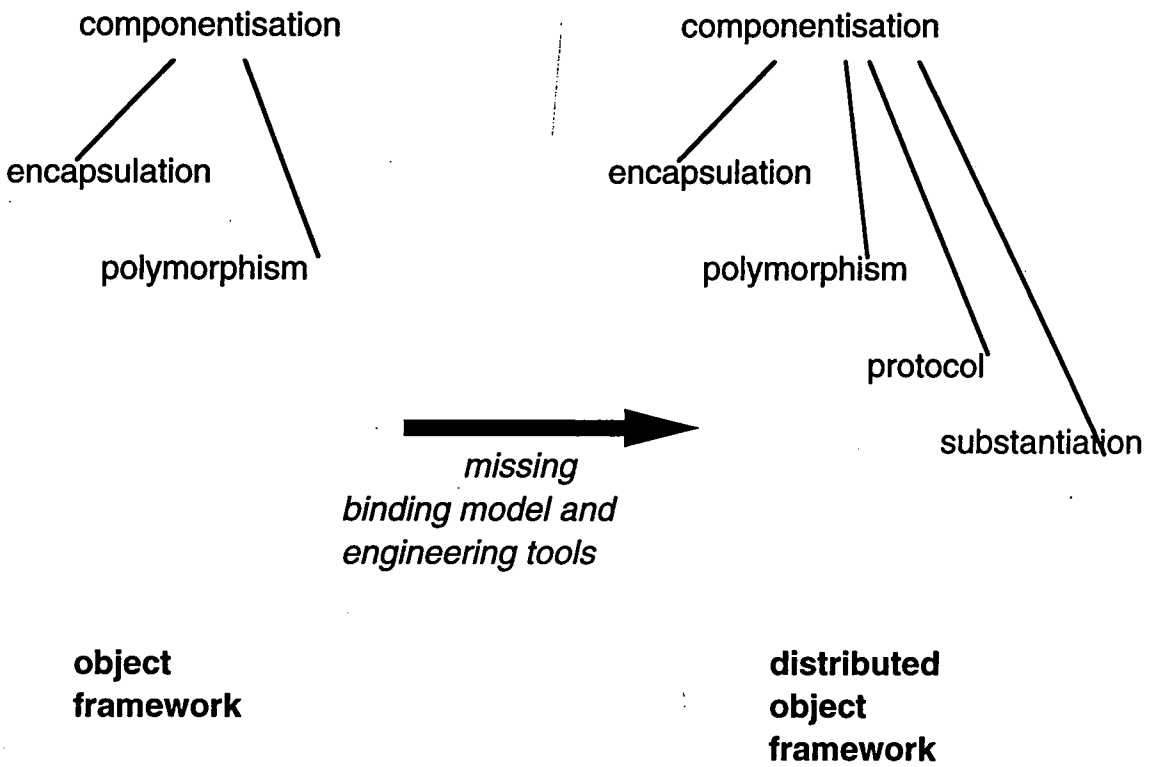


Figure 37 principles of componentisation

The four-faceted principles of virtuality and componentisation are useful to appreciate the real merits of the OpenBase architecture concepts introduced in the next chapter. By taking the classification hierarchy down another level to specific techniques, we can relate high level goals of virtualisation and componentisation to specific choices as shown in Figure 38. This is the basis for rationalising design choices between techniques. Typical goals are defined in chapter 3.

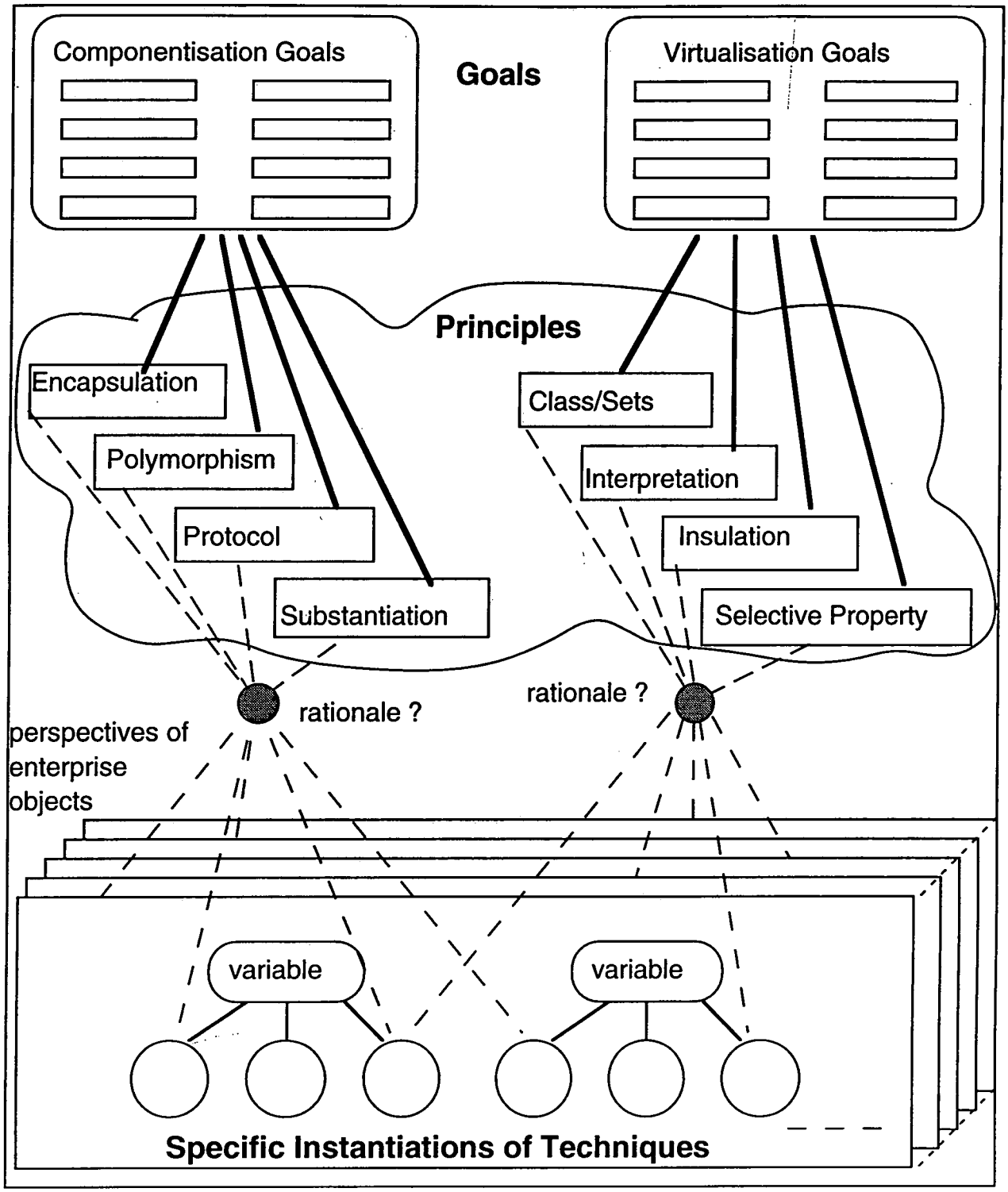


Figure 38 unified design space

6.5 Summary of Part II

The last five chapters have introduced different goals, principles, processes and techniques that can be employed to define a distributed object programming system. Classification hierarchies have been derived to relate these concepts to each other and summarise the results of the survey at different levels of abstraction. The terms used to denote the various classes will be used to describe and define the OpenBase architecture in part III of the thesis. To aid the reader's appreciation of the text, individual items are defined in the glossary in Appendix C: which refers back to the appropriate section where the component is introduced. The complete framework is summarised in the following diagrams, Figure 39 to Figure 48:

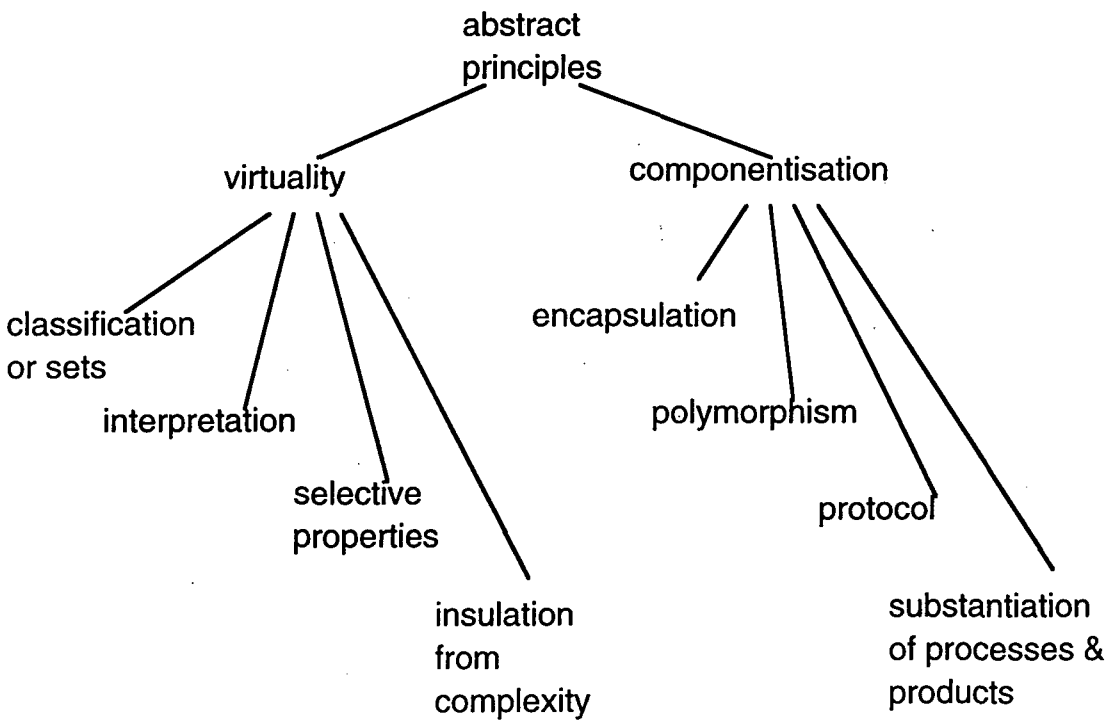


Figure 39 top level principles

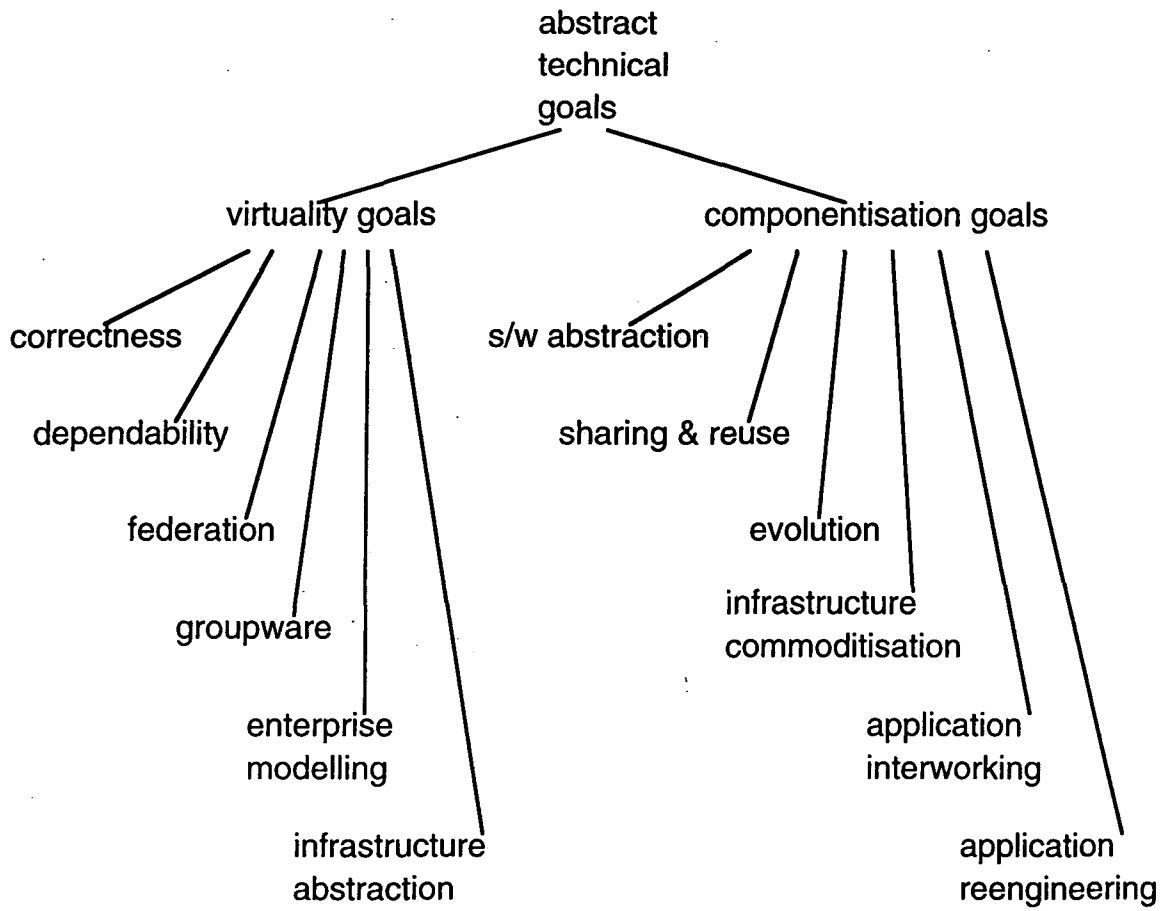


Figure 40 top level goals

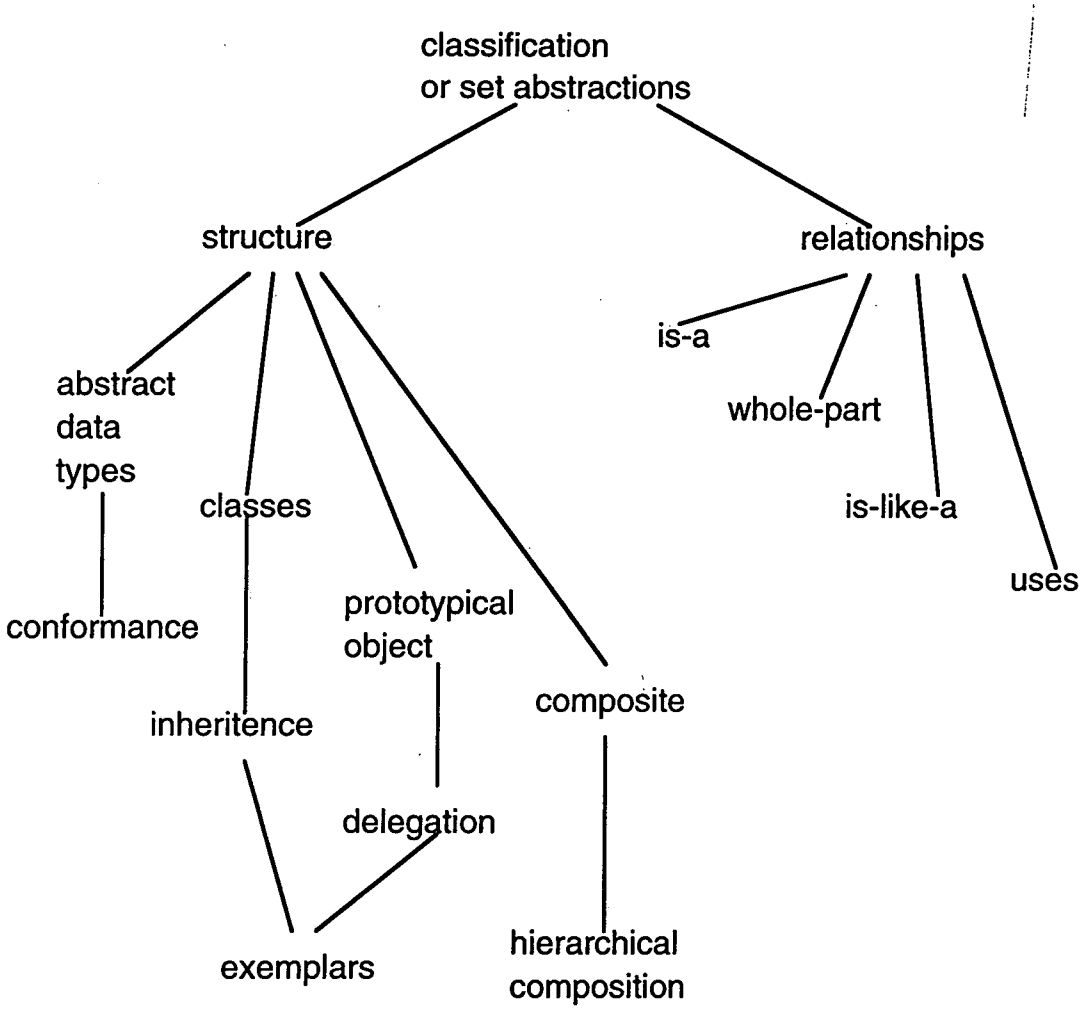


Figure 41 specific classification techniques

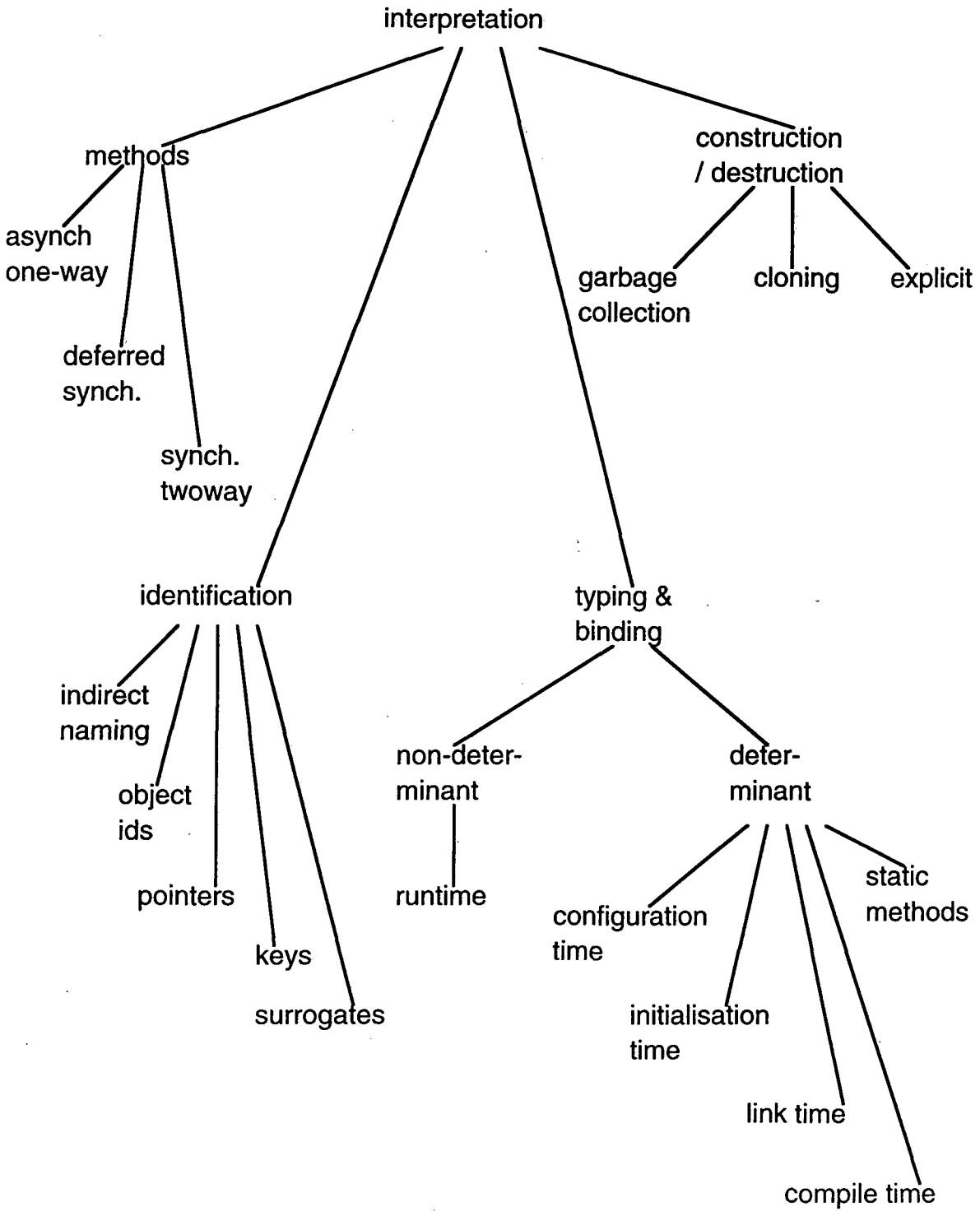


Figure 42 specific interpretation techniques

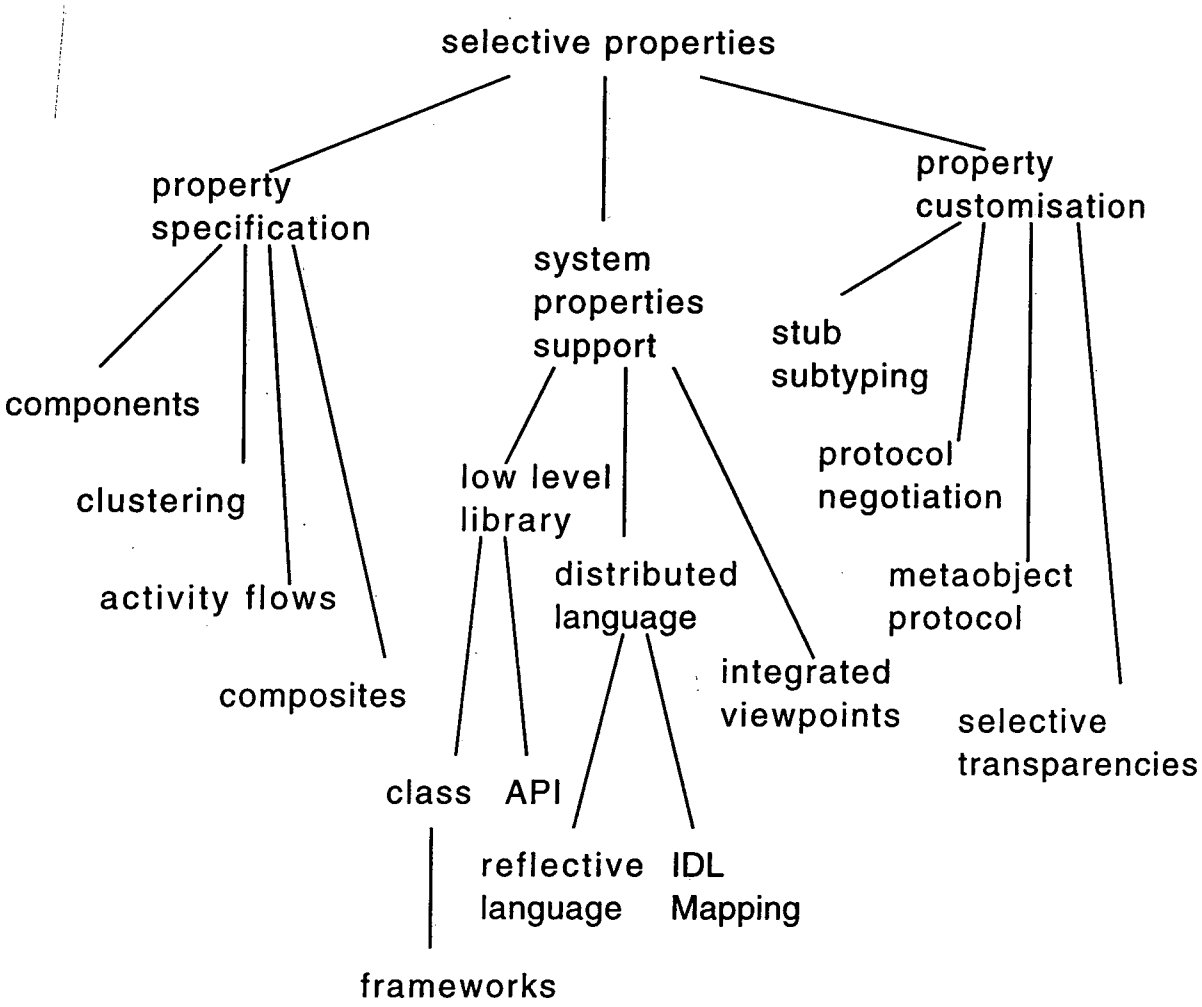


Figure 43 specific selective properties

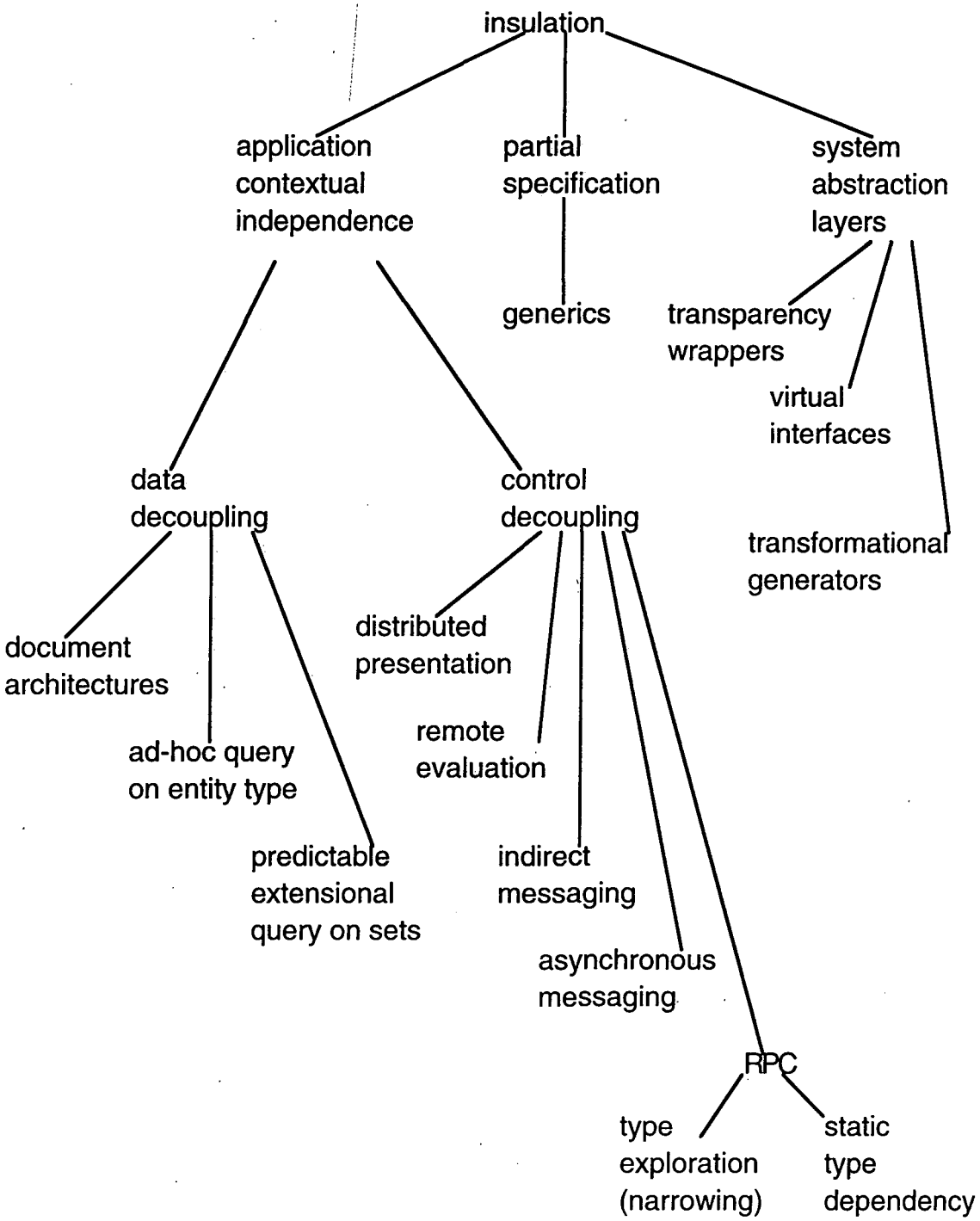


Figure 44 specific insulation techniques

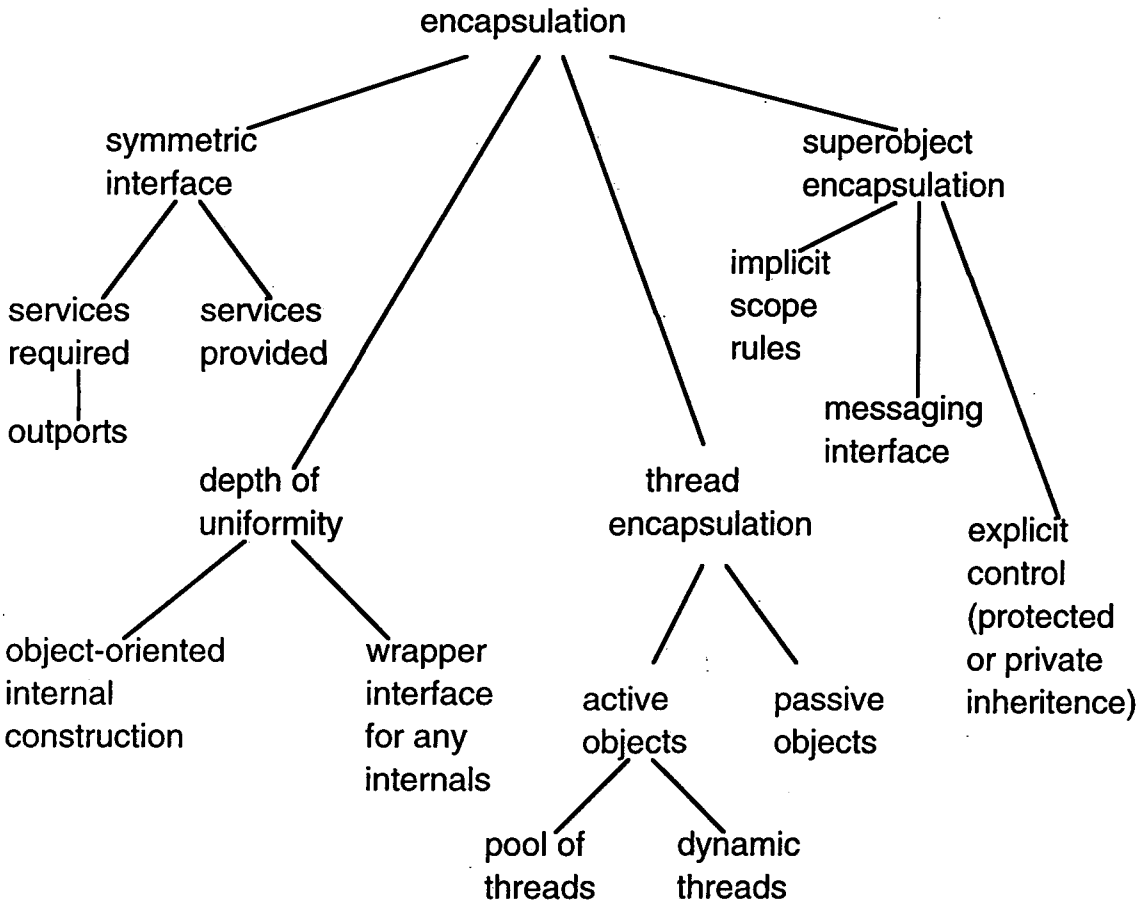


Figure 45 specific encapsulation techniques

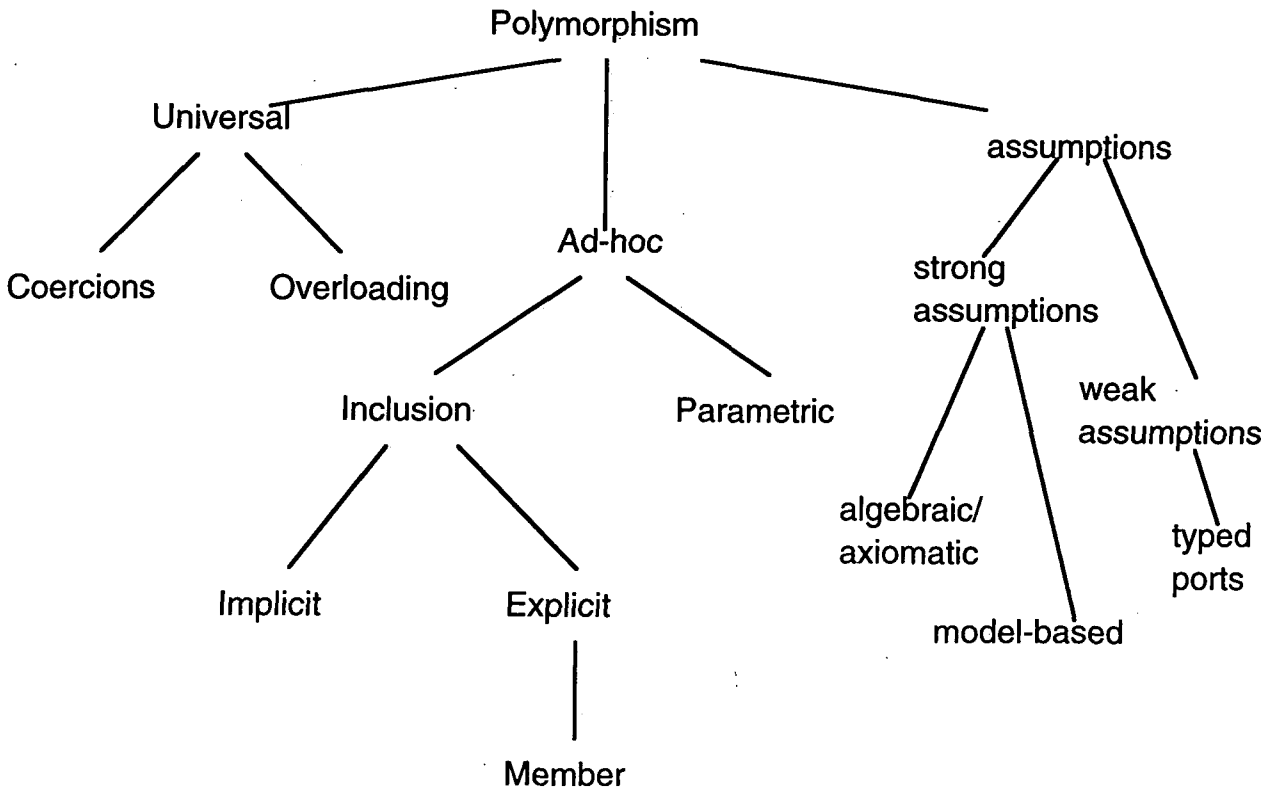


Figure 46 specific polymorphism techniques

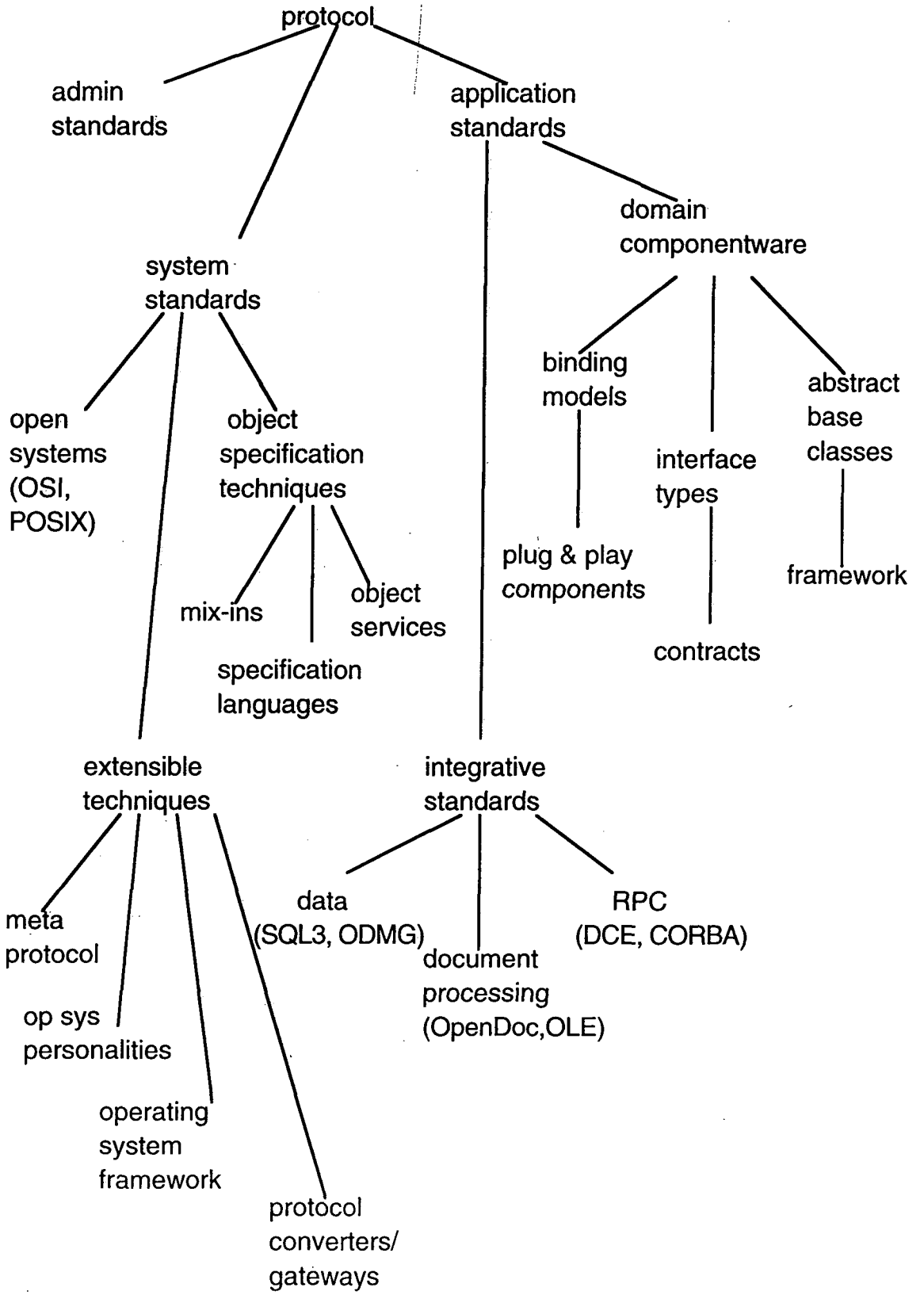


Figure 47 specific protocol techniques

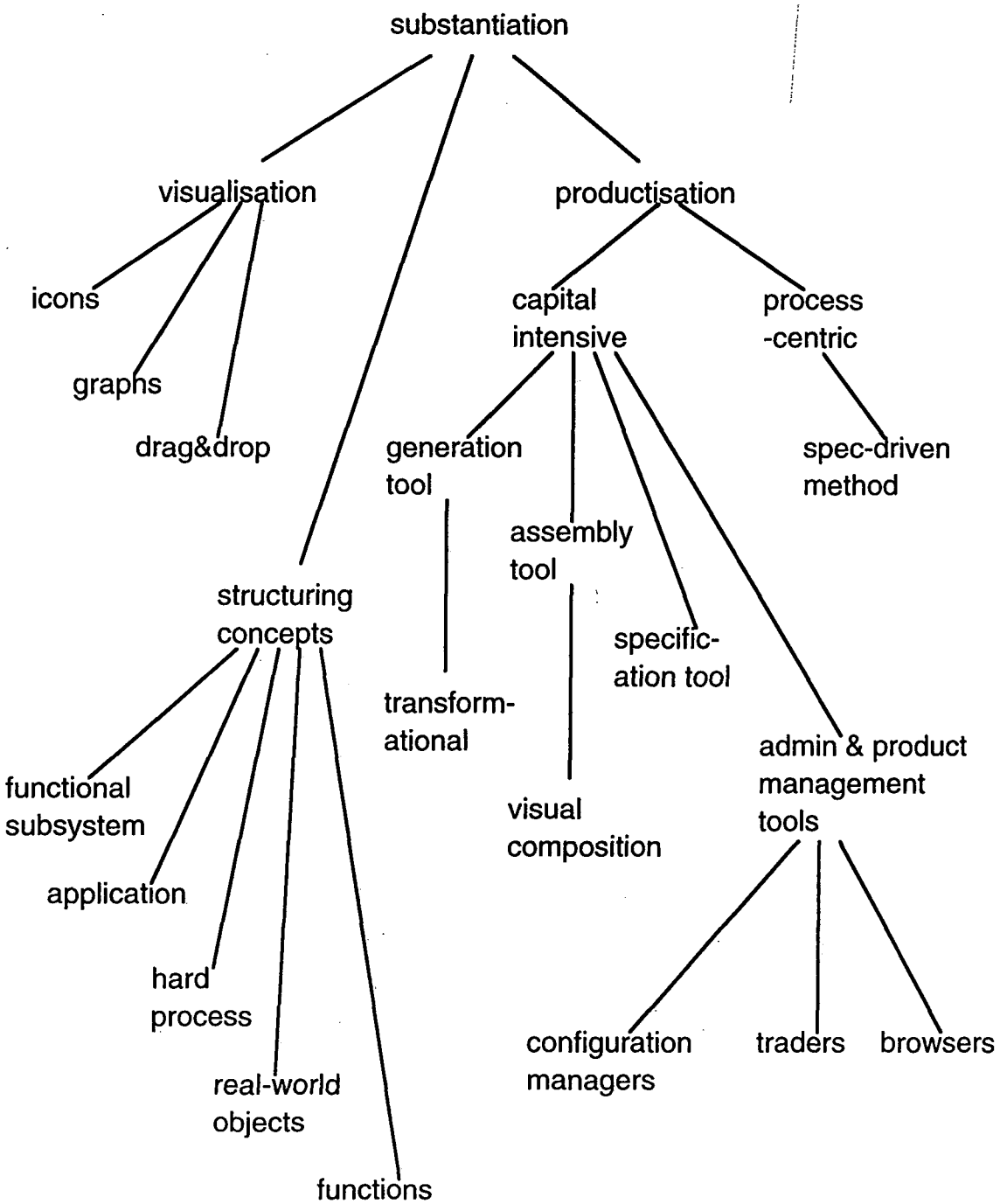


Figure 48 specific substantiation techniques

Part III Evaluation of Architecture

Chapter 7 Overview of Problem and Approach

This chapter summarises some problems in designing the OpenBase programming system architecture and overviews the general solution.

7.1 Summary of General Problems

7.1.1 Overview of Initial Assumptions

The OpenBase architecture seeks to provide an enterprise-wide software integration platform supporting the development and integration of applications for the process and manufacturing industry.

Integrated software systems should be built using a methodology that supports modular production of software; encourages reuse and integration across lines of developers, operating systems and hardware. The object oriented approach to software construction is considered to best support these goals. In contrast, conventional applications are monolithic, insular and proprietary. Common functionality such as reporting is heavily duplicated in independent applications and integration only occurs via low level communication services between whole applications.

Object oriented development provides a terrific boost to software openness. This is partly due to the naturalness of the approach, the emphasis on capturing and exploiting commonality, and also to the rigorous requirement for interface specification. This makes it an ideal model for a standards based integration environment.

The OpenBase integration environment aims to go much further than object technology allows today. It seeks true component oriented development, where the emphasis is on the assembly of prefabricated components, rather than component programming. Constructing an application by component assembly should be like constructing a plumbing system. Plumbers do not need to understand how pipes are manufactured. Instead they know how to seal pipes together using spanners and other tools. Likewise application developers should not need to know how to manufacture a component by programming. Instead component manufacturers should provide them with user specifications which hide the internal construction of components. Application developers then use simple tools like spanners to design and build an application by plugging these components together. Component oriented methodologies emphasise the specification and assembly activities.

Mainstream object oriented development is not yet truly component oriented. This is because object orientation is still seen as an implementation technology, or more specifically a programming technology, rather than a component production and configuration technology. Object oriented programming is still viewed as a labour intensive craft, not a capital intensive engineering discipline. It emphasises construction of reusable components rather than their assembly.

Whilst mechanisms to manage objects in a distributed environment are reasonably mature, there is concern over the programming abstractions provided by these systems. It may be necessary to adapt object orientation considerably in order to rationalise an approach to OpenBase's integration goals for distributed applications.

Interacting application objects do not reflect all the operational behaviour necessary for distribution. Extra levels of system interaction are required to manage objects - for example to create and bind remote objects, to deal with abnormal terminations, to collate replica calls, and to prevent concurrent interference. This may be provided by adding source code to provide this behaviour for example using a system library or framework. However such code would be heavily dependant on the implementation of mechanisms in the environment and it would therefore be difficult to achieve stable standards across mechanisms, to remove semantic ambiguity, or to hide system complexity by using generation techniques. It is better to deal with distribution using declarative extensions to the language. These extensions must be abstract enough to map across different mechanisms.

It is much easier to reason in a language designed for distribution with language concepts directly supporting the tasks of allocation over a parallel network, synchronisation and remote communication, and recovery from partial failures. However such object oriented languages are not yet established. Furthermore formalisms for specifying object allocations over the network, for clustering objects , for specifying scheduling and synchronisation policies for multithreaded applications, for partitioning applications into distributed transactions, and for describing abnormal behaviours to deal with failures, are all universally weak or absent in established methodologies.

A solution may be found by developing an integrated set of development tools The philosophy is simple. If integration of distributed languages with object oriented languages is difficult, why not separate them? Likewise if construction and assembly tends to confuse the tasks of implementation and composition then why not explicitly separate assembly from implementation? An integrated toolset allows us to deal with distribution and computation and with assembly and implementation at different levels using different tools. This allows the choice of the right tool for the job and can simplify programming semantics for each stage of development.

7.1.2 Summary of Shortcomings of Existing Approaches

This section lists some key problems with existing technology that are most relevant to the design of OpenBase. The problems summarise shortcomings mentioned in the survey of part II and includes problems with the reuse goals, problems with methods and tools, and problems with programming system architectures.

There are a number of problems that mean we must revise existing perceptions of reuse - especially for component oriented development of distributed systems:

- Reuse demands a capital intensive engineering discipline that provides a complete set of intuitive tools to organise and assemble components. Object-orientation is still perceived as a programming technology.

- Reusability is such a difficult problem that there is unlikely to be one general purpose reusability approach. Rather there will be specific approaches for different domains. Likewise the approach taken to design-for-reuse will evolve. Today's reusable components will not use tomorrow's model.

- Software reuse is the reapplication of a variety of kinds of development knowledge: domain knowledge, design decisions, architectural structures, code and documentation. Even at the code level there are multiple abstract forms of knowledge about the code, such as functionality descriptions and implementation trade-offs. The types of reuse emphasised by object oriented development are at the code level only and emphasis code implementation rather than knowledge representation.

- Source code introduces a high degree of specificity that inhibits large scale reuse. Distributed requirements demand extra code to support recovery, remote binding, RPC, concurrency and other behaviours. This increases the specificity of components even more. A component should be reusable between different non-functional requirements. To maximise reusability, distributed components should avoid mixing management behaviour and application behaviour, to separate reusable behaviour from contextual dependent behaviour. In addition, components should use abstraction to minimise the assumptions made about other components.

- The process of selecting and assembling components is not well researched, i.e design-with-reuse. It is certainly very different from the process of constructing reusable components for which there is ample methodological guidance and support, i.e. design-for-reuse. Whilst encapsulation attempts to remove implementation dependencies between objects or between objects and the system support, it is still usually necessary to explore the implementation of a class before reusing it. Reuse is still seen as a part of the implementation activity instead of a separate activity. Inheritance and invocations to a reused class are *"programmed"* into a derived class. To change this, there needs to be more explicit rules or constraints on how reuse is to be managed and encouraged.

- In practice, inheritance is more often concerned with implementation than specification. It is frequently used as a mechanism to compose behaviours. A typical example of the way inheritance is being used in the commercial world is to create specialisations of a List class such as OrderedList, SparseList and indexableListwith different implementations which make different performance trade-offs. One specialisation may multiply inherit from an array class to use the array as its internal implementation. Clearly a user of the list should not be concerned with behaviour sharing relations such as this. Ideally specification hierarchies should be provided to capture these trade-offs that are orthogonal to the implementation hierarchies used for sharing implementations. Inheritance is abused to mix the two.

- Different implementations of a list may make different trade-offs, for example internally use arrays or linked lists as above, represent elements as generic pointers or buffers of fixed size values, apply different growth rates. There is no means of showing these commercially significant differences to potential users without violating encapsulation and looking at the implementation itself. These properties are not fully captured by inheritance relations. Likewise if we deal with distributed properties like recovery behaviour in the implementation of a class then we have no easy way of showing our operational trade-offs to potential clients. Nor is there an easy way to show the dynamic properties of a class i.e. what it actually does. Class names and method names can be insufficient for this when the behaviour is complex. Programmers tend to be obsessed with efficiency and are very reluctant to reuse something without knowing this sort of information. If we are to avoid looking at implementations, a modelling interface is required to capture this information for later selection. Modelling interfaces may form the basis of the specification hierarchies discussed above. Alternatively the information may be obtained by exploration of a components behaviour through execution or simulation or theorem-proving.

- Reuse of components in different contexts is made difficult because the interface between objects is incomplete. It fails to capture the outgoing interface for a client. Consequently dependencies between a client and a server can not easily be deferred and resolved by the system integrator. There is no clear binding model that allows clients to be integrated in different contexts without violating their encapsulation. The encapsulation of entry points and not exit points is in line with a model of reuse based on incremental extension by adding custom made clients. The system is reused by extending it with new reusable clients and subclasses, i.e. design-for -reuse. A complete interface on the other hand supports reuse of both clients and servers in new contexts without any custom development, i.e. design-for-and-with-reuse.

There are a number of problems with existing development processes and tools:

- Existing object-oriented methods are weak on requirements specification. The relationships between a requirement model at some level of abstraction and an object design is not always clear. Object oriented methods are therefore imprecise. Specifications risk appearing understood by everyone but interpreted differently. Requirement models based on mappings between roles and objects begin to alleviate some of these problems but are not mature.

- The imprecision is more general than requirements modelling. Object oriented methodologies are generally imprecise, being strong in notation yet weak in processes. Likewise existing tool support is for notations and not design transformations. Second generation methods like Fusion (Jeremaes and Coleman, 1993) are alleviating some of these problems. Another solution is to resolve it in an integrated toolset.

- Methods usually provide at least two disjoint views, for example class models supporting is-a and consists-of hierarchies and dynamic models describing behaviour. The relationship between class modelling and dynamic modelling is poorly explained.Some methods rely exclusively on state charts for describing dynamic behaviour of a single object. It is generally difficult to understand complex method behaviour or behaviour resulting from collaboration between several objects. Developing formal semantics to describe behaviour is an active area of research. Notions like roles and responsibilities unify the views of class structure and behaviour to a degree.

- Distributed object technology is often defined at the level of mechanisms and languages. There is some consensus on the set of techniques and mechanisms used to construct distributed systems, such as RPC, group multicasting, atomic actions. Yet none of these mechanisms help in the design process. Client server systems rely on simple partitionings of presentation, logic and data access functions. This is unsuitable for applications that perform complex processing on each node such as supervisory control systems supervising control processes on several nodes. Higher level formalisms are required.

- Operational, i.e. non-functional, requirements such as responsiveness, generally exist at the process level as constraints on asynchronous parallelism between system objects and environment objects. Yet there is no clear mapping between fine-grained objects and processes. Object oriented design assumes the entities identified in analysis are simple and can be modelled as objects not subsystems. There is no easy way to partition a large design into subsystems that can be sensibly distributed. Too little attention is paid to task management in methods and this results in over-simplistic policies for task scheduling and object allocation to processes, with poor traceability to operational requirements.

- There is a dichotomy between approaches to partition a large application which require analysts to focus on the whole and approaches to identify reusable components which require analysts to focus on the parts. The latter results in fine grained architectures with much coupling between components. The former results in coarse subsystems containing heavily biased functional objects that are not reusable between functions.

There are a number of problems with existing object oriented programming systems:

- Mechanisms for interaction have achieved a degree of maturity as reflected by their inclusion in commercial distributed system standards from the OMG, ANSA and OSF consortia. However these mechanisms merely provide the glue used to compose applications. What is missing is any notion of structure. Consequently it is difficult to develop applications exhibiting anything more complex than the simplest client-server partitionings.

- In mainstream object programming systems, there is no explicit definition of the dynamic structure in terms of instantiations and invocations. Instead these imperatives are embedded in the implementation of a class. The implicit nature of object dynamics makes reconfiguration and evolution of the structure unmanageable.

- Inheritance is a single mechanism unifying the approach to different aspects of programming: type checking, polymorphic binding, and behaviour sharing. This unity is one of the strengths of object oriented languages yet it is also a weakness. By separating mechanisms for interface sharing, behaviour sharing and binding, it is easier to rationalise an approach to extend their semantics to deal with distribution and pluggability. Encapsulation problems arise from the failure to distinguish the two kinds of client of a class: other instances and subclasses. The use of different forms for these two interfaces in mainstream inheritance based languages leads to a duality in the model that must be accounted for when extending the language. For example, adding a management policy for an object inhibits incremental modification of that object if we can't also incrementally modify the policy across inheritence relations. This can be tricky. Hierarchical composition and delegation systems use a messaging interface everywhere and are easier to extend.

- Methods have found it necessary to introduce new concepts to specify reusable properties of classes such as roles and behavioural contracts, yet these concepts are not directly supported in mainstream object languages like C++. Some methods also support concepts that are useful for dealing with large systems at several levels of abstraction. This includes frameworks and subsystems. Yet languages provide little support for any of these concepts. These constructs are useful to specify domain standards for large control systems but are difficult to explicitly define and enforce without programming system support.

7.2 Analysis of Specific Problems

This section analyses the key problems in the design of OpenBase using the evaluation framework of chapter 6 and presents an abstract view of the solution.

7.2.1 Design Space for OpenBase.

The first section positions OpenBase by defining an appropriate profile of the evaluation framework of Chapter 6.

Motivational Perspective

The goal of OpenBase is to provide an enterprise integration infrastructure for the petrochemical and manufacturing industry. The increased need for integration and supervisory control of plants is driven by the needs for increased plant flexibility and performance, less downtime, greater stock diversification and new safety standards. This requires tighter management and a global view of the whole process. No existing products allow integration on the required scale.

Three key infrastructure goals that arise from these needs are reuse, evolution and interworkability.

Businesses need to be more responsiveness to changing markets and manufacturing process requirements. Reuse is important to be able to build on existing resources, rather than redeveloping them time and time again, as business needs continue to change. Fine grained reuse of individual facilities within each application is important to avoid redundant duplication of similar facilities across applications. For example every control application typically provides its own implementation of reporting and alarming facilities.

Responsiveness to changing requirements requires rapid development of new applications or the rapid reconfiguration and integration of existing applications. New manufacturing opportunities may not last for long, and often not as long as traditional development lead times for new applications to support them. Further a solution should not limit further changes and advances. Evolution means rapid prototyping, reconfiguration and incremental modification of existing systems. This should be supported by the infrastructure.

Interworking is important to supervisory control because specialised applications from different vendors are usually provided for individual stages of the same chemical process. This includes specialist controllers, for example predictive controllers, specialist numerical processors, for example for feedback loops, and specialist analytic tools, such as historical data analysis. To provide overall supervisory management requires integration across these specialist applications across heterogeneous platforms and this can be expensive with conventional low level interconnection approaches.

Cognitive Perspective

Distributed systems are much more complex than single systems. They are also complex to debug and maintain. They require a higher level of abstraction across runtime mechanisms to support transparency, dependable service, efficiency, portability and scalability. They must deal with insecure interaction, indeterminate interaction, unreliable interaction and costly interaction. At the least, the system must be able to allocate applications to processes and processes to nodes, enable applications to synchronise and communicate, and deal with failures gracefully.

Transparency properties can mask these complexities to an extent but there are two fundamental assumptions that cannot be overlooked: that effects take time to propagate; and that work can only be done with the right resources. The first means a programmer must be aware of communication latency to meet responsiveness and performance requirements; he must prevent destructive interference between concurrent activities for safety and liveness; and he must ensure serialisability and atomicity of actions for integrity in the presence of aborting activities. The second means resources must actually be deployed and allocated for efficiency; they must be replicated for availability and performance; and the programmer must recover from partial resource failures for reliability. High level language support should be provided to deal with these complexities.

Many configuration decisions regarding distributed properties like reliability, time criticality, availability, are best made by plant engineers on the manufacturing plant. Much time and documentation is currently wasted communicating requirements between plant engineers and system integrators. This is because low level programming is well beyond the scope of a typical plant engineer. In particular the low level communication facilities used to integrate existing applications are often complex to use, provide non-uniform interfaces and can result in programs that are difficult to debug and maintain.

High level configuration tools can bridge this knowledge gap by allowing plant engineers to get more involved in integrating applications themselves. Structuring a collection of objects into an integrated system is a different intellectual activity to the construction of individual objects. A integrated tools based approach allows different people to describe the system at different levels, applying different skills and knowledge. Objects can be implemented by control software vendors using specification tools and programming languages. The plant engineer then would use configuration tools to interconnect objects and encapsulate them in larger composites and to describe the overall system. Plant engineers are not interested in management policies per se. High level tools may describe semantic information that is exploited to automate the choice of management policies. This combines the flexibility and efficiency of a low level interconnection approach with the safety and simplicity of a high level language environment.

Existing object oriented methods are not truly component-oriented, i.e. they do not separate component development from component reuse. The component oriented lifecycle identifies two roles: component programmer and application engineers. These roles cleanly map onto the control software vendors role and the plant engineers/system integrators role. A component oriented approach is ideal in supervisory control because typical manufacturing and chemical processes consist of a series of stages that are each satisfied by specialised components that need to be connected together.

The definition of an interface in conventional client server or object oriented applications is incomplete. Existing IDLs and object oriented languages fail to model exit points (i.e. invocations) in the interface. Consequently dependencies between exit points and other components can not be easily be deferred and resolved by the application engineer when the component is used. A complete interface should also define the outgoing interface of a client. This is necessary for integration of clients in different contexts, without breaking their encapsulation. The traditional client server model is analogous to a domestic electrics manufacturer that used prefabricated sockets but custom made all his plugs. Client server approaches tend to assume clients are not reusable and servers are reused by incremental addition of new custom-made clients. This is not the component-oriented model. For a component oriented system, the exit points or services required by a component should also be captured in the interface.

The above problem may be stated more generally by the preposition that existing object oriented binding technology is weak for component oriented development. It needs to be enhanced by :-

1) providing distinct tools for component assembly and component construction. Graphical assembly tools are more intuitive. Declarative construction tools free implementors from the specifics of how services are achieved in every target system. Tools make the activity of reusing component explicit and enforced.

2) separating the specification interface from the implementation interface. A specification interface should reveal dynamic properties and operational tradeoffs not how internals are constructed. This may be descriptive text, pre-defined parameterised property values or formal specifications. This provides the information required to understand and trust a component without violating its encapsulation.

3) defining pluggability via interface standards and specific binding models for the assembly mechanisms. Pluggability is unlikely to be achievable unless interfaces are simple and common across a domain. Framework standards that are defined across a domain, identify abstract roles that components can play and increase the chances of realising large scale component reuse in that domain. Identifying these frameworks demands an up front investment and co-operation between vendors selling into the domain.

Capability perspective

The programming system must be expressive to allow process control specialists to deal with the complexities of distribution without learning new distributed computing skills. It must have an efficient implementation to support time critical process control. It must be polymorphic to optimise pluggability of components from different specialist control vendors. There exists no widely used distributed polymorphic language. Any language demands significant compromises.

A better approach is to have an integrated programming system that allows different tools and languages to be mixed and interchanged.

In mainstream object oriented languages like C++, the interface between a superobject and a subobject is defined purely in terms of how method calls are delegated between them. It says little about how management behaviour is composed over the inheritance relationships. For example what happens to a concurrency control policy when it is inherited? Object oriented languages have not yet defined a way to compose extra semantics for distribution. Attempts to add distributed management policies orthogonally to an object oriented language are doomed because distributed properties are not orthogonal and must be themselves inherited. Care must be taken in the way distribution is added.

As the interface to objects becomes richer as distributed properties are added, it may also be useful to provide different interfaces for different aspects of object management, for example the replica management interface used to add a new member to the group of replicas should be separated from the messaging interface to another application objects. Most existing object oriented languages do not allow an object to have multiple interfaces thus preventing the clear separation of messaging interfaces from the management interfaces with the system.

Not only is it useful for an object to have multiple interfaces but it is also useful to have multiple programming interfaces to describe different properties of the object. Inheritance makes the composition model used to define objects complex and this makes it difficult to extend object oriented languages with expressive features for distributed properties. Expressivity interferes with the compositionality. A simpler solution is not to deal with distributed properties in the object itself but rather deal with them reflectively in meta-level code that wraps the objects. This code can be generated from meta-programming interfaces. An integrated programming system can provide multiple meta-programming interfaces to deal with management behaviours independently from application behaviour.

Constituency perspective

With the establishment of new trading partners and mergers to open up new markets, businesses also invariably have a diversity of systems, including: geographically distributed data; a variety of mainframe and mini-computer platforms and operating systems; and different communications networks. A consistent global view of the systems involved is needed to manage the enterprise more effectively. This demands a high level of abstraction away from the trappings of a particular technology. Existing systems are defined by specific low level interfaces and proprietary mechanisms. This means doing business frequently requires a merger and unification of IT strategy.

Consistency in management policy does not mean that the system should not tolerate diversity. The solution must provide a sufficient level of abstraction to allow management policies to be optimised regardless of platform. There are a number of important trade-offs that should not be ignored in distributed systems management, for example between reliability and performance, flexibility and safety, communication latency and parallelism. A diversity of mechanisms is needed to ensure that each part of the business is making the right trade-offs for each facility that is supports.

The importance of trade-offs increases with the number of objects that need to be managed. Supervisory control applications for the process industry typically involve large numbers of fine-grained control objects, such as datapoint tags for I/O points, thresholds, alarms. Therefore performance overheads per object can be critical. Solutions require a flexible approach to specifying management behaviour and making appropriate trade-offs between overheads.

Existing systems generally provide all-or-nothing management abstractions. Flexible management mechanisms are required. Performance can be improved by : relaxing reliability guarantees; by using lightweight local RPC; by replicating objects across the network; by load balancing; by clustering; and by increasing application parallelism. However mechanisms that provide this flexibility also encumber developers with the new complexity of selecting and realising different policies and demand new language concepts.

An alternative to solving these issues in the programming technology is to solve them in the assembly technology. Management support in a high level configuration tool can provide an abstract solution that can be mapped across heterogeneous platforms.

Developer complexity can be reduced if the development environment implicitly supports adaptive management. One convenient characteristic of objects is the amount of semantic information that can easily be made visible in interface definitions. An adaptive system can exploit this information. This facilitates a declarative approach to object management. The programmer specifies the properties required of objects, using a definition language and composition tools. It is up to the environment to interpret requirements correctly across different platforms.

Correct interpretation requires us to develop concrete mechanisms for property representation and implementation. Greater simplicity can be achieved by separating concerns through multiple levels of program transformation. An integrated programming system can control the visibility of these transformations by providing distinct programming languages and editors at each level. Adaptive optimisations are possible as component classes are compiled, as application frameworks are constructed from components, as applications are configured from frameworks, and as loaded applications execute at run-time.

The discussion of problems may be summarised using the perspectives of the evaluation framework to categorise the different problems. This is shown in Figure 49.

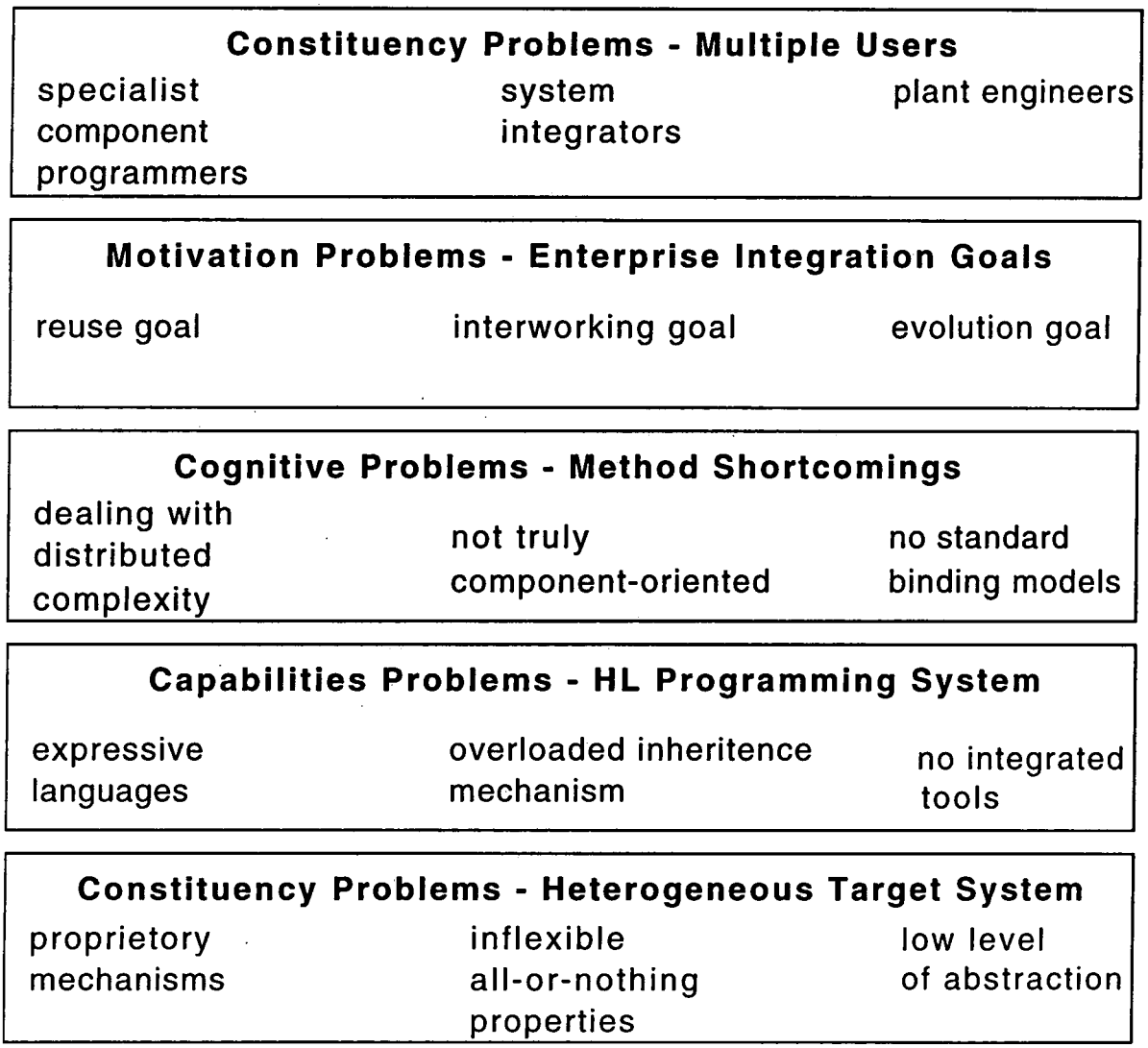


Figure 49 Problem profile for existing systems

7.2.2 General Statement of Problem and Approach

A single programming technology is insufficient to deal with the multiple concerns of : component oriented development, emphasising tangible assembly activities; open distributed computing, emphasising interoperability; and efficient fine grained integration, emphasising trade-offs between a wide range of management options.

A solution is to separate concerns through an integrated set of tools existing at multiple levels: at the programming level, emphasising implementation techniques; at the assembly level, emphasising specification and composition techniques; and at the system level, emphasising generation techniques.

The solution space may be characterised using the abstract principles of the evaluation framework:

- encapsulation is important in this context to prevent components from having dependencies on the implementation of other components. Components are produced in isolation and should be properly specified so that there is no need to look at the implementation. Therefore specifications must include all the properties and information necessary for reuse. This includes resources requirements, expected throughputs and latencies and sizing information.

- classification is important to structure the software around tangible objects that are meaningful to the application engineer. Classes capture reusable behaviours. Inheritance relationships capture patterns for sharing.

- insulation is important to isolate what is reusable from what is specific. Specific details can be deferred, separated or made refinable. Conventional imperative source code demands a high degree of specificity. This increases in a distributed environment when we add concurrency, persistence, recoverability, mobility etc. A reusable component should be insulated from these contextual complexities.

- selective properties are important to allow control over management policies to meet non-functional requirements like reliability, time deadlines, security without having to specify every aspect of management for every object. Declarative properties where the programmer describes the properties required of the system in the interface to a component rather than programming the actual behaviour in the implementation of a component, is important to avoid overloading the programmer with responsibilities.

- protocol at the application level is important to ensure that components fit together. Protocol at the system level is important to ensure that components may be freely distributed across heterogeneous platforms and co-ordinated in a uniform way.

- polymorphism is important to maximise the opportunities for plugging components together. In a polymorphic system, application protocols may be defined at multiple levels of abstraction so a component can minimise the assumptions that it makes about another component thus maximising pluggability.

- substantiation of the process is important so the roles of the component programmer and application engineer are clear. Visual programming is important to allow end users like plant engineers to act as application engineers. Substantiation of the components and sensible choices of names and iconic representation is important so the plant engineer understands what behaviour to expect from components.

- interpretation is important to map the abstract representations of programming concepts onto an executable target. This must be done efficiently and safely. Interpretation support may build on the runtime support of integrative standards to provide a single virtual system across heterogeneous systems.

The general problem is that these principles in combination would be difficult to resolve using a single language. Whilst inheritance mechanisms provide a sufficiently powerful mechanism to support encapsulation, classification, polymorphism and interpretation, the further bundling of open distributed computing principles like selective distribution properties, pluggable protocols, substantiation and insulation between components, demands variations of the composition mechanism that undoes the benefits of using a single simple mechanism like inheritance everywhere. Inheritance embodies behaviour sharing, interface sharing and binding in a single mechanism.

The bundling of principles on a single language is illustrated in Figure 50.

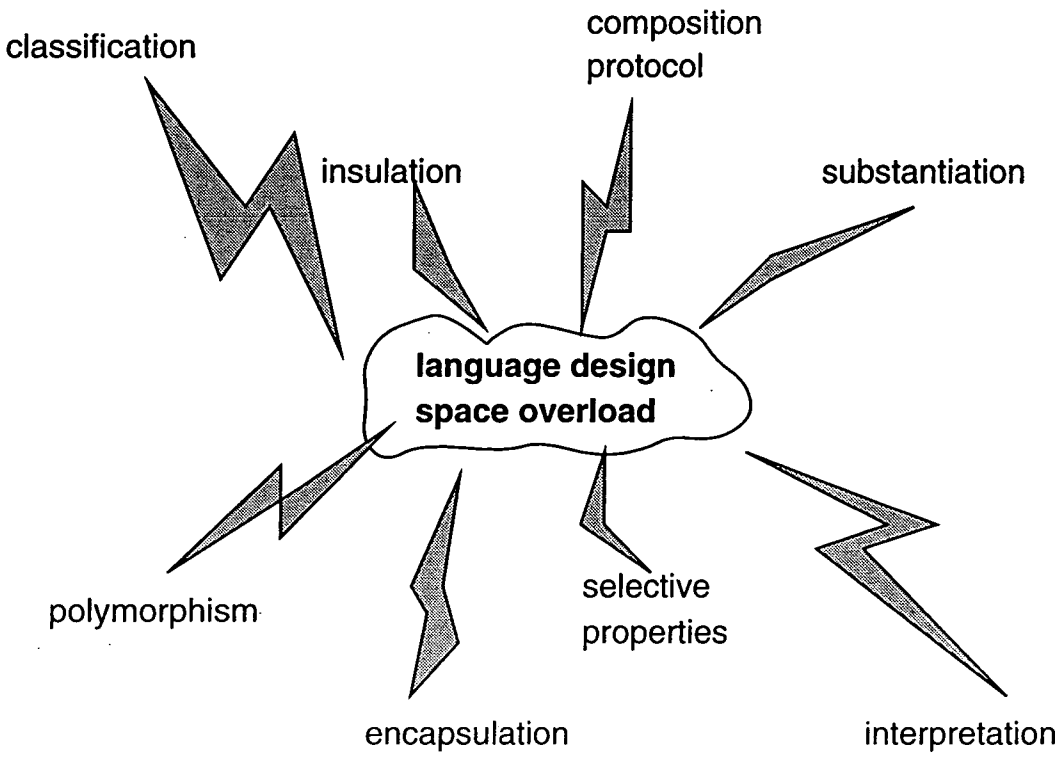


Figure 50 Problem space overload

The solution is to unbundle overloaded mechanisms like inheritance mechanism by opening up the language, to allow the integration of multiple programming interfaces that deal with different aspects and variants explicitly rather than reapplying a single formalism in different ways in an uncontrolled manner. This may not be as elegant, but the separation of concerns and mechanisms makes the whole manageable and extensible with less interference between mechanisms. This approach provides different mechanisms for behaviour sharing (class subclassing), interface sharing (port interface subtyping), and binding (hierarchical composition) rather than unifying them with inheritance.

The unbundling of principles is illustrated in Figure 51.

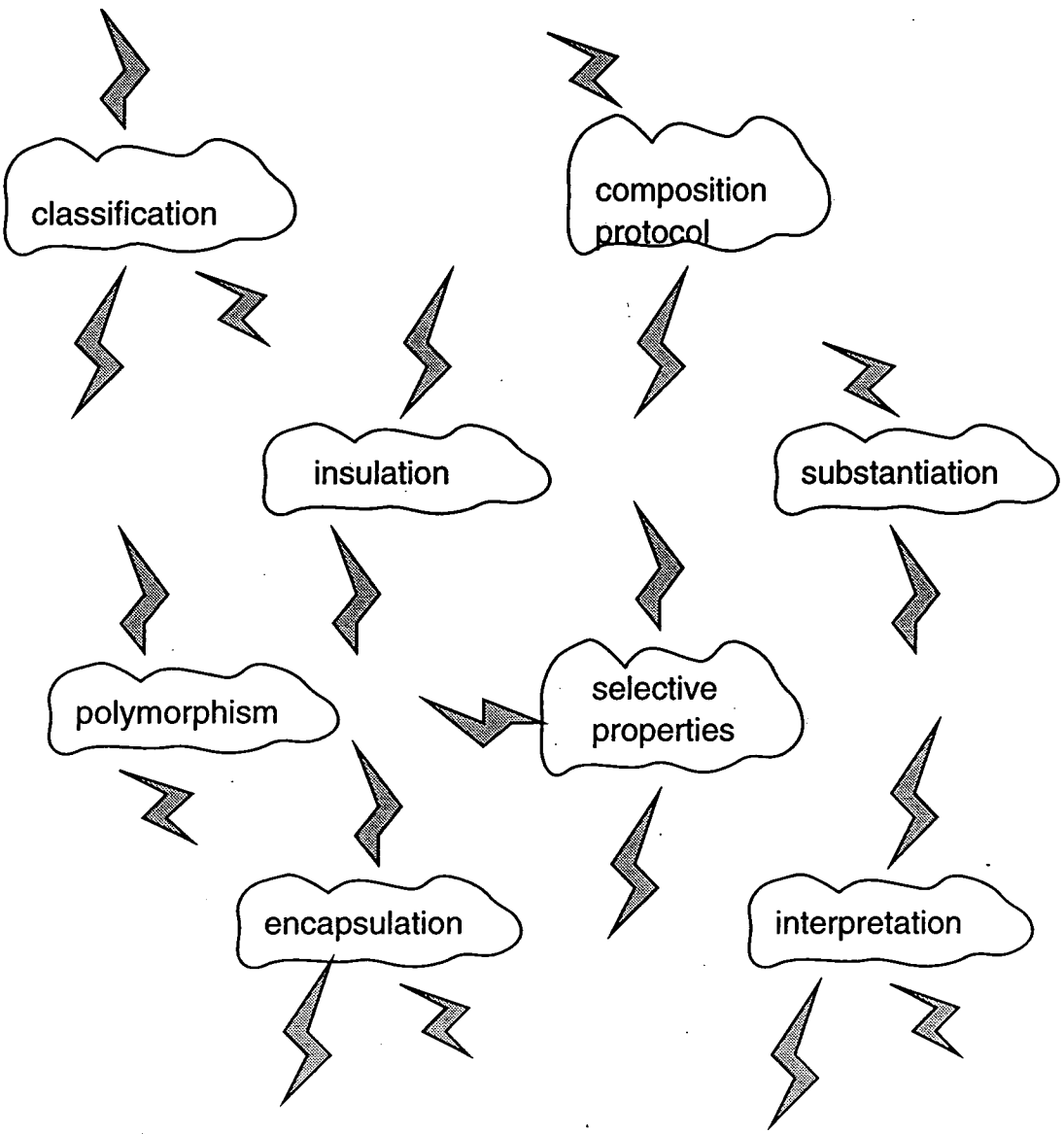


Figure 51 Problem space unbundling

Unbundling the conventional solution space serves several purposes:

- it distinguishes the different types of developer and their distinct roles;
- it separates the different programming interfaces that together provide the integrated programming system;

- it isolates complex distributed object management behaviour from application-specific behaviour.

At a conceptual level, the OpenBase architecture may be viewed as an explosion of the language design space both vertically and horizontally. Vertically , it adds front-end specification languages and tools to a C++ environment (ObjectStore) and a distributed object management system (ANSAware). Horizontally, it separates application engineers from component programmers and distributed object management from application object implementations. This unbundling of the problem space allows the definition of a simpler solution space. Figure 52 shows the pieces of the exploded architecture that make up the solution space.

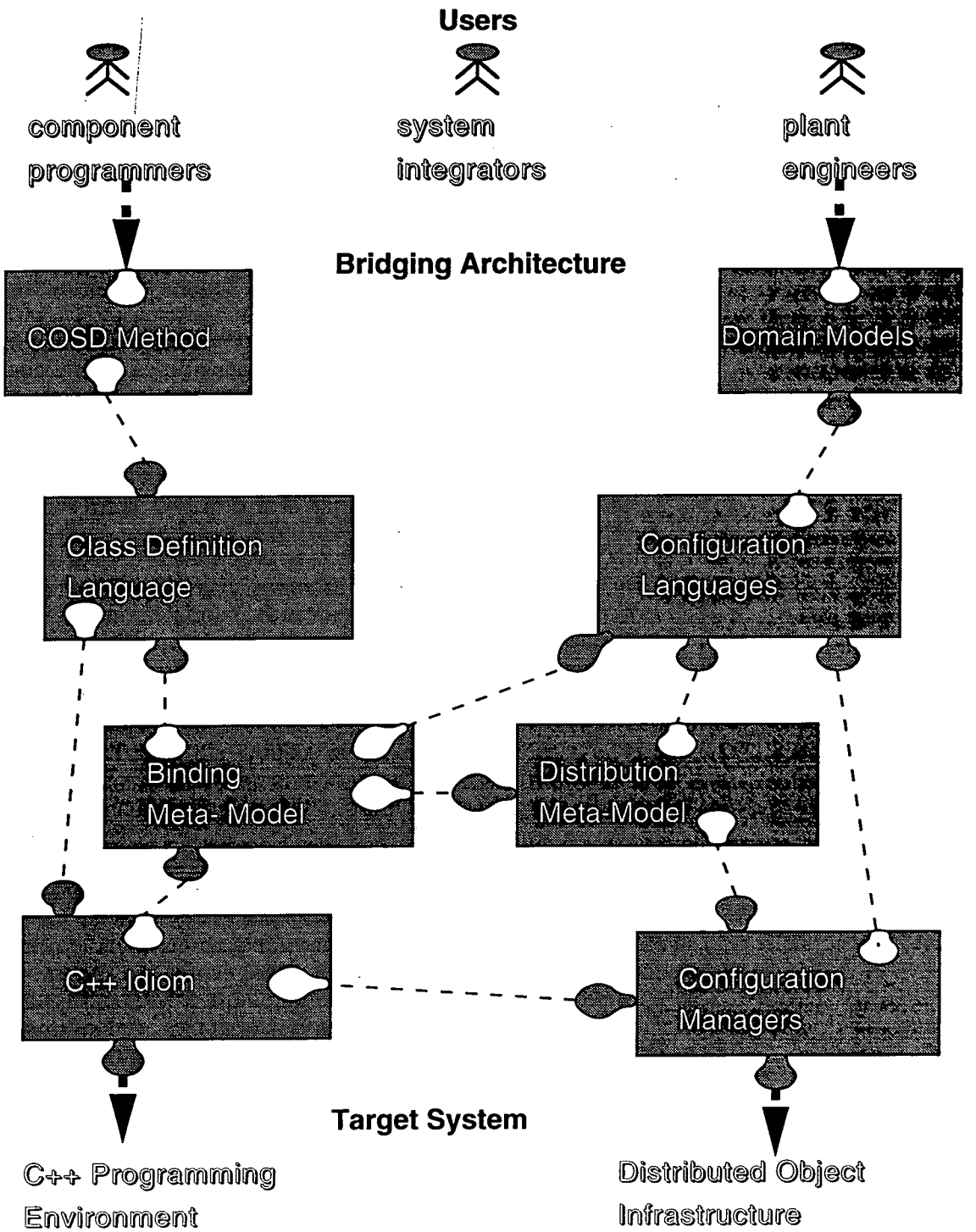


Figure 52 Solution Breakdown into Pieces of Architecture

This architecture provides a suitable separation of concerns that makes it easy to rationalise design decisions. In particular these pieces are well factored both in terms of the abstract principles that each emphasises and in the technical goals supported. The pieces and their key principles and goals include:

- CODM, the component oriented development method provides the guidelines used by component programmers to develop components in isolation of other components. The key principle emphasised here is the use of **classification/sets** to reason about behaviours. The key goal is that of **software abstraction** in organising similar components.

- Domain Models, this piece provides the domain specific support, such as domain specific frameworks, standards, customisations of the programming interface, domain concepts, and libraries providing domain specific functionality such as alarming for supervisory control. The key principle and goal emphasised here is the establishment of domain **protocols** to facilitate **interworking** goals at the application level and system level.

- Class definition Language, used by component producers to specify and register components. It describes classes and automatically generates support to hide the external system interface from the class implementation. The key principle emphasised here is the use of **insulation** techniques that transparently wrap an object with code to provide the required environmental support. The key goal is that of **infrastructure abstraction** to present a uniform virtual system across platforms and management mechanisms.

- Configuration Languages, used by system integrators and plant engineers to configure applications and model the process plant. It describes networks of configured graphical objects and objects with which the software components must interface : - operators, PLC's , nodes, resources, libraries. The key principle emphasised here is that of **substantiation** to make the process of assembling objects and describing the plant configuration intuitive to the application engineer. To do this is must provide a comprehensive set of component management tools to find components, understand components, modify components and bind components, to support the **large scale component reuse** goal.

- The binding meta-model, provides a repository holding definitions of classes and hierarchical networks of objects. These are read by the configuration manager to interpret and load the configuration of runtime objects. It provides browsing, representation and editing facilities to the tools layer and querying services to the distribution model. The key principle emphasised here is that of **polymorphism** in the definition of classes to allow bindings to be flexible yet semantically meaningful. Polymorphism provides mechanisms that support the **flexible sharing** of interfaces and component definitions across similar components. Sharing relations allow the assumptions that a client makes about a server to be expressed at different levels of abstraction.

- the distribution meta-model, models the logic and constraints by which objects are allocated to resources and properties are selected. The distribution model provides a property selection mechanism to allocate resources and select the most appropriate mixture of management services. Once preferences have been resolved, it uses the configuration managers to load the complete application and link in the required runtime support. The key principle emphasised here is that of **selective properties** to allow programmers control when required but to use default policies when unspecified, in order to provide the flexibility to support different **dependability** goals without overcomplicating the programming interface.

- The C++ idiom, is used to implement application classes that can be instantiated to provide the application functionality. An idiom-specific C++ skeleton is generated by the class definition language processor, to be filled with C++. The C++ programming idiom ensures application objects provide a compatible configuration interface to the configuration managers and a compatible messaging interface to other components. The key principle emphasised here is that of **encapsulation** both of incoming and outgoing interfaces to other components and to the system. This hides the internal implementation of components from potential users, including system components, thus isolating volatility in representations and supporting the **evolution** goal and **portability** goal.

- The configuration manager, provides configuration and management services. The configuration manager is built out of system objects which provide configuration services to the modelling layer for configuring a system and use protocol management services of the runtime support system to manage components. System objects may exist in distinct system processes or be linked in-process. The configuration managers also includes the make utility which is generated by the configuration languages. The key principle emphasised here is that of **interpretation** techniques to ensure efficient and safe execution of an abstract specification on a heterogeneous computing network. The key goal is that of **correctness.** This can be taken to include determinance, liveness, fairness and other system properties.

Chapter 7 summarises the techniques that are applicable to each principle in the evaluation framework. The key principles appropriate for each piece of the architecture are shown in Figure 53. These principles and the corresponding techniques are used to characterise the design choices for each piece of the architecture, in the next three chapters.

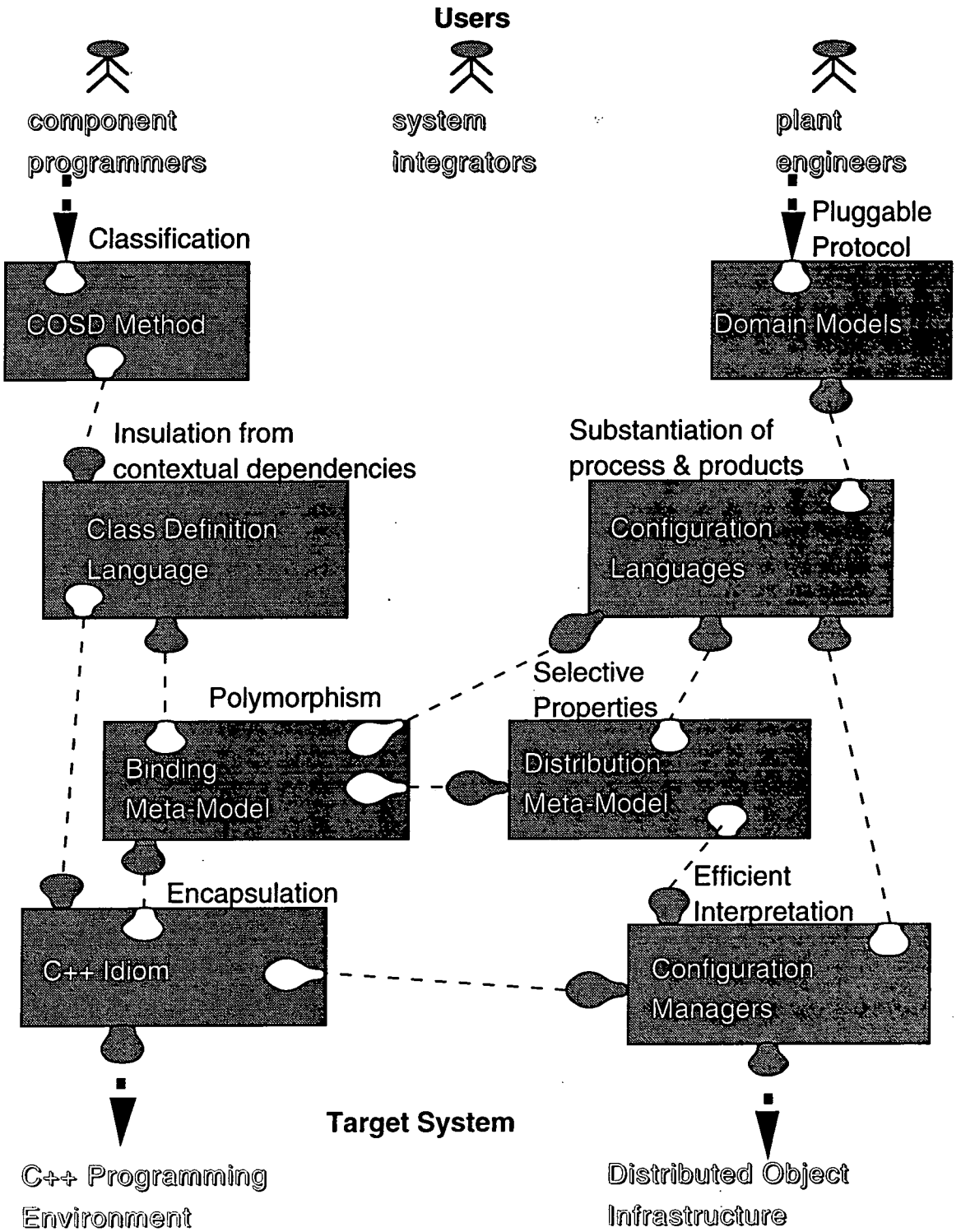


Figure 53 Key Principles Characterising Pieces

7.3 Overview of Conceptual Architecture

This section provides a quick tour of the resulting architecture. The architecture can be viewed as three main layers:

Interpretation Layer

The interpretation layer is responsible for evaluating the high level models into an executable system. It manages the interface to the target system and application code. There are two main components: the C++ idiom used for the internal implementation of components in C++; and the configuration managers which provide runtime support for the dynamic loading and linking of the objects specified in the model layer. The interpretation layer is built on an RPC system which includes runtime services for object concurrency, migration, replication, persistence, atomicity and recovery. The interface between component implementations and the underlying system is hidden from component programmers by stub and wrapper code generated by the tools layer.

Model Layer

The model layer provides an underlying database to store abstract representations of programs and system pictures. This acts as an intermediate target for representing concepts described in the tools layer. It consists of two main parts: the configuration model and the distribution model. Interconnections and instantiations of components are represented explicitly in the configuration model. Policy issues concerning object allocation and management are made in the distribution model. The distribution model models the relations between operational requirements, runtime services and physical components such as nodes, images, devices and processes.

Tools layer

The tools layer provides two types of high level tool to the users. Specification tools describe each component class and populate the model layer with static definitions of object types and ports. Configuration tools describe applications in terms of these component definitions using instantiate and link commands as well as describing the physical plant layout in terms of its components. This populates the configuration model and distribution model respectively.

The breakdown of the architecture into these components is shown in Figure 54

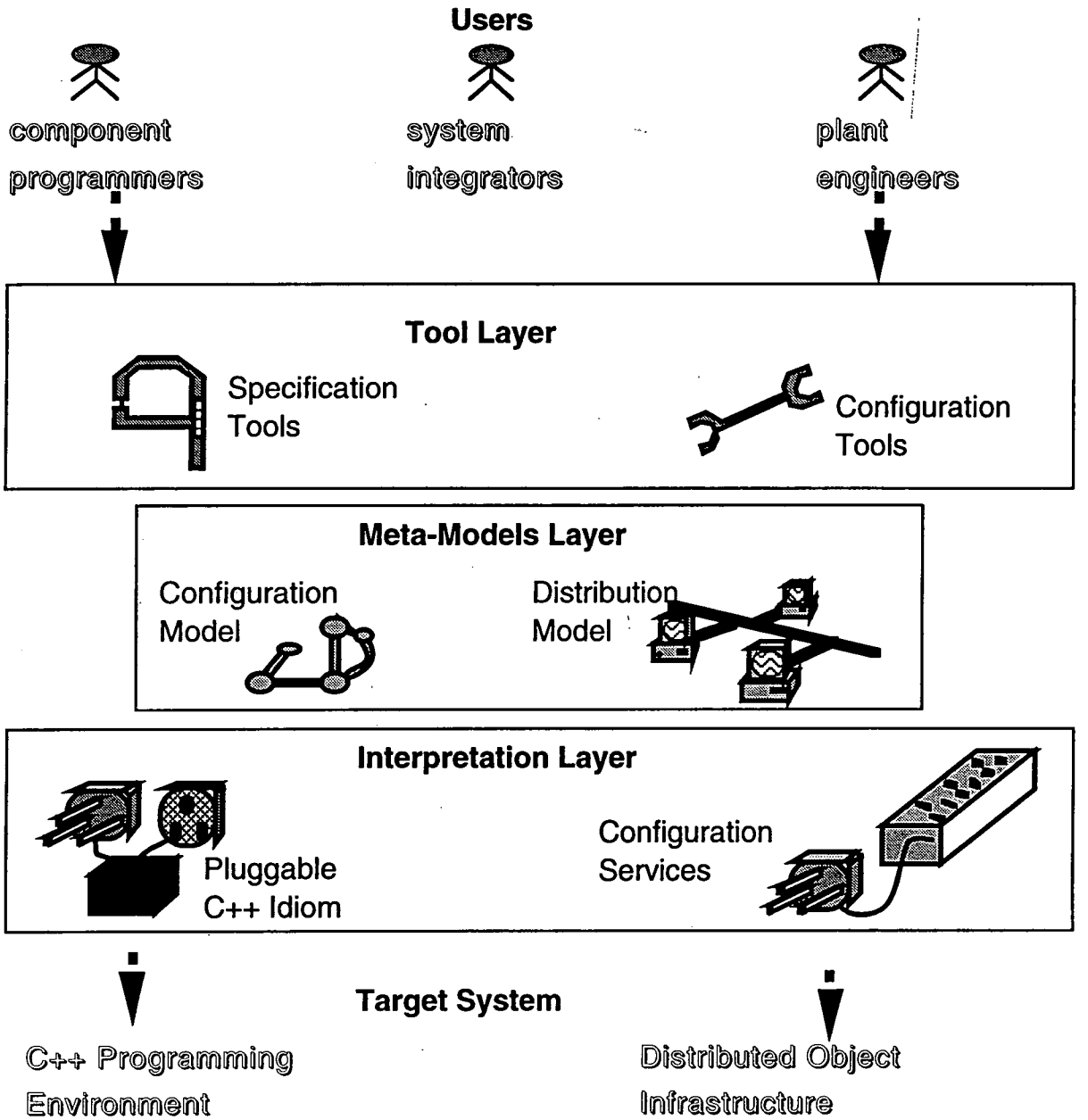


Figure 54 Three layers of the architecture

The C++ programming environment integrates ObjectStore, an OODBMS that includes a C++ compiler, Dataviews, a dynamic graphics toolkit, and MotifApp, an X-based GUI framework.

The distributed object management infrastructure layers ANSAware on a conventional operating system, currently DEC/Ultrix. It provides runtime support for process creation, binding, remote interaction and distributed management.

The three layers of the architecture are described in the next three chapters that take each layer in turn. A more detailed architecture diagram is used throughout and is based on the expansion of this three layer model into constituent parts. The detailed breakdown is shown in Figure 55. The breakdown is as follows:

- The specification tools consist of a class definition language (CDL). The configuration tools consist of a visual editor, a textual configuration language and an environment definition language (EDL).

- The configuration model consists of a type model as well as a configuration model. The distribution model internally consists of three data structures: a requirement graph , a reward graph and a load graph.

- The configuration services consist of configuration managers and UNIX-make facilities used to build the system.

The build cycle is complex. A configuration can be in nine states:

- component construction, where components are being developed.

- editing, where the visual editor is being used to define a logical view of the software as a configuration of component instances.

- allocating, where the policy manager of the distribution model is evaluating the logical configuration against the physical configuration of the system to generate a reward graph determining object allocations and choice of services.

- allocated, where there is a reward graph describing the allocation of the configuration and the configuration is ready to load.

- loading, where the configuration services are interpreting the configuration and constructing/reconfiguring the runtime representation of the configuration.

- loaded, where the configuration is loaded and ready to start. Another configuration may be loading.

- starting, where the configuration is being put into a running state.

- running, where the configuration is executing. This may include monitoring and changes to the physical configuration model that may result in any part of the configuration being stopped and reallocated or reloaded elsewhere and restarted.

- stopping, where a configuration is being put in suspended state prior to re-editing, re-allocating, re-loading and/or re-starting.

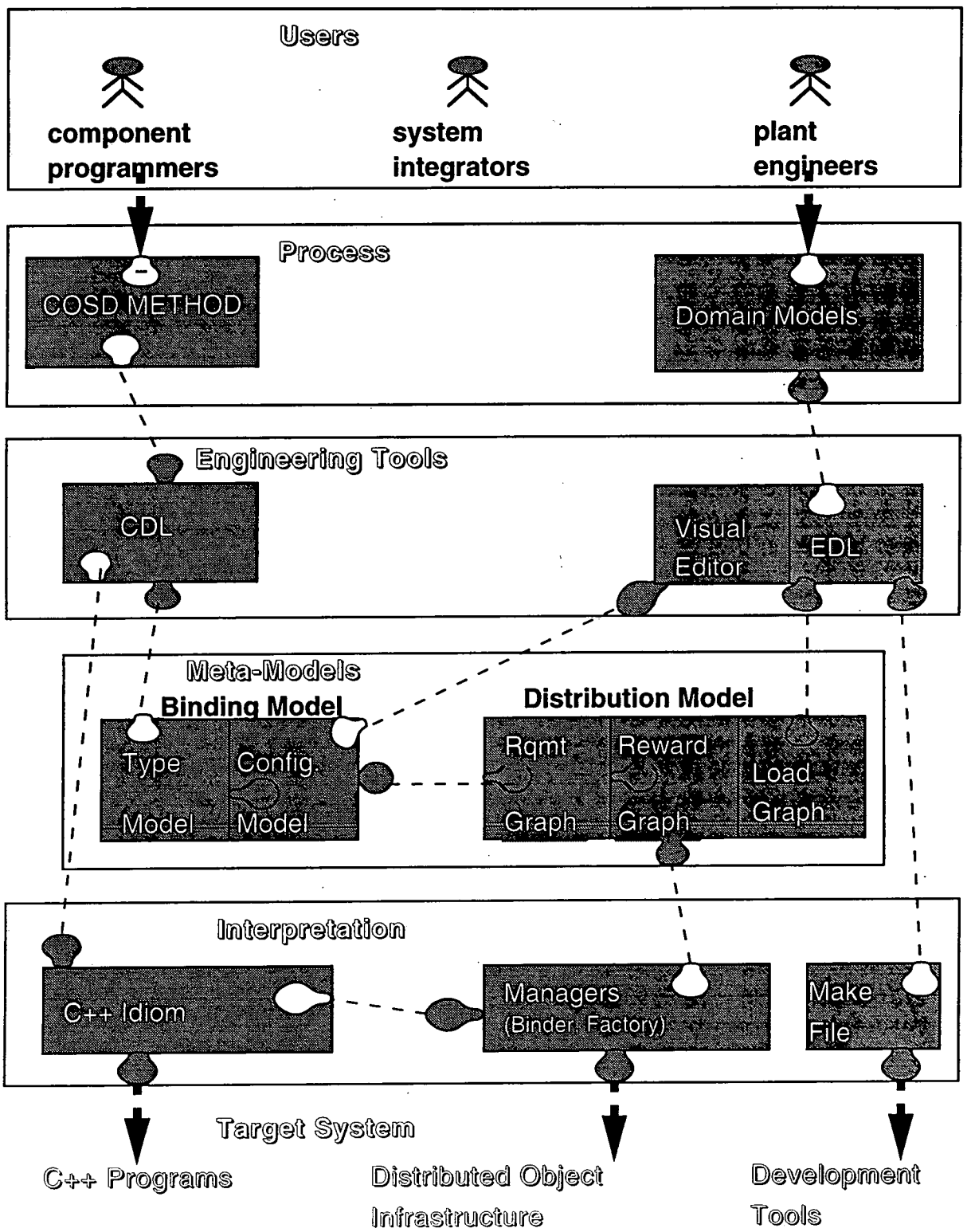


Figure 55 Detailed breakdown of architecture

7.4 Summary of Chapter 7

This chapter has described the high level rationale for separating the programming system architecture into a number of distinct parts. It has done this by decomposing the problem space and solution space according to the abstract goals and principles identified in the evaluation framework of chapter 6.

The ability of a configuration language to relate multiple languages allows the programming interfaces to be customised to a specific task. This simplifies and clarifies the role of the programming tools.

Conventional object oriented programming systems use not only a single language but also a single mechanism, namely inheritance, without any explicit rules to constrain and enforce the development style to adopt to deal with reuse, distribution, or late binding. In a distributed component-oriented environment, this can lead to a complex system in which a programmer needs to address many issues without much explicit guidance.

Our system is manageable because there is a separation of concerns in the architecture and this allows us to define clear explicit roles and rules for each architecture component. The components are described individually in more detail in the next three chapters.

Chapter 8 Interpretation Support for Adaptive Management

8.1 Scope of chapter

This chapter evaluates the design and role of the interpretation layer. The interpretation layer is defined by the components and relationships shown in the dark shaded box in Figure 56. This includes:

8.1.1 Interpretation Layer Components

C++ idiom

Programming idioms are reusable expressions of programming style and conventions. They define how a language is used to solve problems. Whilst syntax shapes a programmers thinking, most of what guides the structure of a program is the styles and idioms adopted to express design concepts.

OpenBase severely restricts the way C++ is used in the overall structure and implementation of a design. It enforces a particular idiom that supports indirect naming through standards-based ports and hierarchical composition through the configuration language and visual scripting tools. This gives a programmer a very specific model for problem decomposition and system composition. The goal is to allow components to be developed in isolation from each other.

Managers

Configuration managers are required to interpret the meta-models and configure the infrastructure appropriately. Primarily this relies on factory services to create components and binder services to connect components together.

Make file

Make files are needed to describe the way the system should be built. OpenBase uses UNIX make supplemented by imake to manage the complex dependencies between generated files and the diverse libraries that must be linked on different machines.

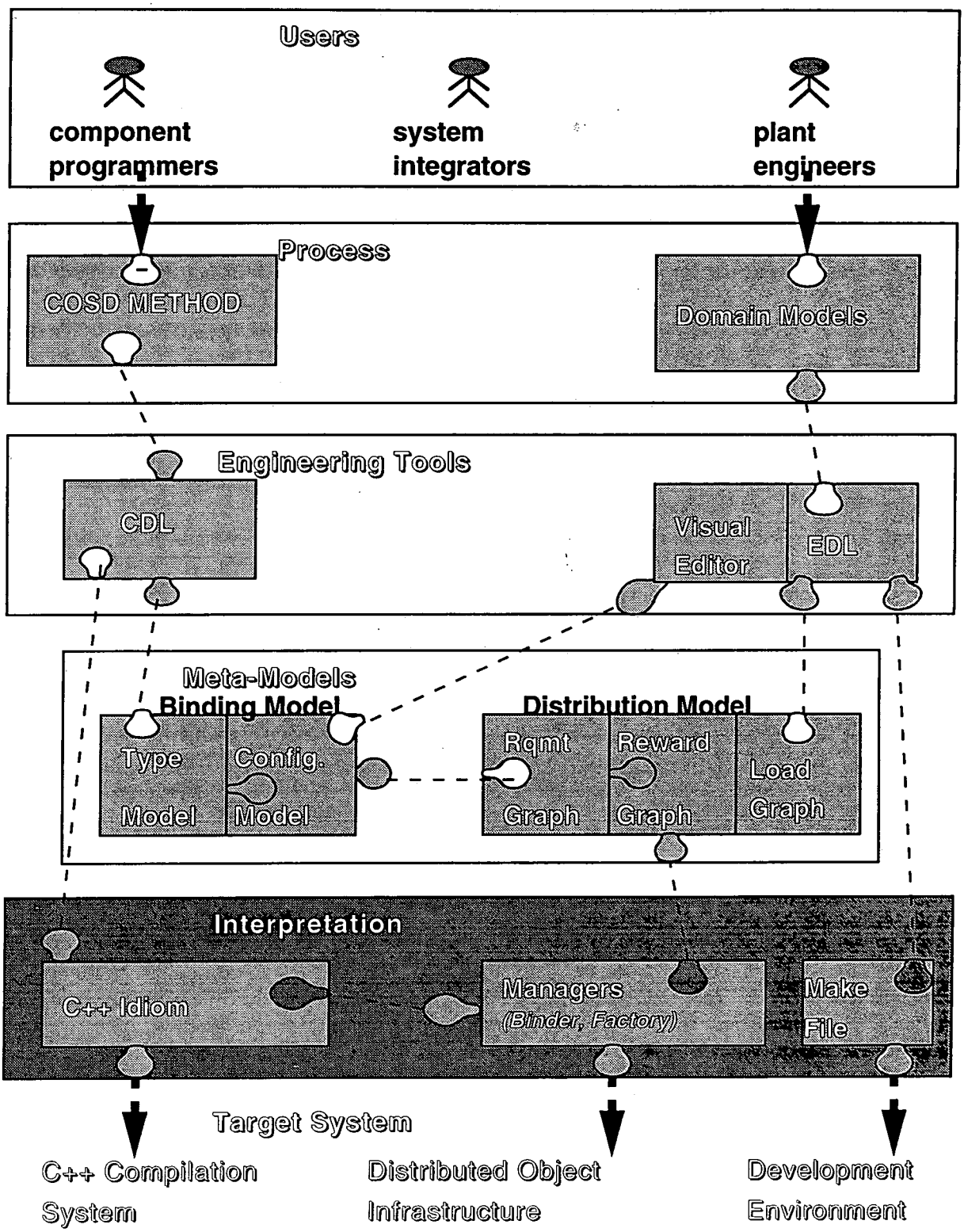


Figure 56 Interpretation Layer Components and Relationships (shown in shaded box)

8.1.2 Relationships between interpretation layer and other layers

Relationship with Class Definition Language

C++ objects are wrapped up in extra stub code which hides the interface to the management protocols. Stubs are generated by class definition language (CDL). They control the interface to other components and to the operating system. This extra code also includes the type-specific element of the configuration services used to create objects and link objects together

CDL also generates skeleton code that further explicates the C++ idiom to be used to implement objects.

Relationship with C++ Compilation System

The ObjectStore C-front compiler is used to compile the wrapped objects. Interfaces to C-based infrastructures are linked using extern "C" statements to avoid C++ name mangling.

Relationship with Environment Definition Language

The Environment Definition Language describes what classes are supported by an image and links the object code for the appropriate configuration services capable of creating and binding together all the objects supported by the image. Objects and link representations in the distribution model are transformed into runtime objects using these configuration services.

Relationships with reward graph in distribution model

There are operational trade-offs to be made between different runtime support services, for example between reliability and performance, availability and currency, dynamism and safety. Management mechanisms carry a runtime overhead. Dependability requirements are met by different options for object allocation, interconnection and interoperation over the network.

The allocation of objects to processes and policies for interaction and interoperation is modelled by the reward graph. On allocating a logical configuration (the requirement graph) to the physical configuration (the load graph) rewards accumulate for different options and are represented by the reward graph. This is described in Chapter 9. The configuration manager objects query the reward graph to select the appropriate interpretation.

Relationship with development environment

The object configuration services are linked into images and the images executed using the development environment of the operating system, i.e. UNIX linker, loader, exec.

Relationship with distributed object infrastructure

The configuration manager uses factory services of the infrastructure to create processes with built-in runtime support for object configuration and remote interaction. These services are provided by infrastructures like ANSAware and CORBA. The configuration manager also uses naming and binding services of the infrastructure to bind these processes together.

8.2 Design Choices

Design choices can be described as instantiations of the design variables identified in the evaluation framework. These are shown in boxes in Figure 57 and Figure 58 which summarise the two most important principles in the design of the interpretation layer, namely interpretation techniques and encapsulation techniques.

8.2.1 Interpretation Choices & Rationale

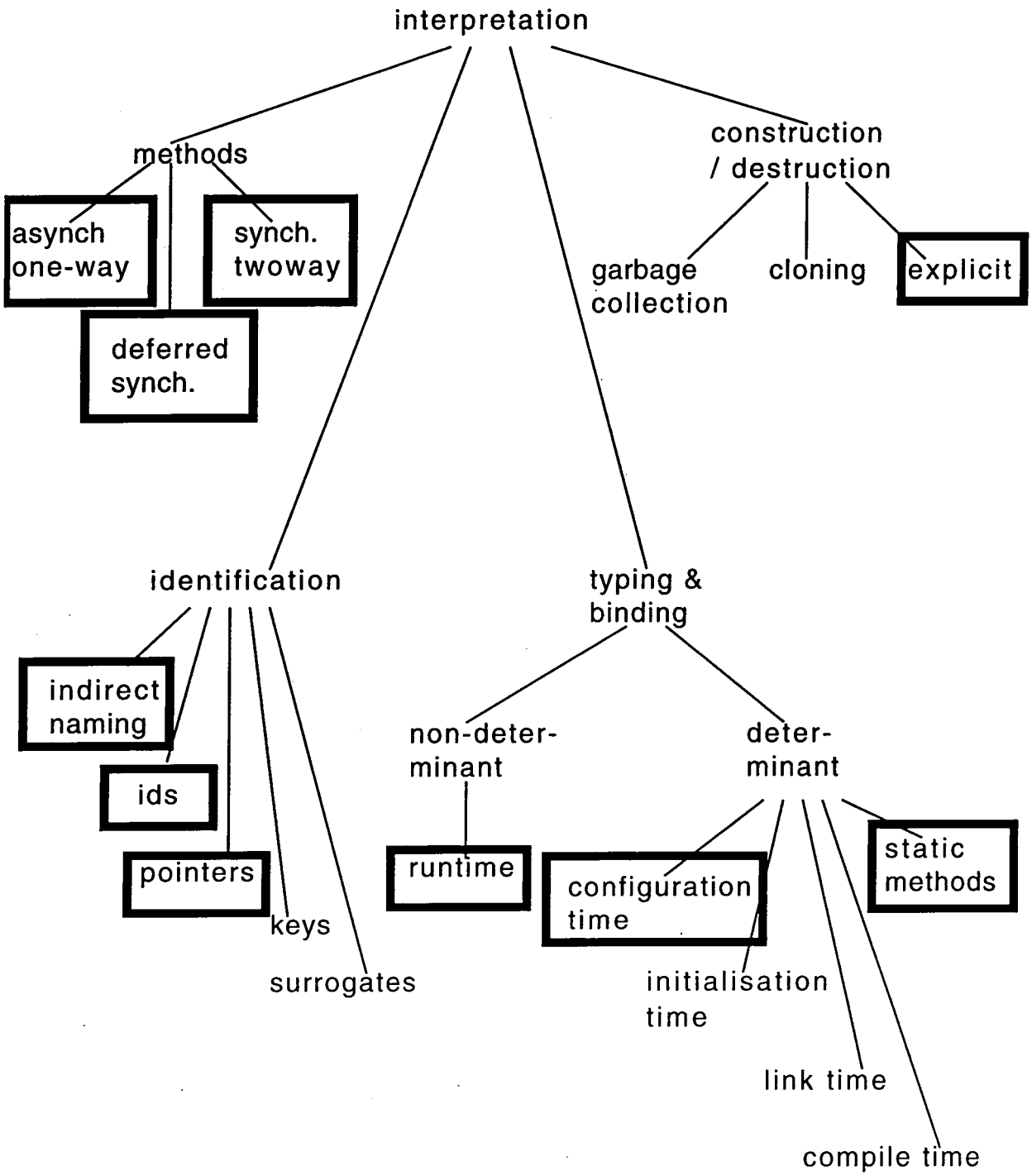


Figure 57 Types of interpretation supported (shown in boxes)

The design choices are as follows:

The RPC system supports the following types of call: synchronous invocations where the client blocks waiting for a reply; asynchronous one-way annunciation messages, and deferred synchronous two-way calls, where the client issues a request asynchronously then later blocks to collect the results. This allows flexibility in the degree of parallism across the network.

The naming scheme relies on *indirect naming* for contextual independence. A client object makes an invocation on an *outport* that is locally named in it own class definition file. Outports also exist as runtime objects and act as locally named reference holders, for runtime references to runtime *inports*. References identify the remote inport. This automatically identifies the host object thus object identities are not required at runtime.

Identities in the configuration model are logical identities and avoid encoded physical addresses. Processes, objects, and ports all have a logical name that is mapped to a *runtime ID value* and a *runtime reference or pointer*. The name service or trader of the infrastructure maps the process name to a runtime reference to a RPC server object. The RPC server object is an infrastructure object that is responsible for delegating calls to and from all application objects in its process. Each RPC server maps port ID values to pointers to local port objects. The RPC server also maps process ID values to references to remote RPC server objects in other processes.

This scheme is optimised for connected protocols since connection patterns are known in advance in the configuration model. Binding occurs at *initialisation time* or *reconfiguration time,* when a configuration is loaded or reloaded respectively, as opposed to runtime. Binding information is cached locally to where it will be used. Binding between ports therefore incurs no runtime binding overhead, other than that of a normal virtual C++ call made by an inport on it's host object. The underlying RPC service of the infrastructure determines the actual form of the runtime connection management policy. For example, an ANSAware implementation only maintains a single connection between processes at one time and must reuse the binding information to reconnect every time a new server process is contacted.

Type checking occurs at *configuration time* , for example when a composite is defined using the visual editor. The visual editor will not allow incompatible ports to be linked together and will not allow the system to be allocated with unbound mandatory ports. Thus the system is determinant, i.e. a runtime type error cannot occur, provided the type model is kept up to date and is consistent across the system. The visual editor also checks for cyclic nesting of synchronous calls that can cause deadlock.

Explicit construction/destruction is specified by configuration commands of the visual editor to create and destroy objects. Currently these are made by the application engineer who configures an application using the visual tool or configuration language. This implies a static runtime configuration is loaded. Reconfiguration currently requires reinterpretation of the configuration model and is too expensive for highly dynamic object oriented applications. Reinterpretation of a configuration is currently intended to support coarse-grained dynamic changes such as application evolution or process migration to recover from node failures or to relieve highly overloaded nodes. Reconfiguration requires a composite to be suspended.

This does not mean the structure is entirely static. Local C++ objects that are used to implement a component, can be created and deleted dynamically.

Runtime interpretation of configuration commands is planned for the future. This will allow the programmer to write scripts of configuration commands as a method that dynamically creates and links objects.

Associated with construction/destruction is initialisation/finalisation. The Class Definition Language generates an attribute data editor dialogue for each object that can be used by the application engineer to define initial values for the attributes of an object defined in the configuration model and these values are read by the interpretation layer to initialise objects.

8.2.2 Encapsulation Choices and Rationale

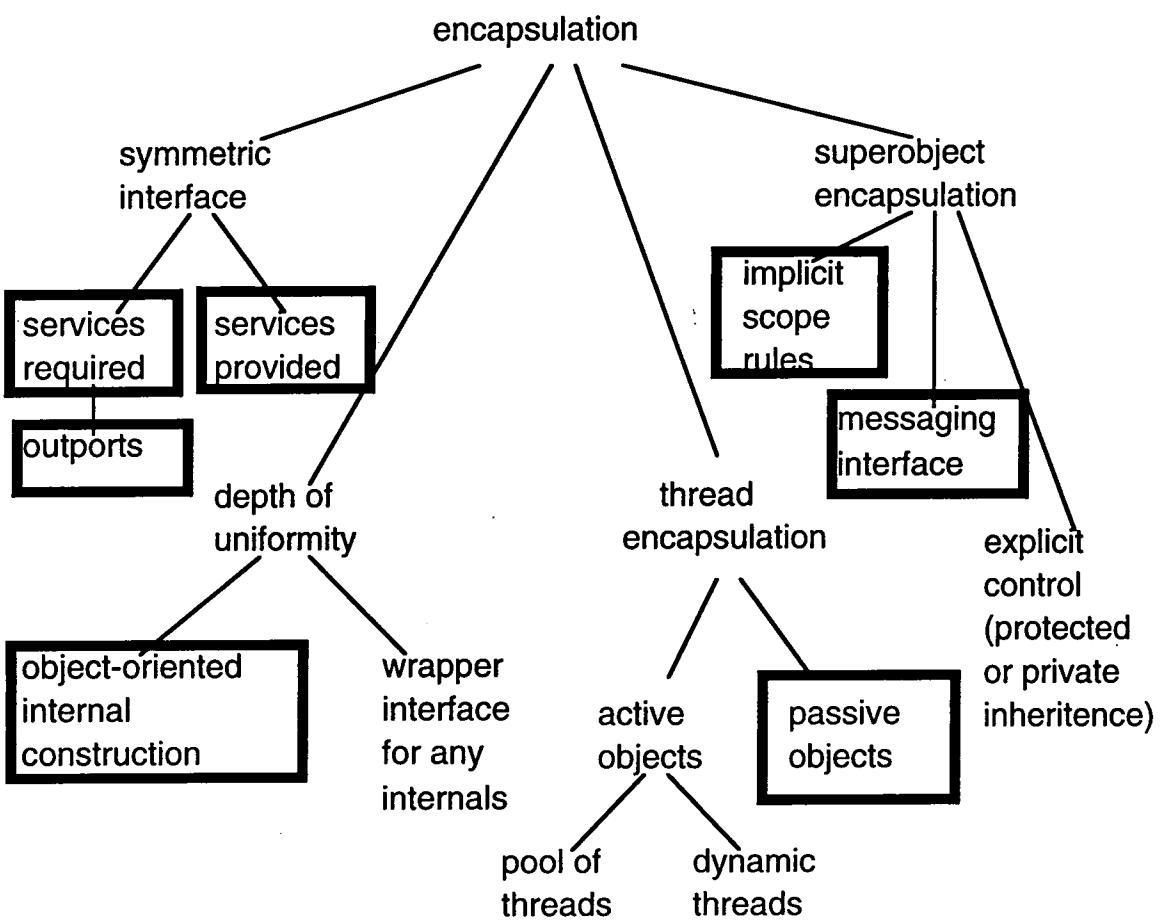


Figure 58 Types of encapsulation supported (shown in boxes)

One design constraint imposed by the sponsors was that the implementation language should be C++ based. The absence of C++ bindings in integrative standards like CORBA, DCE and ANSAware at the time of development and the total absence of hierarchical composition support made it difficult for us to exploit these infrastructures at a high level. Instead it was decided to provide our own IDL, actually called CDL, and our own C++ mapping that does lend itself to hierarchical composition. CDL generates extra code that encapsulates an object implementation and maps the C++ implementation onto the infrastructure support.

In our CDL a component identifies both the *services required* by the component as well as the *services provided*. Thus encapsulation is symmetric between the incoming and outgoing interface to a component. *Outports* are defined in CDL and used in object implementations in place of object references.

Because C++ is used internally, the *internal construction of a component is object-oriented.* C++ objects may be used internally in the normal way to allow further levels of encapsulation, but they can not be referenced outside their address space.

Objects defined in CDL are *passive objects*, i.e. they are inactive until invoked and threads are allocated by the caller and control returns to the caller when a call completes. OpenBase objects do not encapsulate their own threads.

An object can inherit from another component using C++ inheritance with the *implicit scoping rules of C++* being applied. Ports may be viewed as members, therefore a subclass includes all the ports of its base class and can access them using C++ scoping rules. Virtual functions can be used to (re)implement a method that is associated with an inport defined in a base class.

An object may also be configured in a composite with another object and this composite be saved and instantiated as a class. The composite derives behaviour from both its embedded objects by exporting their ports in its interface. If the embedded objects are semantically related as generalisation-specialisations, the composite can be viewed as consisting of a superobject and a derived object. It defines the interface between the derived object and superobject by specifying links between them. A *messaging interface* is used to define the superobject encapsulation primitively at the level of individual members.

8.3 Exploitation of Integrative Standards

An evaluation of different integrative standards was made as part of the MSc research. This evaluation explored various design options for a flexible architecture design that exploits products that conform to the integrative standards but also isolates higher level components from volatility in the choice of product, so that dependencies on a particular product are not critical. The evaluation looked at OSF/DCE, OMG/CORBA and ISO/ITU ODP RM (exemplified by ANSAware) to compare them as alternatives and to evaluate the potential for having an approach that is portable across these standards.

The reasons for considering these standards include:

- One design goal for OpenBase is to be able to integrate and interchange different third party architectural components that provide the runtime support for objects. This allows the architecture to evolve as new products emerge or to make pragmatic trade-offs between existing mechanisms to meet requirements for availability, cost, changing requirements or to meet diverse application needs. To do this we must understand the standards to which third party components conform.

- CORBA and DCE products are now supported by many main vendors and there is a fair amount of commitment by the vendors to supporting these technologies in the future. Hence it is important to consider them to future-proof our approach, to demonstrate compatibility with mainstream object developments and standards and to be in a position to exploit additional runtime support provided by mainstream products.

- Despite the existence of published specifications for CORBA, ISO ODP and DCE, there is still a great deal of confusion over how they impact on software development. One result of the evaluation has been explaining the concepts behind these standards, see Appendix B.

- The integrative standards do not address configuration tools hence OpenBase must layer our own configuration tool standards on top of them. The result of this is that the integrative standards are not visible to OpenBase users, only the system programmers of the internals. This means standards-based applications are not directly compatible with OpenBase applications. However by including the standards in the internals of the architecture, the gap between OpenBase standards and the integrative standards is narrowed, making it easier to provide gateways that interface to applications written to these standards and simplify support for porting OpenBase and for interoperability of OpenBase across platforms.

- The integrative standards address many low level problems that are shared by OpenBase at the implementation level, such as dealing with data type encodings. By layering OpenBase on top of a standard, much research and development effort may be saved, allowing Prism to focus on higher level issues.

- The alternative to building on existing RPC systems would be to use UNIX networking programming services like sockets. Sockets are effective if efficiency is important. Standards-based RPC systems offer higher level functionality that makes it preferable if productivity is important or if portability is important. In the initial prototype, the latter are more important. Prism may well re-implement the design in the future using a socket based transport service.

The evaluation is written up in Appendix B. It is recommended that the reader consider the description of these standards that is presented there before looking at the detailed design of the interpretation layer presented in the next section. In particular, the reader should be familiar with the use of the following ANSAware terms: IDL, factory, node manager, stubs, skeletons, trader, name service.

Appendix B provides a simple functional framework to describe the standards. The programming task is broken down into six steps corresponding to six areas of functionality that a programmer must deal with. Each step is illustrated with example source code taken from a simple banking scenario. These steps are:

- Stub Generation, stubs encode/decode request and replies. They are typically generated by providing a description of an objects interface in an Interface Definition Language.

- Source Code Implementation, the functions or methods in the interface need to be implemented, typically in C or C++.

- Server Registration and Naming, the server needs to register itself with a name that is used by clients on binding. This can be initialisation code within a server or a command line utility that the configuror uses.

- Locating and Binding, before using a server, the system must find and bind the server to the client. A binding service is used by the client.

- Activation and Failure Handling, to use a server it must be actively running. This can be a configuration exercise or automatic (re-)activation of inactive servers can be built into the system. The system should also notify long term server failures.

- Synchronisation and Request Processing, the client must issue a request to the server. The system will deliver the request and return any reply data.

The rest of this section describes some conclusions that were made about how OpenBase may exploit these standards.

At the time of evaluation, the standards were too immature to rely exclusively on one implementation for runtime support. It was decided that we should minimise dependencies on any one standard. Hence the evaluation had to identify what was common and consider techniques that could be used to achieve portability between the standards. These techniques include layering, virtual interfaces, bridges, abstraction, and system or meta-level programming.

Integration of CORBA with ANSAware or DCE is not trivial. It was decided to exploit only the most fundamental common subset of their features to ease migration between them. This approach suffers the least-common denominator effect. Much added functionality needs to be implemented in a proprietary manner. However it avoids resolving the differences that were observed between the standards and those between the configuration programming paradigm and the extended client-server paradigm of the standards. As the standards mature and evolve into fuller specifications, Prism can commit to one and exploit it at a higher level. Alternatively Prism may implement the runtime support themselves and eliminate dependencies on standards based products entirely.

The rest of this section provides the rationale for concluding that integration of different standards and high level exploitation for configuration programming are not trivial.

The relationship between OSF DCE, OMG CORBA and ISO ODP is not strict layering, since CORBA adds entirely new concepts to DCE such as dynamic invocation and ISO ODP (as exemplified by ANSAware) adds new concepts to CORBA such as viewpoints. The relationship is more one of extension both as an abstract layer and in terms of runtime support. This is shown in the following diagram.

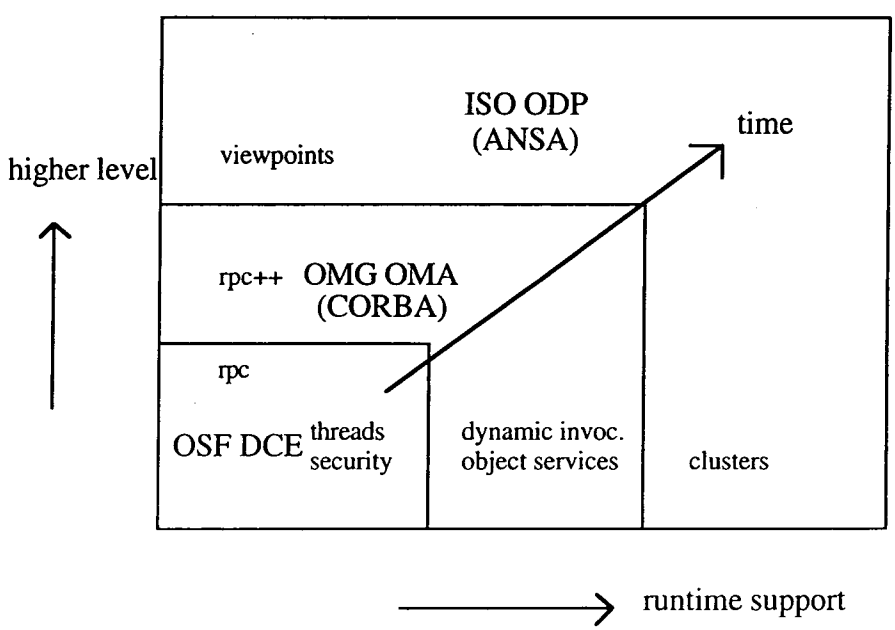


Figure 59 Non-Layered Architecture Relationships

ANSAware provides a partial implementation of much of the ISO ODP standard. This implementation: is open, i.e. the source is available; is rich in features, for example including group interaction protocols, and is relatively mature, compared to DCE and CORBA products. For these reasons, ANSAware was selected as the best platform for the first prototype in 1992. However it is expected that commercial releases of OpenBase will want to build on the more commercially accepted DCE and CORBA standards.

It is expected that CORBA implementations will eventually provide the best platform to launch the product on. The OMG will eventually provide a rich set of services for managing objects that will overlap the features currently supported by ANSAware.

The areas of functionality that CORBA technology may save research and development effort include:

- activation, starting the server if an instance of it is not available
- request dispatch (and scheduling if multi-threaded)
- parameter encoding, also called marshalling.
- message transfer and delivery, shipping the request from the client to the server
- error/exception handling, if the network fails or the server cannot be started,
- internal connection management (hidden behind binding mechanisms)
- synchronisation, between the client and server
- security mechanisms, to prevent unauthorised object manipulation.

In addition, extra functionality will be provided by object services. This includes the following, in rough order of likely availability date:

- name services - to get references to objects based on text names
- events - notification of events to interested parties
- object lifecycle - allowing objects to be created, copied, tested for equivalence and deleted
- persistence - allowing an object to retain data on stable storage
- relationships—the capability to define relationships between objects to retain the object handles of linked objects for graphical tools
- concurrency - concurrent access to one or more objects
- transactions - atomic execution and recovery of one or more operations
- externalise - object externalisation and internalisation
- time - synchronisation of clocks in a distributed system
- security - protected access

- licensing - licensing management
- properties - allowing extra dynamic information to be attached to an object
- query - predicate based operations on sets or collections of objects
- trading - matching of provided service to the service needs of a client object
- change management - identification and consistent evolution of objects including version and configuration management
- data exchange - exchange of some or all of visible state between one or more objects
- replication - group interaction among objects
- archive - mapping between archive and backup object stores
- backup/restore - backup and recovery of objects
- installation and activation - mechanisms for distributing, activating and deactivating and relocating managed objects
- operational control - controlling the dynamic behaviour of objects

The approximate time profile for CORBA object services is illustrated in Figure 60.

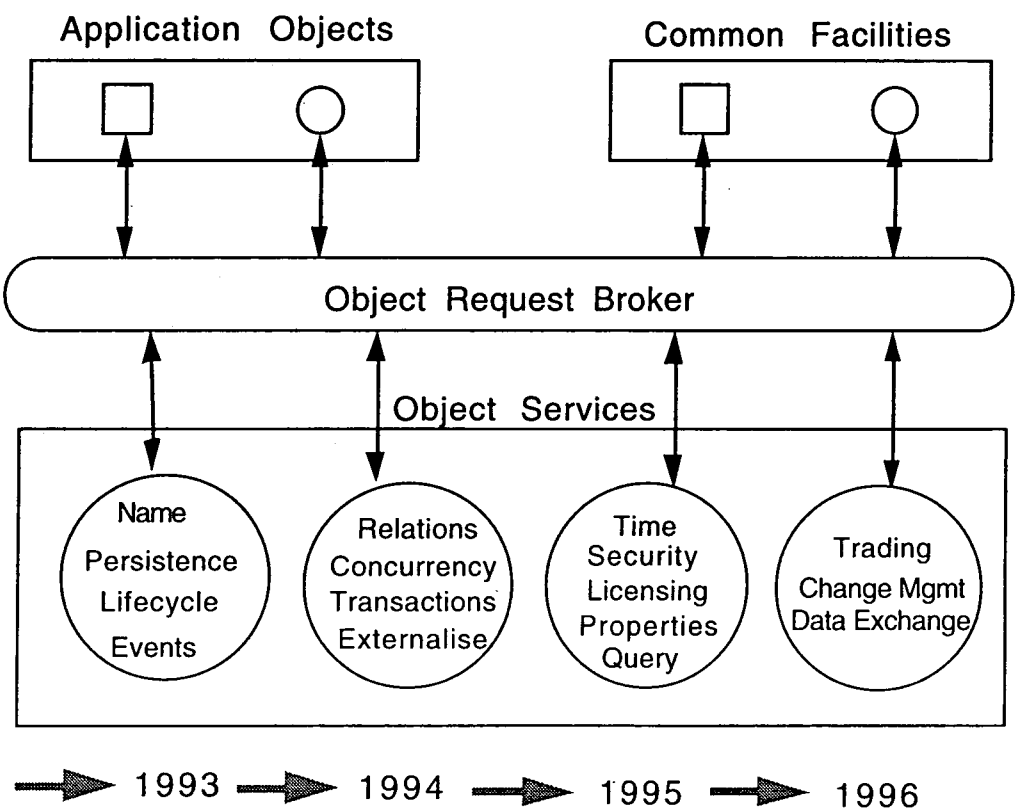


Figure 60 Time Profile for Object Services

The rest of this sub-section focuses on comparing CORBA and DCE and evaluating the techniques that can be used to integrate CORBA and DCE, in order to support the longer term view that CORBA is likely to provide the best infrastructure support and CORBA can not be easily integrated with DCE.

Some people have referred to the relationship between DCE and CORBA as like that between C and C++ (i.e. CORBA has been called RPC++). This is because DCE is not object oriented: it has no interface inheritance. An object oriented style can be superimposed on DCE in the same way as an object oriented style can be superimposed on C.

As a result of it's lack of an object model, DCE is limited in what runtime support for object management it can provide. The OSF intend to evolve DCE to accommodate CORBA object services by adding an object model. However this will inevitably take time

At a more practical level, the following differences can be observed:

- CORBA IDL and binding mechanisms are easier to use than DCE's low level interface that require several calls to register interfaces.
- The low level interface means there is much work to be done to support fine-grained models. DCE is intended for process-oriented servers. CORBA's higher level interface is more productive and can support transparent collocation that makes it appropriate for finer grained object models.
- CORBA IDL is more appropriate to C++ and may even generate C++ headers. DCE is used in C++ in the same way as C and doesn't support interface inheritance.
- CORBA libraries can be linked selectively and provide a smaller system interface whereas DCE generates a large image from a large interface. DCE consists of some 2 million lines of code, mainly C.
- DCE provides security, threads and directory services. Most ORBs lack support for this yet and all lack OMG's blessing. However DCE is not object oriented and is therefore limited in the long term as to what object management services it can provide.

As a result of these differences CORBA is recommended for the longer term. The next few paragraphs look at overlaps, incompatibilities and coverage differences that limit the applicability of integration techniques like layering, abstraction, virtual interfaces and system programming and suggest integration of both CORBA and DCE is difficult.

Some parts of DCE are lower level and can easily be layered on by CORBA implementations:

- DCE RPC, the procedure call subsystem of DCE can be used as the transport service for a CORBA implementation.
- DCE NCS, the data encoding standard.

In addition, system programming techniques can be used to integrate orthogonal functionality that does not overlap.

- DCE is very low level. It is therefore possible to program at the system level easily to customise its behaviour. However this requires a lot of work.

- CORBA products like Orbix support higher level interfaces for system programming such as stub subtyping, filters.

However, there are incompatibilities in overlapping areas and differences in coverage, as described below, that make layering and system programming difficult or restrictive.

Some parts of DCE overlap with CORBA but are incompatible, making abstraction across both difficult, including:

- both provide an IDL. The IDLs are incompatible. Unification of the two distinct IDLs is difficult.

- both provide a name service and binding services.

  DCE only names and references interfaces for server processes not fine-grained objects within a process. DCE names are resolved to the physical address of the server process before invocation. Support for multiple objects within a process is difficult in DCE.

  CORBA supports location-independent object references that can be passed across the network and allows multiple objects per server process.

  Layering is possible if object references are implemented in two-parts: the server is resolved using a location service and DCE cell directory service to establish a direct channel between server processes; the object name within a server is resolved using the CORBA name service.

Some parts of DCE are extensions to CORBA that have not yet been standardised as Object Services. Differences in coverage restricts the suitability of layering, virtual interfaces and bridges that suffer from the least-common-denominator effect.:

- DCE Threads

- DCE Security

Some parts of CORBA are extensions to DCE:

- CORBA Dynamic Invocation Interface, DCE enforces static link-time dependencies between a client and a server's stubs. a call-level interface for the dynamic interface is difficult to implement using a static compilation model such as DCE RPC. DCE has no interface repository holding information about stubs. This support needs to be added explicitly.

- Factory and lifecycle services, to activate processes and create objects.

- CORBA inheritance and object model. DCE does not support interface inheritance or objects. In this sense CORBA may be viewed as DCE++.

- DCE does not provide a type repository nor trading mechanisms. This includes the Property Service, to associate dynamic properties with objects, and the Association Service, to model associations between objects.

- DCE does not support deferred calls where the response is collected later. This is supported by the dynamic interface. DCE threads can be used to spawn a request handler thread to make the synchronous DCE call. The original caller later blocks on the request handler when collecting results.

- DCE is not internally object oriented and has no notion of objects acting as intermediaries.

The rest of this chapter describes how the interpretation layer is designed in an early prototype based on ANSAware. The key goal of this prototype is to explore and prove design alternatives. This prototype seeks to expoit only the common basic functionality provided by all the integrative standards like OMG CORBA, OSF/DCE and ISO ODP - to explore the feasibility of migrating the architecture across the standards and defer commitments until the standards mature.

8.4 Design of Interpretation Layer

This section describes the design of the first prototype for the interpretation layer. This has a simpler design than the final prototype. The first release is described rather than the final prototype design because services are combined in this release and this makes the description more concise and simpler. It is intended to illustrate the way the interpretation layer is implemented. The same principles were employed in the final prototype. Furthermore the first prototype was a solo activity right down to code and testing, whereas other Prism staff were involved in the final design.

The interpretation layer provides the mechanisms to generate the runtime support for OpenBase objects. This consists of four key layers:

- the object layer, application objects are implemented in C++ using a hierarchical composition idiom that is supported by skeleton code generated by the class definition language (CDL) compiler.

- the stub layer, objects interact by calling methods in outport objects that are nested in the client object and being called by inport objects nested in the server object. These port objects provide the stubs for encoding and decoding C++ method invocations and are generated by the CDL compiler automatically.

- the configuration layer, connections between stubs are managed by link objects that are provided by the OpenBase runtime library. Link classes may be specialised to support different interaction policies, by performing different reflective computations on an invocation. The choice of link object is determined by specification attributes in the distribution model. Other configuration objects are also provided in the runtime library to interpret the distribution model. Node configuration objects create and link processes on a given node. Process configuration objects create and link application objects within processes.

- the manager layer, the underlying distributed object infrastructure, i.e. ANSAware, is encapsulated in a portable interface provided by manager objects. These objects map C++ calls onto C calls of the infrastructure, including calls to the RPC service and to any management API.

The design of the interpretation layer is explained in the next six sections that cover the six areas of functionality that have already been used to compare integrative standards. This highlights the difference in approach between OpenBase and integrative standards - namely that configuration dynamics are determined by interpreting a structured high level model rather than through runtime negotiation between components.

In OpenBase, bindings between service providers and service users or process activation and passivation are resolved in the distribution model. Dynamic configuration is supported by interpretation commands as shown in Figure 61. Interpretation queries determine most of the functional steps: linking of stub objects, naming and registration, locating and binding, server activation.

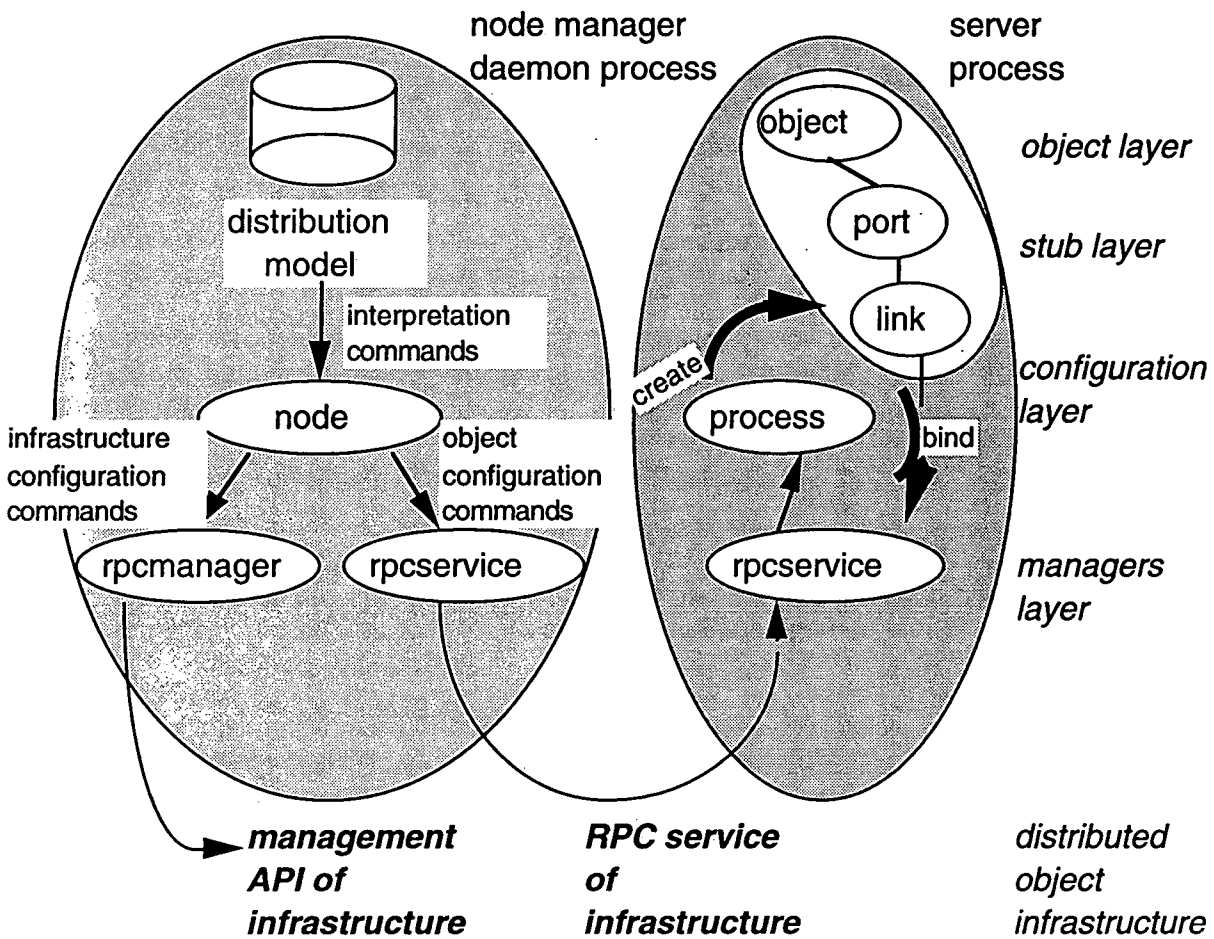


Figure 61 Dynamic interpretation of high level model

This contrasts static models like DCE or server-oriented dynamic models like trading, narrowing and server repositories. The interpretation of a complete model for the software configuration allows coarser grained management policies that take advantage of composite properties and local domain knowledge.

A configuration can be in eight states:

- editing, where the visual editor is being used to define a logical view of the configuration.

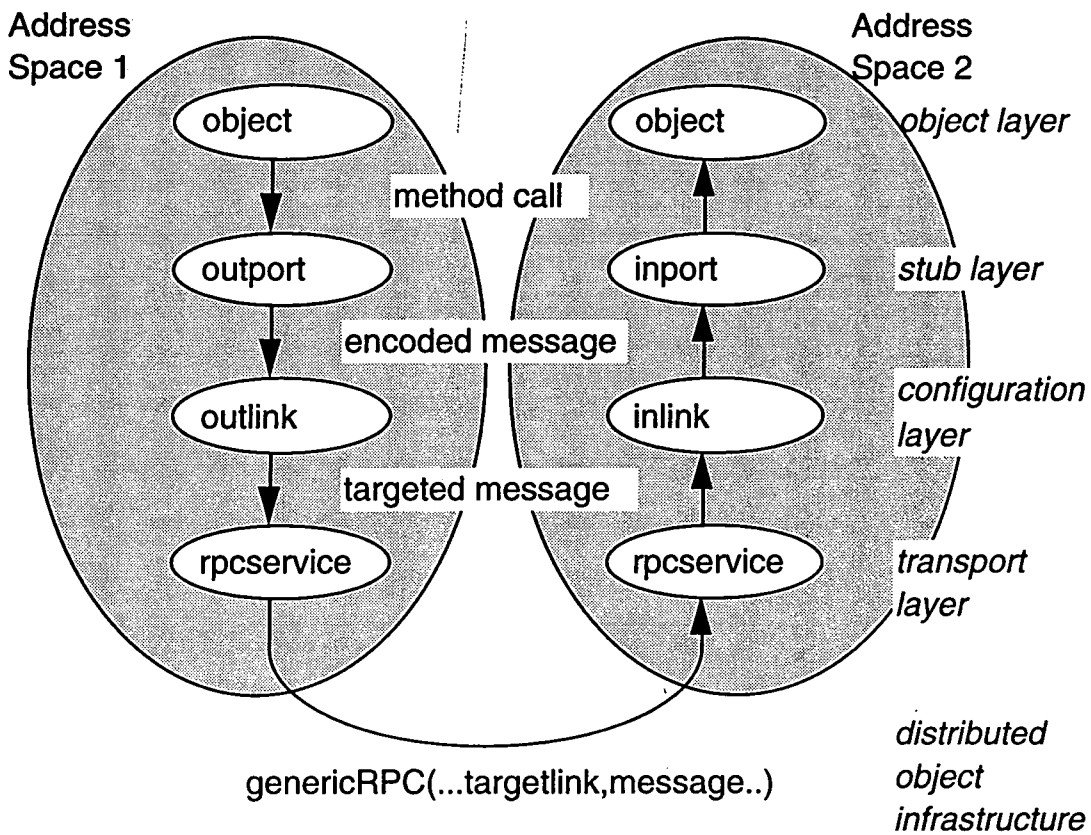


Figure 62 OpenBase layers involved in method invocation

8.4.2 C++ Idiom for Application Method Implementation

As well as generating inports and outports, the Class Definition Language also generates C++ code skeletons that have consistent method names and signatures with those expected by the inports. These skeleton consist of a class header consisting of data declarations, operation declarations and port declarations, as well as port implementation files. The component programmer must provide an implementation file to implement each operation (i.e. method) using the OpenBase C++ idiom. The relation between the operation/port declarations and the implementation files, is illustrated in Figure 63.

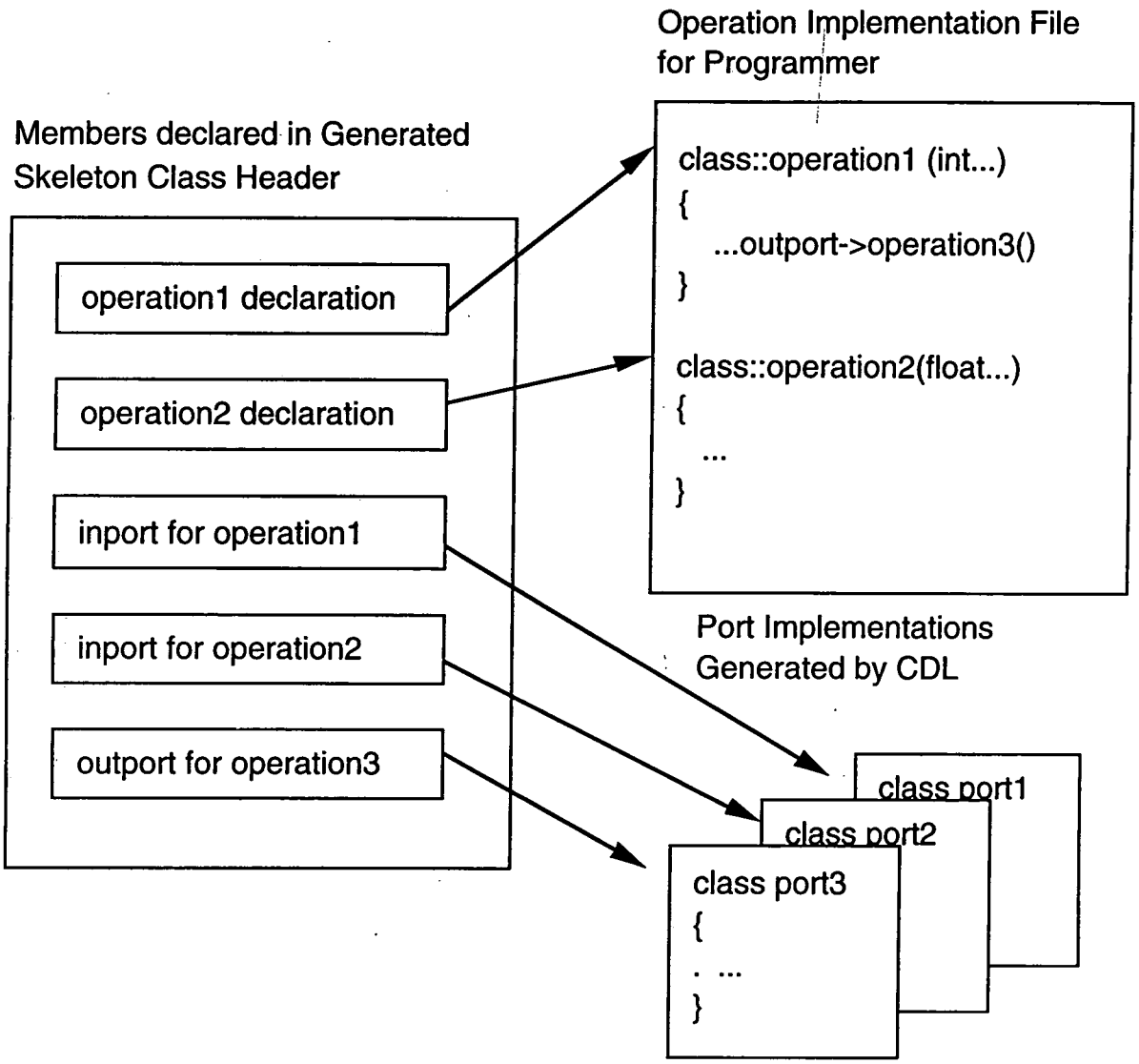


Figure 63 Implementation of Skeleton Code

The OpenBase C++ idiom uses AT&T C++ with the following restrictions on method implementation :

- Any distributed objects must be referenced and invoked via outports declared using the Class Definition Language. A dynamic untyped port interface is provided as an alternative to using statically typed ports but a full discussion of this is outside the scope of this thesis.

- Local C++ objects may be declared, instantiated and invoked in the normal way either on the stack, heap or as a class member, but pointers or references to these objects can not be passed as arguments to inports or outports.

- Outports may be passed as arguments to allow dynamic connection. The semantics of this are that an outport in the target is dynamically bound to the same inport. The marshalling code must create the binding dynamically.

- The ObjectStore C front compiler is used to support persistence. This is integrated with CDL. The CDL compiler generates a metaclass object that supports persistence services. A full discussion is outside the scope of this thesis.

- The ObjectStore C front provides no support for exceptions. OpenBase provides its own exception mechanism. A full description of this is outside the scope of this thesis.

8.4.3 Naming and Registration

Server objects are named when object instances are declared using the OpenBase configuration language or the visual scripting tools, for example by dragging and dropping an instance into a composite.

Server processes are named when composites of application objects are allocated to them using the editor. In reality a unique processID value is usually generated rather than a textual name.

The runtime name of an object consists of these two parts. The process name is used to address the rpcservice object and the object name is used to address objects within a process. The mappings between names and physical addresses is held in two places:

- The process configuration object that is embedded in each process to provide configuration services for objects also caches a catalogue mapping between object names and object pointers.

- The name service of the infrastructure maintains a mapping between process names and infrastructure references to rpcservice objects.

Each rpcservice object includes a pointer to its local process configuration object, and can be used to find the object pointer on binding. Hence a pointer to an object can be obtained from a name in three simple steps: lookup of process name in distribution model; lookup of rpcservice object in name service ; and lookup of object pointer in local process configuration object.

A pointer to an object is required for adding new links on configuration. However pointers to link objects are sufficient at runtime in order to pass messages via ports to objects. The rpcservice generates linkIDs when link objects are bound and maintains a mapping from linkIDs to pointers to local link objects. The binding service passes these linkIDs across the network so that a request message can address the correct inlink object and a reply message address the appropriate outlink object. Link objects encapsulate the linkID of the other end of the link. The linkID is added to the message body by the link object and read out of the message by the target rpcservice in order to lookup the correct link object

The rpcmanager object encapsulates the name service of the infrastructure in a portable C++ interface in the same way as the rpcservice object encapsulates the generic RPC service. In the prototype, the ANSA trader is used. The rpcmanager maps a portable C++-based interface onto the distributed C-based interface of the trader. It also provides an ANSA interface to the management API for the rpcservice object.

For efficient interpretation and simplicity in the prototype, lookup, creation, registration and linking are combined in a single link evaluation action. There is a process configuration service for managing creation/bindings to rpcservice objects and an object configuration service for managing creation/bindings to application objects. The relevant names are cached locally to where the link evaluation action takes place. A lookup is performed prior to linking to determine if the process or object needs to be created and have it's name registered in the cache or whether it is already cached and therefore already exists.

The creation of objects and the registration of the object name in the process object is automatic when evaluating an object configuration command for an object that does not already exist. The creation of processes and the registration of the rpcservice in the name service is automatic when evaluating a process configuration command for a process that does not already exist.

A configuration is interpreted and loaded by visiting all links in the distribution model and performing navigational queries on the associated objects to find object names and process names. The distribution model is stored in ObjectStore which supports efficient navigational access. By combining evaluation actions, fewer queries and messages are needed to access and interpret the model, than if we iteratively created all processes then all objects then all links as this would require revisiting objects for each link.

In the final prototype of OpenBase distinct services are provided for object creation, object initialisation, object finalisation, object destruction, object linking and object unlinking. These services are designed to be idemnipotent, i.e. can be called several times without multiple effects, by combining them with automatic lookup and registration/de-registration. This allows navigational interpretation of composites in the distribution model without worrying about revisiting elements.

The registration of a server process is shown in Figure 64.

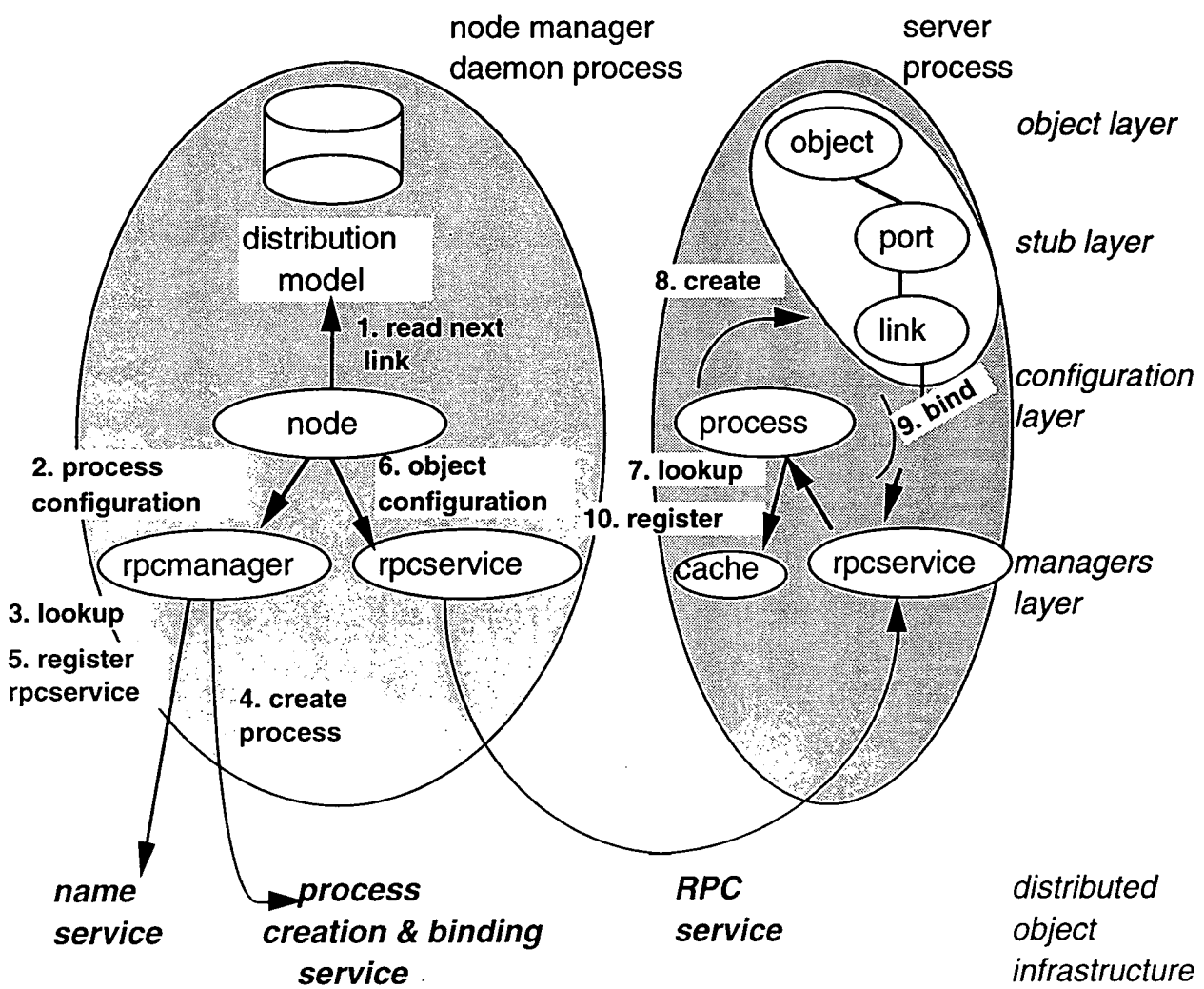


Figure 64 Registration of process name and object name

In the case of links between nodes, the node object forwards configuration commands to the daemon on the required node. The node name for the process can be obtained from the distribution model. Daemon processes have a well known name that is registered in the name service automatically and can be found from anywhere. ObjectStore supports distributed database access and the name service of the infrastructure is a distributed name service so both can be accessed from anywhere, making interpretation of distributed configurations easy.

8.4.4 Locating and Binding

In the prototype, lookup, creation, linking and registration are all combined in a single configuration service for objects. This actually consists of two services: a linkout service and a linkin service. A linkout message is sent to the client object and the client forwards a linkin message to the server object. This is shown in Figure 65. These services pass the linkIDs for each end of the link to the other end so that the destination link object of a request or reply message can be identified to the rpcservice object.

The client is found by using its process name to lookup the rpcservice in the name service, by calling the process configuration service. In the prototype this is called bindProcess. This may result in the creation of the client process.

The client object name is passed in the linkout message together with the server object name and server process name. The client uses the server process name to call bindProcess for the server prior to calling linkout in the server.

Where the server is on another node, the distributed name service will still return the appropriate rpcservice object reference. If the process lookup fails, the process creation and registration will be delegated to the node where the process is to be loaded.

The rpcservice object violates the C++ encapsulation of the trader by the rpcmanager in order to call an ANSA form of bindProcess without mapping to and from C++.. The rpcservice object controls the way the trader interface is presented to RPC service users. The rpcservice objects cache a reference to their local daemon for rapid access to the rpcmanager.

The rpcmanager is designed to encapsulate any enabling management services of the infrastructure such as name services, process activation, RPC service binding. This facilitates migration of the design across distributed object infrastructures and the separation of management interfaces from RPC service interfaces. It provides a portable C++ interface to the node and a distributed non-portable interface to the rpcservice object. The rpcservice is designed to control the way the management interface is presented to RPC service users.

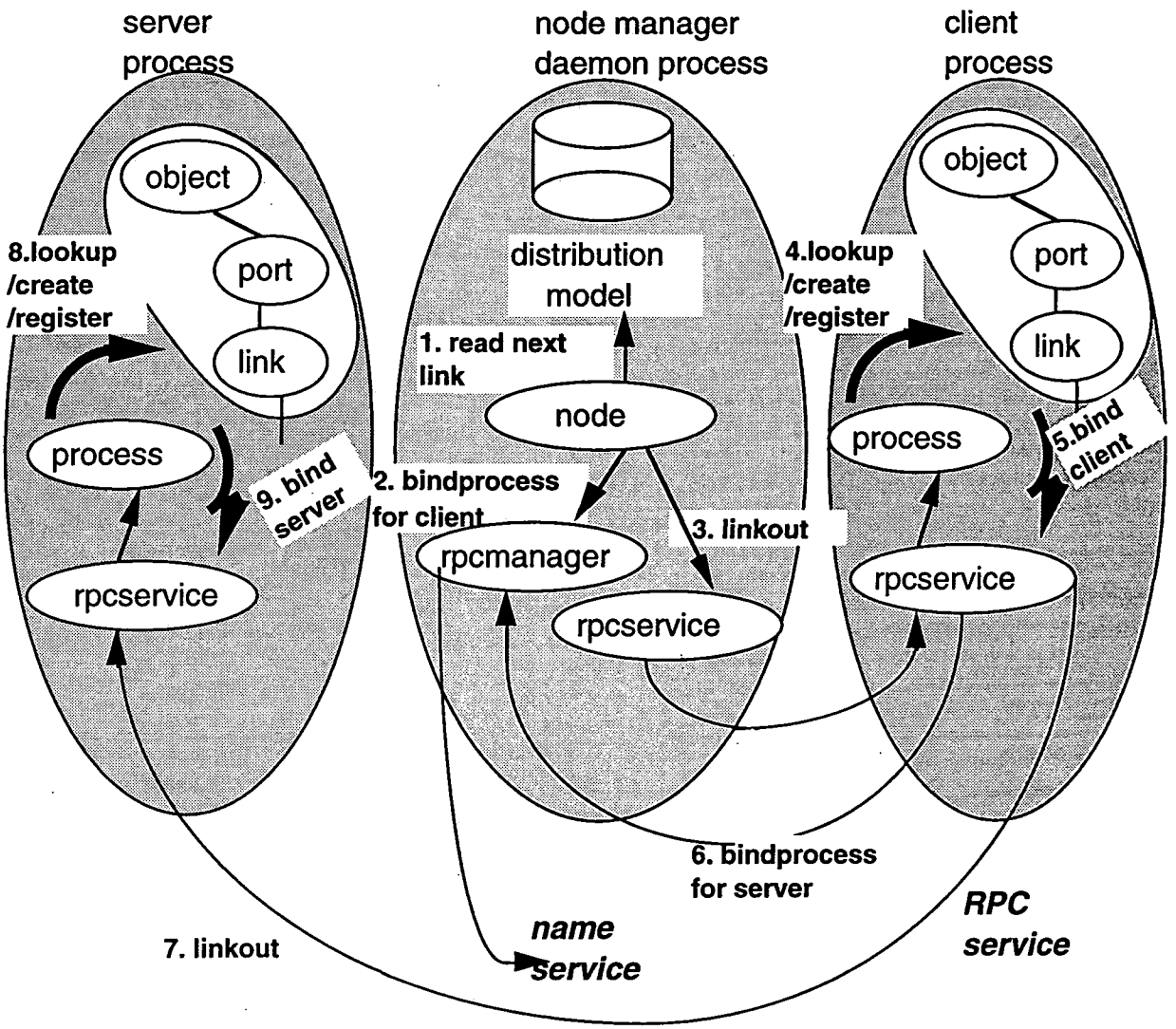


Figure 65 Binding application objects

8.4.5 Server Activation and Failure Handling

Processes are activated automatically by the combined bindProcess service of the rpcservice object. This relies on the infrastructure support. In the prototype, it uses the ANSA factory and trader to create the rpcservice as a managed object and register a reference to it. The reference is also returned to bind the rpcservice objects of the daemon to the server. This is shown in Figure 66.

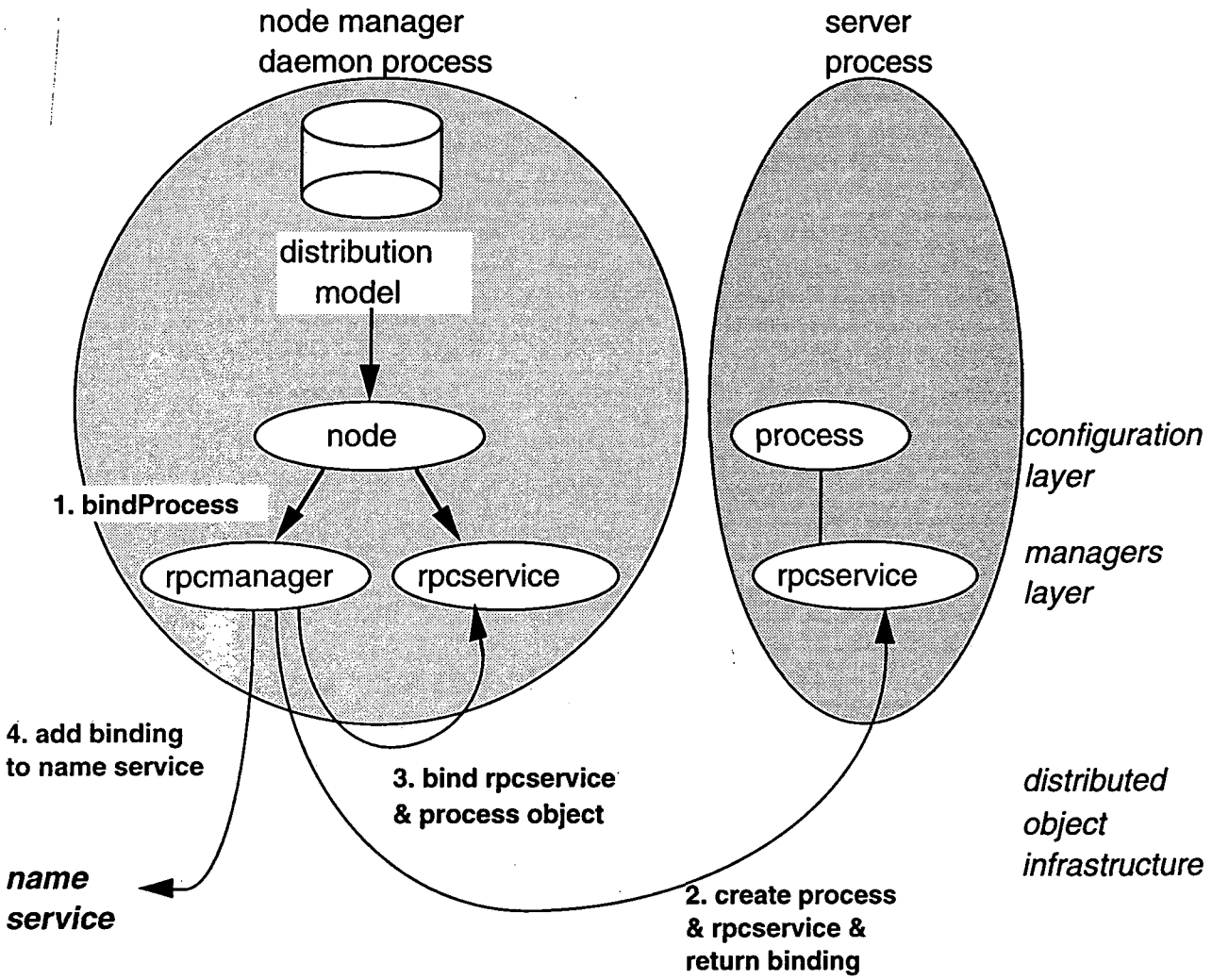


Figure 66 Process creation and binding of rpcservice object

On creating the rpcservice in the server process, a binding to the local rpcmanager is cached for rapid access to the management API. The rpcservice object provides an ANSA service called RMIInitialise which is called by bindProcess.

The process also supports a method for putting all its objects into a running state. In the final prototype, support was also included for putting all objects in a running process into unloaded states for re-configuration and for deactivating unloaded processes to destroy them.

In the prototype, objects are created automatically by the combined linkin and linkout services. Object constructors are called indirectly by calling a generic createobject() method provided by meta-class objects that are generated by the CDL compiler. The process object uses the class name to find the appropriate meta-class object. This is supported by the EDL compiler which describes the classes supported by each image and generates a make file that links the right metaclasses into the process.

In the final prototype support was also provided for unlinking objects and finalising and destroying objects. This allows reconfigurations once the process is in an unloaded state.

8.4.6 Request Processing

Ports are generated by the CDL compiler according to the signature defined for the invocation. Ports perform the data marshalling and concurrency control. Ports support synchronous invocations where the client blocks waiting for a reply, asynchronous one-way annunciation messages, and deferred synchronous two-way calls, where the client issues a request asynchronously then later blocks to collect the results.

Links are specified in the configuration language or visual scripting tools and control message and control flows. Link objects are provided by the OpenBase runtime library. They maintain pointers to local port objects and linkIDs for remote links.

The rpcservice object is actually implemented as two objects: one for incoming requests and outgoing replies, mapping linkIDs to local inlinks; and one for incoming replies and outgoing requests, mapping linkIDs to local outlinks. Each message consists of :

- the linkID of the destination,
- binding information for the source, i.e. the process name and linkID,
- the input buffer for requests or output buffer for replies.

Memory management in the client is such that the input buffer is reclaimed by the system after the call completes and the output buffer is reclaimed on the next call. Memory management in the server is such that both buffers are reclaimed after the reply is sent. Any persistent data not passed by value such as strings should be copied between calls.

Port and link objects perform various computations on object invocations to cope with various object management behaviours. The final prototype supports concurrency controls through locks that are specified in CDL, multicast calls across several links (called fanning-out where a outport is connected to several links), request construction by combining successive calls from different sources (called fanning-in, where a inport receives requests from several links before calling the method), and optimised local calls.

Future support may be included for sampling of continuous streams of input when it is not necessary to process every update and for reversible protocols that change from notifying to polling to trade-off required sample rates against measured rates of change. .

The scheme is extensible through port and link subtyping. The choice of subtype controls the behaviour and can be made transparent to the programmer by introducing specification attributes in the distribution model and adding further management policies to determine and interpret them.

An example class hierarchy for links is shown in Figure 67. Not all of this hierarchy was implemented in the prototype. Link behaviour is transparent to the ports since the base link class provides polymorphic bi-directional interfaces Typically a remote link provides the outgoing link and a inlink provides the incoming link.

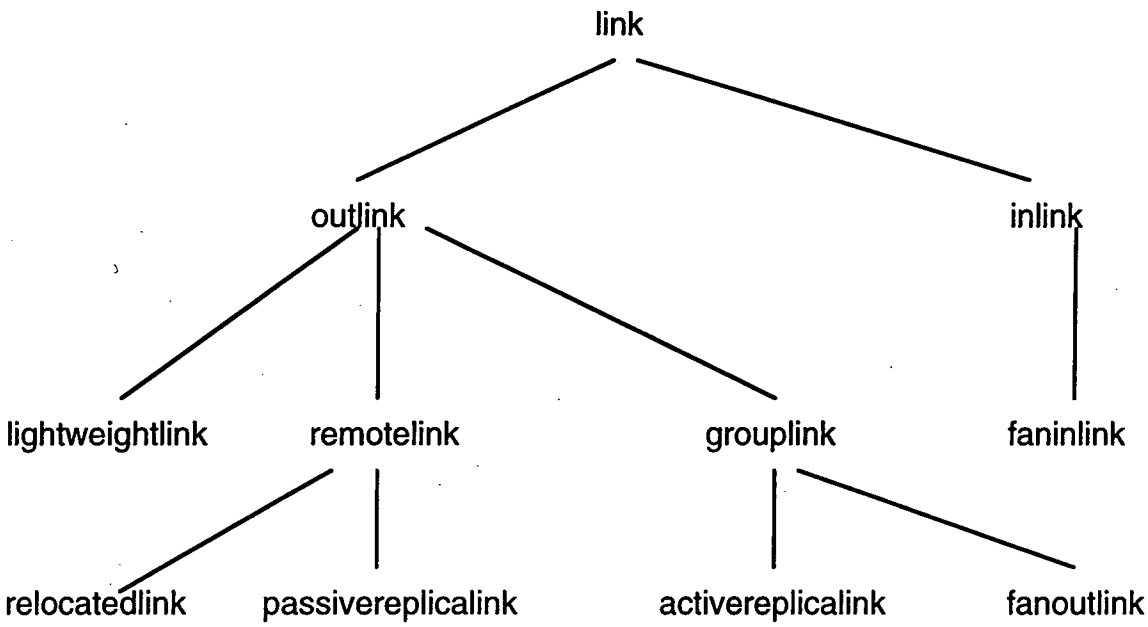


Figure 67 Example Link Class Hierarchy

In the future, link objects may be chained together to compose different management behaviours including migration, replication, transactions, load balancing, recovery, caching. This may also allow control at different granularities by including links bounding composites. Composites may either be clusters of objects or control flows that run across several objects. Links should also be able to change their behaviour, for example providing a *become()* service to support migration, *become (relocationlink)* at the old server, cluster splitting, *become (grouplink)* at the old server, and replica group changes, *become (passivereplicalink)* or *become(activereplicalink)* or *become (remotelink).*

For fine-grained invocations, performance optimisations are critical to avoid marshalling, buffer copying and context switching overheads for local calls. In the prototype optimisations can be made at two levels.

- For invocations between processes on the same node, a lightweightlink can be used. This is implemented in the prototype to use named pipes as the protocol in the rpcservice. ANSA supports named pipes.

- For invocations between objects in the same process, the outport will be connected directly to the inport via a special typed inlined method. Inport implementations are generated with this method. This method is called in place of the generic RPC upcall and avoids all marshalling overheads. This is shown in Figure 68 as a nested call.

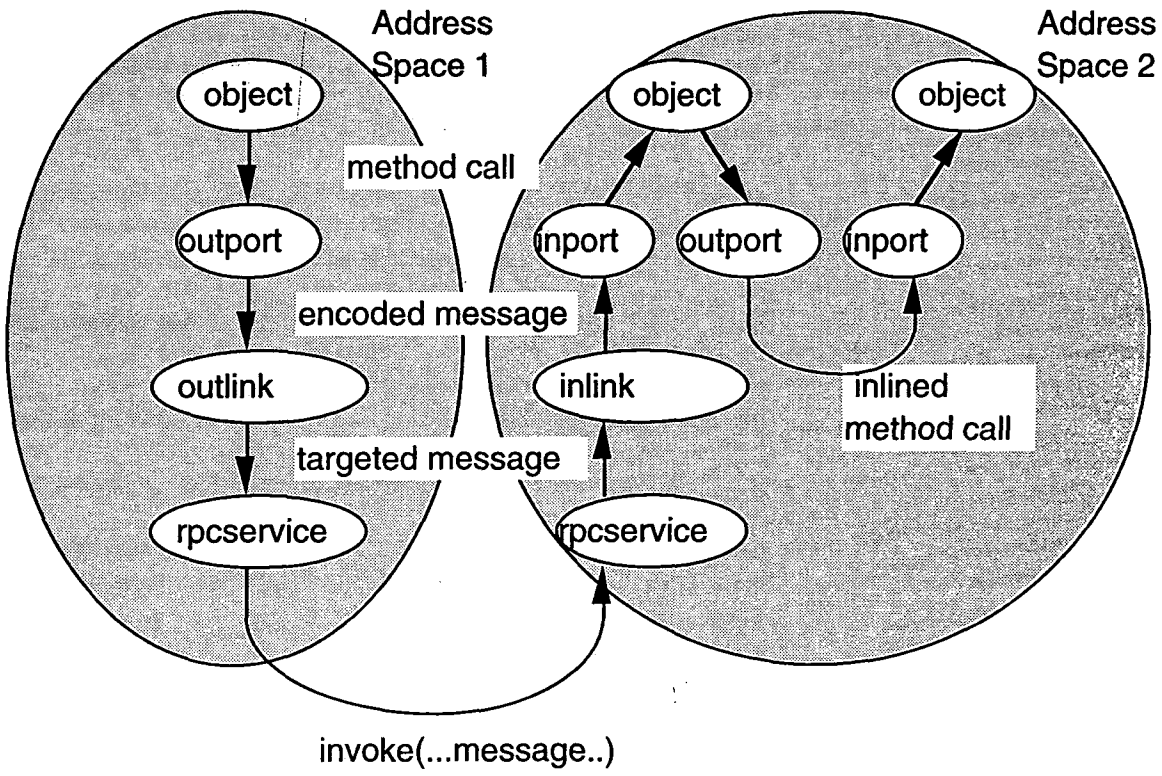


Figure 68 Nesting of local call in remote call

8.5 Summary of Chapter

This chapter has described the design of the prototype of the interpretation layer. The prototype was designed, coded and demonstrated as part of the MSc research.

The prototype provides a portable layering on top of ANSAware. This may be migrated onto another standard such as a CORBA implementation or the entire infrastructure may be implemented in a proprietary manner by Prism in later commercial releases of the software.

This chapter shows how a configuration of objects can be transformed from a meta-level representation in a distributed object oriented database onto an executable runtime environment. The design of the meta-data model is described in the next chapter and the development tools that can be used to define and manipulate the data model are described in Chapter 10.

The description of the interpretation process provided in this chapter assumes the meta-data model is generated by higher level tools and interpreted at build time. However this is not always the case. It would be easy to define runtime configuration monitors that manipulate the meta-data model at runtime to define new allocations and initiate re-interpretation to re-configure the system. Such configuration monitors may provide the runtime support necessary to implement advanced object management policies such as for process migration to deal with node failures and/or load balancing. Likewise it would be easy to re-edit an existing configuration to evolve the software incrementally without bringing the whole system down and doing a complete re-build.

Migration of a process and evolution of a software configuration are traditionally seen as very difficult tasks. Machine dependent information such as network communication paths, data structures in memory, running clocks are all inherently immobile. Machine dependent information is minimised with OpenBase objects. Objects encapsulate what can be moved. Link objects isolate and provide access to the dependent data structures that need to be changed.

Transactional support for changes in the configuration state of nodes, processes, composites and objects should be supported in the final system to facilitate the development of such advanced runtime configuration managers.

In the proposed design, adaptive management can occur in different ways:

- by substituting different runtime support components to choose different services i.e. reconfiguration of the infrastructure itself.

- by allocation of the configuration model to the distribution model.

Both are managed adaptively according to the existing loading and demands on the system. Both can occur at runtime or at build time (or re-build time on maintenance)

Support for build-time adaptive management is described in the next chapter.

Chapter 9 Binding Model and Distribution Model

9.1 Scope of chapter

This chapter evaluates the design and role of the meta-model layer. The meta-model layer is defined by the components and relationships shown in the shaded box in Figure 69. This includes:

9.1.1 Meta-Model Components

Type model

The type model is an object oriented database containing related objects that store the definition of a component type. This includes class definition objects, port definition objects and interface definition objects.

Definition objects capture common properties shared by sets of conformant instances. They contain static semantic information that is specified for all conformant component instances. These sets may be arranged in type hierarchies by having inclusive relations with other definition objects. This allows semantic information to be specified at various abstraction levels up the different type hierarchies.

Configuration Model

The configuration model is an object oriented database containing related objects that model a program as a configuration of component instances. This includes component instance objects, port objects, link objects, composite objects, cluster objects and activity flow objects.

Composite objects, cluster objects and activity flow objects are all set abstractions defining instance hierarchies using inclusive relations to the other objects. The ability of composites to contain instances that are nested composites allows a hierarchical model to be represented.

Port objects have connected-by relations with link objects. Nested composite objects may be related to unconnected ports that are exported from internal instances. These ports will be connected by link objects in the outer composite.

Port objects in turn are related to port definition objects in the type model and instance objects are related to class definition objects in the type model.

The configuration model contains contextual semantic information that is specified for particular instances of a component and can be specified using the various set abstractions of the instance hierarchies. Thus set abstractions exist both in type hierarchies and instance hierarchies and are useful to structure specifications.

Requirement graph

The objects that make up the configuration model and the objects that make up the type model include information that affects allocations. These take default values unless explicitly specified by the component programmer or application engineer.

The requirement graph defines views onto the configuration model that are used by the policy manager to perform queries in determining resource allocations and policy choices. These queries access the relevant information in the type model and configuration model.

Views can be generated in two ways:

- by automatically iterating down the composite hierarchies recursively when the application engineer submits a configuration to be allocated ,
- by explicit definition whenever the application engineer takes explicit responsibility for the management policy of any part of the configuration.

Views may have different consistencies, depending on the decision or policy being selected. They may be branches in the hierarchy of composites or sets of branches defined by cluster objects or sets of link and port objects defined by activity flow objects or individual instances, ports and links. Views provide the unit of management.

Load Graph

The load graph models the physical configuration as related objects representing nodes, images, processes and resources and is queried by the policy manager to determine a set of options to be used in allocations.

The load includes views onto the configuration model for configurations that have already been allocated. Hence allocations are sensitive to the imposed loading on the system. This also includes resources provided by previously allocated objects.

The configuration model and type model may include resource objects that are provided by a component, for example a print driver provides a printer as a resource. Links that span views identify correspondent objects that are also considered as resources, i.e. service resources.

The load graph therefore tracks previously-allocated correspondents, resources provided by objects, as well as general loadings. Loading information is also maintained by runtime monitoring.

Reward Graph

The complexity of management decisions means they are often relatively ad-hoc in practice. OpenBase attempts to provide a more systematic approach by providing conflict resolution policies that are sensitive to the application requirements. In the prototype these policies are algorithmic, and are based on accumulating reward values for different options. Rewards accumulate according to the value of requirement attributes in the requirement graph and load attributes in the load graph.

The reward graph maintains the reward values for the different options. The reward graph also allows several best options to be stored and used later for adaptive runtime management. For example, a domain of several best location may be used by a load balancing manager to migrate configurations off nodes that become overloaded without incurring the runtime overhead of determining best locations.

1.2 Relationships between Models and Other Layers

Relationship with Class Definition Language

The class definition language provides the static semantic information that is used to generate and populate the type model.

Relationship with Visual Editor

The visual editor or configuration language provide the contextual information that defines a configuration of objects and populates the configuration model.

Relationship with Environment Definition Language

The environment definition language defines the physical configuration that is used to populate the load graph with an initial description of the available resources, nodes, images and processes.

Relationship between configuration model and distribution model

The component programmers and application engineers view is of a single world of objects, accessible from any machine without awareness of actual location. This view is modelled in the configuration model and is transformed into a physically distributed model by allocation of the configuration model to the physical model represented in the load graph.

Relationship with interpretation layer

The interpretation layer queries the configuration model and instantiates and initialises runtime objects that correspond to the instance objects in the configuration model and instantiates and links runtime communication stubs that correspond to the link objects in the configuration model. The allocation of runtime objects to processes and the choice of runtime policy is determined by specification attributes that are defined for allocated views.

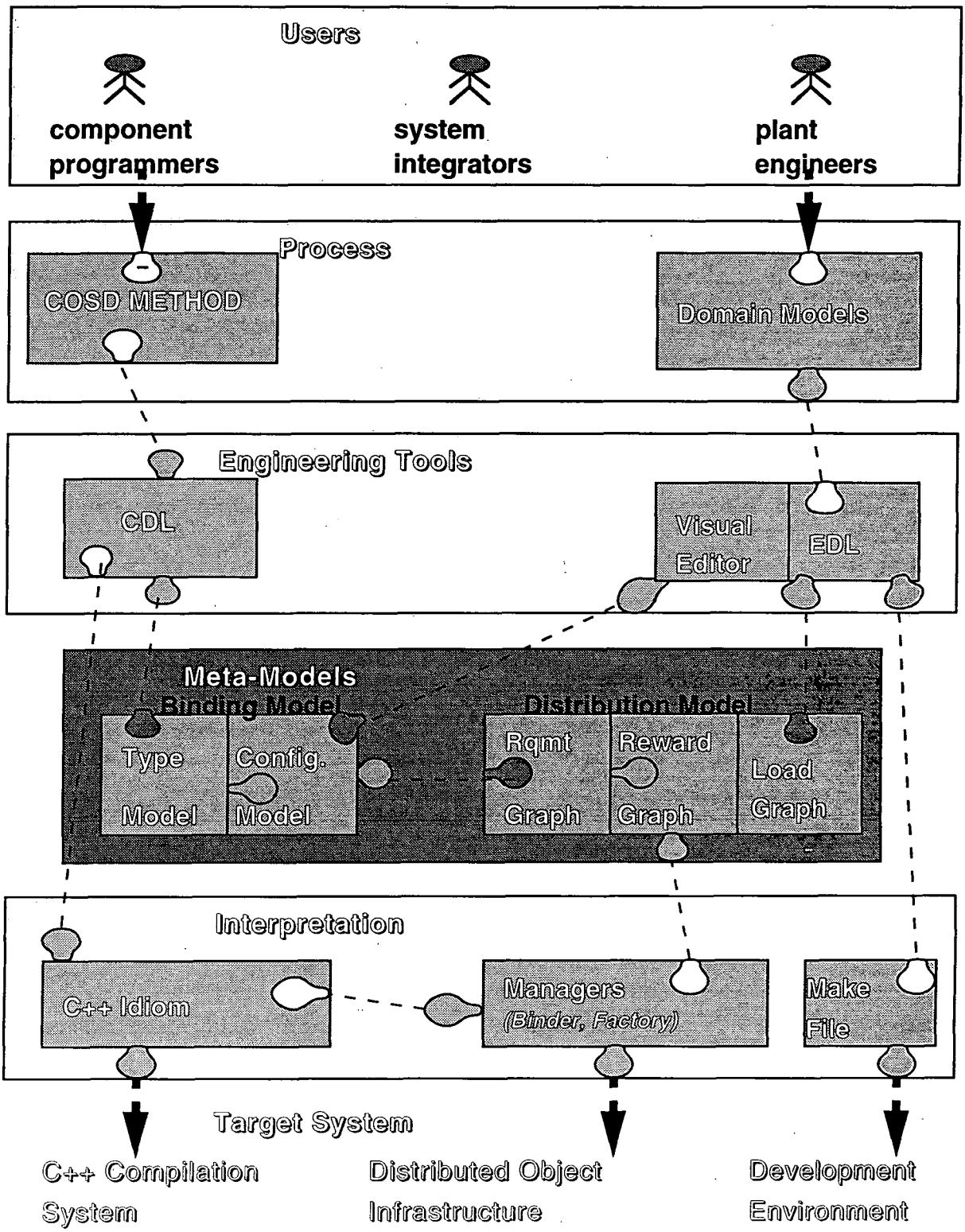


Figure 69 Meta-Model Components and Relationships (shown in shaded box)

9.2 Design Choices

Design choices can be described as instantiations of the design variables identified in the evaluation framework. These are shown in Figure 70 and Figure 71 for the most important principles in the design of this layer, namely polymorphism techniques to ensure pluggability to allow components to be connected safely and selective properties to allow control over the management behaviour.

9.2.1 Polymorphism Choices & Rationale

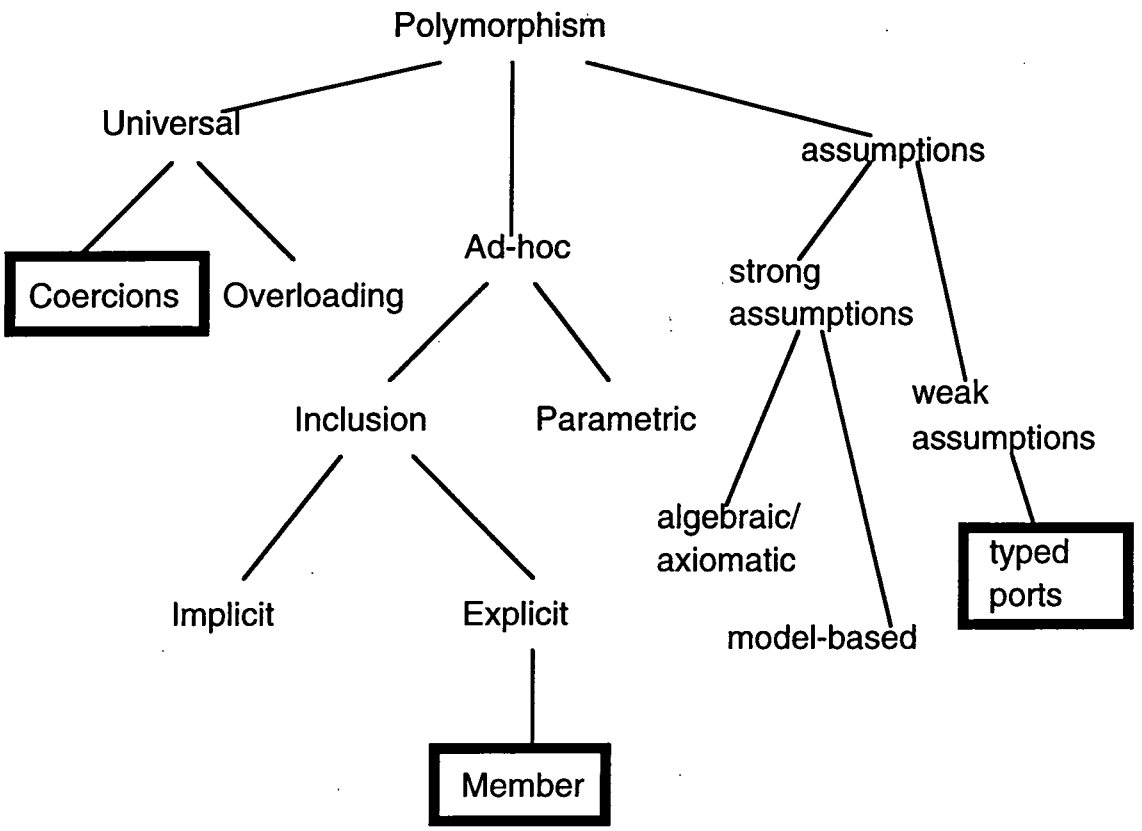


Figure 70 Types of polymorphism supported (shown in boxes)

OpenBase allows assumptions that a client makes about a server to be expressed by specifying *typed ports.* Ports are typed according to interface definitions. These consist of specifications of operation signatures and data types for arguments and exceptions. Components that provide inports with conformant interface definitions may be substituted across a link from an outport. These assumptions are weak. Interface definitions do not currently allow the specification of pre- and post- conditions or other behavioural specifications that express strong assumptions about the behaviour that a client component expects from a server component. This is seen as a weakness in the design.

Substitutions are made by specifying a link between the outport in a client object and an inport in a server object. It is individual members, associated to ports, that are bound together and must conform rather than objects. Interface definitions are distinct from class definitions. This form of *member inclusion polymorphism* is the weakest most flexible form of conformance, as defined in section 2.1.2.

Mainstream object oriented systems support polymorphism across inclusive relationships between classes, i.e. inheritance relations, rather than inclusive relations across individual members of classes, i.e. composition relations. They have no notion of port type. Class inheritance using generalisation-specialisation usually implies some level of semantic consistency for all derived classes and this is used as the constraint.

Primitive port types say little about semantics, i.e. the actual behaviour of a method. Semantic consistency is only achieved with port types if the design methodology focuses on finding abstractions for the interactions that can occur between objects, such as design-by-contract(Meyer, 1992, ref [2]) or framework-based methods discussed in section 4.2.2. In this case, the designer identifies different types of interaction that are meaningful in the application domain and introduces interface types to capture the semantics of those interactions. This is an important aspect of the OpenBase methodology.

Note interface types may themselves have inclusive inheritance relations to other interface types so that constraints on polymorphism can be expressed at different levels of abstraction. Note also that this type hierarchy is orthogonal from the class hierarchy for components. There is nothing to stop interface definitions from mapping to class definitions with each class having a single inport but this is not implicit.

Coercions are supported by the C++ compiler between basic types such between integers and longs. However this only occurs internally in the implementation of a component since port types must be constructed types defined by operation signatures and cannot be basic types.

9.2.2 Selective Property Choices & Rationale

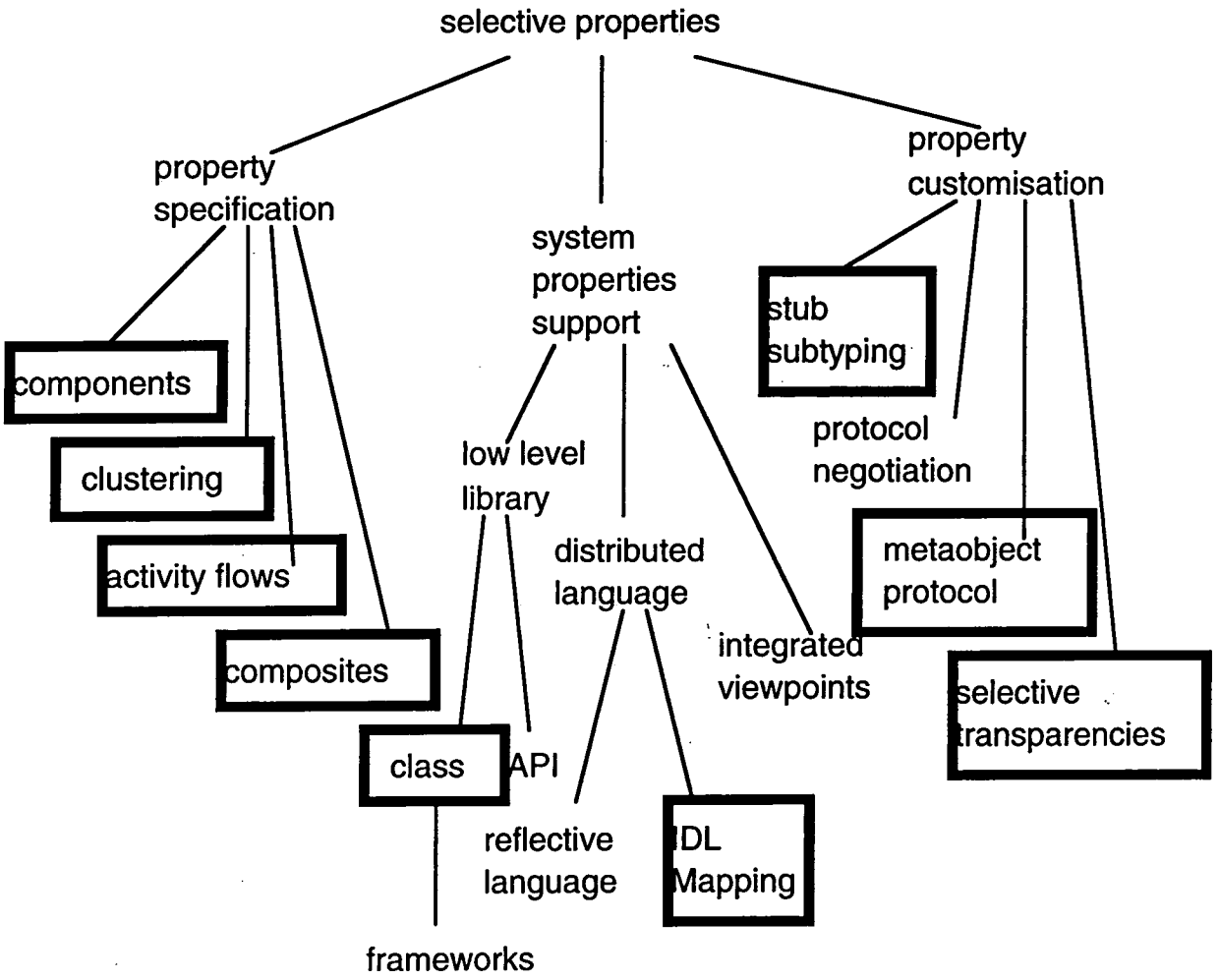


Figure 71 Types of selective property supported (shown in boxes)

Various system properties are realised by adopting different management policies. This includes policies for allocating objects to processes across the network and choosing runtime protocols for RPC, for replication, for atomicity and recovery, etc.

Selective properties are supported by providing both explicit and implicit mechanisms for choosing these management policies. Explicit management is sometimes necessary to allow control over management policies for efficiency or to meet non-functional requirements. Implicit management is important to avoid the complexity of specifying a policy for all management behaviours for all components.

In order to provide selective properties, OpenBase provides automated conflict resolution policies that can be influenced at three levels. In the prototype, the configuration model is annotated with requirement attributes and specification attributes and the policy manager uses the values of these attributes to determine the allocation according to a programmed policy. The three levels of automation include:

- Policy may be explicitly controlled by defining dialogues in the visual editor that allow the application engineer to define policy for selected parts of the configuration. Parts may be *component* instances or *composites* or *clusters* or *activity flows*. This overrides automated resolution.

- Conflicts may be influenced by specifying abstract requirement attributes rather than explicit policies using the visual editor as above or using the class definition language. Rewards accumulate for different options according to the value of these attributes. This leaves the final choice of policy to the system but allows sensitivity to the applications needs and permits best-efforts choices.

- Conflict resolution may be entirely transparent by using default requirement values that are defined in advance for the entire application domain without any programmer effort.

In effect, the visual editor provides a *meta-level protocol* consisting of dialogues that allow the application engineer to define these attributes and modify the policy.

The distribution model provides a more complex *meta-object protocol* to system programmers who can program new policies by subtyping the objects in the distribution model and configuration model to add new attributes and new queries.

In the prototype, the policy is programmed algorithmically by accumulating reward values for different options and selecting the option that has the highest reward. In the future it is intended to add a proper rule engine and explore constraint based reasoning.

Runtime support for selective properties is supported by the interpreter choosing different *stub subtypes* according to the value of the specification attributes. Stub subtypes, or more precisely port subtypes and link subtypes, are provided by the interpretation layer to deal with optimised RPC, replication, etc. as discussed in chapter 8.

Stubs are supported transparently by using an *interface definition language*, or class definition language, to hide the selective runtime interface from the programmer. Stub generation by a language processor allows the programming system to support a more declarative view of the different runtime protocols.

Stub subtypes are implemented using the transparency support already provided by the infrastructure. ANSA supports a number of *selective transparencies*, including location transparency, access transparency, replication transparency.

9.3 Design of Configuration Model

Support for graphical programming requires more than editors for graphical notations. It is important to develop a notion of evaluation. Graphic objects representing programs must be evaluated to generate executable code. Evaluation behaviour may itself be encapsulated in objects and be associated with graphic objects that present themselves on a screen.

In OpenBase graphic objects are used to visualise the program structure as a node edge graph where the nodes are components and the edges are relationships between components. These graphic objects have associated with them, persistent objects that may be evaluated to instantiate components and connect them together. These evaluation objects are stored in an object oriented database that is called the configuration model.

The configuration model provides an underlying repository to store any textual and graphical programs. The underlying node edge graph provides a base-level framework for hooking other types of graphic primitive or textual script that may have richer semantics.

This configuration model consists of the type model and the configuration model.

The type model includes the following types of object as shown in Figure 72:

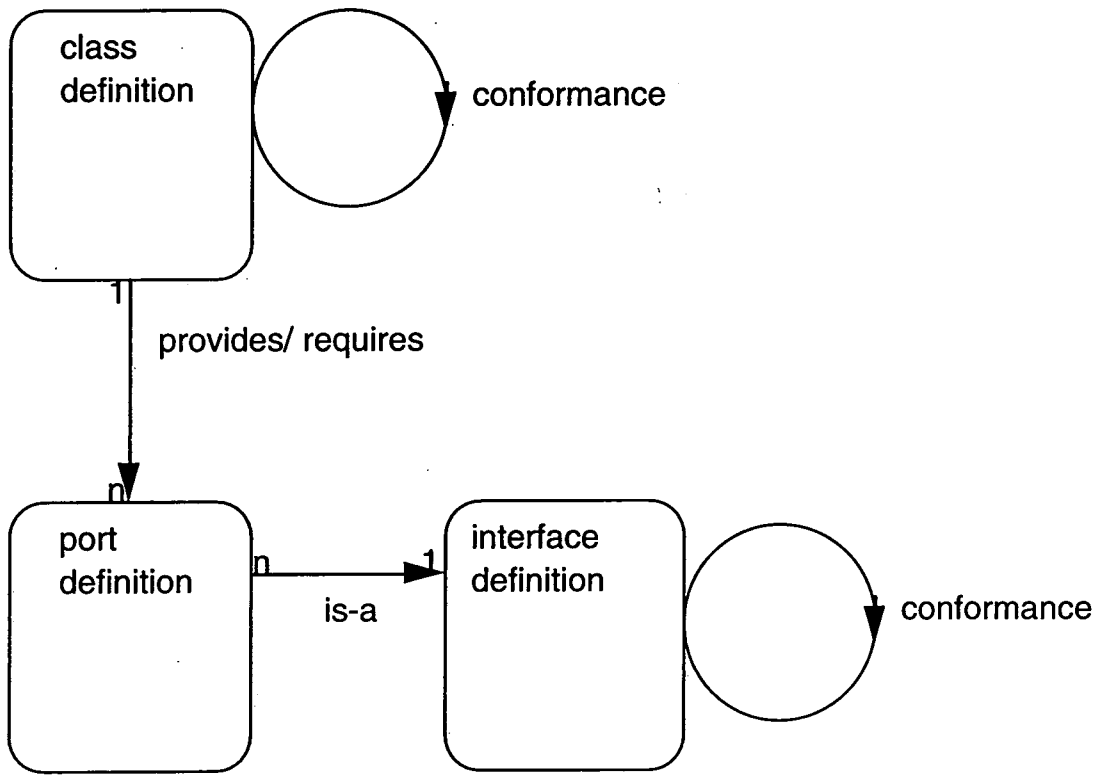


Figure 72 Type Model

Class Definitions - one convenient characteristic of object is the amount of semantic information that can be described in interfaces. An interface definition language or class definition language can be used to describe rich semantic information. This information is stored as attributes of objects in the type model representing class interface definitions. Nodes in the node edge graph are related to class definitions by is-a relationships. Class definitions are related to other class definitions by conformance relationships. Class definitions are related to port definitions by requires and provides relations.

Interface Definitions - semantic information can also be described for interface types. These may be orthogonal to class definitions. This information is stored as attributes of objects representing interface definitions. Interface types are related to ports by is-a relationships. Interface definitions are related to other interface definitions by conformance relationships.

Port definitions - semantic information regarding inports and outports is stored as attributes of objects representing port definitions. Ports have direction, in or out, and mode, such as optional, asynchronous, dynamic. Port definitions are related to interface definitions.

The configuration model includes the following types of objects as shown in Figure 73.

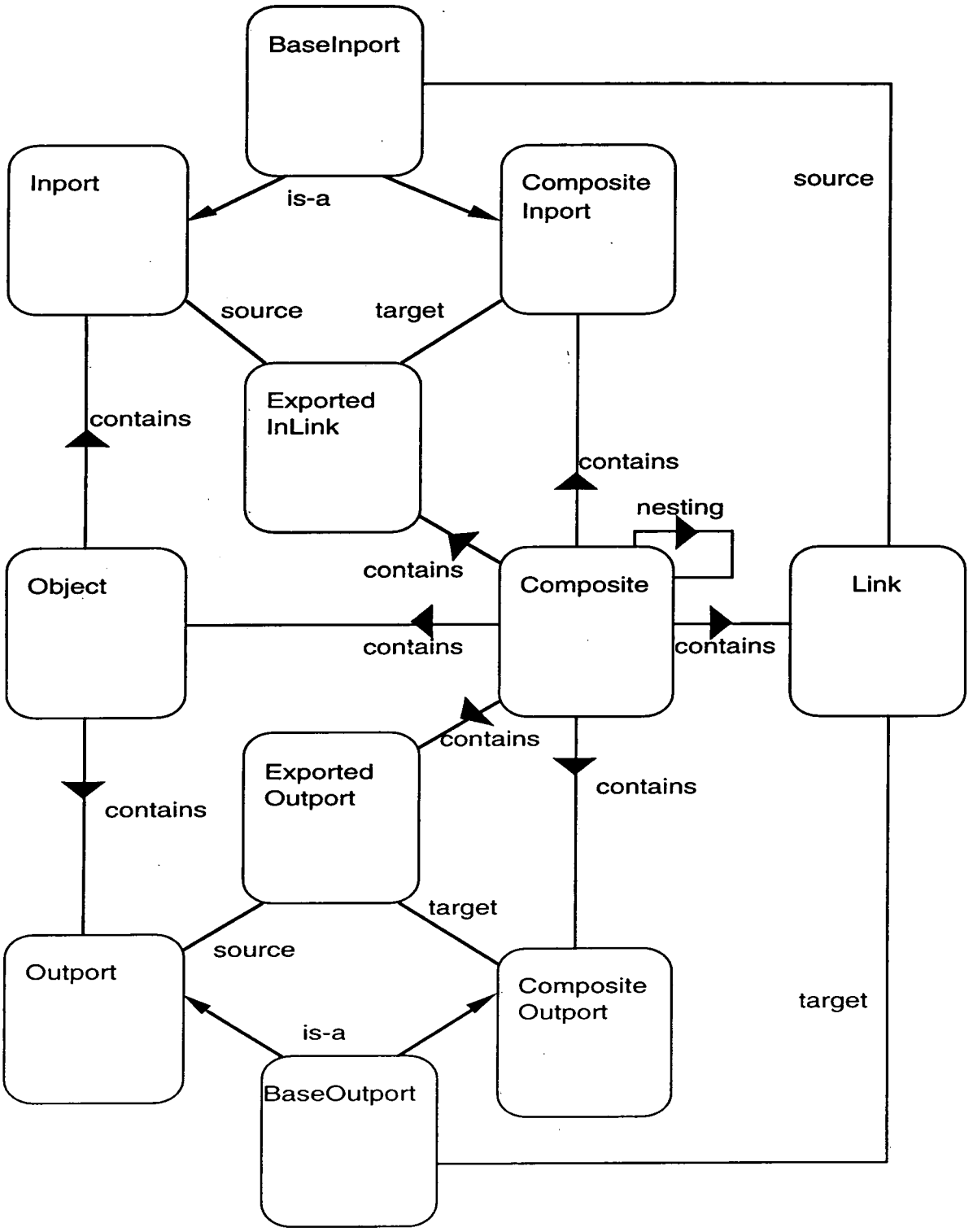


Figure 73 Configuration Model

The *Object* class provides the meta-model representation or runtime instances of classes. This is related to the class definition from which that object is instantiated and this relation is navigated to explore any static properties share by all instances of that class. *Object* instances may be annotated with contextual properties only relevant to that instance. They are related to outports and inports and these relationships must be consistent with the relations in the type model between the corresponding class definition and port definitions.

Composite provides the meta-level representation for configurations of objects. Composites may act like instances and be nested within higher level composites to form a hierarchy. Composites may also be saved and instantiated in several places.

Outport and inport represent the exit and entry points to an object respectively. They inherit from a base class.

Composite outport and composite inport represent the exit and entry points to a composite respectively. They also inherit from a base class.

Exported inport and exported outport represent a link that exports a port to the composite interface. This means that the port is linked in some outer composite. The *source* relation identifies either an object's port or a nested composite's port. A port may therefore be exported up several levels in the hierarchy before being linked. The latter is not shown on the diagram to keep it simple.

Baseinport and baseoutport allow ports from objects and ports exported from nested composites to be treated identically so that instances of composites may appear like instances of classes. They are related to port definitions. In the case of composite ports, this is the port definition of the source port that is exported. They derive static properties shared by all instances of the class which owns the port. They may also be annotated with contextual information for that instance of the class.

Link objects represent links between inports and outports. They may be annotated with contextual information determining the communication policy.

There are a number of different types of composite that have not been shown in Figure 73 and will be discussed in Chapter 10. There are also two objects that may be viewed as collections that have nothing to do with the connectivity shown in the figure:

Activity flows are collections of link objects that allow contextual properties that apply to activities spanning several objects, to be modelled at a more global level.

Clusters are collection of objects that allow contextual information that applies to structures of objects to be modelled at a global level. Clusters are not restricted to structures defined as composites but may be any collection.

Both the type model and configuration model may be annotated with extra attributes and nested objects to be queried and set by the policy manager in the distribution model. This includes :

Property Specification Objects - the evaluation objects may be annotated with detailed specifications whose attributes explicitly determine the runtime behaviour of the system, through different choices of management policy. The globality of a specification is determined by which type of evaluation object the specification object is attached to. The application engineer bounds the scope of a specification for example by selecting individual elements in the node edge graph or clusters of nodes or by selecting flows along multiple edges. Such selections may be called management composites as they act as the domain in which management policy is specified, i.e. this can be called management-in the-large, as opposed to configuration composites that act as the domain in which objects are configured, i.e. called programming-in-the-large.

Priority Requirement Objects - evaluation objects can be annotated with structured requirement descriptions that are evaluated to property specification objects according to priority trade-off rules that are encoded in the distribution model. This supports higher level specification of operational (i.e. non-functional) requirements like availability, reliability, time criticality.

Resource Requirement Object - evaluation objects may be annotated with information about resource requirements and services required such as the requirement for binding or memory.

Resource Objects - elements of the graph or definition may be annotated with information about capabilities that a component offers to the system such as resources or special services. For example, in process control, a PLC driver object provides the PLC resource for tag objects.

Reward evaluation objects - the distribution model provides primitive evaluation objects like priority trade-offs, thresholds, dependencies, sets and constraints that can be used to program evaluation rules for requirement objects. This are described more fully in the next subsection.

The reward evaluation and requirement objects are shown in Figure 74.

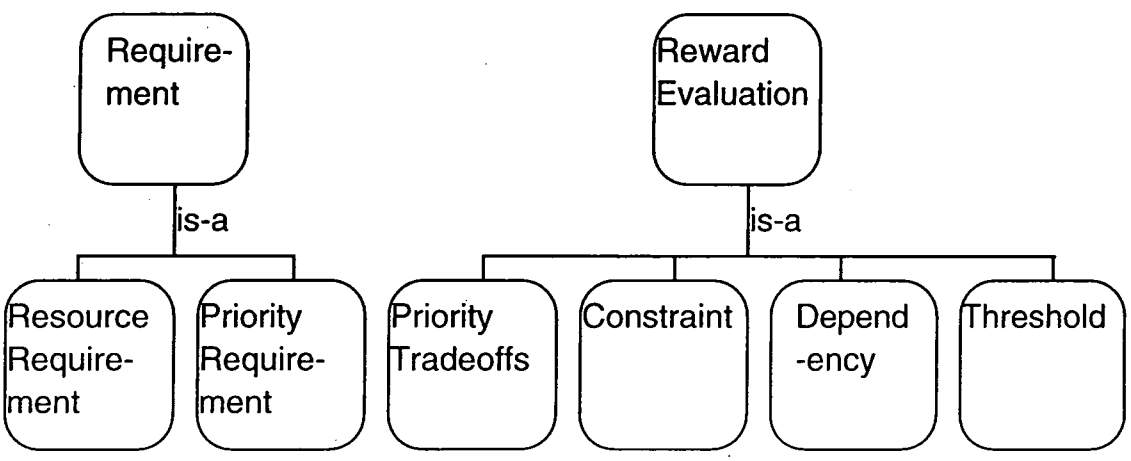


Figure 74 Requirement and Reward Evaluation Objects

9.4 Design of Distribution Model

A Framework for Build-time Adaptive Management

Existing object oriented methodologies offer poor support for implementing process/processor architectures. This project completely removes the need to design a process architecture. Programmers generate a logical model which is not explicitly distributed. Instead allocations are based on constraints and requirements that are described by the programming tools and kept up to date by the distribution model, including the existing system loading and expected resource requirements. The transformation of requirements to a process architecture design is entirely automated.

Evaluation of the configuration model is not limited to allocation of composites to locations. There are also protocol choices to be made.

Both allocations and protocol selections are represented by annotating the meta-models with requirement attributes and specification attributes or objects. Requirement attributes are queried by the policy manager to determine the specification attributes. Specification attributes are interpreted by the interpretation layer to select the right infrastructure service. The policy manager objects controls the logic for defining specification attributes based on requirement attributes and the physical configuration.

The policy manager used in the prototype takes a systematic approach to resolve specifications, by attempting to quantify requirements with a reward value. The system is characterised with three interrelated data structures : a requirement structure for the configuration being allocated; a reward structure for options and an availability or load structure for physical resources and configurations already allocated, as shown in Figure 75. The system accumulates a reward value on allocating services. Expected rewards for different options are based on probabilities of meeting requirements. This will depend on knowledge of the application, including expected object sizes and throughputs. The reward value also depends on requirement priorities, for example to resolve the trade-off between reliability and performance. Expected rewards are used to select alternatives.

Initial physical resource availability is described in EDL. Requirements, priorities and application requirements are made available using CDL and the visual editor.

The logic for determining expected rewards is programmed into reward evaluation objects, as described previously in Figure 74. This includes:

- priority trade-offs, expressed as equations in terms of cumulative priority variables for different properties, such as reliability, throughput.
- thresholds, expressed as rewards that accrue when the threshold triggers, such as the overload threshold for a CPU.
- dependencies, that are rewards triggered according to allocations of other parts of the configuration, such as correspondent objects.
- constraints, less fuzzy predicate objects that exclude options outright, for example the constraint that an image must support the class of the object being allocated.

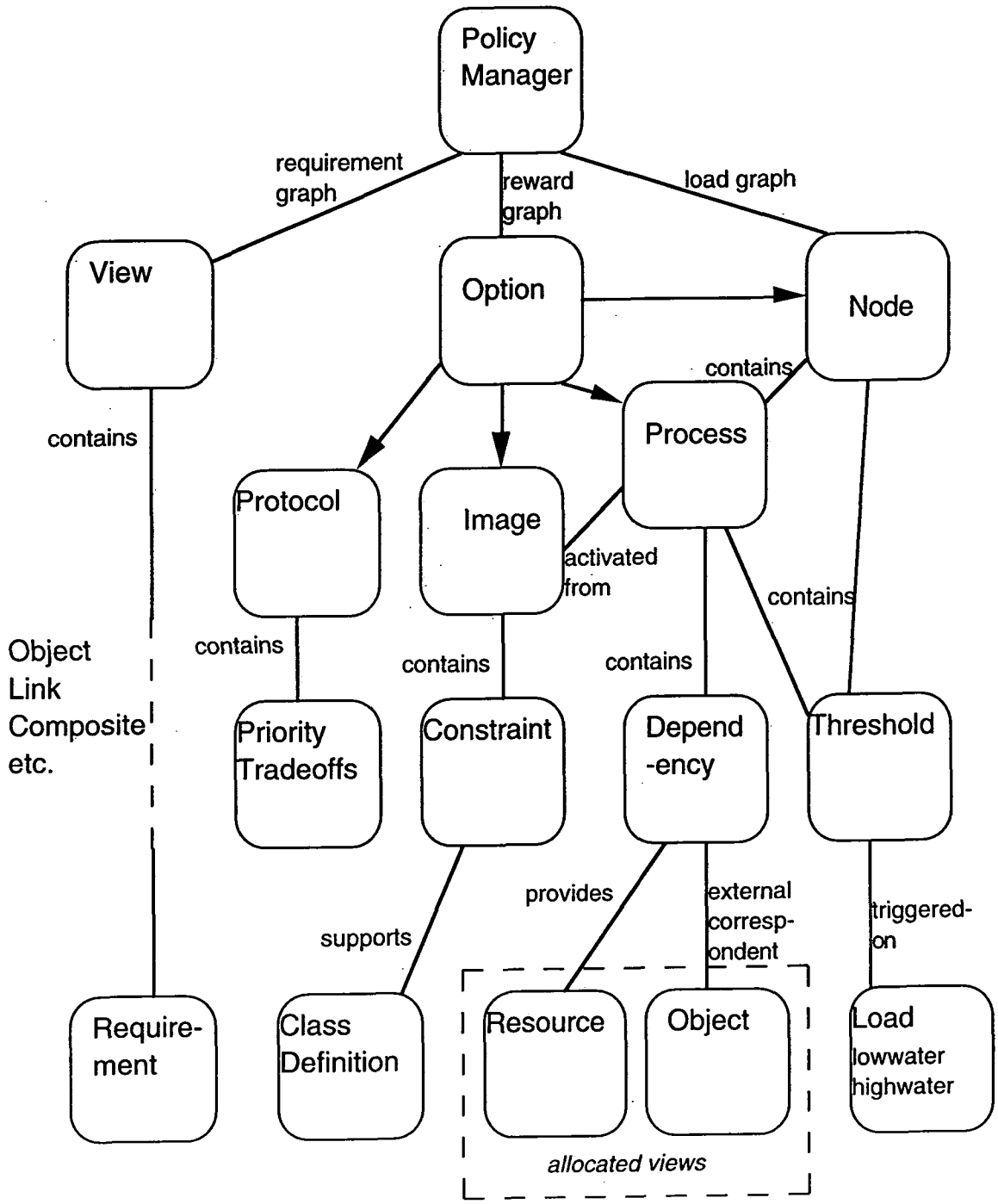


Figure 75 Example of distribution model classes

The reward and availability structures are not restricted to load-based resources such as memory, storage, CPU. The policy recognises the significance of co-location decisions in meeting operational requirements such as deadlines, performance targets, reliability and currency goals. The availability structure tracks correspondent objects so that rewards can accumulate based on proximities to related external objects that are in a different view and these external links may be annotated with requirement priority properties like time-criticality, throughput expectations, bandwidth, latency constraints.

Initial allocations use all requirements and store domains of best locations. Any runtime policies such as load balancing, replication and recovery are designed to restrict allocation decisions to this domain and reward fewer requirements. A two threshold representation is used for load availability to reduce information dissemination overheads, i.e. state changes in rewards are restricted to underloaded, mediumloaded and overloaded states.

Design of Initial Prototype

The first prototype provides some very simple policies merely to prove the feasibility of automated support. In the future it is expected that Prism will further research this in particular constraint based-reasoning to define automated policy management rules more rigorously.

The prototype supports automated allocation of objects based on two threshold representations of loading that is maintained as an attribute for each process and node object in the distribution model. As well as ensuring objects are allocated to underloaded nodes and processes and away from overloaded ones, as a crude way to ensure load balancing, this attribute is also used to determine when to create new processes from an image and on which node the new process should be created. The relative size of objects can be determined from the class definition and appropriate thresholds be hard-coded by the system configuror.

The only protocol selection that is supported by the initial prototype is the choice of communications policy. The system provides the following communication services: a two-way reliable service, a one-way reliable service, a one-way unreliable service, an optimised service for local calls within a process, and a lightweight service for calls between processes on the same node. The choice of communication policy depends on two specification attributes in the distribution model : the location attribute and the RPC mode attribute. The distribution model will define these attributes when evaluating a configuration. The choice will be determined by a combination of factors each contributing to the net reward for different modes. Factors include: the physical constraints, system loading, service availability and requirement attributes in the configuration model.

The model is intended to be extensible with other requirement attributes, corresponding specification attributes and protocol selection policies. Future specification attributes include: replication attributes to define the activeness/ passiveness and multiplicity of replication to meet availability and performance for relatively static data; timing attributes for hard or soft deadlines; extensions to the RPC modes attribute to include time critical communication services; clustering attributes to ensure regular correspondents are collocated; relocation domain attributes, defining groups of best locations for replicas or migrations. Future protocol selections include: replica group interaction; real-time monitoring; communication services and scheduling; recovery and load balancing services that migrate servers.

The distribution model is designed to be extended in several ways: by adding sets, constraints, dependencies and rule evaluation as explicit objects to permit selective iteration through the configuration model; by adding constraint nets for dependency driven evaluation of the model; and by adding rule conflict resolution mechanisms according to the order of evaluation, the knowledge used, the context of the decision, and the specificity of the rule.

9.5 Conclusions to Chapter 9

This chapter has described the design of the meta-model that represents component definitions and configurations of these components.

The model is queried by the policy manager to select management policies and by the interpreter to load and link an executable for the configuration. The main benefit of this model is that it opens up the language to allow integration and automated selection of management policies.

The model is defined with high level tools like the class definition language, environment description language and the visual editors, that will be described in the next chapter.

Chapter 10 Specification and Modelling Tools

10.1 Scope of chapter

This chapter evaluates the design and role of the tools and process layer. Whilst the programming concepts and tool interrelationships were defined by the research on which this MSc is based, the detailed design of the tools, the syntax and implementation of the language processors and visual editors, was carried out by other members of Prism. This chapter does not define the tools in detail, rather it defines requirements placed on them and illustrates an example of what they might look like, that was provided by the research as input to the detailed design.

The tools and process layer is defined by the components and relationships shown in the shaded box in Figure 76. This includes:

10.1.1 Tools Components

Class definition language

Component programmers provide definitions of component interfaces, data and exception types , operations, both ingoing and outgoing, and service properties, both requirements and capabilities.

Environment definition language

The system manager describes the plant layout in terms of computer nodes, executable images available on each node, processes and their associated resources.

Configuration language

The application engineer can optionally use a static configuration language to define composite components, which can optionally be saved as composite types.

Visual editor:

The application engineer can also use a graphical editor to define composite components using a visual composition metaphor. This provides commands to browse component libraries organised as palettes of icons representing the components, to select individual components from the palettes, to drag and drop icons on the screen to instantiate components, and to click and join icons to link component inports to outports.

10.1.2 Relationships between Tools and Other Layers

The class definition language processor generates extra code to wrap runtime C++ objects and populates the type model with class definitions, interface definitions and port definitions.

The visual editor browses the type model and populates the configuration model with composite definitions.

The environment definition language generates make files and populates the distribution model with a definition of the system resources, both physical resources (nodes, CPUs, storage etc.) and executables.

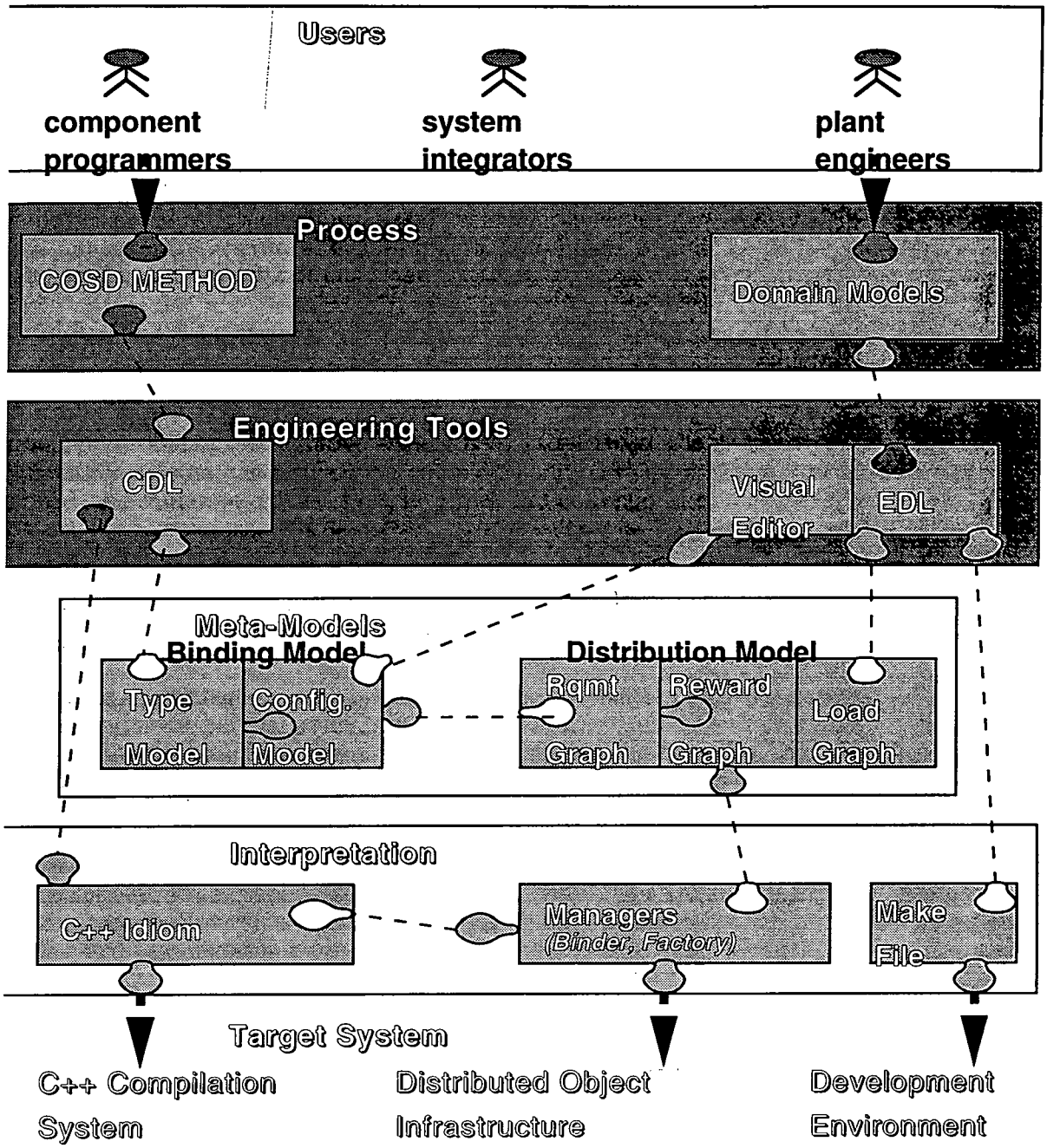


Figure 76 Tool and Process Components and Relationships (shown in shaded box)

10.1.3 Process Components

Domain Modelling

Domain modelling provides the domain specific support, such as domain specific frameworks, interface standards, domain concepts and facilities such as alarming for supervisory control as well as components conforming to the domain model. These generic models are then specialised for a specific requirement and the relevant components for the domain are integrated using the visual editor.

Component Oriented Software Development Method

The component oriented development method provides the guidelines used by component programmers to develop components in isolation of other components using the standards defined in the domain model.

10.2 Design Options

Design choices can be described as instantiations of the design variables identified in the evaluation framework. These are shown in Figure 77 to Figure 81 for the most important principles in the design of this layer.

10.2.1 Protocol Choices and Rationale

System protocols

Open distributed computing standards (integrative standards like CORBA, DCE and ISO ODP) by themselves do not yet provide the complete solution to users goals of interoperability and portability. This is because they are continuously evolving immature partial specifications.

The OpenBase platform provides a virtual interface that hides and manages migration to these standards. It does this by hiding the standards interface behind high level tools and abstraction layers. It defines a higher level protocol.

Middleware like OpenBase can provide a vital role for companies in managing the complexity of dealing with open systems standards across producer-supplier channels. Collaboration is the key to open systems. Collaboration in easier when focused on a specific domain like process plants.

The open object-oriented design of OpenBase allows customisation and specialisation at any layer of the architecture to extend the system or migrate it onto new platforms. The runtime support is represented as objects and is therefore open to customisation by meta-object protocols. For example, configuration objects of the interpretation layer such as runtime link objects that hold binding information, may be specialised by a system programmer to use a different transport service.

The existing design uses standards-based remote procedure call (RPC) mechanisms for the transport service to send requests and replies across machines. Remote procedure call is a natural choice to support remote method invocation. DCE, CORBA and ANSAware all support RPC across a heterogeneous network.

Domain Protocols

OpenBase defines a specific *binding model* based on combined data and control flows between typed ports. Ports have a direction according to whether they are inports or outports. The binding model does not support conditionals or looping constructs in ports as these can be provided by the components behind them.

Ports are typed by an *interface type* definition that may support more than one operation. This is usually orthogonal to component class definitions.

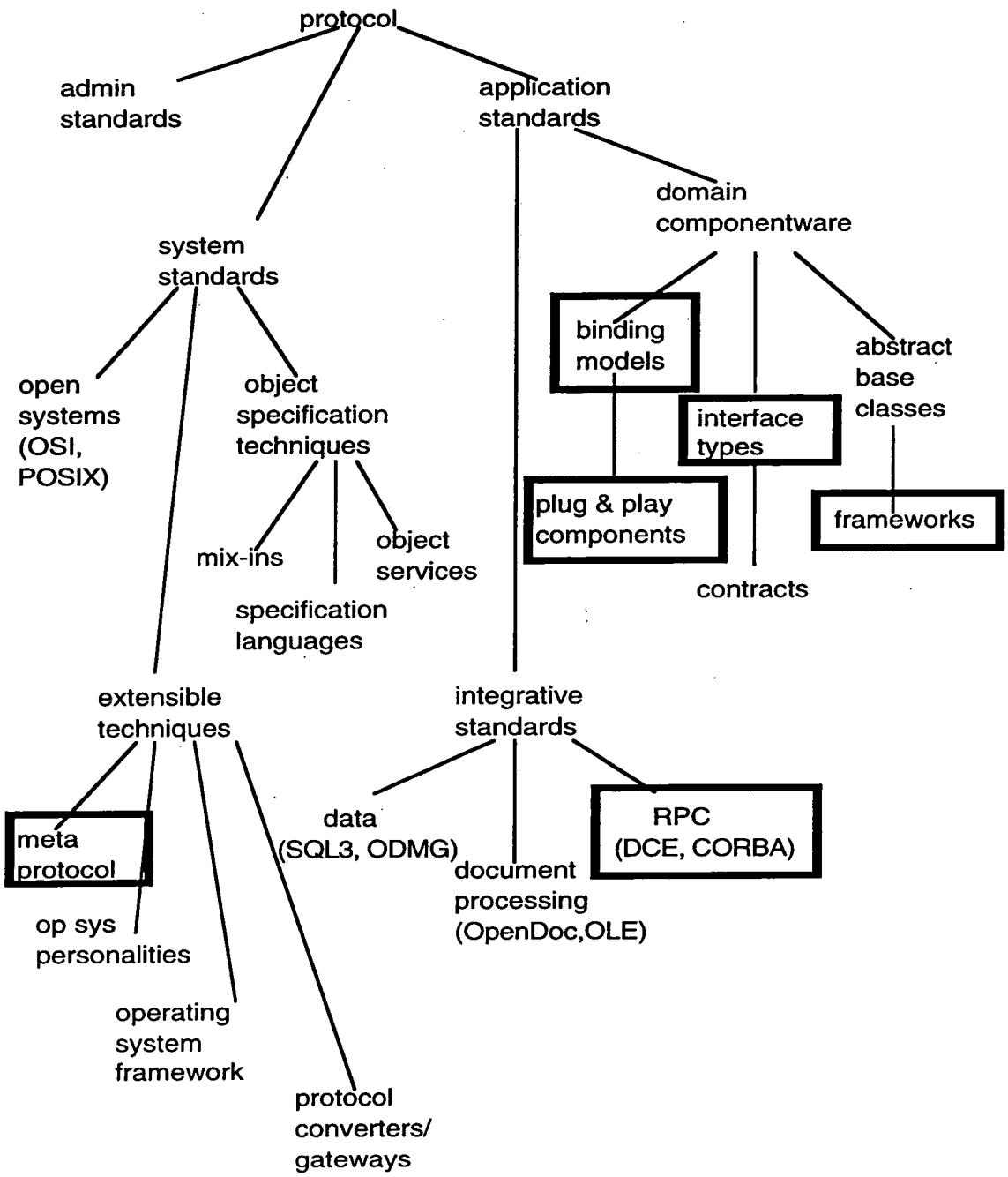


Figure 77 Types of Protocol (shown in boxes)

The development lifecycle proposed first establishes domain model standards to standardise on interface types in each application domain. This optimises opportunities for vendors to implement plug compatible components. Prism have committed to maintain a registry of interface types and co-ordinate the de facto adoption of interface standards.

A *framework* is an abstract pattern of interaction between collaborating objects. These patterns define control structures that are reused in different designs. They may themselves be organised at different levels of abstraction, for example a control structure for Processing Plants that may be specialised into one for Chemical-Plants, one for Paper-Plants, one for Food-Processing Plants. Discovering these abstract patterns is therefore important to standardise solutions for process control software.

Frameworks can be identified by identifying abstract types of interface between abstract objects. Methods based on role modelling are good for this. These abstract interfaces are then published as standards for the domain to which vendors provide specialised conformant objects. Frameworks can be specified and saved as a composite definition by instantiating and linking objects and links that use the most abstract definitions. These frameworks can then be used by cloning them and replacing the abstract objects with more specialised objects.

Ideally it would be nice to be able to leave a component of the network specified only as an abstract deferred class for which any compatible subclass could be specified. At present a CDL definition must be used to define the abstract class rather than allowing icons to be created on the fly.

10.2.2 Classification Choices and Rationale

Ports conform to abstract data types, called interface types, that are defined independently of classes. This allows abstraction of the relationships or contracts that objects make about each other. It also reduces the need to duplicate class code across a network in order for a client to access a remote server, only interface type definitions need be duplicated.

In all, there are three notions of type, defined separately:

- interface types, used to define conformance across links by providing abstract data types for port objects at both ends of a link. Interface types are defined in CDL in a similar way to classes but using the keyword "interface" instead of "class". Interface types are not used to instantiate any objects. Ports are instantiated along with the object in which they are embedded and not by explicit instantiation of an interface type.

- class definitions, allowing classes to share behaviour. This includes sharing of embedded port objects. Class definitions are defined in CDL and can be instantiated using the configuration languages or factory services of the interpretation layer.

- composite types, configuration composites provide an interface by exporting unbound ports of instances that are defined within the composite. Composites are defined using the configuration language or visual editor. They can be saved and cloned in several places, thus treating them as a class.

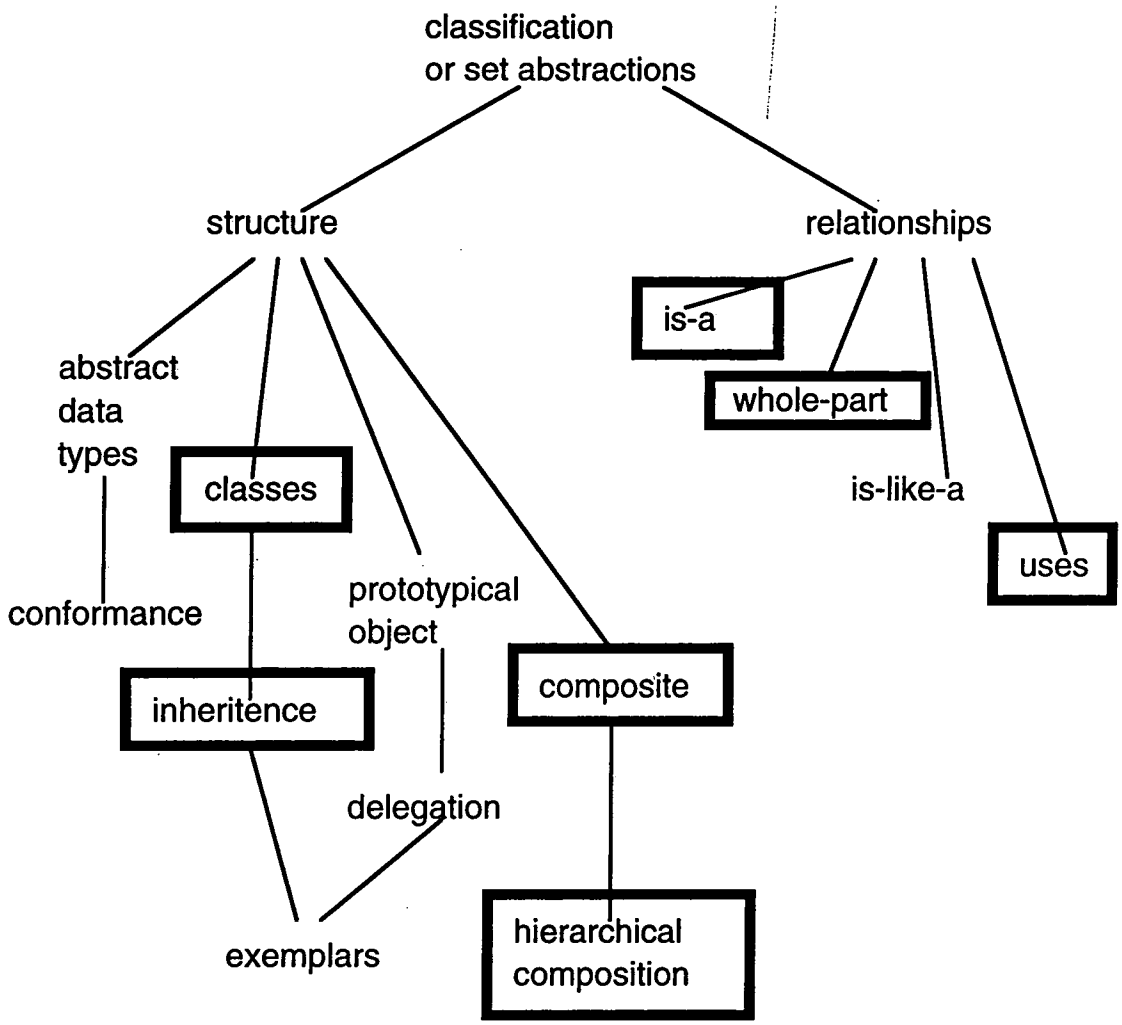


Figure 78 Types of Classification or Sets Supported (shown in boxes)

This separation of interfaces supports three distinct inclusive relations:

- subclassing relations between classes to share implementations and embedded ports.
- subtyping relations between interface types to allow abstraction of interfaces for binding.
- hierarchical composition of instances and composites within other composites to define any configuration using explicit links between ports.

Note that subclassing and subtyping relations are specified at the level of types whilst hierarchical composition relations are specified at the level of ports. This is more primitive and can even be used to simulate inheritance as shown in the Figure 79.

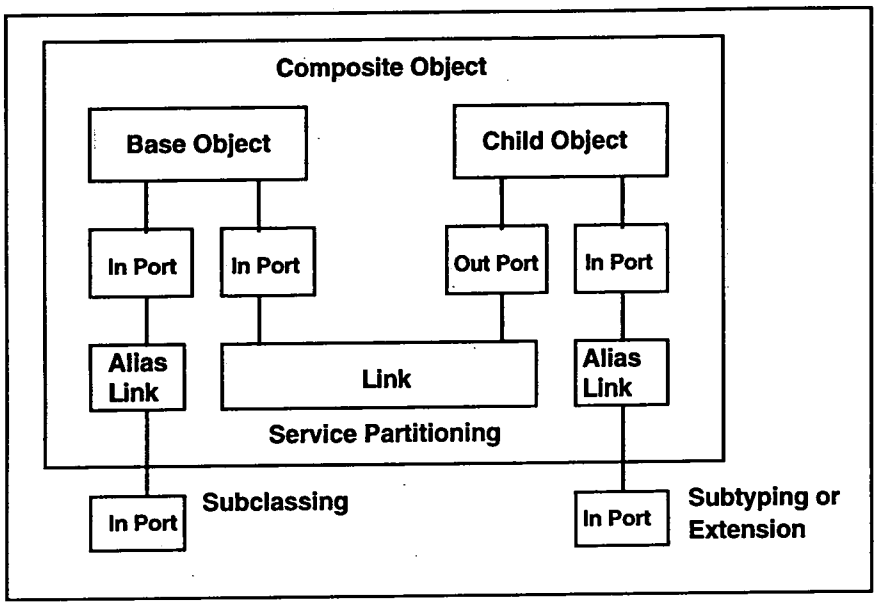


Figure 79 Using Hierarchical Composition to Synthesise Inheritance

Composite types can be synthesised by defining ports for the composite and associating these ports with the ports of embedded objects. Conventional inheritance is also supported at the programming level. The classes instantiated as objects in a composite can inherit from other classes. Consequently synthesis operations occur at many levels.

10.2.3 Insulation Choices and Rationale

OpenBase supports *partial specification* through transparency mechanisms and automated support for selecting management policy.

Transparent properties include locations, access mechanisms, bindings, replication, recovery, concurrency. A component programmer does not need to specify these properties. The interface between components and the system supports transparency in these aspects. Instead these properties are defined either explicitly by the application engineer on configuring components or on allocating a configuration model to the distribution model. The component programmer and application engineer specifies semantic information that is exploited to make a sensible choice. In this way management decisions are split between the component programmer, the application engineer and the system.

OpenBase also supports partial specification of data initialisations since the CDL compiler generates an editor that can be used at configuration time to specify the initial values for attributes and these camn vary between instances.

There are three times that a partial specification can be fully resolved: during configuration using the visual editor, during allocation based on automated management policies and at runtime using generated code.

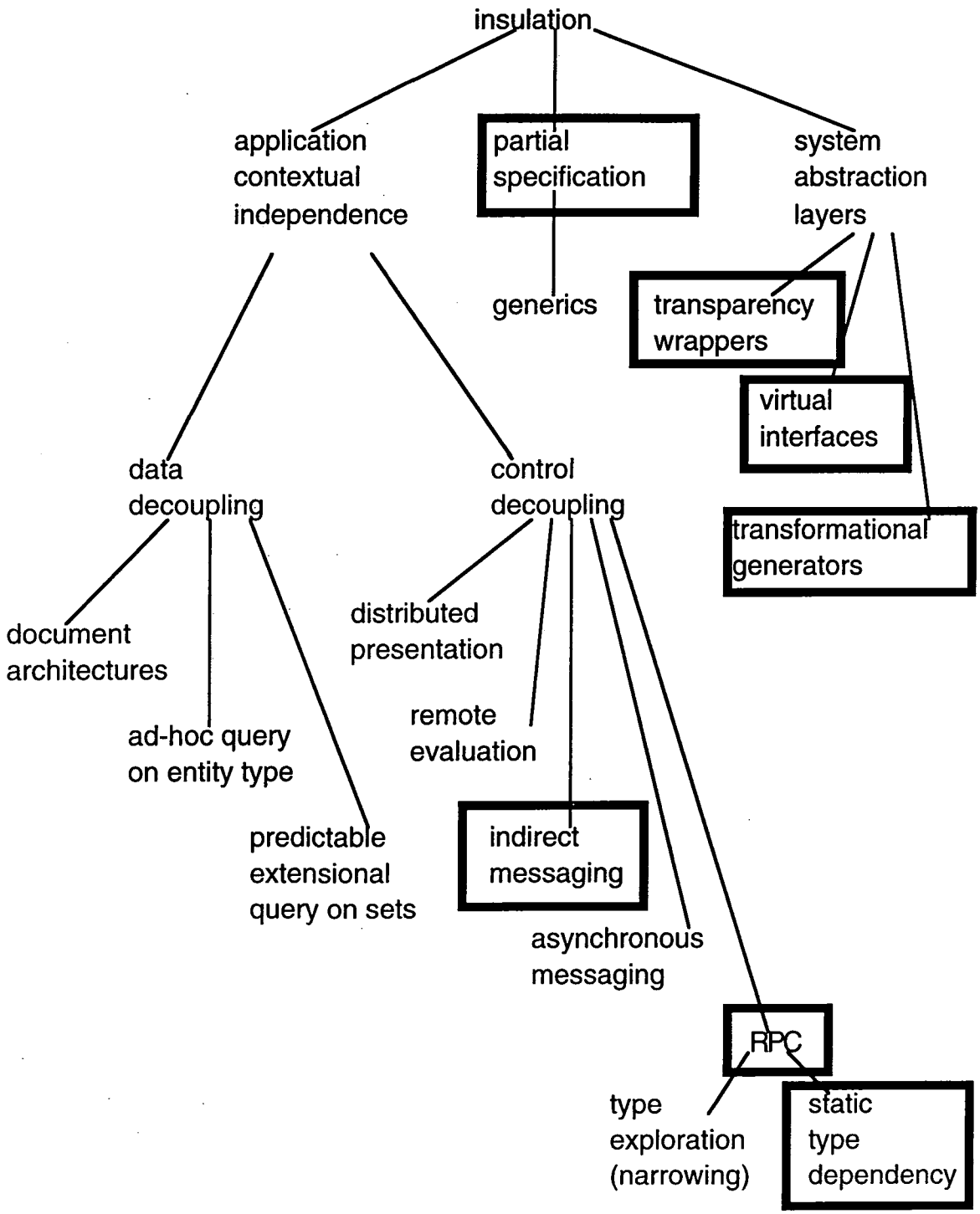


Figure 80 Types of Insulation Supported (shown in boxes)

Runtime support for resolving partially specified components relies on the *automated generation* and instantiation of *transparency wrappers* in the executable code. Transparency wrappers apply a layer of code to each component to hide the normal complexity of using the runtime support of the underlying technology. This includes stub objects and metaobjects and are generated by the CDL processor and configured by the interpretation layer. These hide details such as location, operating system usage, persistence mechanisms, bindings to other components, data formats and so on.

Through the use of the Class Definition Language to generate wrappers that are portable across platforms, OpenBase provides a single *virtual interface* to the operating system services. This includes configuration services and method invocation services as well as conventional operating systems resources and services. The OpenBase environment may be viewed as a virtual assembly machine that presents to the programmer a world of independent configurable objects that send and receive messages through standard interfaces.

OpenBase supports *indirect messaging* through typed ports. Bindings are resolved externally to an object, rather than the component being responsible for importing a binding, as in trading systems, or for interrogating interfaces to derive a binding at runtime, as in narrowing systems. Outports are implemented as embedded typed objects.

An alternative implementation for typed ports could have merely used public attributes that pointed to the destination. These attributes could be externally configured because they are public. However this does not make the encapsulation model clear nor does it allow special port behaviours such as optional ports, deferred synchronous ports, etc.

OpenBase ports use *statically-defined RPC* as the general mode of communication. This means a client needs to know the interface of services required in advance and include that interface definition in declaring the outport. The CDL description generates the typed outport object to be linked and embedded in the object implementation.

Dynamically-defined RPC, such as the CORBA dynamic invocation interface, provides an alternative scheme. By treating type information as enumerated data values at runtime, a generic port can be written to interrogate the interface of any inport at runtime and compose an appropriate request message on the fly. This avoids type dependencies between an object and the interface type of its outports. However little advantage is gained unless the implementation is also written to explore type information of services used at runtime as in narrowing systems.

Writing narrowing code would be unbearable for general application programming. However a dynamic interface is useful for certain components that manipulate many types of objects such as configuration managers, protocol converters or debuggers and is therefore supported as a special option.

10.2.4 Substantiation Choices and Rationale

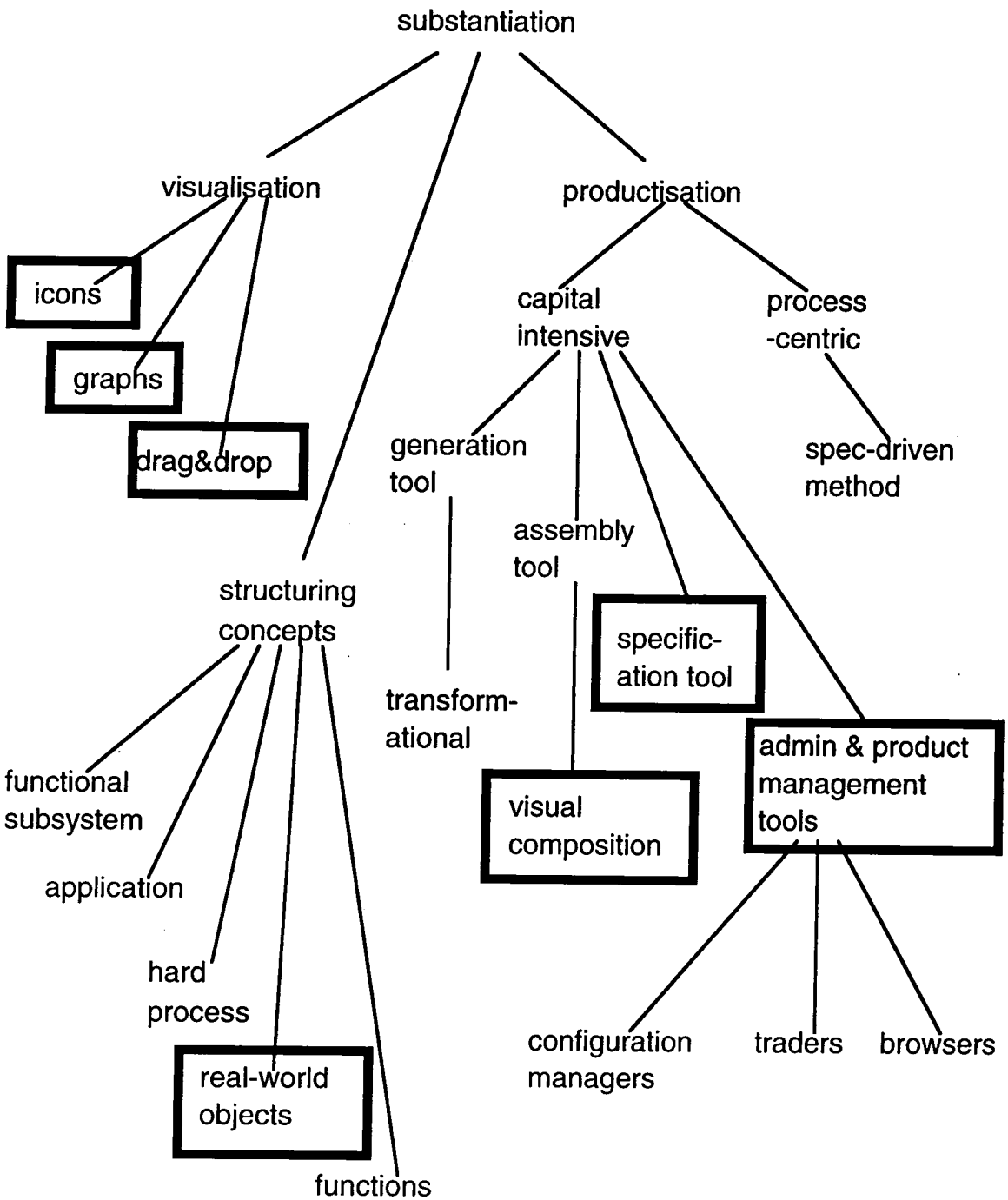


Figure 81 Types of Substantiation Supported (shown in boxes)

OpenBase supports program visualisation using icons for components, and visual composition to define graphs by dragging and dropping icons from a library.

Components are modelled using an object-oriented approach. This produces a more natural model than conventional approaches and identifies objects that correspond to real-world concepts familiar to end users.

The overall approach is capital intensive, demanding an up front investment in infrastructure, tools, domain standards and new development skills. The tools layer of OpenBase currently provides a limited selection of administration and management tools such as browsers, imake and configuration monitors. Much further work is required to extend the tools to offer complete support for all aspects of component development, integration and reuse.

The configuration management features of OpenBase support system management. Dynamic reconfiguration of an on-line system is even possible since only the affected processes must be put in a quiescent state.

OpenBase can also be used as a procurement planning tool. It would be possible to buy the icons without the runtime support for components in order to analyse and evaluate possible applications of components. New icons may be added to model existing applications and describe improvements

The EDL can be used as a system description language to model and plan changes to the physical network configuration.

10.3 Requirement for Graphical Tools

Most people think using pictures. Most programmers design their programs using a combination of text and pictures. Currently they must translate their conceptualisations back and forth on development and maintenance. The aim of graphical programming systems is to work directly with conceptualisations using a variety of pictures and text which reflect different viewpoints and aspects of the system. It seems unlikely that a single programming language can effectively support all these views. OpenBase provides a consistent framework that allows languages to be combined and related. The adoption of a primitive repository allowing flexibility in the representation of graphs, facilitates the development of a number of programming styles and visual metaphors.

The requirements for a graphical programming system include:

- the programming system should provide multiple views of the same model. Different views can be unified by manipulating a common representation or by defining translators between representations. Distributed systems are often too complex to represent at a single level of abstraction such as code views. Different views are needed to deal with different issues in isolation at different levels of abstraction.

- the programming system should provide a complete general environment. It should allow the structured description of design knowledge and requirements as well as source code.

- the programming system should equally support both textual and graphic languages. Textual languages are necessary to express complex data structures and algorithms and can be used by highly skilled component programmers to implement components. Graphical views are particularly useful for presenting large volumes of information to synthesise a collective understanding of a large program and can be used by application engineers to compose systems and navigate the structure.

- the programming system should be extensible so that new or custom tools can be developed, for example by providing a meta-programming interface. It should provide facilities to simplify language definition such as representations for common language evaluation mechanisms.

- there should be a consistent support framework that provides general purpose primitives for building tools and languages. It should not enforce specific mechanisms.

10.4 Description of OpenBase Tools

This section describes the OpenBase tools.

In supervisory control systems, the required system knowledge to make management decisions is frequently divided between the plant engineer who knows requirements, the application developer who knows object implementations and the system engineer who knows the management mechanisms. The OpenBase development tools are designed to bridge this division without exposing engineers to new complexities. This is possible because issues are clearly separated and distinct programming tools are provided for the application developers and plant engineers to express their different knowledge of the application properties.

The following programming interfaces are provided:

- OpenBase/C++, i.e. standard C++ using a skeleton file generated by a preprocessor.
- the Class Definition Language, a declarative class description and modelling language.
- the Environment Definition Language, a textual system description language for describing the physical configuration of the plant.
- The configuration language, a textual configuration programming language.
- the visual editor, a graphical configuration programming language.

Flexibility is built-into these languages using: hidden defaults programmed by the system programmer, expressions which describe abstract properties, and expressions which are explicit requests. Naive users express their knowledge as properties. More advanced users express precise instructions.

The detailed design of these tools was not part of the thesis research. Rather than define the tools as they were implemented by Prism, this section illustrates an example of what the programming interfaces might look like. This was provided by the research as input to the language design to articulate the technical vision supported by the prototype. No responsibility is taken for the final implementation of the languages and tools, as described in (OpenBase, 1993).

The runtime platform presents a flat object model to the applications layered on it. Various structures may be imposed on the flat model in order to represent different aspects of system behaviour. These structures consist of sets of interconnected objects called composites. They contain other composites and thus hierarchies may be constructed. A composite is identified by its name and is defined by the objects and links it contains and the interface it exports to other composites.

The configuration language and visual editor provide commands to instantiate and link objects within a composite. In the visual editor, objects are dragged and dropped as icons from palettes and their ports are selected to define links between objects using dialogues that are activated on clicking on an icon. Links may also be specified between objects and the interface exported as the composite interface. In the configuration language there are keywords like *inst* and *link* that allow objects to be created and ports connected in a similar manner to CONIC described in chapter 5.

Figure 82 shows an example composite for a storage tank, consisting of instances of the classes: PressureTag, Threshold, Alarm, Logger, and PrintQ. The pressure I/O point is linked to a threshold which signals an alarm if the threshold is exceeded. This composite can be saved as a simple framework and be instantiated at several points in the hierarchy. The hierarchy is viewed in the lower window.

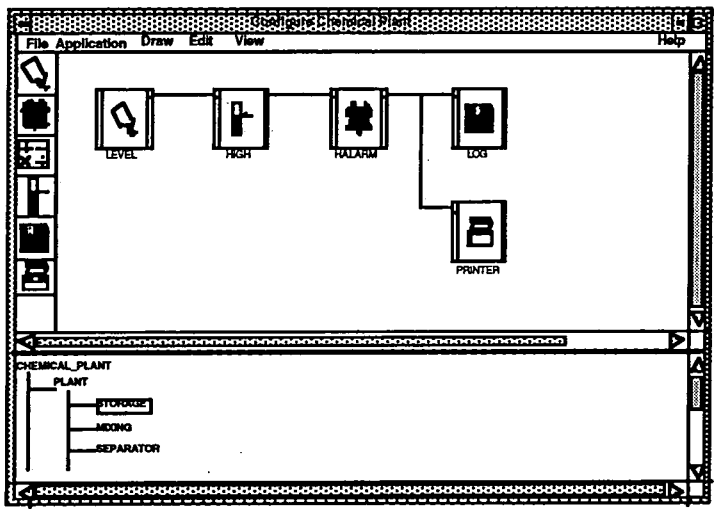


Figure 82 Example Graphical Editor

Figure 83 illustrates IDL for the PressureTag class and the Threshold class. Ports are typed according to the interfaces, IntIn and BooleanIn. These interfaces can contain groups of services or contracts. However the example shows singular contracts, providing the services reset and trigger.

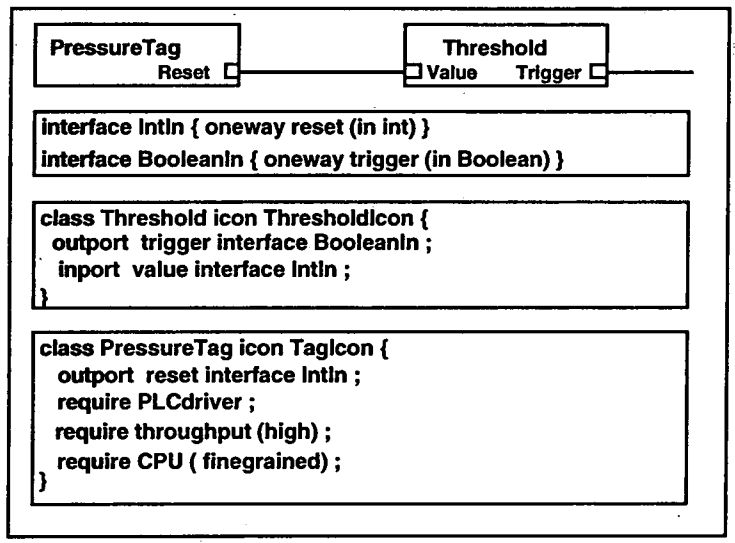


Figure 83 Example Interface Definition Language

The convenience of interface definitions for describing properties is shown for the PressureTag class. This class has a hidden system port for communicating with the PLC driver. This is declared by a "require PLCdriver" statement. This declaration increases the likelihood of the pressure tag being located near the PLC driver.

The other require statements show the sort of property that may be modelled. This was provided merely as input to the design of the languages and still needs to be formalised before implementation. Instead expressions with a free format for naming and setting any variables were supported i.e. throughput and CPU become attributes taking the parenthesised values of high and fine-grained. These variables are matched with provides statements that are usually defined in classes modelling physical resources. This mechanism needs to be formalised to recognise and support different categories of constraint such as priority trade-off, threshold, dependency, constraint, matching the reward evaluation objects of chapter 9. In teh example, the two require statements are intended to reflect the following properties:

- Pressure can change frequently. This could be declared explicitly in a "require throughput(high)" statement. This increases the likelihood of the threshold object being located near the tag. It can also increase the likelihood of an optimised RPC scheme being used for communication.

- I/O tag classes are not CPU intensive. This can also be declared, increasing the likelihood of multiple I/O points in a single process. This is important if a loading threshold is based on the number of objects in the process and does not take into account actual sizes.

Devices like Programmable Logic Controllers (PLCs) are associated with I/O points using dialogue boxes, as in Figure 84 below. Unlike application objects, device drivers for these PLCs are explicitly allocated to a fixed location using EDL. The relationship between an I/O point and the device is a strong mobility constraint since data-point sampling is best done locally to the device driver. This is integrated with the "require PLC statement" in the interface definition above and is fully supported in the prototype.

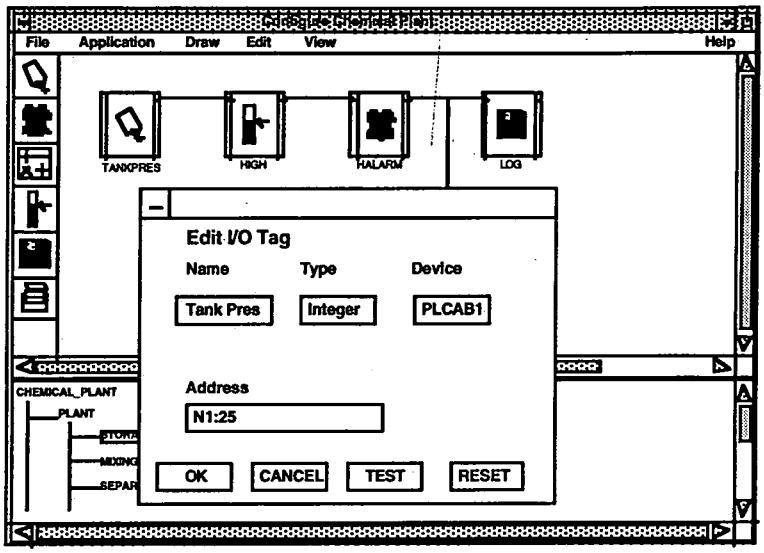


Figure 84 I/O Driver as mobility constraint

Implicit control over allocation is supported by capturing these mobility constraints as properties of the configuration using the languages and visual tools.

Explicit control over configuration structure and allocation is also supported. This is achieved by providing different types of composite. The type may be selected by pointing and clicking on the appropriate icon using the visual editor or through the use of different keywords for composites that enclose inst and link commands in the configuration language. The different types of composite include:

Configuration groups - these are logical composites of application components. They also define the lifetime or activation policy for objects. A configuration group may be: preload, where it is configured at loading time and unloaded at reconfiguration time; discardable, where it may be activated and passivated to a persistent store to conserve resources when not needed at runtime; or unloadable, where it is discardable but using the same initial state on each activation.

Processor composite - models the allocation of configuration groups to processes of the operating system. These may be generated using the automation support in the distribution model. The visual tool provides a command to submit configuration groups to be allocated to processors.

Distribution composite - models the allocation of processor composites to nodes. This too may be automated if configuration groups are submitted for automatic allocation.

Other composite types will be added to model atomic actions, replications. These may be activity flows across links in a configuration group or clusters of objects , orthogonal to the configuration group hierarchy.

Figure 85 shows how an Environment Definition Language can be used to describe resource availability and physical constraints. Standard templates such as the PLCLIB library and the RISC nodetype, simplify the job of modelling the plant. Extra resources such as the printer on node A, are added to those provided by the templates. Templates are parameterised to allow easy configuration. In the example, the location of the PLC driver for PLCAB1 is explicitly named as a template parameter and this introduces the mobility constraint for datapoint objects that require a PLC driver.

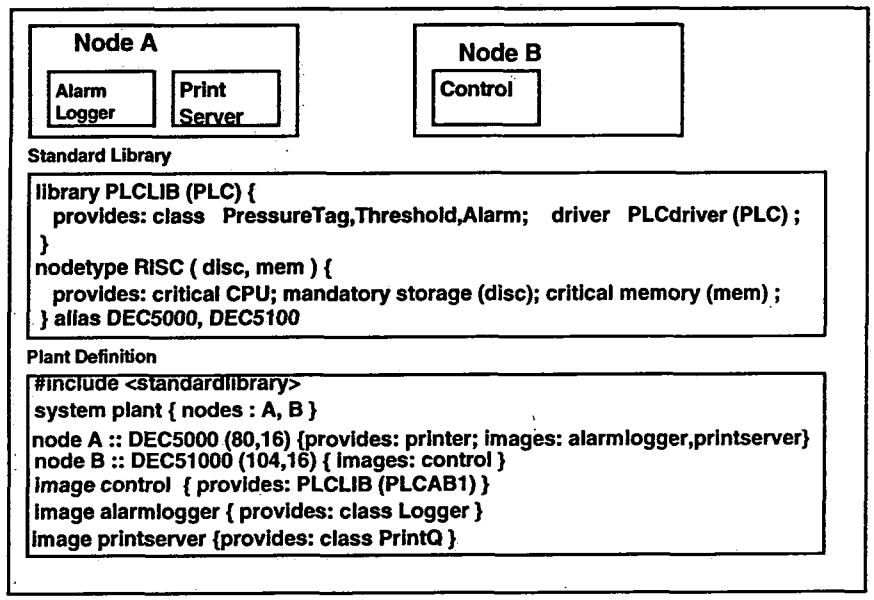


Figure 85 Example Environment Definition Language

Class specific constraints are described in IDL. Physical constraints are described in EDL. Figure 86 shows how a role specific constraint or contextual constraint could be described on a activity flow using the graphical editor. For example, if the tank stores dangerous material, then alarm conditions are critical in this context.

The final choice of location and RPC service depend on the combined set of class specific, role/context specific and hardware specific constraints.

Note that OpenBase supports system level and application level functionality that is usually hidden in the visual editor. This is added-value functionality provided specifically for the process industry. This includes PLC drivers as above and alarm viewers. Hidden aspects can be selectively viewed as in shown in the figure.

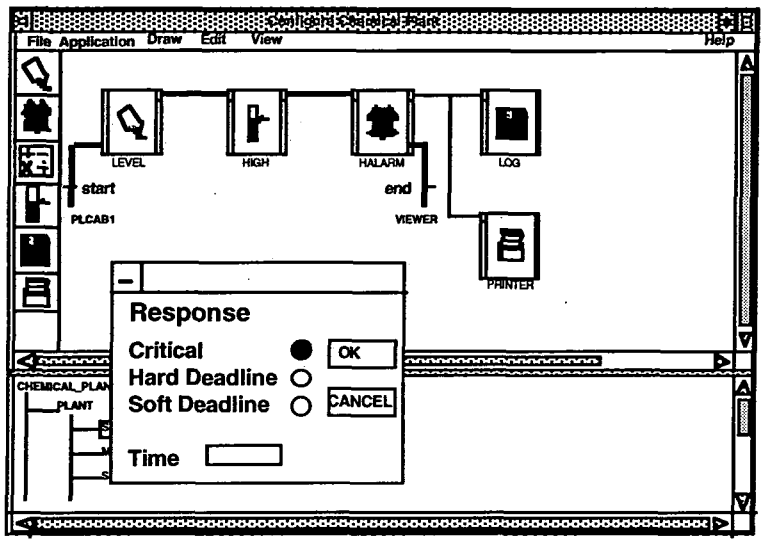


Figure 86 Example Control Flow Requirement

10.5 Conclusions to Chapter 10

This chapter has described the design choices that were made in the design of the OpenBase development tools and illustrated the design with simple examples showing one way that the adaptive programming concepts might manifest themselves in the configuration. This was not implemented as part of the MSc research and although most of the concepts are supported by the prototype, the interface is incomplete.

Object management mechanisms are now well established. The biggest barrier to their commercial adoption is the complexity they present to programmers. The OpenBase programming languages do not encumber the programmer with detailed system knowledge. Instead they provide specification languages to describe the environment and the operational requirements as properties. They hide many decisions behind defaults appropriate to the process industries and provide tailored concepts and notations. This approach bridges the gap between the commercially acceptable object oriented languages that do not support distribution and the emerging distributed object oriented languages that have not gained acceptability.

Initial resource availability is described in EDL. Requirements, priorities and application knowledge are made available using CDL and the graphical editor.

Existing object oriented methodologies offer poor support for implementing process/processor architectures. This project completely removes the need to design a process architecture. Programmers generate a logical model which is not explicitly distributed. Instead constraints and requirements are described. The transformation of requirements to a process architecture design is entirely automated.

The reward and availability structures are not restricted to resources such as memory, storage, CPU. The policy recognises the significance of co-location decisions in meeting operational requirements such as deadlines, performance targets, reliability and currency goals. Properties may be specified on activity flow between objects so that rewards can accumulate based on proximities to related services.

Chapter 11 Conclusions

This thesis has described a design for an integrative architecture for distributed, multi-vendor, object-based applications. This chapter describes the key contributions and conclusions of the research and makes some recommendations for future work.

11.1 Contributions of Thesis

The research efforts in developing this thesis, have made the following contributions:

surveyed different approaches to programming system design - this re-examined the purpose and basic principles of programming system design and identified a number of the practical problems that need to be overcome to meet the future challenges of application integration.

extended an analytic framework of architecture goals and abstract design principles - this extended framework can be used informally to evaluate architecture.

defined an approach to architecture design - this is based on purposive modelling of the key architectural components. A purposive approach unifies evaluation/ selection activities and modelling/ design activities. This makes it better suited to the emerging world of collaborative off-the-shelf infrastructures where selection and integration issues dominate over bespoke development.

designed an architecture for enterprise-wide integration - in which diverse applications, infrastructure technologies, and system platforms can be integrated into an open distributed system. This applied the purposive approach. The end result provides a commercially exploitable architecture. Enterprise integration is growing in importance in a number of market sectors: telecommunication , banking, manufacturing, utilities, government and the petrochemical industry.

provided solutions to some practical problems of programming system architecture design - this includes object specification, configuration, object library management, object wrapping, object distribution, object messaging, object persistence, and runtime object management. The discussion highlighted integration problems for different techniques.

realised support for a new development paradigm based on adaptive graphical objects - this paradigm differs from that of conventional programming systems, in particular, in the emphasis placed on integration rather than development. This manifests itself in the requirement for a complete environment for integrating applications from different vendors rather than a universally applicable language for developing them. It is a product management system as well as a development environment.

clarified and provided background information for the paradigm - by elaborating on four essential characteristics: component based lifecycles, binding models for visual composition, adaptive object management and integrative standards.

The overall contribution explored the issues and tested-out the limitations of a new design approach to integrative architectures.

11.2 Conclusions of Thesis

The key position argued by this thesis is that a separation of concerns in the programming system architecture through intermediate layers of representation and interpretation helps to integrate a range of techniques that can solve the problems of application integration.

In order to justify this position, the hypotheses listed in the research objectives of Chapter 1 have been investigated. In evaluating these hypotheses, the research has sought to find answers to the following questions:

1. What are the key challenges of application integration?
2. Do the shortcomings of existing solutions require a shift in programming system architecture?
3. What are the characteristics of a more appropriate solution?
4. What architectural separation of concerns will simplify such a solution?
5. How does this separation of concerns help integrate techniques?
6. How valuable is the resulting integrative architecture?

The following conclusions have been made about each hypothesis:

hypothesis 1: more radical solutions are required to solve the individual challenges of application integration.

Existing solutions to the key challenges have serious shortcomings :

development of applications spanning different types of machine

This is being addressed by open distributed computing standards provided by bodies such as the OMG, ISO, OSF.

dealing with the complexities of programming distributed systems,

Distributed systems introduce complexity such as new forms of partial failures, network latency, indeterminate orderings of messages, heterogeneous representation schemes, application partitioning decisions. It is much easier to reason in a language designed to address these complexities, that supports clustering and partitioning to allocate objects over a network of processors, scheduling and synchronisation policies to deal with concurrency, abnormal behaviours to deal with failures. Yet there is no widely used distributed language that supports these features directly. These features need to be integrated into existing programming systems.

integration of applications that have been developed independently by different vendors,

In order to establish a mature cross-vendor development culture, the emphasis in tools must shift from process-centric development tools to product-centric component management tools. A capital-intensive investment in product management tools and frameworks is required to help locate, understand, and assemble independently developed components.

In order to avoid vendors needing to examine foreign code to collaborate, the emphasis much change from implementation to specification of components. Inheritance is promoted as the reuse mechanism. In practice today, inheritance more often concerned with implementation than specification. Programmer's paranoia about the efficiency of unknown code can only be overcome if the properties dictating poor implementation choices are captured abstractly in

interfaces. These properties can be complex. For example, the relative efficiency of a collection for different intended access patterns and data types is dependent on its memory management strategy: taking into account member size, dynamic growth etc.; and its access structure: array for fast direct access; stack for nested accesses; cyclic buffer for iterative.

reuse of application components in different parts of the enterprise.

Conventional decompositional approaches tend to focus on the whole, with the finer parts defined in the context of larger parts making them more contextually dependent and less reusable. Likewise development focused at a specific target platform introduces system dependencies that inhibits reuse when ported across the enterprise. Reusable component developers should work with a compositional model within a virtual environment that isolates him from contextual or system specificity.

Reusability is such a difficult problem that there is unlikely to be a single general purpose model. Rather there will be different approaches for different domains and the approach will evolve. Component developers need to be insulated from the reusability framework itself. One way to do this is to explicitly separate the reuse mechanism from construction mechanism.

hypothesis 2: a paradigm shift is required to meet the combined challenges of application integration.

new lifecycles

The whole notion of traditional system development lifecycles must go in order to gain the strategic advantages of development speed, flexibility, reuse and integration across vendors. The need to rapidly develop and reconfigure systems introduces incrementality and iteration into the lifecycle. The need to overcome cultural impediments to reuse mean that we need to distinguish tasks of construction from reuse. The need to support increased collaboration in the development splits lifecycles across vendors and users.

need for explicit binding model

Ideally the way components are interconnected should be selected from common models not invented for each occurrence. Binding models should be explicit to allow components to be developed in isolation. Choosing effective sets of rules for composing independent components is not easy. In typical systems today the binding model is not designed but encoded within the participating classes. Collaboration is facilitated by agreeing on a specific binding model that may limit the universality of the programming environment.

Client server models for interconnection are too restrictive since the interface between clients and servers fail to capture exit points. Hence clients can not be reused in different contexts without violating their encapsulation and examining their exit points. More flexible reuse can be achieved by publishing exit points in interfaces.

need for adaptive management

Objects need to be adaptive to allow for:

selective control over distributed management policy - to meet different dependability requirements. Complexity can be masked by high level abstractions but an all-or-nothing policy may result in poor performance if used everywhere.

more varied lifecycle and environment - The context and purpose of a component is deferred till the component is instantiated in a composite. The runtime environment is deferred until the composite is actually allocated to a processor. If context or environmental assumptions are introduced prematurely, the range of usage of a component across the enterprise can be overly restricted. Reusable objects are adapted to their context and environment by configuration. The behaviour of an object can only be partially resolved at

each stage of its development and different parties contribute: the producer, any consumers defining another service, the user or application engineer configuring components and the system according to policies.

direct manipulation using visual representations - Programmers have had a monopoly over users and seek to deskill and distance them. Programmer and user are traditionally two distinct roles. The IT industry has been delivering complete insular solutions that hide complexity rather than manage it. Little work has been done to present more user-friendly views of program complexity. This has resulted less flexible and maintainable programs. The intent of the program is hidden in its translation to source code leading to the overhead in tracing from requirement to code representation. Visual notions such as symbology, adjacency and connectivity can be used to represent the users intent more directly, to allow complexity to be adaptively managed by the user.

application level standards

Commercial distributed system standards are now gaining widespread acceptance. However to date they merely define the glue used to connect heterogeneous applications. There is no industry-wide consensus on structure. Application level interface standards are required for plug-compatibility.

With the establishment of new trading partners and mergers to open up new markets, businesses invariably have a diversity of systems and networks, policies and users. Standardisation is required at higher levels of abstraction to allow higher level integration and to unify technology/ mechanism diversity in lower layers. Various protocol definition techniques have been discussed and incorporated into the proposed architecture.

hypothesis 3: adaptive graphical objects provide an appropriate paradigm shift.

The proposed solution has the following characteristics that make it appropriate:

component oriented lifecycle

A component-oriented lifecycle separates the task of component integration to build applications from the task of component production. This distinction defines the two roles of component developer and the application engineer. This is particularly useful in highly specialised collaborative markets where different specialist vendors each contribute part of the solution and a third party is responsible for integration.

binding model through visual configuration programming

The proposed visual tools realise a graphical configuration programming tool. The premise of the configuration programming approach is that a separate, explicit structural description and representation of the application structure is essential to support all phases of software development, including: abstract system and framework specification as a configuration of component specifications; construction of the system as interconnected components by construction tools; installation as a physical configuration of processes across a network; runtime management as adaptive changes to the configuration; and evolution as user-driven changes to the configuration.

adaptive management

Adaptive management can occur in many ways, including:

1. by direct user manipulation of a visual representation,

2. by refining a partial specification, throughout its lifecycle,

3. by generation of specific code from a common declarative specification,

4. by substituting different runtime support components such as types of communication channels,

5. by re-allocation of objects to physical resources.

Each of these approaches is combined in the prototype.

Visually adaptive tools make programming intuitive thus opening component integration up to less technical audiences, removing the users dependency on highly skilled system integrators. Most users do not want to deal with complex management issues. It is important that they can describe their requirements abstractly and leave it to the system to adjust.

The partial specification approach introduces intermediate abstract representations that hide the complexity of mechanisms. The component developer uses declarative interfaces to describe requirements and properties rather than mechanisms. The application engineer uses graphical tools to annotate the software configuration with context-specific requirements and properties. These abstract requirements and properties are represented by the configuration model state. The system must interpret this state correctly.

A generation-based approach makes components specified in a common specification language portable. This defines an abstract assembly machine that can be mapped across heterogeneous platforms and management mechanisms. Component programmers need not be aware of the context in which the component will be used. Likewise application engineers need not be aware of the physical location of the objects. The objects used may actually move between machines (migration) or provide local copies (replications) without affecting correspondent objects.

Substitution of infrastructure services is made possible by selecting services according to the configuration model state. In the prototype this has been demonstrated for the policies governing the location of objects, their bindings and choice of interaction protocol. Selection policy may be tuned for specific market sectors, such as using process industry defaults for supervisory control.

Selective control over the allocation of objects to processes is supported by providing overridable allocation policies. Allocations can be specified explicitly by the programmer when designing a configuration or by allowing the system to distribute the processes automatically using built-in rules about the resources available, supplied by the application engineer. This means a reasonable system can be generated quickly yet highly optimised systems can be allocated by hand.

domain-specific standards

Domain standards are important for cross-vendor collaboration. Well targeted integrative architectures setting domain-level standards in their industry sector can provide a vital role in stimulating collaboration across producer-supplier channels. Collaboration on a smaller scale within a specific industry like the petrochemical industry is easier to achieve than the degree of collaboration required for general purpose international open distributed system standards. The proposed architecture has been involved in higher-level standardisation efforts in the petrochemical industry such as POSC.

hypothesis 4: a purposive approach can result in a suitable separation of concerns.

The problem and solution space for the programming system design were composed and decomposed according to the abstract goals and principles identified in the evaluation framework.

There are several observations about this separation of concerns:

explicit roles makes complex system manageable - Conventional object oriented system architectures not only use a single language but also often use a single mechanism, namely inheritance, without any explicit rules to constrain or enforce the development style to adopt to deal with distribution, reuse or late binding. The programmer must deal with many issues without much explicit guidance. In contrast, the separation of concerns in the prototype programming system made it's development and usage manageable because there we were able to define clear distinct explicit roles and rules for each architecture component and each programming task.

visible rationale - the purposive approach adopted by the thesis resulted in an architecture whose high level structure could be rationalised. This rationale gave focus and direction to the development of each of the components. The purposive approach helped the architect review design decisions and separate concerns.

difficulty in formalising creative architecture design process - however the generality and rigour of this approach should be questioned. Developing an appropriate framework for programming system design is a very creative task. The framework did not necessarily help identify design options only capture and organise them in retrospect. The creative process of architecting is difficult to formalise by imposing some framework. Rather it draws on experience and intuition.

limited generality - as a result of this, the framework adopted here is closely coupled to design ideas and may have limited generality in other architecture design problems or as a general purpose evaluation framework.

co-ordination techniques more important - the purposive approach is also insufficient from an implementation viewpoint. Its success depends on the architect identifying appropriate implementation techniques for the relationships between different parts of the architecture. The research identified and applied a number of co-ordination techniques that can be used to separate concerns: pre-processors, metaobject protocols, generation, projections. Success depends on these techniques not exclusively the separation of concerns.

value of goal driven approach - in spite of the limitations, there is definite merit in attempting to drive creative processes from higher level goals and attempting to define abstract principles. Such principles should be recognised as heuristics. Empirical data is needed to validate these heuristics, hence this approach is best mixed with prototypical proof-of-concept studies.

hypothesis 5: a separation of concerns facilitates the integration of different techniques that solve the basic challenges of enterprise integration.

The separation of programming concerns to an integrated set of tools has a number of benefits:

open language design framework - conventional tensions in language design between efficiency and expressivity are resolved by allowing developers to optimise a high level language using additional programming interfaces. This may exploit meta-object protocol or adaptive techniques.

developers can use the right tool for the job - the separation of concerns simplifies the semantics of each tool and allows developers to select the right tool for the job at hand. An enterprise includes a wide range of developers of differing ability, demanding different tools to perform different functions. Issues of computation, interoperation, interconnection and distribution are clearly separated. This allows developers to use mainstream object oriented language like C++ for defining computations to be performed and still reason about

distributed properties using an appropriate language interface. Programming interfaces may be customised to each particular task.

explicit development roles and programmer tasks - As well as supporting each task, the separation of tools makes the distinction between programming tasks explicit. This simplifies the role and responsibilities of each developer. This can overcome many of the cultural, motivational and competency issues in collaborative development.

separation makes mechanism integration and extension easier - inheritance is an overloaded mechanism, dealing with type checking, polymorphic binding, behaviour sharing. Introducing additional features such as concurrency mechanisms causes semantic interference with the compositionality semantics of inheritance. By separating mechanisms for type checking, binding and sharing, it is easier to extend their semantics to deal with distribution and pluggability or to integrate other mechanisms/techniques. The price paid is a loss in elegance. The overloaded use of inheritance may be perceived as one of the strengths of object oriented languages yet it is also limiting in complex environments.

configuration meta-model sets solid foundation for extensibility - A clear and concise structural description of the software in terms of types, instances, connections and composite elements provides a powerful extensible foundation for designing additional advanced tools to support specification, design, management and evolution. Tools can be developed in isolation using these common elements as primitives to build higher level concepts .

higher level tools results in integrated design - Conventional object-oriented approaches are limited by the weak process for relating class, dynamic and processing views of the design. Integrated design environments provide precise programming interfaces for dealing with different modelling aspects. This formalises the techniques used to integrate these views.

split management policy avoids complexity overload - Flexible management strategies are particularly important for fine-grained objects where management overheads can be costly if the wrong policy choice is made. The required knowledge to make policy decisions is usually split between the component developer who understands the class semantics, the plant engineer who understands the contextual requirements; and the systems engineer who understands the engineering trade-offs of different choices. Splitting decisions across these parties, avoids overloading any one programmer with overall responsibility. Adaptive management techniques allows decisions to be deferred and integrated.

integration of diverse techniques - the proposed architecture provides a synthesis of techniques. This includes RPC (synchronous, asynchronous and deferred synchronous), indirect naming, explicit construction/ destruction, configuration time type checking, ports, passive objects, implicit and explicit superobject encapsulation, property specification at different levels of abstraction and different granularity, IDL processors, stub subtyping, metaobject protocols, selective transparencies, standardised binding models, distinct interface types, domain frameworks, inheritance, hierarchical composition, partial specification, indirect messaging, wrappers, virtual interfaces, transformation/generation, object graphs, drag-and-drop, visual composition, specification tools, and product management tools.

realises a cross-platform development strategy - the interpretation layer provides a common interface to the runtime support making the architecture ideal where complex applications must span a diverse heterogeneous network. The virtual interface hides and manages migration to evolving standards. It does this by hiding the standards interface behind high level tools and abstraction layers.

hypothesis 6: the proposed architecture has general value as an integrative architecture in different market sectors.

Using the classification hierarchies of the evaluation framework, we can relate the techniques employed in its construction to all eight abstract principles defined in chapter 7: encapsulation, set abstractions, polymorphism, interpretation, selective properties, universal

protocols, substantiation, component insulation. The architecture provides both a virtual infrastructure that masks users from technological complexities, and a component-based development environment that allows users to freely integrate components developed by different vendors.

We can also relate the principles to the supported basic goals and challenges of chapter 3: software and infrastructure abstraction, flexible sharing, correctness, software and infrastructure evolution, dependability, application interworkability, large scale re-use. This profile of goals suggests a flexible infrastructure that has potential to enable and facilitate cost-effective enterprise integration.

The architecture meets the enterprise integration requirement in its target domain. The key technical requirements of the petrochemical industry are for improved reuse, evolution and integration to provide increased plant flexibility and performance, less downtime, greater stock diversification, and new safety standards. The architecture meets these requirement by building on existing resources rather than redeveloping them time and time again, by avoiding redundant duplication of functions, by flexible management, by allowing rapid application reconfiguration, by integrating products from multiple specialised vendors and by providing integrated global views of the entire process plant.

The approach may be overkill for many systems that only require a subset of the goals described above. For example UNIX IPC may be sufficient for data sharing where application interworking, evolution and reuse are not important. Client server databases may be sufficient where data is not distributed. Object oriented databases may be sufficient for applications with distributed data which do not require visual tools for rapid integration and reconfiguration, and where object sharing is sufficient for application interworking where there need be no complex messaging patterns. X windows may be sufficient to allow remote users for any application.

Rather than applications that use distribution as a means to achieve dependability, the architecture is intended for use where data, resources and users are inherently physically distributed, and there is a significant amount of local processing on each node. It is in these situations that simple client-server architectures are inadequate and peer-to-peer networks of objects are needed to provide flexible enough application partitionings. In process control , local processing at the pumps and valves of a chemical plant is essential to ensure rapid reaction yet a remote view of the data is essential for overall supervisory management. Similar systems include network management and command and control.

The ability to compose applications quickly makes it appropriate for rapidly changing decision support systems such as support systems for the new trend in manufacturing processes like just-in-time or small batch.

There are restrictions on the sort of application that can be addressed:

dataflow/control flow interconnection model - the choice of binding model implemented in the prototype restricts the applicability to application domains that lend themselves to data-flow or control-flow programming such as process control where components act as function-blocks to manipulate data flows, multimedia where medias are combined, imaging where images are transformed.

systems with complex structure - Like the various forms of bubble-charts used in structured design methods, visual composition is most appropriate when there a lot of bubbles with complex flows between them. Many applications have the majority of behaviours described by two bubbles and do not gain much from such graph-based notations. This makes it most appropriate to applications with complex processing. These applications typically arise in defence, telecommunications, scientific systems and process control.

complex behaviour restriction - in structured design the bubbles themselves are specified elsewhere and may be internally complex. Visual composition on the other hand relies on the application engineer understanding a component's behaviour from its iconic presentation on the screen. This restricts the applicability in domains where the behaviour of components is internally complex, for example as arises frequently in financial applications.

General Lessons

There are a number of general lessons that can be learned from the experiences gained on this project:

- object orientation unifies the approach to achieve several goals: abstraction, sharing, evolution, interpretation, interworking and dependability. Objects are useful for both graphical display, for runtime components and for unifying intermediate control, modelling and evaluation abstractions. Object not only encapsulate problem solving abstraction but they also encapsulate what can be managed allowing uniform management protocols to be defined. Traditionally difficult tasks like load balancing become easy.

- object oriented databases like ObjectStore provide an efficient repository for graph based structures of objects that are accessed by navigational queries. This is ideal enabling technology for repository based integration of development and configuration tools.

11.3 Limitations of Results

The goals that remain unaddressed seriously limits the universal applicability of the product developed by the research across different domains:

groupware goal - Failure to support co-operative working or groupware limits its appropriateness in business enterprises where groupware is important in some part of the business. This is an unfortunate omission as groupware is an ideal candidate for visual composition.

enterprise modelling goal -Failure to incorporate mainstream enterprise modelling approaches like business process re-engineering, workflow models, enterprise database modelling distances the product further from the business marketplace. Visual composition is an applicable metaphor for workflow specifications.

re-engineering goal - Failure to address application re-engineering by providing explicit support for migration and integration of existing legacy applications, distances the architecture from commercial systems even further. The product uses its own programming model that even makes it difficult to integrate mainstream object oriented software.

dependability goal - Dependability support is limited. This makes it inappropriate in the following situations:

- for computationally expensive parallel applications, such as simulation, modelling or imaging systems. The extra communication overhead introduced by port and link objects may be greater than the time saved by subdividing a computation. Active objects and parallel language extensions are unsupported. Parallel extensions are more appropriate since they put more of the onus for support on the compiler.

- for reliable data processing applications such as reservation systems, order processing systems due to the current lack of support for transactions.

- for conventional real-time systems. This is a serious omission in process control. Individual components may be designed internally along real-time principles but

activity flows across components do not have deterministic bounds unless the components are allocated explicitly. Explicit control undermines the underlying philosophy of masking complexities. There is a dichotomy between transparent adaptive management of systems using a "best efforts' philosophy and the predictability demanded by conventional real-time systems that generate inflexible designs based around cyclic executives. However the conventional approach is proving to be unsuitable for the growing number of large-scale real-time systems. More flexible adaptive approaches to the design of real-time software is an area of active research (Burns and McDermid, 1994).

correctness goal - Correctness is supported by static type checking at configuration time and by exception handling and exception throwing at runtime. Exception handling puts the onus on the programmer to ensure correct behaviour. However this may be inadequate in a distributed system due to a whole new set of potential errors. Correctness in the presence of partial failures is very difficult to enforce or verify. Debugging is more complex and requires distributed monitors and new debugging tools.

There are a number of general limitations in the approach:

ambiguity of intuitive tools - visual tools have been adopted for their intuitiveness. Intuitiveness is an important requirement for an engineering discipline. However intuitiveness introduces conflict with the programming language requirement for precision necessary for a program to execute on a digital machine. This conflict is minimised by restricting the use of visualisation to programming-in-the-large. However as behaviours become more complex, there is a need to trade intuitiveness for formality introducing model-based, algebraic, logical or hypergraph formalisms.

formality needed for validation/maintenance - the weak interpretative semantics of the configuration model and visual tools make validation difficult. If more formal semantics were defined it would be possible using a proof system to take semantics and prove properties of specifications such as safety, liveness, fairness, or timing properties say in temporal logic. The configuration model could even be represented in logic e.g. using prolog. Programs could then be generated by deductive synthesis using explicit rules and policies, rather than hiding the policy in generation algorithms.

need for exploratory tool - an alternative to formal validation would be to use the tool in an exploratory manner, constructing composite structures and testing assumptions about behaviour on the fly. The view of a composite as an executable specification compensates for its lack of formality to some extent. An Exerciser tool could be provided to make exploratory development and testing more efficient.

limited scaleability of repository-based tools - the use of a large central meta-repository may lead to a unwieldy and inflexible environment in which to add multiple new tools. The use of translators and transformation tools to exchange information between tools would lead to a more open extensible tool architecture.

11.3 Recommendations for further work

The author needs to gather more empirical data to investigate different trade-offs and overheads in the design of different management policies. The research was limited to formulating an architecture and evaluating it. No real experimental data was taken.

The distribution model was designed using rather ad-hoc algorithmic techniques. Different logics and AI techniques such as constraint based reasoning should be researched more fully and applied to the problem of automated policy management.

Researchers need to explore more open language technologies focusing on feature integration and semantic interference. They should define clearer semantics for metaobject protocols, frameworks or patterns, and composition using inheritance, hierarchical composition or delegation. They should also define semantics for distribution including clustering and partitioning, synchronisation and scheduling, adnormal behaviours.

Practitioners need to refocus on evaluation and integration methods for Component-off-the-Shelf applications and infrastructure components. There are a number of open practical issues that need to be resolved before component-oriented development can become a reality. For a start components need to be organised and retrieved. The question as to what makes a good component will persist for some time. How big or small is a component? How generic and how adaptable can a component be? How complex can a component be?

Prism need to instil more discipline in the way they conduct their research and development activities. This thesis describes first attempts to formulate a conceptual architecture description and programming model to be used to explore design decisions rationally. The whole approach of Prism is lacking formality and rigour. For example, the hybrid use of inheritance and hierarchical composition is unclear. It is not only theory that is lacking. The examples used by Prism are over-simplistic, typically consisting only of a tag, a threshold and an alarm. Yet real world scenarios are much more complex. Further research should start with more realistic CASE studies, to define the requirements more clearly.

The architecture is originally targeted at the specific domain of process control. The infrastructure and policies should be optimised for this domain before broadening its scope.

Extra tool support needs to be developed to make the environment complete. This includes debugging tools, tools to analyse configuration with respect to deadlock, liveness fairness, tools to find and select components, specification tools to aid understanding of components, tools to capture and retrieve other forms of design knowledge.

Additional management behaviours need to be supported at runtime in order to widen the scope for exploitation, such as planned dynamic reconfiguration using programmed change scripts, transactional control and integrity constraints for unplanned dynamic reconfiguration, atomic actions, multimedia streams, group interaction. The distribution model needs to be extended to define management policies for these behaviours. New dialogues and properties need to be added to the visual editor and Class Definition Language.

Part IV Appendices

Appendix A: References

ACA, 1992, DEC ACA Services, System Integrator and Programming Guide, Part AA-PQKMA-TE, Digital Equipment Corp., April 1992.

G.Almes, A.Black, E.Lazowska, J.Noe, 1985, The Eden System: A Technical Review, IEEE Transactions on S/W Engineering SE-11 (1), Jan 1985, p43

ANSA, 1989, ANSA: An Engineers Introduction to the Architecture, available from APM Ltd, Cambridge, UK, Architectural Report TR.03.02.

ANSA, 1991, A Model for Interface Groups, available from APM Ltd, Cambridge, UK, Architectural Report AR.002.00

ANSA, 1992, ANSA Atomicity Model and Infrastructure, available from APM Ltd, Cambridge, UK, Architectural Report AR.004.00

ANSA, 1993, Architecture and Design Frameworks, available from APM Ltd, Cambridge, UK, Architectural Report AR.38.00

ATOS, 1994, Review of ATOS, AI Watch, Vol 3 No 7, July 1994, p 1-10.

H.Bal, J.Steiner, A.Tanenbaum, 1989, Programming languages for Distributed Computing Systems, ACM Computing Surveys, Vol 21 No 3 Sept 1989

Beta, 1987, The BETA Programming Language, by B.Kristensen, O.Madsen, B.Moller-Pedersen, K.Nygaard, In: (Wegner, 1987).

R. Bellinzona, M. Fugini, V de May, 1993, Reuse of Specifications and Designs in a Development Information System, Proc. IFIP WG8.1 Working Conference on Information System Development Process, Sept 1993, p79-96, also in Visual Objects, Report , Ed. D.Tsichritzis, Universite de Geneve, 1993, p.247-264.

J.Bennett, 1987, The Design and Implementation of Distributed Smalltalk, OOPSLA '87 Proceedings, p.318-330, October 1987.

S.Bijnens, W.Joosen, P Verbaeten, 1994, A reflective invocation scheme to realise advanced object management, in Proceedings of the ECOOP 93 Workshop on Object Based Distributed Programming, R. Guerraoui, O Nierstrasz, M. Rivell (Ed.), LNCS 791, Springer-Verlag, Kaiserslautern, Germany, 1994.

K.Birman, T.Joseph, 1991, Exploiting Replication in Distributed Systems, Chapter 15 in (Mullender, 1989)

A.Birrell, B.Nelson, 1994, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems., Vol 2 No 1, p.39-59, Feb 1984.

G.Blair, J.Gallagher, J.Malik, 1989, Genericity vs Inheritence vs Delegation vs Conformance, JOOP, Sep./Oct. 1989, p.11-17.

G.Blair, J.Gallagher, D.Hutchinson, D.Shepherd, 1991, Object Oriented Languages, Systems and Applications, Book, Pitman Publishing, London, 1991.

G Blair, R.Lea, 1992, The Impact of Distribution on Support for Object-Oriented Software Development, S/W Engineering Journal Vol 7 No 2 March 1992

G.Bock, 1994, Examining Agents in the Workplace, Patricia Seybold Group, Workgroup Computing Report, Vol 17, No 12 Dec 1994.

B.W.Boehm, 1988, A Spiral Model of Software Development and Enhancemem-nt, IEEE Computer, May 1988, p.61-72.

G.Booch, 1991, Object-Orientated Design with Applications, Benjamin/Cummings Publishing, 1991.

S.Brost, T.Malone, 1986, Towards Intelligent Message Routing Systems, in Computer Message Systems- 85, Ed. R.Uhlig, Proc. of second international symposium on computer message systems, Amsterdam, Holland.

A.Burns and J.McDermid, 1994, Real-time, safety-critical systems: analysis and synthesis, Software Engineering Journal, Vol 9 No 6, Nov 1994.

R.Campbell, N.Islam, Choices: A Parallel Object-Oriented Operating System, p393-451.

L.Cardelli, P.Wegner, 1985, On understanding types, Data Abstraction, and Polymorphism, Computiong Surveys, Vol 17 No 1, p477-522.

Carriero, N.Gelertner, 1986, The Linda S/Net's Kernel, ACM Transactions on Comuputer Systems, Vol 4 No 2, May 1986.

P.Chen, 1976, The entity-relationship model, ACM Transactions on Database Systems No 1 Vol 1, 1976.

S.Chiba, T.Masuda, 1993, Designing an extensible distributed language with a meta-level architecture, in Proceedings of the ECOOP 93, O Nierstrasz (Ed.), LNCS 707, Springer-Verlag, Kaiserslautern, Germany, 1993, p483-502.

R.Chin, S.Chanson, 1991, Distributed Object-Based Programming Systems, ACM Computing Surveys, Vol 23 No 1, March 1991.

CLOS, 1989, Object-oriented programming in common lisp: A programmers guide to CLOS, book by S.Keene, Addison-Wesley, MA.

P.Coad, E.Yourdon, 1991, Object Oriented Analysis and Object-Oriented Design, two books, Prentice Hall, 1991.

T.Coatta, G.Neufeld, 1992, Configuration Management via Constraint Programming, Procs. of International Workshop on Configurable Distributed Systems, IEE / Imperial College, London, Mar 1992, p.90-101.

P.Collinson, 1992, The Network File System, SUNEXPERT Magazine, Nov 1992, p26-33.

S.Cook, J.Daniels, 1994, Designing Object Oriented Systems - Object Oriented Modelling with Syntropy, Prentice-Hall, New York.

B.Cox, 1990, Planning the Software Industrial Revolution, IEEE Software, Nov. 90, p. 25-33.

T.DeMarco, 1978, Structured Systems Analysis, Yourdon Press, 1978.

F.DeRemer, H.Kron, 1976, Programming in the large vs Programming in the small, IEEE Transactions on S/W Engineering Vol SE-2 (2), June 1976.

L.P. Deutsch, 1989, Design Reuse and Frameworks in the Smalltalk-80 System, in Software Reusability, Vol. II, (eds. T.J. Biggerstaff , A.J. Perlis), ACM Press, 1989, p. 57-71.

M.Endler, J.Wei, 1992, Programming Generic Dynamic Reconfigurations for Distributed Applications, Procs. of International Workshop on Configurable Distributed Systems, IEE / Imperial College, London, March 1992, p.68-79.

E. Fieume, D. Tsichritzis, L. Dami, 1987, Temporal Scripting Language for Object-Oriented Animation, Proc. Eurographics'87, North-Holland, 1987.

D.Gelernter, 1985, Generative Communication in Linda, ACM Transactions on Programming Languages and Systems, Vol 7 No 1, p.80-112.

M.Glykas, P.Wilhelmij, T.Holden, 1993, Object-orientation in enterprise modelling and information systems design, IEE Colloquium on Object-Oriented Development, Digest No. 1993/007, Jan 1993.

I.Guffick, G.Blair, 1992, Building Configurable Distributed Systems using the Kitara Object-Oriented Language, Procs. of International Workshop on Configurable Distributed Systems, IEE / Imperial College, London, Mar 1992, p.60-67.

A.N. Habermann, D. Notkin, 1986, Gandalf: Software Development Environments, IEEE Transactions on Software Engineeering, vol. SE-12, no. 12, Dec. 1986, p.1117-1127.

D.Harel, 1987, Statecharts, A visual formalism for complex systems, Scientific Computing Programming, No 8 , p231-274.

W. Harrison, H. Ossher, M. Kavianpour, 1992, Integrating Coarse-Grained and Fine-Grained Tool Integration, Proc. CASE '92, July 1992.

R. Helm, I. Holland, D. Gangopadhyay, 1990, Contracts: Specifying Behavioural Compositions in Object-Oriented Systems, Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, vol. 25, no. 10, Oct 1990, p 169-180.

A.Herbert, 1989, The Computational Projection of ANSA, Chapter 19 of (Mullender, 1989).

HOOD, 1991, HOOD Reference manual, Issue 3.1, July 1991, HRM/91-07/ V3.1.

D. Ingalls, 1988, Fabrik: A Visual Programming Environments, SIGPLAN Notices, vol. 23, no. 11, Nov. 1988, p 176-190.

I.Jacobson, M.Christerson, P.Jonsson, G.Overgaard, 1992, Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, Wokingham, England.

P.Jeremaes, D.Coleman, 1993, FUSION: A second generation object-oriented analysis and design method, IEE Colloquium on Object-Oriented Development, Digest No. 1993/007, Jan 1993.

J.Johnson, D.Hudson, 1993, OLTP Monitors, Patricia Seybold Group, Distributed Computing Monitor, Vol 8 No 12, Dec 1993.

E.Jul, H.Levy, N.Hutchinsom, A.Black, 1988, Fine grained mobility in the Emerald System, ACM transactions on Computer Systems, Vol 6 No 1 feb 1988, p109-133.

A. Julienne, L. Russell, 1993, Why You Need Tool Talk, SunExpert Magazine, vol. 4 no. 3, March 1993, p. 51-58.

G. Kappel, J.Vitek, O.Nierstrasz, S.Gibbs, B.Junod, M.Stadelmann, D.Tsichritzis, An Object-Based Visual Scripting Environment, ITHACA Report, p123-148.

M. Kass, 1992, CONDOR: Constraint-Based Dataflow, Proc. SIGGRAPG '92, p. 321-330.

G.Kiczales, J.Rivieres, D.Bobrow, 1991, The Art of the Metaobject Protocol, The MIT Press, Cambridge, Massachusetts, London.

S.Klerer, 1988, The OSI Management Architecture: An Overview, IEEE Network, Vol 2 No.2 , p.20-29.

J.Kramer, M.Sloman, 1987, Distributed Systems and Computer Networks, Section 1-4, Prentice-Hall, 1987.

J.Kramer, J.Magee, K.Ng, 1989, Graphical configuration programming, IEEE Computer, Vol 22 No 10.

J.Kramer, 1990, Configuration Programming - A Framework for the development of distributed systems, Proc. Int. Conf. on Computer Systems and Software Engineering, Tel Aviv Israel, May 1990.

J.Kramer, J.Magee, M.Sloman, N Dulay, 1992, Constructing Object-Based Distributed Systems in REX, S/W Engineering Journal, Vol 7 No 2, 1992.

M.Kramer, 1993, Enterprise System Management, Patricia Seybold Group, Distributed Computing Monitor, Vol 8 No 6, June 1993.

M.Kramer, 1994, Message-oriented middleware, Patricia Seybold Group, Distributed Computing Monitor, Vol 9 No 6, June 1994.

G.E. Krasner, S.T. Pope, 1988, A Cookbook for Using the Model-View-Controler User Interface Paradigm in Smalltalk 80, Journal of Object-Oriented Programming, p.26-49, Aug./Sept. 1988.

K. Lieberherr, I Holland, 1989, Assuring Good Style for Object-Oriented Programs, IEEE Software, Sept. 89, p. 38-48.

H.Lieberman, 1986, Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems, Procs. Conf. on Object-Orientated Programming Systems, Languages and Applications (OOPSLA '86), Portland, Oregon.

B.Liskov, 1988, Distributed Programming in Argus, Comms of the ACM, Vol 31, No 3 , Mar 1988.

A.Macintosh,1994, Enterprise Modelling : Review of Existing Work, TIP Report, Logica Cambridge.

R.Maclean, 1991, Object Oriented Analysis and Design - A Critical Review, Logica Cambridge Report, Cambridge UK, ASE TR5.

J.Magee, J.Kramer, M.Sloman, 1989, Constructing Distributed Systems in CONIC, IEEE Transactions on S/W Engineering Vol 15 (6), Jun 1989.

J.Magee, N.Dulay, J.Kramer, 1992, Structuring Parallel and Distributed Programs, Procs. of International Workshop on Configurable Distributed Systems, IEE / Imperial College, London, Mar 1992, p.102-117.

R.Marshak, 1993, Action Technologies Workflow Products, Patricia Seybold Group, Workgroup Computing Report, Vol 16, No 5 May 1993.

R.Marshak, 1994, IBM's FlowMark - Object Oriented Workflow for mission-critical applications, Patricia Seybold Group, Workgroup Computing Report, Vol 17, No 5 May 1994.

D.McCue, 1992, Selective Transparency in Distributed Transaction Processing, Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, England, April 1992.

B.Meyer, ref [1], 1992, Eiffel: the language, Prentice Hall, 1992.

B Meyer, ref [2], 1992, Applying 'Design by Contract', IEEE Computer, Oct. 1992, p. 40-51.

R.Milner, 1980, A Calculus of Communicating Systems, LNCS Vol. 92, Springer verlag, 1980.

N.H.Minsky, D.Rozenshtein, 1987, A law based Approach to Object-Oriented Programming, OOPSLA '87 Proceedings, Oct. 1987, p. 482-493.

R.Moran, 1992, ISIS News, in ISIS Reliable Distributed Object Manager, Product Summary , Nov 1992, ISIS Distributed Systems Inc., now owned by Stratus Computers, New York.

J. Morel, J. Faget, 1993, The REBOOT Environment, Proceeding of the Second Inter'l Workshop on Software Reusability, March 1993, p 80-88.

S.Mullender, 1989, (Ed) Distributed Systems, ACM Press, Addison-Wesley Publishing.

G.Mullery, 1979, CORE - A Method for Controlled Requirements Analysis and Design, Proc. of 8th IEEEE International Conference on Software Engineering, Munich , 1979.

O.Nierstrasz, S.Gibbs, D.Tsichritzis, 1992, Component Oriented Software Development, Communications of the ACM, Vol 35 No 9 Sept 1992, p160-165, also in Object Frameworks, Report , Ed. D.Tsichritzis, Universite de Geneve, 1992, p.1-10.

O.Nierstrasz, M.Papathomas, 1990, Viewing Objects as Patterns of Communicating Agents, in Object Management, Report , Ed. D.Tsichritzis, Universite de Geneve, 1990, p.255

ObjectStore, 1995, ObjectStore Technical Overview, available from Object Design Inc., Swindon, UK (Tel. 44-1793-486111)

OMG, 1990, The Object Management Architecture Guide, Revision 1, Document No. 90-12-1, Object Management Group, Cambridge, MA (now superceeded by Revision 2, 92-11-1).

OMG, 1991, The Common Object Request Broker Architecture, version 1.1., Document No. 91-12-1, Object Management Group report, Oct 1991, Cambridge, MA.

OpenBase, 1993, OpenBase Technical Overview, available from Prism Technologies Limited, Team Valley, Gateshead, England.

Orbix, 1994, Orbix Programmer's Guide, version 1.2 feb 94, available from Iona Technologies, Dublin ,Ireland.

G.Parrington, 1992, Programming Distributed Applications Transparently in C++, Myth or Reality, Report available from Computing Laboratory, University of Newcastle upon Tyne, England.

POSC, 1992, Petrotechnical Open Software Corp - Industrial Strength User Power, Software Futures, November 1992, p.6-8 (see also October 1994 p.10-12)

R.Prieto-Daiz, 1991, Implementing Faceted Classification for Software Reuse, Communictaions of the ACM, Vol 34, No. 5, May 1991, p.89-97.

A.Profrock, D.Tsichritzis, G.Muller, M.Arder, 1989, ITHACA: An integrated toolkit for highly advanced computer applications, in Object-Oriented Development, Ed. D.Tsichritzis, Universite de Geneve, 1989.

J.Rambaugh, M.Blaha, W.Premerlani, F.Eddy, W.Sorensen, 1991, Object Oriented Modelling and Design, Prentice Hall, 1991.

T.Reenskaug, E.Andersen, A.Berre, A Hurlen, et al., 1992, OORASS: Seamless support for the creation and maintenance of object-oriented systems, JOOP, Oct 1992.

S.Reiss, 1987, Working in the Garden Environment for Conceptual Programming, IEEE Software, Nov. 1987, p16-27.

T.Rodden, I.Sommerville, 1989, Building Conversations using Mailtrays, In Proceedings of EC-CSCW, the first European Conference on CSCW, Gatwick-Hilton, Sept. 1989.

L.Rowan, 1992, Visual Programming, Patricia Seybold Group, Guide to Workgroup Computing Report, Vol 15, No 11 Nov 1992.

W.W.Royce, 1970, Managing the Development of Large Software Systems: Concepts and Techniques, Proceedings Wescon, Aug, 1970.

M.Rozier, J.Mantis, 1987, The CHORUS distributed operating system: some design issues, in Distributed Operating System Theory and Practice, Springer-Verlag, Berlin, Heidelberg, p262.

J.Rymer, 1992, Message Express, Patricia Seybold Group, Distributed Computing Monitor, Vol 7, No 4 , April 1992.

J.Rymer, 1993, IBM's System Object Model, Particia Seybold Group, Distributed Computing Monitor, Vol 8 No 3, p1-24.

J.Rymer, 1994, Middleware Roadmap, Distributed Computing Monitor, Vol 9 No 5, Patricia Seybold Group.

R.Rymer, M.Guttman, J Mathews, 1994, Microsoft OLE 2.0 and the road to Cairo - How object linking and embedding will lead to distributed object computing, Patricia Seybold Group, Distributed Computing Monitor, Vol 9, No 1 , Jan 1994.

K. Schumucker, 1986, Object-Oriented Programming for the Macintosh, Hayden Books, Hasbrouck heights, N.J.

Seybold, 1994 , Guide to Client/server and Distributed Computing Architectures, Patricia Seybold Group.

Seybold, 1994, ref [2], General Majic Hopes to put the Network to Work, Patricia Seybold Group, Distributed Computing Monitor, Vol 9, No 3, p.20-25.

J.Shirley, 1992, Guide to Writing DCE Applications, O'Reilly & Associates, Sebastopol, California, 1992.

S.Shlaer, S.Mellor, Object Oriented Systems Analysis, Modelling the World in Data, Yourdon Press/Prentice Hall, 1988.

S.Shrivatava, 1991, An Overview of the Arjuna Distributed Programming System, IEEE Software, Jan 91, p66.

S.Shrivastava, G.Nixon, G.Parrington, 1987, Objects and Actions in Reliable Distributed Systems, S/W Engineering Journal Vol 2 No 5 Sept 1987.

SIMULA, 1967 , O.J.Dahl, B.Myhrhaug, K.Nygaard, SIMULA 67 Common Base Language, Norwegian Computing Centre, February 1968,1970,1972,1984.

SMALLTALK, 1983, A.Goldberg, J.Robson, Smalltalk-80: the Language and its Implementation, Addison-Wesley,1983.

D.C.Smith, J.Susser, 1992, A Component Architecture for Personal Computer Software, in Languages for developing User Interfaces, Ed. B.Meyers, Jones and Bartlett Publisher, p31-56.

D.Steedman, 1990, ASN.1 The Tutorial and Reference, Technology Appraisals, The Camelot Press, Trowbridge, Wiltshire, UK.

S.Stepney, R.Barden, D.Cooper, 1992, A survey of object-orientation in Z, S/W Engineering Journal Vol 7 No 2 1992.

W.Stevens, UNIX Network Programming, Prentice Hall, New Jersey, 1990.

B.Stroustrup, M.Ellis, 1990, The Annotated C++ Reference Manual, Addison-Wesley, Reading, Mass., 1990.

H.Tonks, 1994, Improving Traditional Object models, Journal: Objects in Europe, Winter 1994.

D.Tsichritzis, 1989, Ed. Object-Oriented Development, ITHACA Report, 1989.

D.Unger, 1984, Generation Scavenging: a non-disruptive high performance storage reclaimation algorithm, ACM Sigsoft/Sigplan Practical Programming Environments Conference, pp 157-167, April 1984.

VisualAge, 1995, Product Literature and Demonstration, available from IBM, US Tel 1-800-3-IBM-OS2.

I.Walker, 1992, Requirements of an object-oriented design method, S/W Engineering Journal Vol 7 No 2, 1992.

J.Walpole, G.Blair, D.Hutchison, J.Nicol, 1987, Transaction Mechanisms for Distributed Programming Environments, S/W Engineering Journal, Vol 2 No 5, Sept 1987.

P.Wegner, 1987, The object-oriented classification paradigm, in B.Shriver and P.Wegner (eds.) , Research Directions in Object-oriented Programming, MIT press, 1987

R.Wirfs-Brock, B.Wilkerson, 1989, Object Oriented Design: A Responsibility Driven Approach, Procs. of the OOPSLA'89 Conference, SIGPLAN Not. (ACM) 24,10, New Orleans Oct 1989.

R.Wirfs-Brock, B.Wilkerson, L.Wiener, 1990, Designing Object Oriented Software, Prentice Hall, Englewood Cliffs, New Jersey.

R.Wirfs-Brock, R.Johnson, 1990, Surveying Current Research in Object Oriented Design, Comms of the ACM 33 (9) Sept 1990.

K.Woods, J.Mellor, J.Russell, 1993, ACUMEN: A configurable,distributed platform for object-oriented control systems, IEE Colloquium on Object-Oriented Development, Digest No. 1993/007, Jan 1993.

A.Yonezawa, E.Shibayama, T.Takada, Y.Honda, Modelling and Programming in an Object Oriented Concurrent Language ABCL/1, in Object Oriented Concurrent Programming, ed. M.Tokoro, pp. 129-158, The MIT Press Cambridge, Massachussetts, 1987.

J.A.Zachman, 1987, A framework for information system architecture, IBM Systems Journal, Vol 26, No 3, 1987 p. 276-292, Reprint order no G321-5298.

P.Zave, 1986, An Operational Approach to Requirements Specification for Embedded Systems, Software Specification Techniques, Addison-Wesley Publishing, 1986.

Appendix B: Survey of Integrative Standards - CORBA, DCE and ANSA

B.1 Introduction

This section describes the findings of the survey of integrative standards. This includes OMG CORBA, OSF/DCE and ANSAware. ANSAware provides a partial implementation of the ISO/ODP Reference Model standards effort. The purpose of this survey is to illustrate the way these integrative standards are used, so that we can both position OpenBase and provide background to the underlying technology behind OpenBase.

The introduction first provides background information on the different standards. It then describes the overall scenario that will be used to illustrate the basic concepts using example code. Finally it defines the functional framework used to structure the rest of the survey.

B.1.1 Basic Mechanisms of Integrative Standards

The integrative standards are based on remote procedure call mechanisms. This section describes this basic mechanism and the behaviours that are common across the integrative standards.

In RPC systems, an application consists of clients and servers. In object-based systems clients may themselves be servers for other clients and the term client and server can be taken to be roles that objects play in a interaction between objects.

All the integrative standards provide an interface definition language that is used to generate stub code in the client that acts as a local proxy for remote servers and skeleton code in the server that acts as a local surrogate for remote clients.

The client will make a request to invoke a method or procedure call on a server. The system may decide which server is to be used and where that server resides. This frees the client of knowledge about the location and activation state of servers.

When the client wishes to talk to a server, this is what happens:

- The client raises the request possibly with arguments.
- The system intercepts the request and encodes the arguments ready for transfer to the server's environment. This is done by what is called the stub.
- The stub sends the request and the server's skeleton receives it.
- The skeleton decodes the arguments and invokes the real server object's method or procedure call, passing to the method or procedure call the arguments.
- The server processes the request and passes back any result to the skeleton.
- The skeleton encodes the results ready for transporting across the network to the client's stub.

- The client's stub receives the results and decodes them.
- The stub returns the results to the client.

The following diagram illustrates this.

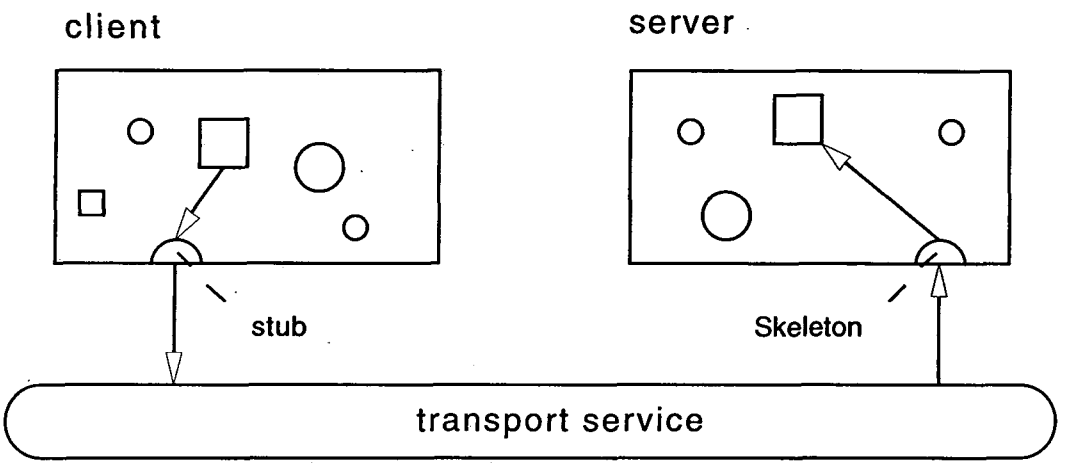


Figure 89 The Stub/Skeleton relationship

The following code shows an example of binding and invocation using Orbix. In Orbix the IDL compiler adds a static *_bind* method to the C++ class of any server. This is used to bind the client and can optionally take a name as parameter. Stubs are generated both on importing a reference as in the call to *_bind* and on passing or returning a reference as in the call to method *newAccount*.in the example. The system ensures that after the calls, both *obj* and *acc* pointers address the local stub for the corresponding remote object. The Orbix example looks like this:

```
//C++

foo()

{

   bank *obj = bank::_bind("Union Bank");

   account * acc = obj->newAccount("Joe Bloggs");

}
```

The system usually handles the following:

- name services, to get references to servers based on text names, like "Union Bank".
- activation, starting the server if an instance of it is not available,
- request dispatch (and scheduling if multithreaded)
- parameter encoding and decoding, also called marshalling,

- message transfer and delivery, shipping the request from the client to the server,
- error/exception handling, if the network fails or the server cannot be started,
- internal connection management (hidden behind binding mechanisms)
- synchronisation, between the client and server,
- security mechanisms, to prevent unauthorised object manipulation.

Each of these behaviours brings a performance overhead and new requirements on the programming interface. Optimisations can be defined by adopting different approaches to each behaviour. In particular, most behaviours are not necessary for calls between collocated objects and local optimisations should be built into the system to by-pass the mechanisms.

B.1.2 Background to OMG/CORBA

What is the OMG?

The Object Management Group (OMG) is a group of organisations and individuals formed in 1989. Today OMG's members number 330. This includes most if not all the major industry players including IBM, DEC, HP, Microsoft, Borland as well as end-users . All these people are trying to work together to develop specifications to maximise the portability, reusability and interoperability of commercial software.

The OMG works by organising committees and task-forces to choose specifications based on their members ideas. Obviously it is not an easy job to get consensus between these organisations. One of their key strategic goals is to catch distributed object technology before it becomes entrenched in market politics and customer preferences. The OMG have positioned themselves as a fast track to open distributed systems.

In November 1990, the OMG published the Object Management Guide outlining a conceptual framework in which to position and focus standardisation efforts. In October 1991, the OMG announced its adoption of the CORBA 1.1 specification. CORBA provides the communications component of the object management architecture. In 1994, the OMG announced the Common Object Service Specification which standardised the first group of added-value object services, naming, lifecycle and event notification.

B.1.3 What is the Object Management Architecture?

The object management architecture is an architectural framework with supporting detailed interface specifications, which seeks to drive the industry towards interoperable, reusable portable software components. The Object Management Architecture provides:

1)An Object Model, which provides a taxonomy of concepts and common semantics that characterise object models in a standard and implementation-independent way. It describes a core set that any conformant system should provide, called the Core Object Model, but also defines compatible extensions, called components, and profiles that group pertinent components for given technology domains.

2)A Reference Architecture, which attempts to map out the areas in which technology proposals may position themselves. It identifies and characterises the components, interfaces and protocols that compose the architecture but does not itself define them. Figure 90 show the architecture components, including:

- The ORB component enables objects to make and receive requests and responses.
- The Object Services component is a collection of value-added management services with object interfaces that provides basic functions for realising and maintaining objects. Examples include transactions, persistence and security.
- The Common Facilities is a collection of classes and objects that provide general purpose facilities for many applications. Examples include editors, widget libraries, device drivers, agents, printing facilities, electronic mail.
- The Application Objects are collections of particular end-user application components that are assumed to be less common and thus will come from a single vendor or developer. Application objects may migrate into common facilities by de facto processes as they become popular and widely used.

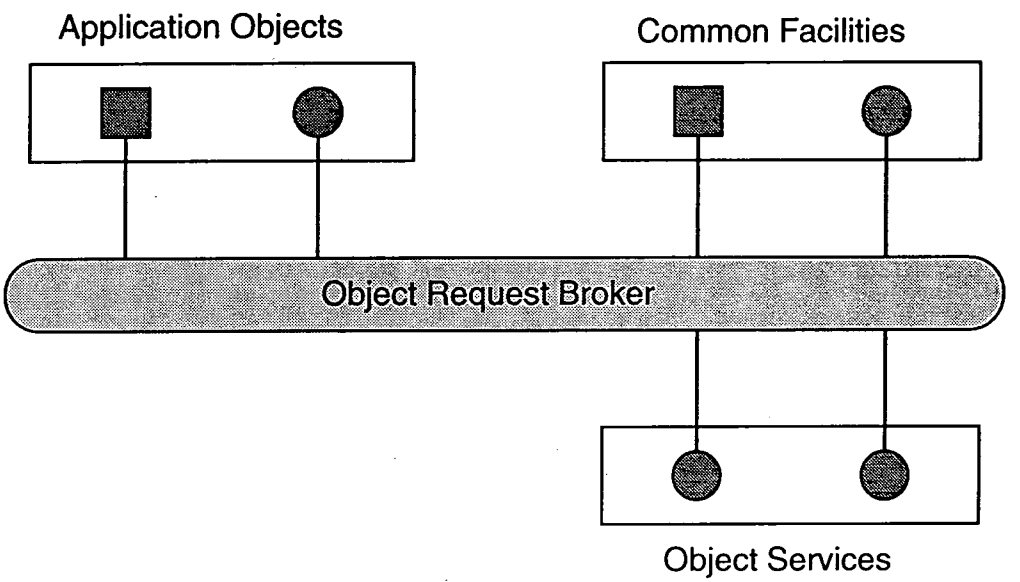


Figure 90 The Object Management Group Architecture Overview

B.1.4 What is CORBA?

The Common Object Request Broker Architecture (CORBA) is one component of the Object Management Architecture. It allows one process to talk to another process in either the same address space or across a network. In object terms, this would be a client object requesting the services of a server object. A 'marshal' is needed to control the request and give the results back to the client. This 'marshal' is the Object Request Broker (ORB). The ORB allows client/server or peer-to-peer relationships between the communicating objects.

The ORB provides the RPC mechanism for CORBA objects. OMG have a more general diagram for describing CORBA than that given in the lat section:-

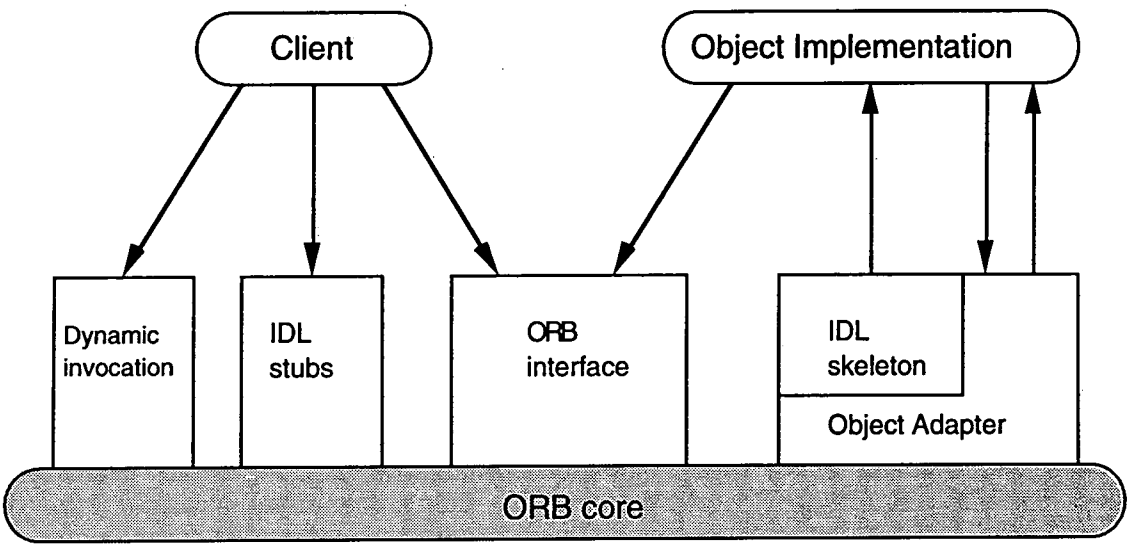


Figure 91 Common Object Request Broker Architecture (CORBA)

Each of the components in this more generalised architecture are described below.

Client

These are the client application object.

Object Implementation

These are the server objects.

IDL stubs

IDL is the Interface Definition Language. The interface to a remote server object may be described in IDL. The IDL compiler generates the source and header files for the proxy and surrogate in the appropriate language. These are then linked with the client and server implementations. This interface is simple and easy to use, but if the interface needs to be changed, the client and server require recompilation.

Dynamic Invocation Interface (DII)

The dynamic invocation interface (DII) can be used to invoke remote object methods dynamically without statically linking to an IDL definition. The client must provide metadata at runtime to describe the composition of the request in terms of method name and argument types. This is particularly flexible when combined with repository services that allow the client to explore interface metadata. CORBA 2.0 will extend the interface repository into a type repository by adding constraints, error messages and other type specific information. This will increase the amount of information that can be explored dynamically, making dynamic invocation more useful.

ORB Interface

This is used to control certain aspects of the ORB, especially on initialisation. This includes the client telling the ORB that it wishes to use an ORB object e.g. the client will bind to an object via the ORB. Also, the main body of the server object will tell the ORB that it is ready to start accepting requests once it has finished initialising. This is an example of when the client/server explicitly uses the ORB. Clients and servers will also be made aware of the ORB due to network failures or protection violation.

IDL skeleton and Object Adapter

This is the surrogate, it acts like the IDL stubs in reverse.

B.1.5 OSF/DCE Background

What is the OSF?

The Open Software Foundation is an industry funded, non-profit making organisation sponsored by a number of leading vendors including IBM, HP, Digital, Hitachi, Bull, Siemens-Nixdorf. The OSF seeks to define and provide open (vendor neutral) software solutions for the computing industry. They adopt a process of selection, certification and integration of software technologies.

Unlike the OMG, the OSF gets involved in centrally stitching technology contributions together into vendor distributions. This gives tighter control over the implementation of standards. However recently the style of funding is being changed to external development teams (probably due to the growing complexity of distributed computing environments).

The OSF have adopted the mach kernel and Unix SVR4 interface as the OSF/1 operating system standard. They have also released the Distributed Computing Environment (OSF/DCE) platform to support the development of distributed applications in a distributed environment. The other main area is in application user interfaces (OSF/Motif). Work in progress includes the definition of the Distributed Management Environment (OSF/DME). DME is an extensible framework for unified management of systems, networks and applications in a multivendor environment and complements other OSF technology.

The OSF have stated their long term intent to add an object model and evolve the DCE standard towards the ISO/ ODP de jure standard. In the shorter term they are also looking to evolve DCE towards OMG standards. DME currently includes a slot for OMG object services in its architecture.

What is DCE?

The Distributed Computing Environment (DCE) is an operating and network independent software platform intended for the development of distributed applications in an open environment. The main component subsystems of the DCE architecture are shown in the following diagram:

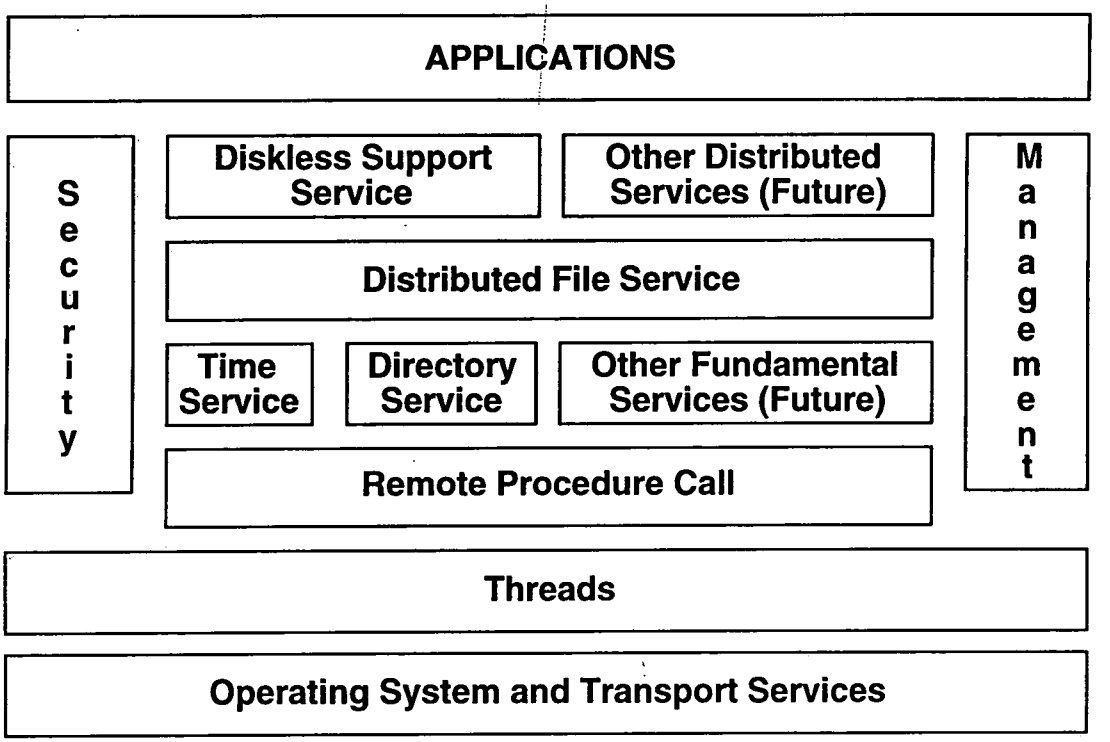


Figure 92 DCE Architecture

The DCE model is client server. A DCE client invokes functions in a DCE server. It is bundled as a complete environment. Intervendor interoperability is good. For scaleability, DCE supports the concept of a cell which is a logical grouping of one or more nodes in a network (for example a business may have cells for engineering, sales, finance). The main components are described below:

Distributed File Service- the DCE file system is extended Carnegie-Mellon Andrew File System (AFS) V4 from Transarc (essentially a robust secure NFS layered on DCE RPC). DFS is interoperable with standard NFS. It supports transparent file replication, file sets to move connected files together using the name service.

Directory Service - names resources including printers, RPC servers, clocks, maildrops, so users can find them. The name service is a hybrid based on DEC DNS Digital Naming Services and X.500.

Distributed Time Service - provides global network time for global cooperation and application synchronisation. It supports delay adjusted network time synchronisation and time tracking services.

Remote Procedure Call - The RPC system is based on network Computing System (NCS) from Apollo/HP, enhanced by Digital. It provides IDL. It is maintainable by configurable logging options

DCE Security - based on MIT Kerberos, DCE Security provides authentication, authorisation and encryption services.

DCE Threads - allows multithreaded clients and servers so that a server can handle multiple clients concurrently or a client can schedule other tasks while waiting for a server.

B.1.6 ANSAware Background

What is ANSA?

The Advanced Network Systems Architecture (ANSA) is an architecture that seeks to enable application components to interoperate despite diversity in programming language, operating system, computer hardware, networks, communications protocols and management and security policies.

ANSA specifies an architecture consisting of: components, that form the building blocks and tools of the architecture; rules, constraining the way components are combined; recipes, to obtain certain properties when applying the tools to the building blocks; and guidelines, to aid design decisions. The ANSA architecture is characterised by the properties of it's components and their composition rules and is principally structural though it does not prescribe a fixed architectural structure. Nor does it prescribe any particular design process or methodology but tolerates a variety of tools and methodologies.

ANSA is structured as a set of projections representing different viewpoints of distributed systems architecture: enterprise, information, computation, engineering, and technology.

The ANSA workprogramme is a collaborative industry effort to advance distributed systems technology. It originated as an Alvey project, then was further developed as the ISA Esprit project and has recently entered a phase of commercial exploitation. ANSA has been very influential in the OSF/DCE and the OMG/ CORBA.

Representives of APM are the editors of the ISO/ITU ODP RM standard, the de jure integrative standard. The ideas behind ANSA have had a significant impact on ISO/ODP and ANSAware provides a partial implementation of these ideas. The OMG have very much positioned themselves as the fast track to ODP.

ANSA also collaborates with TINA-C, producing tools for Telecoms Intelligent Networking, submits technology to OSF, and previously to UI Atlas.

The ANSA workprogramme seeks to :

- contribute to standards
- remain vendor neutral
- prove concepts in advanced technology prototypes
- provide a technical resource for sponsors

What is ANSAware?

The ANSA work programme has developed an infrastructure product in which the consortium have proved research ideas. ANSAware is supplied as a suite of ANSI-C software programs for building ODP systems, providing a basic technology independent platform as well as program generators and system management applications. It is supported on UNIX, MSDOS and VMS and ported to Chorus.

ANSAware consists of the following :

- application objects, an ANSAware object is an encapsulation of application and data, providing services via interfaces in a client server model.

- traders, act as directory and management facility used to advertise services, and for matching offers and requests for services, using service names, interface types and service properties.
- system management tools, to manage the environment.
- factories, provides runtime services to dynamically create capsules (or processes) containing object templates and then objects from the templates and to destroy objects and capsules.
- node managers, providing configuration facilities by combined use of traders and factories. They startup and control both static and dynamic services. Offers can be registered in the trader from passive objects that are activated by the node manager on demand.
- nucleus, a low level resource manager to provide runtime support. The programming interface to the node manager is provided as a library linked into applications.
- IDL, an interface definition language, and stubc, a language processor for IDL, to specify the operations available in an interface and generate stub routines and header files for inclusion in C programs.
- DPL, a distributed processing language, and prepc, a preprocessor which extracts control commands embedded in C and translates them into calls to the stub routines provided by stubc.
- X11 toolkit, to support X11 from within ANSA applications.

The key properties that the architecture seeks to support are:

- state of the art, the architecture is innovative and up-to-date.
- portable, the architecture should be portable across a wide range of computers, operating systems, languages and networks.
- interoperable, the architecture should allow applications to operate in mutivendor heterogeneous environments.
- generic, ANSA is suitable to many fields of application including telecommunications, manufacturing, banking, sales, cooperative working, health service.
- scaleability, the architecture is modular and scaleable, providing interworking between autonomously managed networks.
- distribution policies, ANSA supports a wide range of distribution policies for different dependabilities like security, fault tolerance etc
- high level, the focus of ANSA is on application requirements instead of technological diversity.
- tool oriented, programmers state which application properties they desire in programming tools which automatically insert functionality desired to achieve the desired properties.

ANSAware embodies the following basic design assumptions:

- remoteness, all components are assumed to be physically remote from each other. Co-location is considered an opportunity for optimisation.
- indirect data manipulation, local data is viewed as a co-located special case, hence all data manipulations must be indirect.
- unknown data representation, remote data may be on a system of dissimilar design, hence data can only be manipulated through defined interfaces (abstract data types)
- any request can fail, since all services are potentially remote, hence design must cater for extra types of failure, inlcuding partial network failures.
- indeterminism, events are assumed to be unordered and non-simultaneous, hence if order is important it needs to be determined explicitly.
- data inconsistencies, cannot be avoided in decentralised systems with multiple sources, and must be accommodated.
- unambiguous references to entities, are required for smooth cooperation betwen members of different communities. The most extensible approach without compromising autonomy and scaleability, is to use context relative names that require extensions when accessed in a wider context.
- selective transparencies, are provided to compensate for unwanted properties like concurrency, latency, and failures. They are supported by sets of mechanisms in the infrastructure that can be influenced by application writers without detailed knowledge of the individual mechanisms.
- concurrency and synchronisation, mechanisms are not portable, hence programmers are encouraged to indicate potential and required concurrency and use tools to map this pseudo-concurrency to the available resources.
- bindings, are late to allow flexibility to continuous changes in a distributed system, yet with early type checking on configuration to reduce the risk of unpredictable behaviour.

B.1.7 Example Scenario

The comparison of the integrative standards is illustrated by simple code examples that use a banking scenario. A banking scenario is used because it is simple, familiar, easily extended and used in vendor examples. The scenario is a simple brokerage system that allows an independent broker to create accounts, make withdrawals and lodgements and close accounts on any bank in the network.This illustrates the way the infrastructure manages dynamic objects, representing accounts, that are created, accessed and deleted within objects that represent bank branches. Each bank branch has its own process and is used to illustrate registration, naming and activation of process level objects.

The scenario is supported by three implementation files: one for accounts, one for branches and one for clients. The client provides a simple dialogue that is presented to the user to select transactions and provide transaction data. Figure 93 shows the messages that are sent when a broker creates a new account called "Ken" in the branch called "Ely_High_Street" on the Ely server.

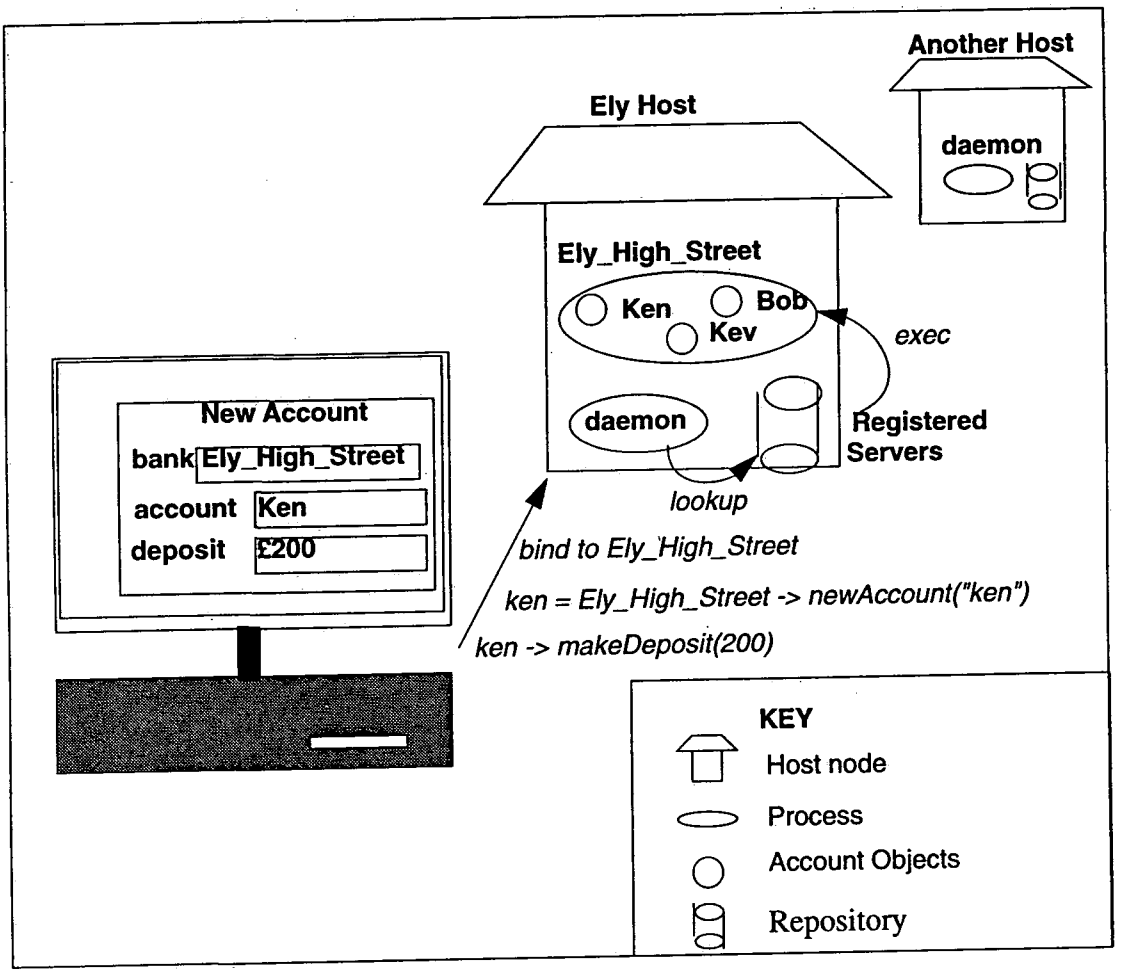


Figure 93 Scenario to Create a New Account

The object code generated by compiling the accounts and banks files are linked into a server executable. The client code and dialog is linked into a client executable. The server executable may be activated as a process several times to represent different branches on different nodes in the network.

There are three main ways in which a client binds to a server:

- Passing pointers or references as arguments in a method call, such as references to account objects may be passed by a lookup method in the bank.
- Calling a binding service for an object that already exists and has been registered for access by the service, such as using the ANSAware trader or ORBIX _bind service.
- Using a factory to create the server, which returns the binding to the caller.

Associated with binding is the generation of the marshalling stubs and skeletons and initialisation of reference counts for memory management.

In CORBA, the server is registered in the implementation repository and activated by the ORB when a broker binds to that branch and may use a timeout to terminate itself automatically.

In DCE the server is started manually from unix.

In ANSAware two schemes are supported. The server is registered in the node manager and activated automatically by the node manager when a reference to the server is imported from the trader. Alternatively the server may be started by the client using the factory service.

In an object mapping for DCE, dynamic object creation for account objects can occur in the server. However the client can only obtain a direct binding to the new account object if a new server process is created for each account. Otherwise accounts are only accessible via the bank interface and accounts are not treated as distributed objects.

With all three integrative standards, this scenario can be run across different heterogeneous networks without any effort to port the example source code. Interoperability comes for free in these environments.

The scenario is run in two steps. First the system is configured. This involves registration of servers in various repositories. In DCE this also involves starting the server process which will register itself in the name service. Once configured the client process can be started on any machine to provide the command interface to the broker.

B.1.8 Overview of Functional Framework

It is difficult to talk about a partial standard without refering to a reference implementation. ANSAware and Iona's Orbix are used in this section to exemplify ISO/ODP and OMG/CORBA respectively.

The survey of integrative standards is written up as a comprehensive tutorial on CORBA, ANSA and DCE and provides a conceptual guide to implementations of all of these standards. There are a number of concepts and areas of functionality that are shared by all, hence the report is structured around the common concepts and functionality not the products. Furthermore there are actions that must be performed to provide this functionality. The survey is written up as a series of steps that must be performed to use CORBA, ANSA and DCE. For each step the tutorial describes the following:

- Overview
    - A brief description of the areas of functionality associated with the step and an overview of the goals and the impact of implementation decisions.
- ANSAware Concepts
    - A conceptual description of the functionality provided by ANSAware.
- ORB Concepts
    - A diagram showing which architectural components of CORBA are involved in providing the functionality. This relates the behaviour to the architectural components identified in section 2.

- A conceptual description of the functionality provided by Orbix and CORBA. The distinction is made between existing standards, expected future standards and proprietary extensions.

- DCE Concepts
  - A diagram showing which architectural components of DCE are involved in providing the functionality. This relates the behaviour to the architectural components of DCE.
  - A conceptual description of the functionality provided by DCE.
- Example
  - A description of how the area of functionality affects the banking scenario.
  - Source example for the banking scenario.

The 6 step guide to DCE, ANSAware or CORBA covers the following functional scenarios:

- B.2 Stub Generation
- B.3 Source Code Implementation
- B.4 Server Registration and Naming
- B.5 Locating and Binding
- B.6 Activation and Failure Handling
- B.7 Synchronisation and Request Processing

B.2 Stub Generation

Terms: Interface Definition Language, Stubs, Skeletons, Smart Proxies, Interface Repository, Dynamic Invocation Interface (DII)

B.2.1 Overview of Functionality

The Remote Procedure Call RPC mechanism provides high level support for distributed programming. The chief objective is to elevate distributed programming to the level of procedure and function calls, rather than low level message buffer handling and transmission. Thus programmers familiar with procedure and function calling from classical programming can easily make the transition to distributed application programming.

RPC systems typically have three levels in their architecture:

- in the application level, the client makes an RPC call on a remote server and the server provides a definition for the procedure called.

- in the stub level, a clients call to a remote server is intercepted by a **stub** that supports the servers interface. The stub encodes the call and the arguments into a message which it passes to the transport layer. At the other end the **skeleton** receives a message from its transport layer and decodes the call and arguments and performs the call on the server.The skeleton then receives return values and encodes them into a reply message. The stub receives the reply and decodes the return values and returns control to the client.

- in the transport level, the source receives request messages from the stubs and sends them over the network to the destination. The destination waits for incoming requests and forwards them to the appropriate skeletons. On return the destination receives the reply message from the skeleton and transmits it across the network. After transmitting a request message, the source waits for a reply message and forwards it to the stub.

This behaviour is shown in Figure 94.

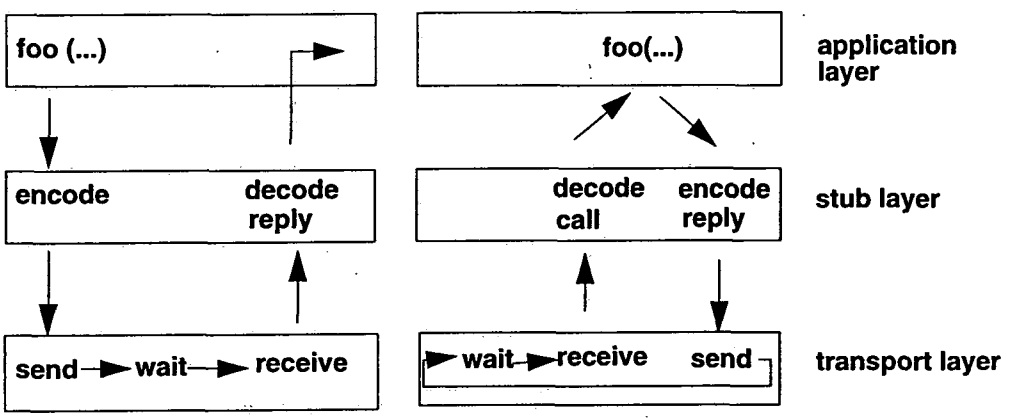


Figure 94 Implementing RPC

The application layer is programmed in a procedural programming language to define the behaviour of the client and the server.

The stub layer provides the presentation protocol for the remote server. This consists of typed procedure calls.

The transport level is responsible :

- for managing connections.

- for multiplexing and demultiplexing messages between multiple stub/skeleton pairs or multiple callers/callees pairs when there is more than one caller or callee in the same process.

- for request scheduling at the destination and reply scheduling at the source, including dispatching requests to server threads and dealing with synchronisation.

- for fragmenting large messages and retransmitting lost packets. Typically ordering information must be embedded in each packet.

- for detecting lost or corrupted or out of sequence messages and retransmitting them. The protocol can be optimised for normal RPC processing by using replies to acknowledge sends rather than using an enq-ack protocol.

- for notifying and tidying up abortions on partial failures such as long term communication failure or node failure.

Rather than hard coding stubs for each procedure, a stub generator can be used to generate the stubs from a higher level description of the procedures. An **interface definition language** is one way that procedures can be described and this has been adopted in both CORBA, ANSA and DCE. The interface definition language compiler generates the stubs. This hides the interface between the application and the stubs. The transport layer itself is hidden by the stubs. Consequently both DCE and CORBA can be used without any knowledge of the stub and transport behaviours described above. An interface definition language consists of four main parts:

- a specification of the composition of the interface, this may include interface inheritance and scoping constructs like modules and class.

- a specification of data, including declarations for attributes, structures for exceptions and composite data types like sequences and unions which support extra metadata for marshalling.

- a specification of operations, typically extending conventional operation signatures in a C style with : qualifiers for arguments to determine whether they are input, output or both; extra exception handling features to deal with the additional errors that occur in a distributed environment; extra synchronisation qualifiers for procedures to indicate they are unidirectional and can be invoked asynchronously; qualifiers for idemnipotency to deal with reincarnations on process restarts; extra context definitions.

- property specifications, not yet incorporated in CORBA and DCE only ANSAware. Extra property information is useful to deal with the additional complexities of distributed open systems and may be added in the future for selecting servers based on other semantic properties that supplement the essentially syntactic operation specifications.

IDL definitions are provided for all distributed objects in much the same way that C++ class headers are provided for all C++ classes.

B.2.2 ANSA Concepts

ANSAware provide an IDL. The ANSAware IDL compiler, stubc, parses IDL definitions and generates stub files for marshalling and unmarshalling and header files to be included into object implementations to define encodings for any data types defined in IDL. The current compiler only supports C stubs and headers. IDL defines a common data representation used to communicate data between heterogeneous machines.

B.2.3 ORB/ORBIX Concepts

The IDL compiler for CORBA generates the stub and skeletons. It also populates the type repository with typing information that can be browsed at runtime to explore interfaces. CORBA IDL code looks similar to a C++ class header, supplemented with extra data type and exception information and omitting support for overloading, constructors and destructors, and default or ellipse arguments. The output of the IDL compiler is shown in Figure 95.

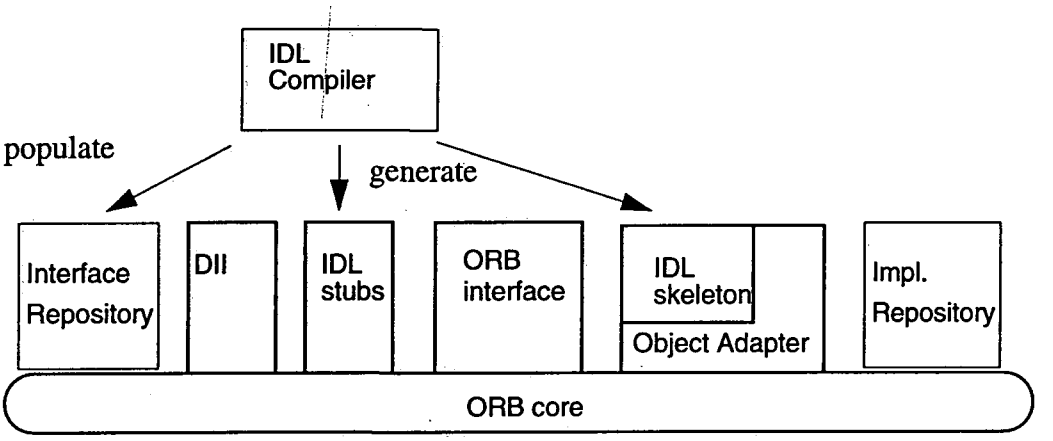


Figure 95 IDL Compiler output in CORBA

The IDL compiler populates the type repository with information about servers.The interface repository effectively maintains enumerated typing information in a hierarchical data structure that can be queried using module names, class names,operation names and argument names. This data is stored in the file system in a nominated directory and itself accessed as an IDL server. CORBA 1.1 specifies the access interface but not the manner in which it is populated.

The DII and interface repository in combination provide a mechanism for bypassing static type dependencies by treating types as enumerated data values in the application's call interface. This allows the use of tagged untyped values as parameters, where the tag value indicates the type. Clients need not statically link to stubs for all servers in advance. The DII interprets the tag values and marshals the request appropriately.

C++ stub mappings are not yet standardised. Orbix provides one mapping.

In Orbix, stubs and skeletons are C++ classes and can be subtyped to modify their behaviour, for example to implement caching strategies for performance sensitive servers. Stub subtypes are known as **smart proxies**. Stub subtyping is not specified by CORBA but is part of the ISO ODP standards.

The CORBA interface definition language has a syntax that is very close to C++ except:

- it adds a new scoping entity called module that allows definitions of interfaces, exceptions, constants and typedefs to be grouped into a hierarchy.
- it does not support private or protected inheritance nor private or protected access controls.
- replaces keyword "class" with "interface".
- no constructors or destructors.
- no signed/unsigned qualifier for char and int must be explicitly short or long.
- adds composite types for sequences, discriminated unions

- restricts "templates" to sequences and strings.
- no ellipse or default arguments.
- extended syntax for operations including exceptions, context definitions, and directional qualifiers like "oneway" for operations and "in", "out", and "inout" for arguments. Refers to section B.6 for more on exceptions and section B.7 for contexts and qualifiers.

B.2.4 DCE Concepts

DCE IDL is similar to CORBA IDL in that both generate stubs to marshal requests. DCE uses a unique identifier which a client uses to identify an interface. This is generated by a utility called *uuidgen*. The output from DCE IDL is shown in Figure 96.

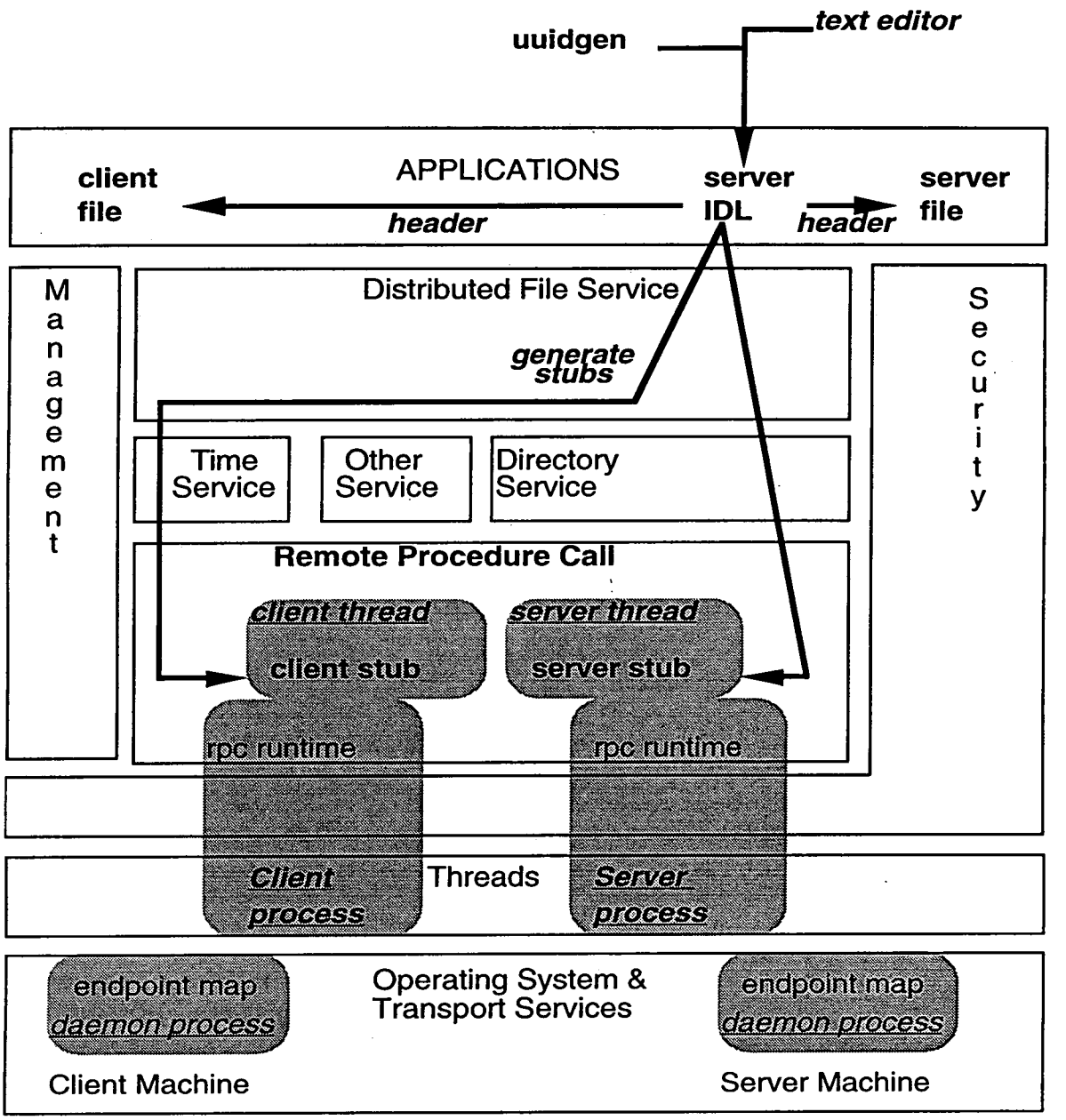


Figure 96 IDL Output in DCE

DCE IDL differs from CORBA IDL in that:

- restricted support for pointers - DCE makes an explicit distinction between pre-allocated reference pointers that must point to preallocated memory and remain immutable and full pointers that can be compared, changed or assigned a NULL value. The stubs do more work for full pointers.

- it does not allow references to be passed i.e. binding handles can not be passed, only textual names, DCE pass-by-reference and pointer-passing is really pass-by-visit semantics, where data is allocated and copied by the stubs before or after making the call. After a call the client and server no longer point to the same copy of the data - so references cannot be maintained between calls.

- no templates
- no interface inheritance
- no class attributes
- operations are invoked as free functions not methods with an implicit 'this' pointer. Objects state is complex to handle.

DCE supports pipes for transfering a large volume of variable data.

B.2.5 Example

The banking example provides two interfaces: an account interface and a bank interface.

In Orbix the interfaces look like this :

```
// bank.idl

interface account {
        readonly attribute float balance ;
        void makeLodgement (in float f) ;
        void makeWithdrawal (in float f) ;
} ;

interface current account : account {
        attribute float overdraftlimit ;
};

interface bank {
        exception reject { string reason ; } ;
        account newAccount (in string name) raises (reject) ;
        account newCurrentAccount (in string name, in float limit) raises (reject) ;
        account getAccount (in string name) ;
        void deleteAccount ( in string name ) ;
} ;
```

In DCE the interfaces look like this:

```
// bank.idl

[
uuid (e49d6bd3-3126-11cd-b624-08002b326291),
version(1.0)
]

typdef struct account_data {
        float overdraftlimit ;
        float balance ;
        char [50] name ;
        account_data *next ;
} * head ;

interface account {
        void makeLodgement ([[in,string,ptr] char* name, [in] float f) ;
        void makeWithdrawal ([in,string,ptr] char* name, [in] float f) ;
```

```
};

interface bank {
        void newAccount ([in,string] char [50] name) ;
        void newCurrentAccount ([in,string] char [50] name, [in] float limit) ;
        account _data *getAccount Data([in,string,ptr] char* name) ;
        void deleteAccount ( [in,string,ptr] char* name ) ;
};
```

Note that DCE does not support inheritance. In the example, overdraft limits have been conveniently inserted in the original definition of an account data and will be present for all accounts. However there are other data that may become important, for example credit ratings or loan expiry dates Account specialisations can not add extra data or methods incrementally.

DCE is a static process level architecture. DCE is also static in the sense that the client must link statically to the stubs generated by the IDL compiler on the server, hence there are static build dependencies. There is no DII.

DCE does allow a server to allocate dynamic memory for a new account and return a pointer to it as in getAccountData. However the data is copied by value into allocated memory in the client's stub. DCE does not allow object references to be passed. Consequently bindings to accounts cannot be maintained between calls to newAccount and makeLodgement. DCE has no notion of fine grained objects within a server and is restrictive when dealing with dynamic data that is accessed across the network.

B.3 Server Implementation

Terms: IDL language mappings, interface inheritance, class inheritance, ties

B.3.1 Overview of Functionality

Once an IDL definition has been provided, a definition of each operation must be implemented in the same way that procedures or functions must be implemented in classical programming. The language mapping and inheritance model severely restricts the OO language usage and is an important evaluation criteria.

The application programmer must take into account the IDL mapping to the implementation language, including: the encoding of IDL data types in the implementation language; mappings for contexts and exceptions; and memory management and multithreading policies.

Another important issue in program construction is the capability to share operation implementations between interfaces that are related by IDL inheritance. Interfaces can be shared down a hierarchy by **interface inheritance** in IDL. Many distributed systems require that the whole interface is reimplemented for each node in the hierarchy, i.e. subtyping or interface inheritance. This is like restricting C++ to only allow virtual methods and insisting that all methods are implemented in each class. If implementations of methods can also be shared, then we can say that the implementation language mapping supports **class inheritance or subclassing**. The term class is generally used to signify both an interface and an implementation. Class inheritance implies sharing of both interfaces and implementations.

Other techniques for code sharing that have been popularised in research but not CORBA 1.1. include delegation, enhancement, exemplars, hierarchical composition.

Neither DCE nor CORBA IDL support sophisticated encapsulation protocols like C++ protected and private access controls or protected and private inheritance.

B.3.2 ANSA Concepts

ANSAware describes a distributed system in terms of potentially distributed objects with defined interfaces through which the services are accessed. ANSA separates the type from the templates from which objects are instantiated. An object may be a collection of more than one interface.

ANSA supports subclassing (IDL declaration *IMLEMENTATION IS COMPATIBLE WITH*) as well as subtyping (IDL declaration *IS COMPATIBLE WITH*).

Applications must implement the operations of an interface in C. A naming convention is used to associate C functions to IDL operation names.

Object declared as 'managed objects' using the *PREPC MANAGED* declaration support lifecycle services that can be used by factories. Applications must explicitly implement constructors and destructors as C functions for all managed objects. Again a naming convention is used to associate the C functions with object names used in the declaration (i.e.*Create_<objectname>_Object()* and *Destroy_<objectname>_Object ()*).

In early versions of ANSAware object state is associated with objects using objectIds with a hash table to lookup state. In later versions, all state is associated directly with interface references. State can be accessed using the interface reference of type Object obtained from the current execution context of a thread executing an operation in the interface (via thread_getInterfaceState). Alternatively state can be obtained using the interface reference, via a call to awifref_getInterfaceState for a specified interface. (Object state is implemented in C by associating state with the socket used to support the interface.)

B.3.3 CORBA/ORBIX Concepts

CORBA 1.1. provides a mapping from IDL data types to C and allows IDL interfaces to be implemented as C functions. There has been much debate about whether C is an appropriate launch point for CORBA technology due to its absence of OO support.

Orbix provides a mapping for C++. One design goal of Orbix was to make the transition to Orbix straight forward for programmers who were already competent in C++. IDL interfaces are implemented by C++ classes. Making a remote call on an IDL interface appears as similar as possible as making a local member function call on a C++ class. A C++ mapping will be standardised by the OMG .

CORBA IDL supports interface inheritance, including multiple inheritance but actively avoids any commitments to implementation issues like class inheritance. Some language mappings will support class inheritance, most will do this in a restricted way, and many will only support interface inheritance.

Orbix provides restricted forms of class inheritance.

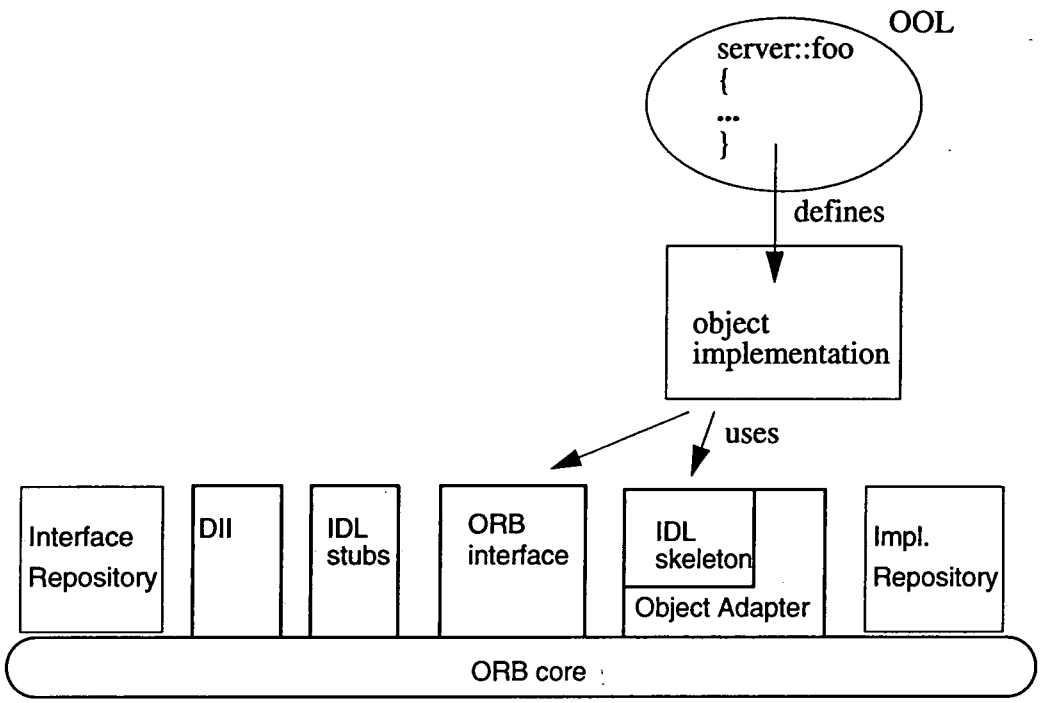


Figure 97 Implementing ORB Objects

In Orbix, the IDL compiler allows implementations to be associated with IDL interfaces in two ways:

- an implementation class can derive from the class generated by the IDL compiler
- an implementation class can be associated to the class generated by the IDL compiler by instantiating a special typed C++ template, known as a **tie.**

Classes generated by IDL only support pure virtual methods. An IDL subtype class inherits from its IDL base and adds further pure virtual methods. The implementation class for a base class must derive from the base IDL class and the implementation class for the subtype must derive from the IDL subtype class. Hence the bases implementation can only be shared by the derived implementation class using multiple inheritance.

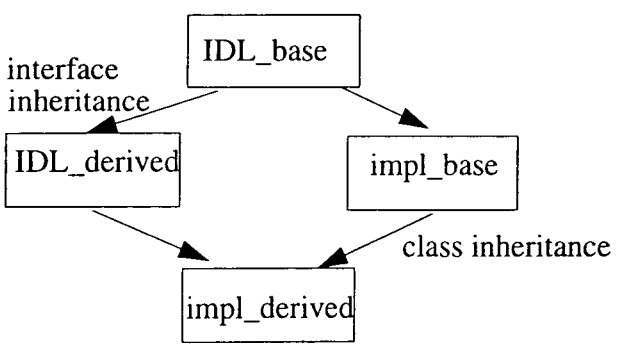


Figure 98 Inheritance restriction without ties

The above figure shows the need for multiple inheritance to share implementations. In fact, in a real implementation, it is even worse than this - the shared methods must be put in a distinct class that is multiply inherited by impl_base and impl_derived.

In turn, the use of virtual multiple inheritance inhibits the casting of pointers. Yet pointers are used as object references in the Orbix C++ mapping. This leads to interoperability problems with the C mapping of CORBA 1.1. which uses void * pointers.

A tie template is expanded for each interface-class implementation-class pair. Tie objects are instantiated from this expansion to delegate calls between interface objects and the implementation objects. Tie objects impose an extra level of indirection. They maintain pointers to the associated implementation object. The Orbix skeleton invokes IDL methods via the tie object. An implementation object must be created and deleted by creating and deleting a tie at the same time. Ties in effect explicitly separate the interface hierarchy used as the type hierarchy for safe remote access from the implementation hierarchy used for C++ implementations. This allows flexible mappings between C++ classes and IDL interfaces and more flexible patterns of implementation sharing .

B.3.4 DCE Concepts

DCE does not support interface inheritance at all so the question of how to support class inheritance does not arise.

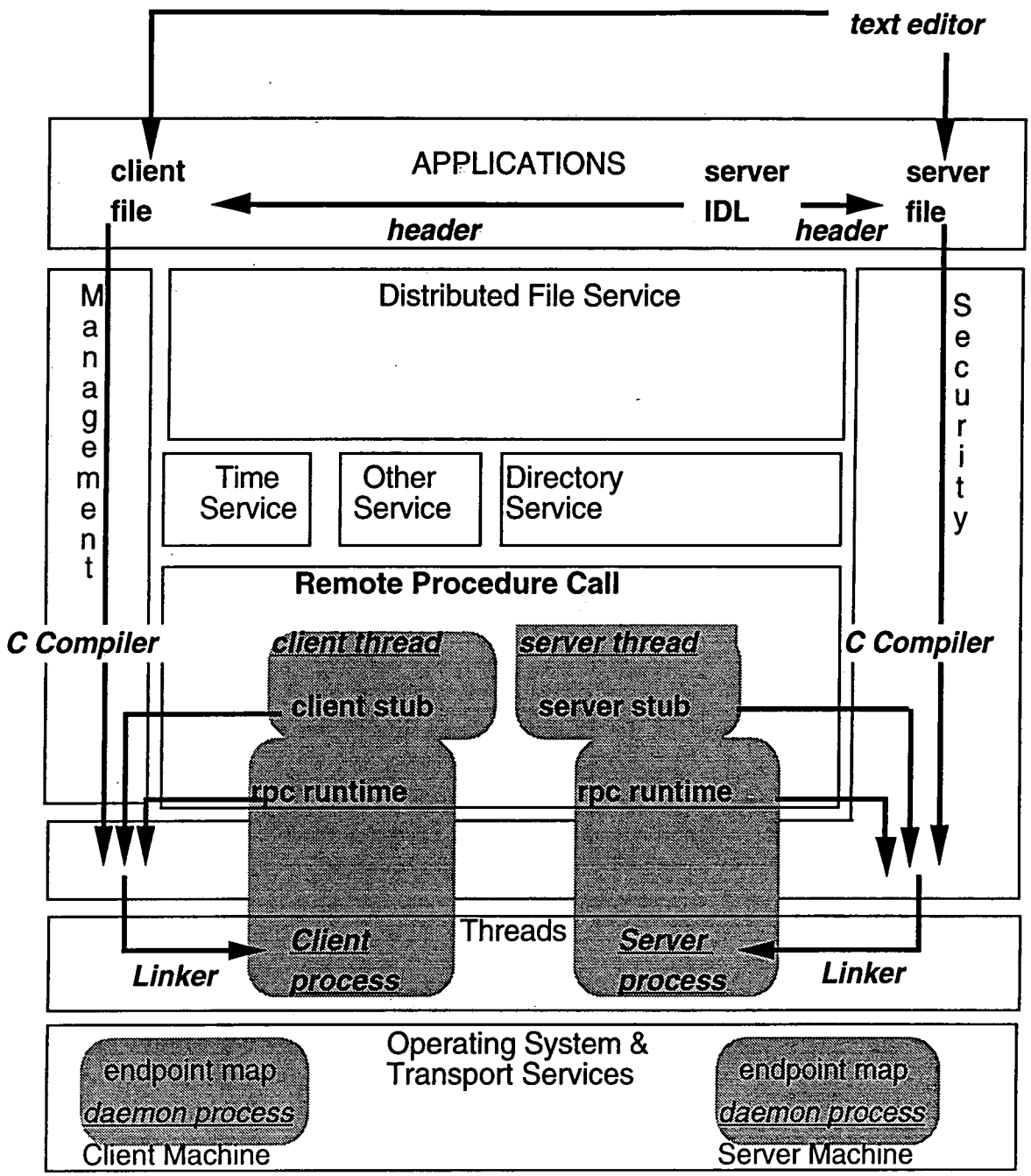


Figure 99 Server Development in DCE

The DCE IDL compiler generates a C header file and C stub files for the client and the server. An implementation is associated with an IDL interface by including the C header file generated by the IDL compiler and implementing the procedures defined in the interface. Both the client and the server must also compile and link the appropriate stub file.

DCE only provides C linkage. Servers must implement their DCE operations as C functions, hence if DCE is used with C++, the server must be implemented as free functions within an extern "C" statement to prevent name mangling by the C++ compiler. The object may be passed as an object identity value in an argument, the free function being implemented to map ids to pointers and forward the call to the relevant object.

There are other problems using C++. DCE provides its own reentrant system libraries and these can be abused easily to use non-reentrant code.

B.3.5 Example

In the banking example we must implement the account and bank interfaces described in section B.2.5.

In Orbix, tie objects are used as shown in the figure below. A tie for the *bank_i* implementation class supporting the *bank* IDL interface is declared by including *DEF_TIE(bank,bank_i)* in the header file for *bank_i.*. A tie for the *account_i* implementation class supporting the *account* IDL interface is declared by including *DEF_TIE(account,account_i)* in the header file. When new bank objects and account objects are created, a tie object must also be created. A code example implementing the operation *newAccount* in the *bank* interface looks like this:

```
account* bank_i::newAccount (char* name, CORBA::Environment &pe)
{
    // ... code to check uniqueness of name ...
    account_i *p _obj= new account_i(0,name) ;
    account *p _tie = new TIE(account,account_i) (p_obj) ;
    p_obj->p_next =m_head ;
    m_head = p_tie;
    p_tie->_duplicate() ;
    return p_tie ;
}
```

Note that it is tie objects that are passed as arguments and return values. An account object must be assigned to the tie object before passing the tie. The tie object can then be used to access the account remotely, for example to make an initial lodgement.

Note also that memory management policy has implications on the programmer. When returning an object reference, the implicit reference count is decremented on the assumption that it was incremented when it was passed. Hence to ensure that *m_head* is still a valid reference after returning, we must first call Orbix function *_duplicate* to increment the reference count. If the reference count becomes zero, Orbix will delete the tie object automatically and this will recursively delete the account object. The Orbix function *_release* should be used in place of normal C++ delete.

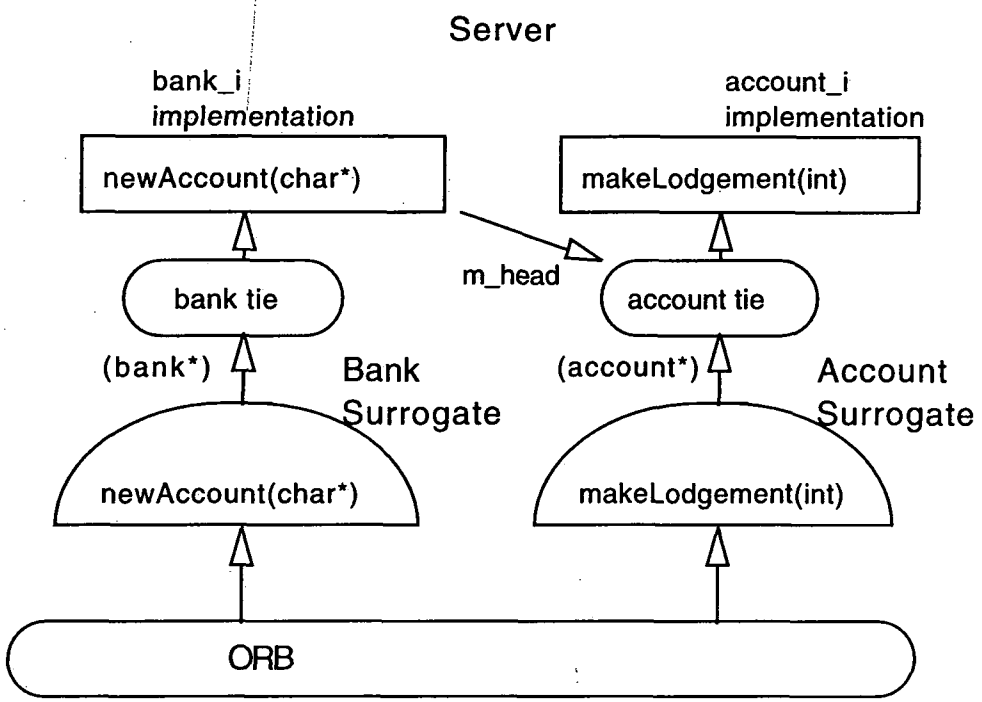


Figure 100 Server Implementation in Orbix using Ties

In DCE, the interface must be implemented as C functions.

B.4 Server Naming and Registration

Terms: name server, cell directory service, implementation repository

B.4.1 Overview of Functionality

In order to identify application servers we must be able to name them. A name could be a raw network address such as a tcp socket address. Alternatively a name service can map text names in any format to network addresses used in binding.

Where there are many fine-grained object per process, a common transport service should be shared by multiplexing all calls to object in the process, to avoid linking generic transport behaviour into each object. Each object then has a network address for the transport service and a local object address for the object, normally represented as a pointer from the transport service.

An object naming service should map text names to network addresses and object addresses. The simplest scheme for object addresses would map names to identifier values rather than transmitting memory addresses between address spaces. The identifier values should be mapped to local memoryaddresses locally by a transport layer catalogue within each address space. These identifier values may be called plug numbers at the client and socket numbers at the server. The socket number is used to identify the server object for client requests. The plug number is used to identify the client object for server replies.

Some system services have well known addresses that are hard coded into the runtime support, for example as environment variable, such as the name service itself and any daemon servers used in binding.

A distributed name service that allows a client to bind to a server without knowledge of the locality of a server or network address is said to support location transparency. Section B.5 will discuss other ways of achieving location transparency such as trading.

A client will bind to a server using the servers name to lookup the address for binding in the name service. A name service itself is a distributed application. It may be replicated or centralised, with or without stand-bys for fault tolerance. Caching at the clients end can be used to improve lookup speeds. Forwarding pointers at the server end can be used to support migration of servers between nodes.

A complete address for an object includes the host node address, the port of the server process and an address for the object within the process. Not all systems allow multiple objects per process, eliminating the need for the latter object identifier. The different parts of an identity may be obtained in different ways, in multiple steps. In particular, volatile server port addresses and identifiers for objects may not be replicated across a network.

Names can be arranged into groups or a heirarchical name space, extending names with a context or allowing the use of group names to make weaker statements about which particular instance is actually referenced. Likewise a name space may be federated using different naming schemes in different domains.

B.4.2 ANSA Concepts

ANSA objects export their services to a repository of services known as a trader. Each server must register in the trader the services it is offering to other objects. A command line tool is provided to edit and add registration entries in the trader.

An entry in the trader consist of an abstract data type together with a set of attribute values associated with the object. These attributes are called properties. Clients import references to servers by specifying an abstract type signature together with property values. This is a yellow pages types service as compared to the name service of DCE and implementation repositories of CORBA which are white pages.

B.4.3 ORB/ORBIX Concepts

The Common Object Service Specification of the OMG's Object Services defines a Name Service specification for mapping textual names to network addresses.

CORBA also defines an implementation repository that maps server names to executable images. The implementation repository not only acts as a name service but may also activate servers by executing the executable, see section B.6.

In Orbix, server images are registered in the implementation repository using a configuration tool with a simple command line interface that makes the repository look like a unix file system. Orbix does not yet support the Name Service.

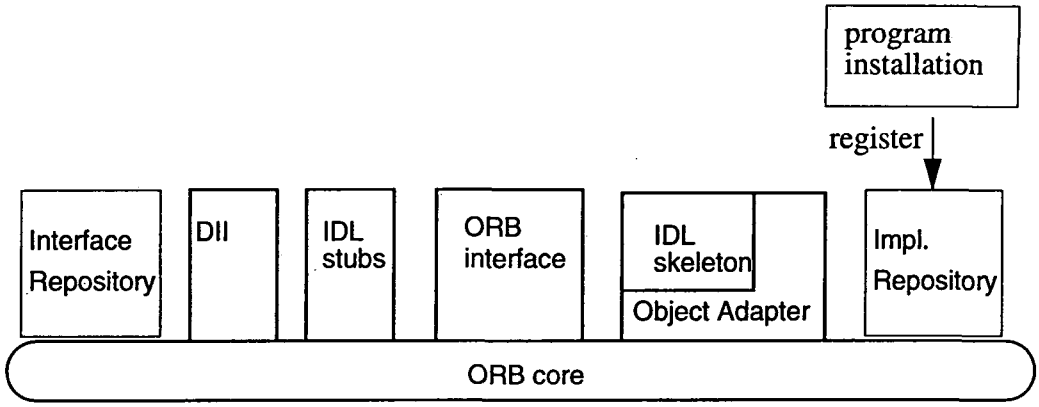


Figure 101 Server Naming & Installation

An executable can be registered with different names, representing different servers that share the same image. An executable can be registered with the same name on different nodes, notifying Orbix that a server can be run on a number of machines and that Orbix should choose.

In Orbix, the name refers to the server process. An additional name can be used to identify a server object within a process, called an object marker.

An implementation repository is maintained on each node. The Orbix locator service maintains a mapping between server name and host names on which the server is available. The host name is used to find the local implementation repository. Hosts can be arranged into groups to allow group names to be used as a shorthand for lists of several nodes.

B.4.4 DCE Concepts

DCE uses a centralised master name server, called a cell directory server, to map textual names to network addresses. The port address of the server is ommited from the address returned by the cell directory server. Instead, the mapping from names to port addresses, called the endpoint map, is managed by a local daemon process running on every node. This means if endpoints change due to process restarts, the cell directory server remains unaffected.

B.4.5 Example

Bank objects need to be named and registered for latter lookup by the broker who acts as a client.

Orbix allows banks to be named in three ways: as a server process name; as a server process name and object marker; or automatically named by the IDL interface name of the object reference being bound. The following commands show these 4options respectively: (The -h argument is optional and determines the physical location of the server. The unix host name merton is used in the example) :

option 1 : putit -hmerton Merton_High_Street server
option 2 : putit -hmerton -shared NatWest -marker Merton_High_Street server
option 3 : putit -hmerton bank server
option 4: putit -hmerton bank -method newAccount server

Usage : putit -hnode [-shared] servername {-marker markername} executable

The nodes represent regional servers for the banks. Here we can think of merton as a city. This makes the mapping between physical names and invented names simple. The host name can be used by the client to restrict the binder to select a server in a specific location. For example, a client could bind to any bank in the locality of merton by by specifying the host name merton. A server may also be registered in merton with name Merton_College_Branch and the binder would choose randomly between Merton_High_Street and Merton_College_Branch. The client need not specify the host name, i.e.Orbix supports location transparency. In this case the binder would chose a branch on any node.

Option 1 allows each server process to be named. A client uses the server name Merton_High_Street, in binding. Binding is described in more detail in section B.4.

Option 2 introduces a composite name for each object consisting of a server name and a marker name. Clients can bind to any Natwest bank by using the server name only or to any bank with a Merton_High_Street branch by using the marker only. For example, *server* may be registered for Midland as well as Natwest in Merton_High_Street. The host name could also be used with either part of the composite name for example to select any Natwest bank in the locality of merton. *-shared* is necessary to allow multiple objects per process.

Option 3 allows type specific binding where the client chooses to bind to any server supporting the IDL interface of its reference. This option must be used if the client is not going to use names other than the host name.

Option 4 uses the -method argument that implies a server is activated for each request to the specified method and terminated when the reply has been sent. The lifetime ofthe server makes it unsiutable to this example without persistence. Also all methods would need to be registered in this way.

The executable may be run in an X-window using "*xterm -e server*" in place of *server* above.

In addition, a server must define markers before any clients try to bind. This is usually done in initialisation code in main before executing the ORB call to listen for incoming requests. It can be done at any time and names can change.

void main {

```
    account *acc = new TIE(account,account_i) (new account_i) ;
    acc->marker ("merton_high_street") ;
    //listen for requests ....
}
```

The repository can be queried using *lsit.* and entries removed using *rmit..*

Serever must also register in orbix.hosts and orbix.host groups files for th erelocation service.

DCE allows banks to be registered by initialisation code in the servers main body. The server must be registered in the local end-point map and in the DCE namespace. This is done in the following source example. Note that the programmer must manage network addresses, interface identities and protocols explicitly using arguments like binding_vector and constants like bank_v1_o_s_ifspec, which is generated by the IDL compiler to identify interfaces. This makes the interface much lower level than with Orbix where this is hidden from the programmer:

```
/* register interface using ifspec handle generated by IDL compiler */
rpc_server_register_if (bank_v1_0_s_ifspec, NULL, NULL, &status) ;

/* select all protocol sequences */
rpc_server_use_all_protseqs(rpc_c_protseq_max_reqs_default,&status) ;

/* get binding information allocated for the server by above registrations */
rpc_server_inq_bindings(&binding_vector,&status) ;

/* advertise the endpoints in the local endpoint map*/
rpc_ep_register(bank_v1_0_s_ifspec,binding_vector,NULL,"merton_high_stre
et" ) ;

/* advertise the server in the CDS name space */
rpc_ns_binding_export(rpc_c_ns_syntax_default,"merton_high_street",bank_v
1_0_s_ifspec,bidning_vector,NULL,&status) ;
```

This example clearly show the difference between high level tools like the Orbix implementation repository and lower level APIs like DCE's binding functions. CORBA and DCE environments differ greatly in the level of complexity in the programming interface.

DCE does provide tools, cdscp and rpccp, in which names can be added to the directory service but this is mainly for administration and the server must still include the initialisation code to register itself. In Orbix, all this work is done by a single command of the implementation repository and initialisation code is only needed for naming individual objects within a server should this be required. Reference passing means that finer grained objects can exist as unnamed entities.

B.5 Locating and Binding

Terms: network addressses, trading, property service, daemons, collocation, lightweight RPC

B.5.1 Overview of Functional Scenario

In order to invoke objects, we must be able to find and bind to them.The transport protocol allows network addresses to be used to establish connections or to route datagrams between objects.The main problem is how to select servers and how to find the complete network address and object address.

Servers may be selected by name or part-name, by physical location, or by querying properties. Competing instances of a server that satisfy the selection criteria may be chosen randomly in several ways: on a nearest-first basis, by a load sharing algorithm or by imposing an order or priority on servers when they are registered. The server must support the interface expected by the client hence selection should also be type specific. In some configurations, type conformance is the only criteria for selecting a server. Servers can be collected into groups or heirarchies in a heirarchical name space to limit the extent of the selection.

Servers may also be selected explicitly by an application engineers in a programming in the large tool like OpenBase or configuration programming systems.

Addresses may be embedded in object references when they are created and passed with references. Likewise addresses may be encapsulated in proxy objects that are associated with references and new proxies are generated when references are passed. Alternatively addresses may be held in catalogues that map identifier values to addresses.

Catalogues and name servers can themselves be accessed by embedding their address in object references or by using well known addresses such as an environment variable. Some advanced systems support broadcast or multicast protocols to find object addresses for remote objects. Others use centralised or ring-based topologies with point to point connectivity between name servers and catalogues. Most allow addresses to be cached and replicated across the network. Local caches with fast lookups can be used as an alternative to embedded addresses.

Another issue is whether optimisations are supported for local calls. This is often a bind time decision.

Interprocess calls on the same node may use a lightweight RPC mechanisms, for example using shared memory or named pipes as the transport mechanism. Few systems support leightweight RPC properly as advanced techniques are required [Berstad]. By managing a call stack in shared memory, buffer copying overheads can be removed. It is even possible to by-pass context switching and dispatcher overheads using hands-off scheduling as supported in Mach. Lightweight connections are usually established at bind time when the protocol is negociated.

Calls between objects in the same process can by-pass all RPC overheads. For example, if the stub/skeleton interface is symmetrical, a client can bind directly to the server rather than via the stub and skeleton. Thus a local RPC call can be as efficient as a conventional procedure call. This is called collocation and is easier to support than lightweight RPC.

B.5.2 ANSA Concepts

ANSA interfaces are identified by a data structure known as an interface reference. Interface references encode the address of the object providing the interface. They may also encode the address of a relocation catalogue should an object migrate between processes or encode the addresses of a group of objects should an object be a member of an interface group such as a replica group. More recently interface references have been extended to encode quality of service parameters.

A client object can obtain an interface reference for a server in three ways:

- by the process of trading, a client imports the interface reference from the trader by specifying an interface type and property attribute values expected of the server. The import service queries offers of service registered in the trader, as described in section B.4.2. Type conformance and property values constrain the search.
- by passing interface references as parameters in service invocations.
- by creating a new server using the factory service, as described in section B.6.2. The interface reference of the new server is returned to the caller of the factory service.

The trader allows for the classification of server instances into a type heirarchy. This restricts the search space to interfaces whose type conforms to the expected type.

The trader allows the classification of server instances into an administrative heirarchy which is similar to a directory structure of UNIX. Each node in the heirarchy is a context that defines a context space as the branch of the heirachy from that node.

When exporting, a server must specify a server name, a context space into which the offer of service should be registered and properties (e.g. *traderRef$Export("name", "/context/...", property, value)*). On importing, clients may specify the context to constrain a search (e.g.*traderRef$Import ("name","context/...")*).

The trader allows two policies with regard to how matching offers are handled: random selection of a single offer or return of all offers.

An offer may be a proxy offer where the proxy itself is used to find the server rather than the binding information being including in the trader. For example node manager may register proxy offers for passive servers that need to be activated before use, as described in section B.6.2. Proxy offers may also be registered by other traders in a federated trading space consisting of several trading domains each managed by a different trader. Likewise a context space within one trader may be bound to context name in another trader, for example to bind several traders to a single trader nominated as the master trader.

B.5.3 ORB/ORBIX Concepts

The Object Services Architecture includes Property Services that allows dynamic named attributes to be associated with an object. Such properties can be used to select objects. The Trading Service provides the mechanisms for matching provided services to services required by a client and may use the property service. Both these services are not yet standardised, although a number of ORBs have implemented similar services e.g. Sun DOE.

The ORB 2 task-force is looking at extending the interface repository into a full type repository by including constraints and other properties with interfaces. This may form the basis of some selection process in the future but has not yet been standardised.

Currently the OMG have standardised the Name Service described in Section B.4.

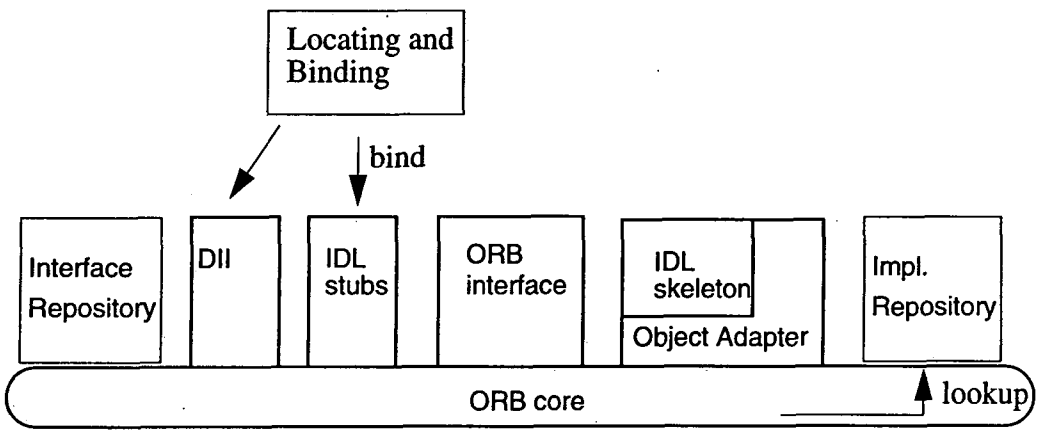


Figure 103 Binding in CORBA

Orbix does not yet support the Name Service. However it does implement an implementation repository that maps names to executables and a locator service that can be used to find the local implementation repository for a server.

Orbix has a symmetric stub/skeleton interface and supports optimised collocation, both explicitly when a server itself only uses local nested servers and automatically at any time when binding to a local server. Explicit collocation is set and unset by calling an Orbix function in the ORB interface and this inhibits remote binding for any objects in that process. Orbix does not provide any lightweight RPC for IPC.

Binding services are supported by the generic transport layer which is linked with the application code. Two libraries are provided ITclt and ITsrv. ITsrv supports : incoming IDL calls, explicit collocation, and callback functions (an Orbix extension). ITclt is linked into images that only act as clients for remote servers.

The orbix IDL compiler generates a *_bind* method that is supported as a static class method in the IDL stub. This makes a downcall to the binding services in ITsrv or ITclt. Alternatively binding calls are made from the dynamic invocation interface which provides an API as described in section B.7.

ITclt uses the locator service to find the appropriate node, step 1 in Figure 100. ITsrv uses the locator service if the server is not collocated and explicit collocation is not in force. The locator service maintains a mapping between server name and host name . The mapping is held locally on each node. Each node also maintains a pointer to the location service on another node.This is used if the server is not cached locally, step 2 in Figure 100. This pointer is configurable, so lookups can be configured in any topology such as a ring or a star. The locator can be bypassed if the client identifies the host as an argument in the call to _bind, i.e. non-transparent binding.

Once the server node is identified, the binding services access the implementation repository on that node using a daemon process that runs on every node, step 3 in Figure 100. Daemons have well known ports, defined as a variable in the system or orbix configuration files. The implementation repository can determine if the server is active and activates a server if one is not active already. Server activation is described in section B.6. It then returns a port address for the binding services of ITsrv that are linked into the server process itself. These services are then used to establish a connection to the server object, step 4. Alternatively for connection-less protocols the final binding is deferred till a request is processed.

If no server name is specified, the _bind method uses the interface name as the server name by default. This allows type specific binding without explicit names.

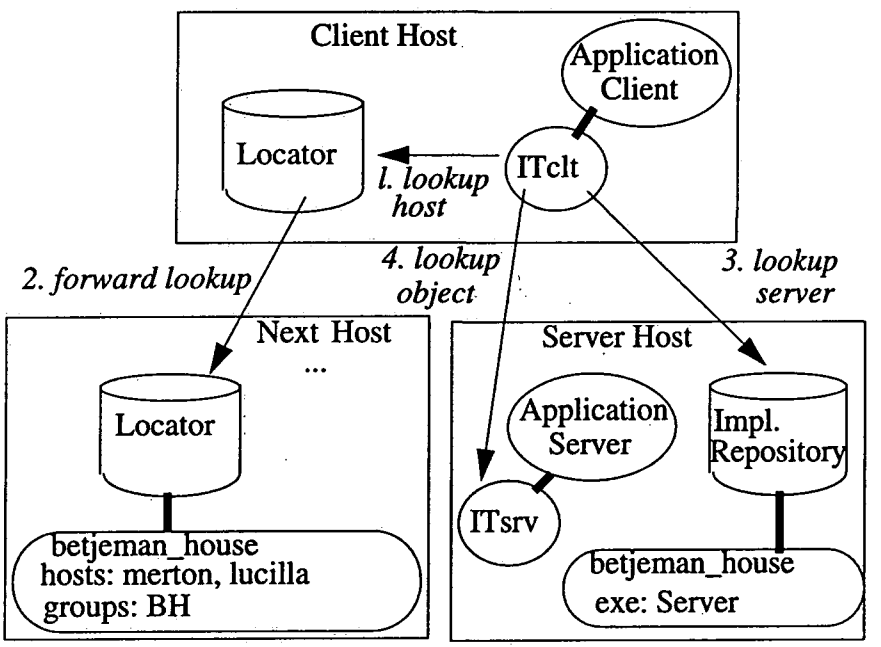


Figure 104 Binding in Orbix

B.5.4 DCE Concepts

DCE provides a Name Server, called a Cell Directory Server, that allows servers to be selected by name. It also supports named groups of servers that can be selected randomly or by prioritisation within the group.

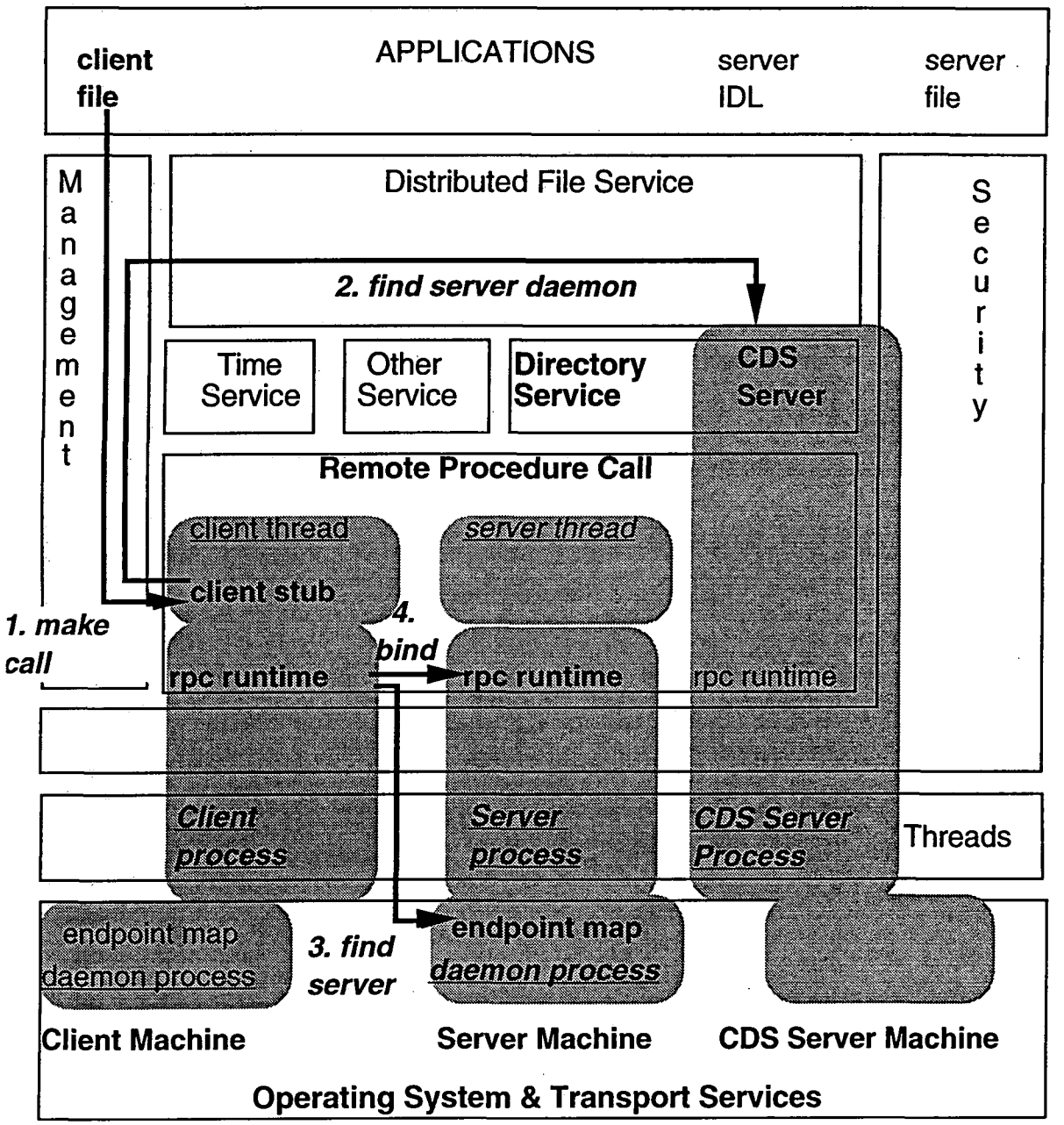


Figure 105 4 Step Binding in DCE

The cell directory server architecture is based on a central master server with replicated caches on each node. Every time a lookup is performed, the cell directory server entry for that name is cached locally on the client machine.

When a client tries to bind, a lookup is first performed in the local cache,as in step 1 in Figure 106. If the server is not found, the master cell directory server is then queried, step 2. The cell directory server returns the host address and protocol for the server or an exception to say that the server is not found. This does not determine the process. The actual end point for the server process is looked up in the end point map managed by a daemon process running on each node, step 3.

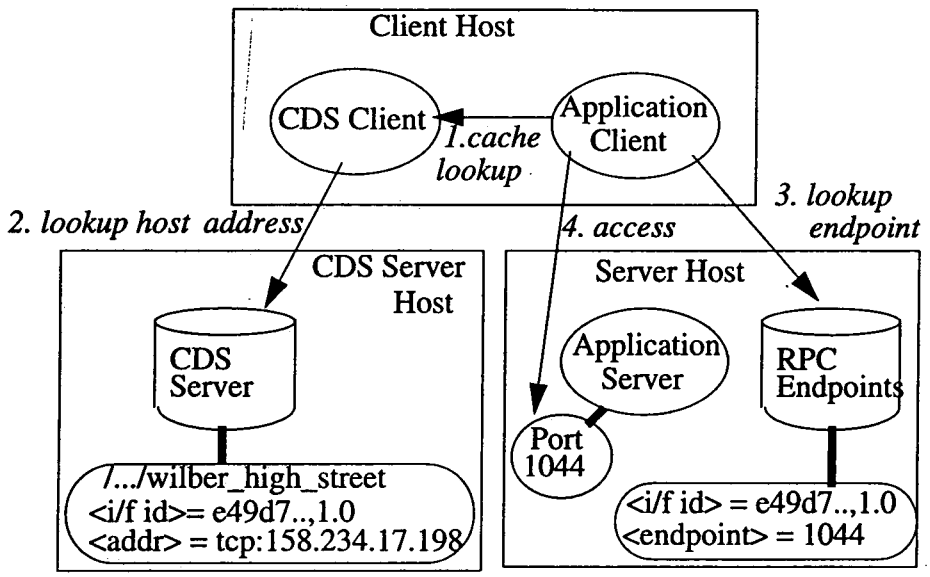


Figure 106 Binding to Servers

B.5.5 Example

In the banking scenario, the broker client must bind to the appropriate bank server.

In Orbix, binding is simple. The client must call the *_bind* method on a pointer to a bank object. *_bind* is generated automatically via the IDL preprocessor. There are several options for arguments to this method that correspond to the options for registering servers in section B.4.5.

*usage: interface *ptr = interface:_bind("marker : server " [, location]);*

location is the optional hostname and can be derived from the locator service if not provided. *marker* is the object name, *server* is the server process name. The server defaults to the interface name. The *marker : server* pair also defaults to the interface name if not present and to an object name if the separator ":" is absent. The following options illustrate some of the possibilities:

```
bank *ptr = bank::_bind(":merton_high_street","merton") ;
// binds to server called merton_high_street on host called merton.

bank *ptr = bank::_bind(":merton_high_street" ) ;
// binds to server called merton_high_street using location service to find node.

bank *ptr = bank::_bind("merton_high_street:NatWest","merton") ;
// binds to object called merton_high_street in server called NatWest on merton.

bank *ptr = bank::_bind("merton_high_street:") ;
// binds to object called merton_high_street in server called bank on any node.

bank *ptr = bank::_bind(":Barclays","merton") ;
// binds to any object in server called Barclays on merton
```

```
bank *ptr = bank::_bind() ;
// binds to any object in a server called bank on any node
```

The second, fourth and sixth options illustrate how location transparency can be provided using a location service.

The pointer can then be used to invoke methods just like a local C++ pointer. This is specific to the Iona C++ mapping:

```
ptr->newAccount("kevin poulter") ;
```

In DCE, there are three ways in which a client can bind to the bank server:

- automatically, by setting an environment variable, RPC_DEFAULT_ENTRY.
- implicitly by setting global data that is accessed by the stub
- explicitly by passing binding data to the stub as parameters.

The binding information can be obtained in a number of ways. For example, when selecting from a group we need to iterate through all entries until we get a valid server using the following code:

```
/* get lookup_context from RPC_DEFAULT_ENTRY */
rpc_ns_binding_lookup_begin (rpc_c_ns_syntax_default, NULL,
bank_v1_0_c_ifspec, NULL,1 , &lookup_context,&status) ;

/* get binding_vector from name service */
while (binding_vector)
{
      rpc_ns_binding_lookup_next(lookup_context, &binding_vector,&status);

      /* get entry for binding from binding vector */
      while (binding)
      {
            rpc_ns_binding_select(binding_vector,&binding,&status) ;

            /* resolve binding by asking the RPC Daemon to lookup endpoint */
            rpc_ep_resolve_binding(binding,bank_v1_0_c_ifspec,&status) ;

            /* test server is alive*/
            rpc_mgmt_is_server_listening ( <arguments to do >) ;
      }
}
```

The binding can then be passed as the first argument in an RPC call or set to the global data , for example :

```
newAccount( binding, "Kevin Poulter") ;
```

The use of global data is not illustrated here.

Both examples above exclude any error handling, this is added in the next section.

B.6 Activation and Failure Handling

Terms: exceptions, object adaptor, loader, null proxy

B.6.1 Overview of Functional Scenario

Distributed systems have the potential to remain operational in spite of the failures in individual nodes. Sophisticated systems provide mechanisms to preserve consistency of distributed information in the presence of concurrency, partial failures and aborting activities. This includes locks, mutex and condition variables, rendezvous, path expressions and monitors for safety; 2 phase locking, sequenced locks, timestamping, transaction ordering and domain relative addressing for serialisability, to prevent aborting activities undoing other activities; rollback logs, shadow replicas and 2 phase commit mechanims for atomicity to ensure activities either complete or have no effect; and checkpointing and replication support for robustness.

In spite of these protection mechanism, the programmer must also be able to specify the recovery behaviour to execute should a server or communication line fail. An exception handling facility is the most common solution to this. Exceptions may be thrown by the programmer or by the system. Some RPC systems make error handling explicit in the request statement, for example adding parameters to specify the number of retries or the timeout period to use before aborting. Others attempt to incorporate standard C++ exceptions. An issue here is how to migrate application code from compilers that don't yet support C++ exceptions to those that do. Many systems provide macros. Without real exceptions, a mechanism is required to stop latter calls in a sequence of calls from having an effect if an earlier call raises an exception.

One way to characterise an RPC system is by the failure semantics. It is quite common for RPC systems to guarentee exactly-once semantics in the absence of failures and at-most-once in the presence of failures. Simpler systems support at-least-once semantics.

Communication failures such as lost, corrupted or out of order messages are usually determined and recovered by the transport layer. The most common error visible to the application is long term communication failure and server process failures. The most useful facilities to define recovery behaviour are facilities to manage the allocation of servers across the network. This includes detecting failed servers and reincarnations of a failed server; reselecting distributed servers from groups; activating server processes from executables or persistent stores; and migrating servers to another node.

The most basic facility is to be able to start and terminate a server process and to detect and report process and long term communication failures.

B.6.2 ANSA Concepts

In ANSA a process is maps onto the concept of a capsule.

ANSA provides the concept of managed interfaces, managed objects and managed capsules to allow the writing of generic code to manipulate objects and capsules to implement configuration managers.

All ANSA interfaces support the Management Interface that allows a set of different interfaces to be obtained, including: an interface to "ping" the interface; an interface to manipulate the enclosing object; and an interface to manipulate the enclosing capsule.

Managed objects are defined using the PREPC MANAGED declaration, which takes as arguments a list of object names that a capsule can create. All capsules are managed capsules and support the Capsule interface that can be used to call an object instantiation service (called Capsule$Instantiate) to create managed objects by name and to call the object finalisation service (called Capsule$Terminate())to destroy managed objects by name. This uses templates defined in the application code. Applications must provide constructors and destructors for all managed objects, as described in section B.3.2..

ANSA provides factories that create objects in two stages. First they create capsules containing object templates using the Factory interface and supplying a search path and executable for the capsule. Then individual objects are created from the templates using the object instantiation and finalisation services of the Capsule interface.

Node managers combine the use of traders and factories to startup and control both static and dynamic services. Offers can be registered in the trader for passive objects and corresponding executables are registered in the node manager. The executables are activated by the node manager on demand when a client imports an offer for a server that is not already active.

Recent versions of ANSA also generate support for migration, activation from stable storage and passivation to stable storage. This is achived by using the PREPC declaration *STORAGE SERVER typename STATE (a IDLa, ..., n IDLn)* in the server and *STORAGE CLIENT typename* to generate the operations *migrate, activate* and *passivate*.

ANSA supports exception handling with functions to signal exceptions and recovery blocks specified in the invocation form of DPL. The action to be taken on process exit by the Capsule library can be controlled to exit gracefully, the default, or perform a core dump.

A relocation service to migrate services to another node, or activate and passivate objects on stable storage is also provided.

B.6.3 ORB/ORBIX Concepts

The implementation repository of CORBA allows a client to use a server that is not active. The implementation repository will activate the server if it is not already active. There are two types of activation: implementation activation, which occurs when no implementation for an object is currently available to handle a request, i.e. no active executable or process; and object activation, which occurs when no instance of the object is available to handle a request. There are four main policies to activate servers and objects :

- shared server - multiple objects of a given implementation share the same server process.
- persistent server - as shared server but activated from a persistent store by a specialised adaptor, not the basic object adaptor.
- unshared server - a single server process is activated for each object.

- server-per-method - a single server process is activated for each request and the server is terminated at the end of each request.

The choice of option requires a tradeoff between the poor throughput of a single shared server and the latency and excessive resource requirement to activate multiple servers.

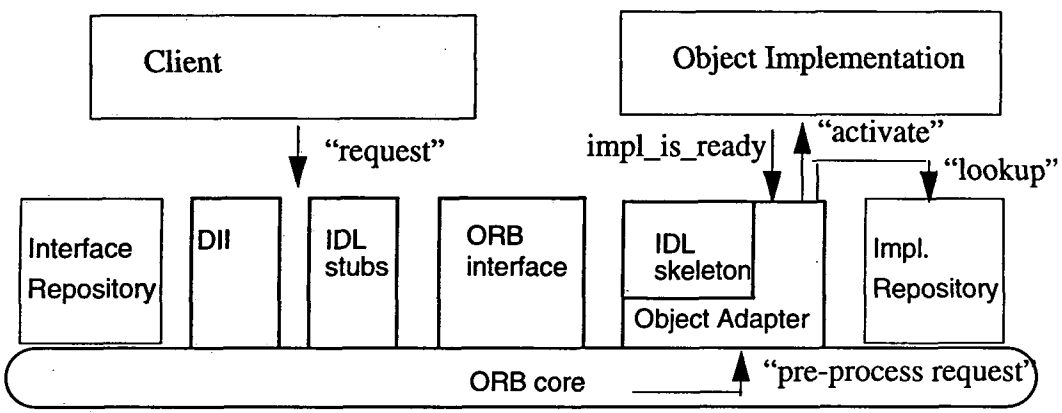


Figure 107 Preprocessing a request - activate implementation

On processing a request, the object adaptor will lookup the activation state of the server and object instance in the implementation repository as shown in figure 1. If a server is already active and the shared server policy is in force or if the object instance is already active and the unshared policy is in force, the adaptor will deliver requests and object activations to the server. Otherwise, the object adapter starts (i.e. "activates") a new server. On activation, the server initializes itself then notifies the adapter that it is ready to receive requests by calling *impl_is_ready* or *obj_is_ready* respectively for the shared and unshared policies. A server remains active and receives requests until it calls *deactivate_impl* or *deactivate_obj*.

Orbix allows timeouts to be specified with *impl_is_ready* or to use event loops that accept one request at a time by calling *processNextEvent* instead of *impl_is_ready*.

No lookup, activation or notification of readiness is required under the server-per-method policy as a distinct server is activated for a single known request and deactivated on sending the reply.

CORBA also defines an exception handling interface. Struct-like data structures can be defined for user defined exceptions in IDL. These structures should include an identifier for the exception and any return values. User defined exceptions are raised by creating an instance of this structure and assigning its data then assigning it to the environment variable that is passed as a parameter in all methods. CORBA also specifies some standard runtime exceptions that are raised by the ORB for failures such as communication, memory allocation, resource and data handling.

Orbix provides macros for C++ "try" and "catch" statements so that exceptions can be treated as C++ exceptions even if the compiler does not yet support exceptions. Exceptions are encapsulated in the Environment class which is passed as a default trailing argument to all invocations. Orbix also supports the concept of null proxies that merely propegate exceptions when requests are issued to them. This allows a sequence of invocations to effectively stop if one of them raises an exception. For any call, the environment variable argument is examined and if an exception is already raised, the request is ignored. Any output data is initialised to zero or to a null proxy. Subsequent requests issued to null proxies merely propegate the exeception and initialise their return data.

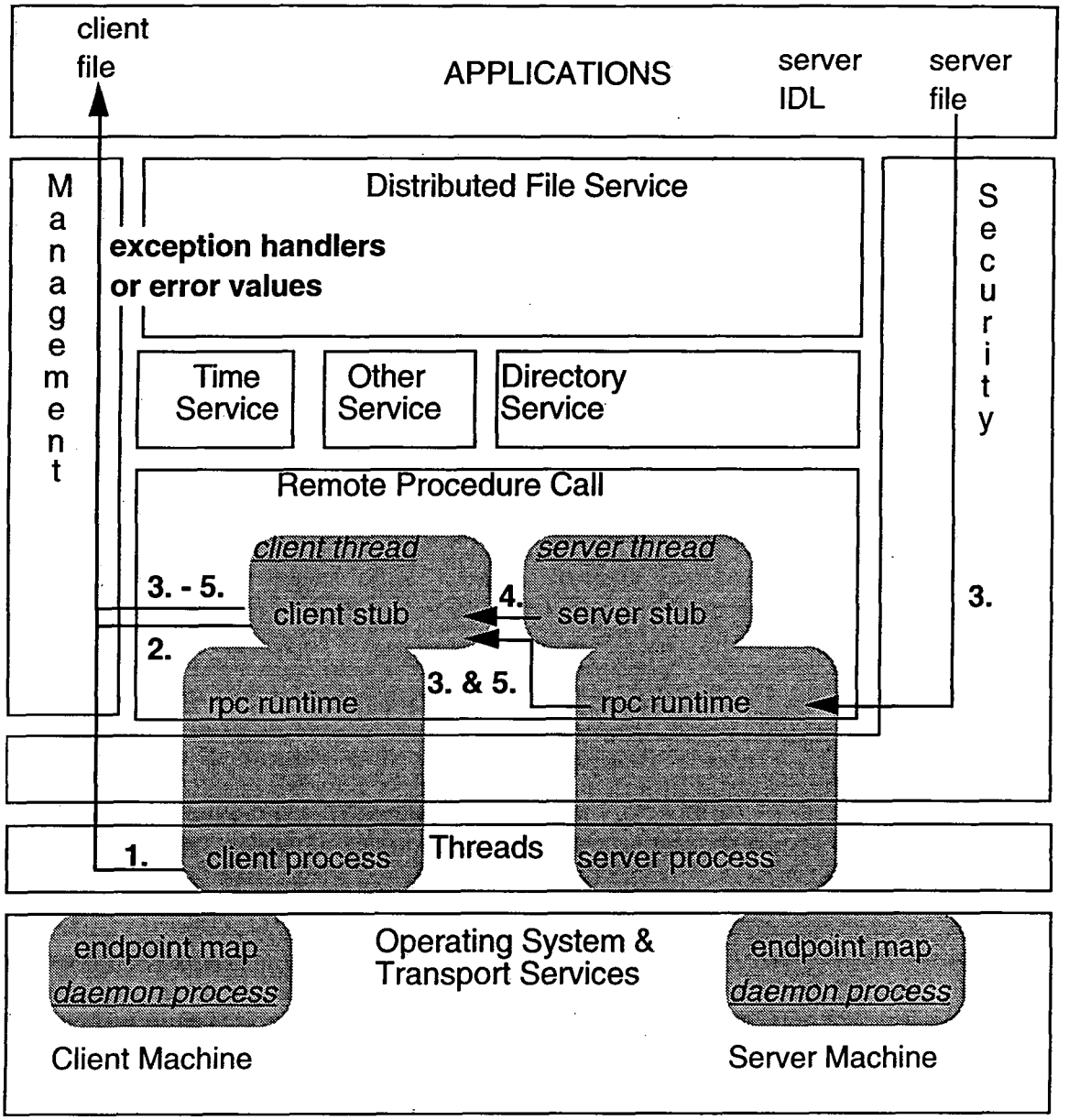
If a server process fails, any robust implementation of the CORBA specification should notify the client through a system exception. It would be easy to restart the server but this does not ensure consistency and atomicity.

Orbix may be integrated with Tuxedo to ensure state changes are made within transactions that can be rolled back consistently on failure and commited atomically using a 2 phase commit protocol.

B.6.4 DCE Concepts

DCE does not support process activation. Processes must be started explicitly to configure the system.

DCE supports exceptions for system and user defined server errors. Exception handling code needs to be written in the client to recover or gracefully exit an action. Macros are provided to simplify the programming of exception handlers.



Types of Exceptions

1. System exceptions on client e.g. out of memory
2. Communication exceptions, e.g. server crash
3. Application exceptions, e.g. call faulted
4. Server stub errors, e.g. buffer memory allocation
5. Server runtime support errors, e.g.

Figure 108 Types of exceptions in DCE

An Attribute Configuration File can also be used to simplify error handling by adding error status parameters to the operation argument list. If the parameters comm_status and fault_status are included in the ACF file, errors are communicated to the client as data values in these parameters rather than by raising exceptions.

DCE provides services to detect failed servers, *rpc_mgmt_is_server_listening*, and to reselect servers from groups, *rpc_ns_binding_lookup_next* as described in section B.5. DCE does not distinguish reincarnations so errors can occur if a process is restarted in an unexpected state.

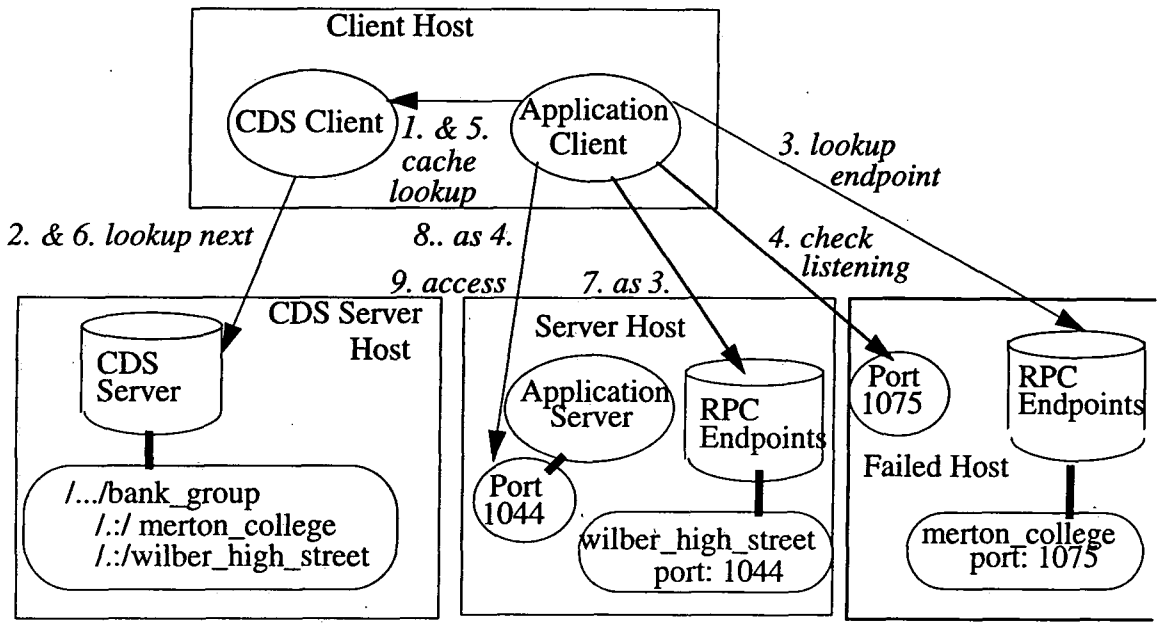


Figure 109 Use of Groups to Recover by Selecting Another Server

DCE also allows failure semantics to be relaxed by qualifying an operation with the *maybe* modifier in IDL.

B.6.5 Example

The server in the Orbix example can use any activation policy. The policy is determined by the choice of arguments when registering servers in the implementation repository as in section B.4.5. The arguments *-shared* and *-method* makes the choice explicitly the shared policy and per-method policy respectively. Otherwise the unshared policy is assumed when markers are used and the shared policy is assumed when markers are not used.

The main body of the server must be consistent with the choice of policy. In particular any markers must be defined before notifying the object adaptor that the server is ready.

```
void main {
        // create bank objects and ties using new operator
        // set markers for bank objects using marker(name)
        impl_is_ready()
}
```

In the banking scenario, it may be possible for the bank to reject a request for a new account. The IDL interface may define an exception called *reject.*

```
// IDL for bank

interfac ebank {
```

```
        exception reject { ...// attributes
                           string reason ;
                         } ;
        void newAccount (in string name) raises (reject) ;
...}
```

The implementation of newAccount will raise this exception by creating an exception object and assigning it to the environment variable.

```
env = new bank::reject ("Bad debtor") ;
```

A client must then protect his call to newAccount by testing for the user defined exception as well as any system exceptions. The TRY macro introduces the IT_X environment that must be passed to every call. The CATCH macros read the IT_X variable and execute the recovery code specified.The Orbix Environment structure also supports stream operators to simplify reporting exceptions. Note that if an exception is raised, a null proxy is returned and assigned to *acc* . In this case,*makeDeposit* does nothing other than propagate the exception.

```
TRY { ...
        acc = bank->newAccount ("Ken",IT_X) ;
        acc->makeDeposit(200,IT_X) ;
}
CATCH (CORBA::StExcep::SystemException,s) {
        cout << "System exception : " << s << endl ;
        ... // try to restart the server on another node
}
CATCHANY {
        cout << "Awkward bank" << IT_X << endl ;
        ... // retry another bank
}
```

In DCE the processes must be manually started by typing on the relevant node:

```
% client

% server
```

Before doing this it may be necessary to login to the DCE system, to create any principles and keytab entries used by the security service (using rgy_edit) and to set up environment variables such as RPC_DEFAULT_ENTRY if automatic binding is used.

In DCE, the exception macros look like this:

```
TRY { ...
        newAccount("ken") ;
        makeDeposit("ken",200) ;
}
FINALLY
{
        puts("Error occurred in transaction") ;
        .../* retry another bank */
}
ENDTRY
```

If an error occurs in the TRY section, the code in the FINALLY section is executed. Alternatively DCE provides CATCH and CATCHALL macros which can be used in place of FINALLY in a similar style to Orbix.

B.7 Synchronisation and Request Processing

Terms: thread dispatcher, DII, filter

B.7.1 Overview of Functionality

A client makes a request. The form of expression used to make invocations may be the same as for local calls - this is called access transparency. Alternatively a different syntax or even embedded language may be used to allow for the differing failure, concurrency, memory management and argument passing semantics. An issue in the former case is how to allow for the extra complexity and latency inherent in remote calls. A client must defend against communication failure and make allowance for the inherent latency in sending a message across a network (typically more than a 1000 times slower than a local call.)

An IDL compiler or stub generator is a valuable gift. Stubs hide the complexities of interfacing to the communications system behind the higher level binding and invocation mechanisms of procedural programming languages. Specifically, the IDL compiler builds the appropriate code to manage proxies, dispatch incoming requests in a server, and manage any underlying object services.

However the use of IDL and stub generators restricts the contextual independence of the client. The client can only use servers whose interface is known in advance. Some applications and tools such as browsers, management tools, assembly tools and interactive interfaces do not want to restrict clients to use a specific pre-fabricated interface. A generic API is an alternative to stubs. Here the issue is how to deal with typing information. Some simple APIs use unix string formatting to deal with type information generically. Utilities such as scanf can be used to format simple data types. However this makes it difficult to pass constructed data types like sequences, unions and object references. An alternative is for the RPC system to provide an API that enumerates all the supported data types so that a client can pass typing information generically as enumerated values. An issue is how to get the appropriate type information at runtime since if it is known in advance then a stub interface is simpler to use. Type information may be obtained from repositories.

With the clear departure from typed procedure call, many APIs also offer richer functionality to deal with distribution and concurrency.

Concurrency may be built into the computational model of a RPC system. Remote procedure call systems normally include a scheduler which presents the semantics of a single thread which is dynamically allocated to passive objects as a single logical thread runs through the call stack. Remote calls are inherently between two distinct threads, in distinct processes, and this is hidden by the RPC system. The simplest schemes use remote communication points as scheduling points with a "run-to-block" policy. Without scheduling or thread support, a complete process will block on a single synchronous two-way procedure call, severely reducing the degree of parallelism in a network.

More sophisticated RPC systems allow asynchronous oneway calls and deferred synchronous two way calls where the asynchronous issuing of the request is separated from the synchronous collection of results.

For a greater degree of concurrency, a concurrent object oriented language may be used. Most adopt an active object model where threads are allocated permanently to objects and communication is always between threads. Alternatively an orthogonal API may be used to manage threads explicitly as with POSIX or Sun thread libraries. In this case, the programmer must take care to make the code thread safe.

For availability or performance, an object may be replicated and stored locally to applications across a network. This brings additional problems in managing interactions within groups of replicas. Changes may be propagated peer-to-peer or master to slave. Multiple calls must be collated into a single reply. Members of the group must be synchronised and groups must be able to recover from member failures.

B.7.2 ANSA Concepts

ANSAware provides a distributed processing language, DPL, and accompanying processor, prepc. DPL was designed as a concrete syntax for writing programs that conform to the abstract semantics of the ANSA computational model. DPL fragments may be embedded in application programming languages to provide facilities relating to binding and invocation. The prepc compiler augments computational objects with code that provides the required transparencies. Prepc provides a simple syntax for dealing with exceptions and recovery behaviour. Retries can be specified as a parameter. Recovery procedures can be identified explicitly.

ANSA supports a point to point communication protocol and a group communication protocol to synchronise interactions between replica groups of objects. The point to point protocol supports bulk data delivery through fragmentation strategies, rate based flow control so as not to overload a server, deferred synchronous calls that issue tokens to be redeemed when the client wishes to block on the return values, and asynchronous messages with at most once semantics. The lower layers of the communication software use Berkely UNIX socket interface, principally to the TCP/IP protocol suite

ANSA uses a declarative notation to impose concurrency control, called path expressions). Many other techniques make specific assumptions about specific concurrency mechanisms that inhibits portability.

ANSA provides a dispatcher to provide integrated scheduling and communication. This will provide user level threads on systems that are not multithreaded. ANSAware has also been ported to run over DCE/POSIX threads and use DCE RPC as the underlying transport protocol.

B.7.3 ORB/ORBIX Concepts

The most common type of interaction in CORBA is synchronous procedure calls using stubs. The client blocks waiting for the return values. IDL allows operations without any return values to be qualified as "one-way" in which case it is possible for the client to adopt an asynchronous model and to carry on processing after issuing a request.

The dynamic invocation interface is more flexible and allows three types of request scheduling:

- synchronous - this is a normal invocation. The client blocks waiting for the reply.

- deferred synchronous - this is a two way call in which the request is sent asynchronously with the client carrying on processing then later synchronising by a second blocking statement to collect the results.
- multiple deferred synchronous - this allows a client to talk to multiple servers in parallel, by issuing multiple requests asynchronously and then blocking to collect the results one at a time in the order they come back. This means the client can be processing the fastest replies in parallel with the slowest servers.

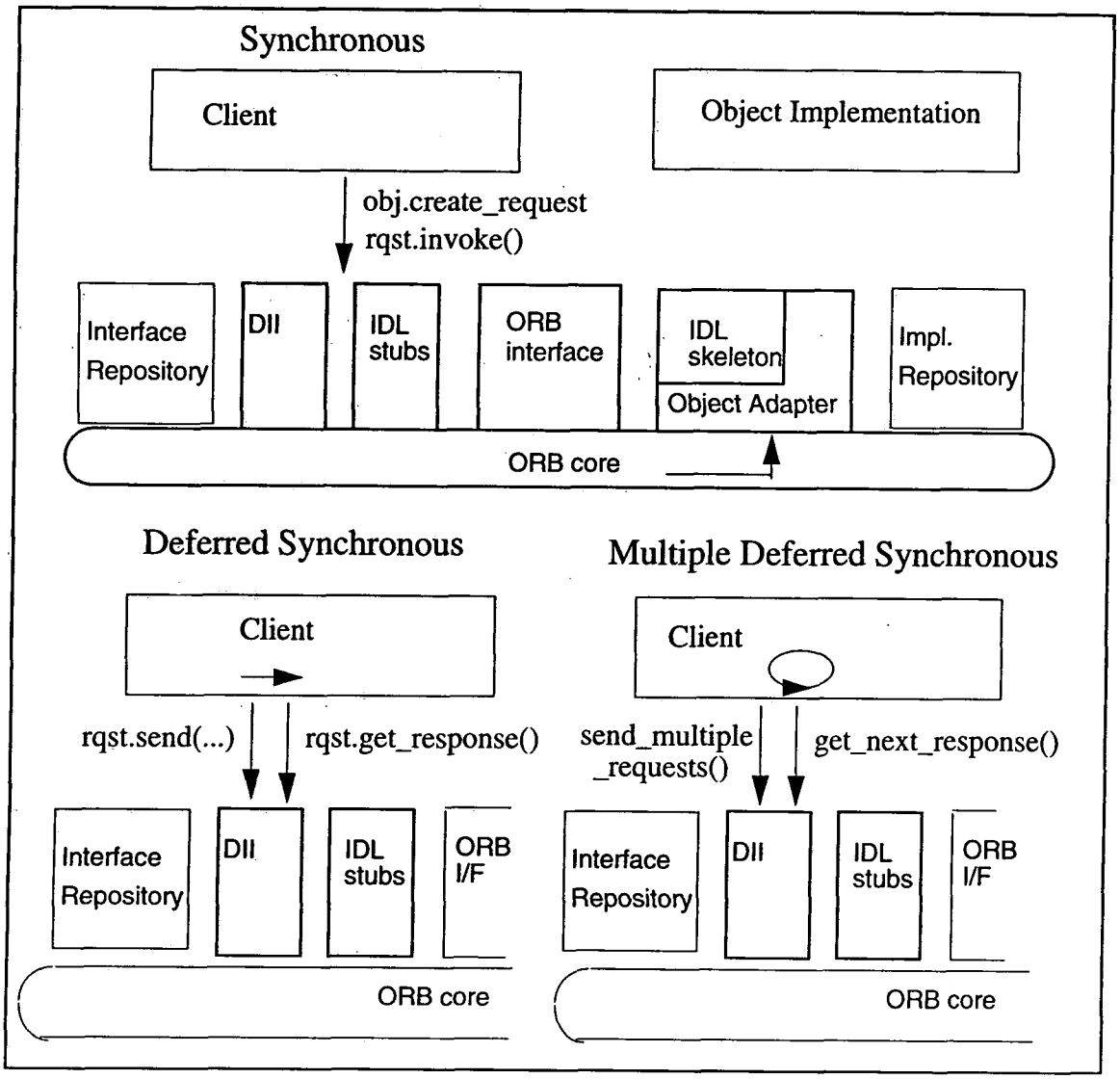


Figure 110 Issuing a request

The dynamic invocation interface allows invocations to be constructed at runtime by specifying as data values the target object, the operation name and the parameters instead of by linking to a stub generated from the server. Type safety is achieved by constructing a list of named values (an NVList) with each value representing an argument value and including a tag which indicates the type of that argument. The ORB at the server end will assert that the NVList conforms to a method. The server itself does not care whether the request was dynamic or static.

Clearly the client uses a different binding mechanism to that for stub invocation. A C++ pointer is typed and can't be used in a generic interface. Instead the client issues a request by constructing a request object using the generic method Object::create_request. This takes the context, the operation name, the parameter list and the result. The parameter list can be omitted in which case it should be added in a stepwise manner using Request::add_arg. The request object is then issued by calling Request::invoke or Request::send/send_multiple_requests and get_response/get_next_response as shown in Figure 110.

The object reference for dynamic invocation can be translated to and from a string thus allowing bindings to be treated also as data values and imported at runtime from a repository or name server.

Type information may be imported from the interface repository by browsing, by invoking get_interface on any referenced object or by invoking lookup_id on a repository object whose repositoryid is known. The interface repository allows clients to explore interfaces and compose appropriate NVlists on the fly.

In Orbix the C++ mapping is such that a C++ pointer is used as the object reference for static invocation and a normal C++ invocation is made to issue a remote request via the IDL stubs. There is one additional argument, the Environment variable.

Orbix has a cleaner interface to the DII than that standardised. It provides a Request class which hides the interface to the NVList behind stream operators that are overloaded for each C++ type to tag the list with the appropriate type information automatically.

Multithreading can be introduced in Orbix by programming a filter that is fired every time a request is received by a server. This filter makes a system call to spawn a new thread to handle the request. This is only possible if Orbix is running on a multithreaded operating system. Orbix does not implement its own run-to-block style of scheduler.

B.7.4 DCE Concepts

A DCE call appears like a normal C call - but has different semantics for memory management of pointers and for constructed data types as discussed in section B.3.

A call request comes in over the network and is placed on the request queue for the endpoint. A server can select more than one protocol sequence on which to listen and each protocol sequence can have more than one endpoint. The RPC runtime library dequeues requests and places them on a single call queue. Requests can be processed from this queue concurrently using threads. If a thread is available the request is automatically dispatched to it, otherwise the request waits on the call queue until a thread is available. If the request queue or call queue become lost, the next request is rejected.

Before passing the request to the stub, the interface specification in the request is compared to the interface specification supported by the server and the request is rejected if unsupported.

The components involved in processing requests and replies are shown in Figure 111.

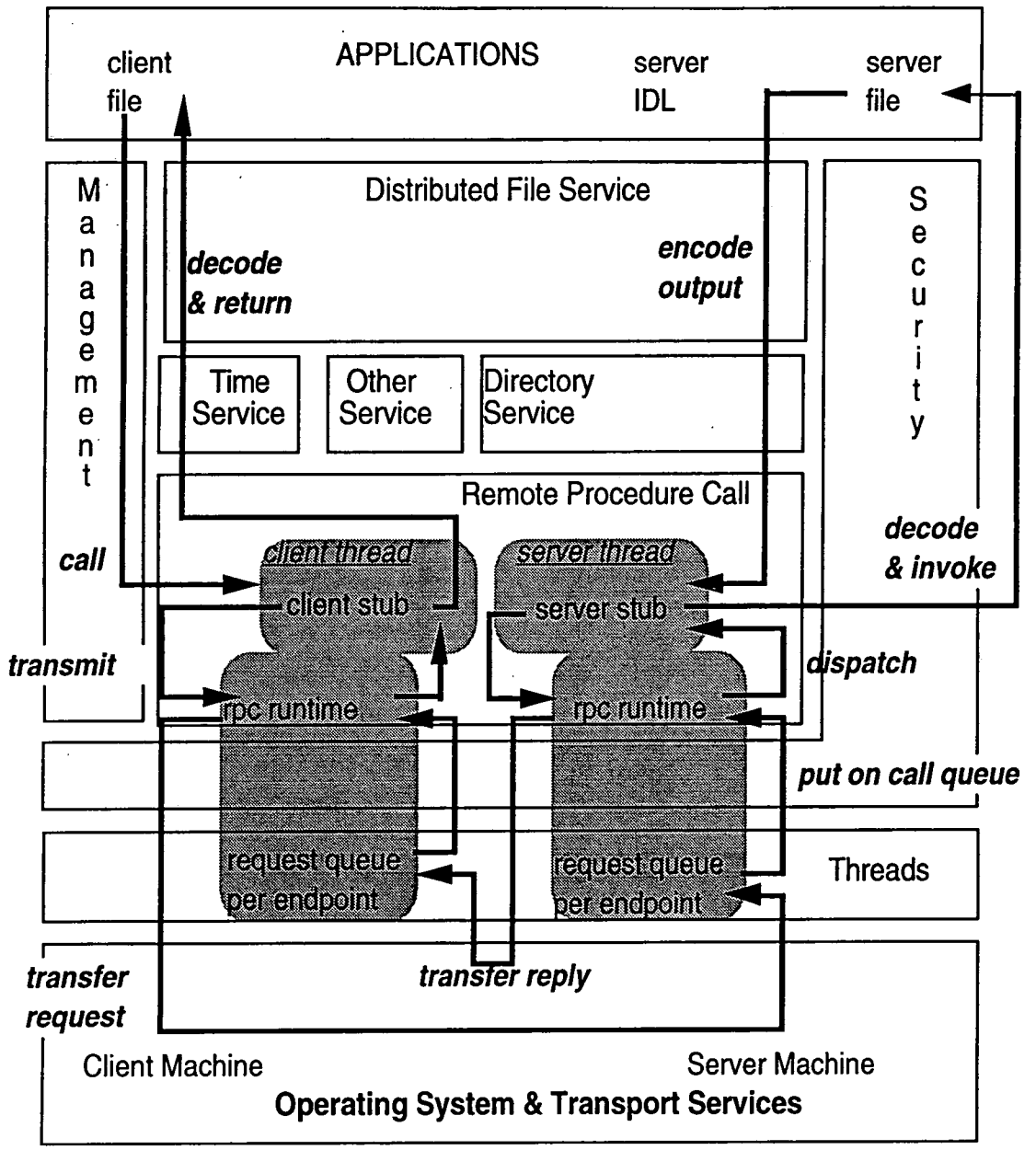


Figure 111 Request and Reply Processing in DCE

ACFs modify the way the IDL compiler generates stubs and consequently affect the request statement. ACFs can add parameters to the operations parameter list and exclude operations from the interface to a given client. They also allow control over the size of the stubs and performance by determining whether marshalling code is to be inline or out of line. Parameters are added for explicit binding to pass the binding handle for a client and for error handling to signal errors by passing data values rather than raising exceptions.

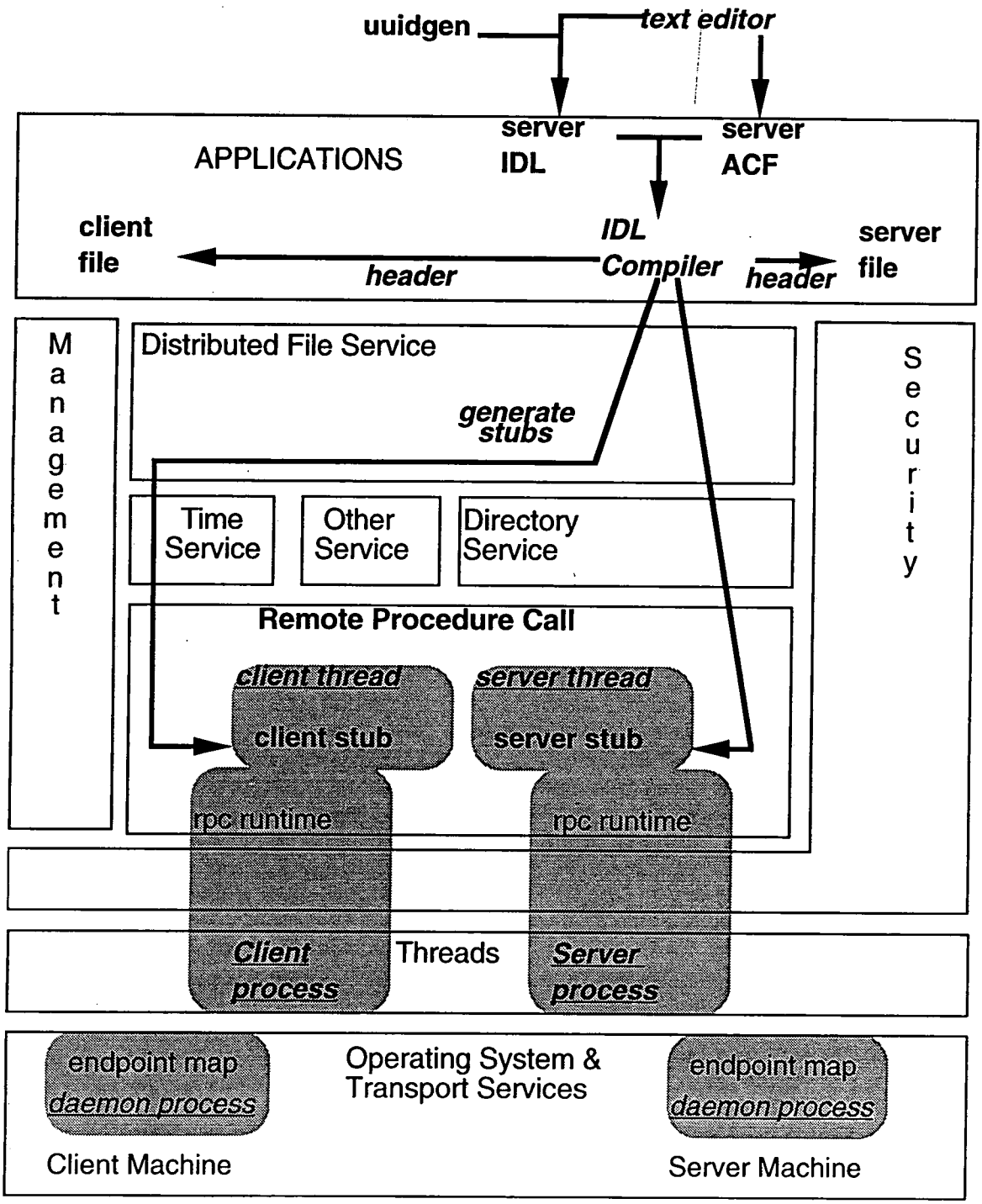


Figure 112 Use of ACF file to modify IDL compiler output

DCE provides no DII interface or generic API for making calls.

The DCE threads subsystem is integrated with DCE RPC to support multithreaded clients and servers. However DCE does not support deferred synchronous calls or asynchronous oneway calls.

DCE also supports broadcast RPC to all servers by qualifying an operation with the *broadcast* modifier in IDL.

B.7.5 Example

In the banking scenario, the broker program must invoke methods in the bank server.

```
TRY {.../// get the bank and account names and the deposit amount
        Bank *bank = bank::_bind("Ely_High_Street",IT_X) ;
        Account * acc = bank->newAccount("Ken",IT_X) ;
        acc->makeDeposit(200,IT_X) ;
}
CATCHALL { cout << "Exception :" << IT_X << endl ; }
```

The same methods can be invoked in Orbix using the dynamic invocation interface provided by the Request class. This class overloads the stream operators for each C++ class:

An orbix example for dynamic invocation would look something like this :

```
foo()
{
        ... // get the bank and account name and deposit amount
        // bind to name and select the method
        CORBA::Object bank("Ely_High_Street");
        CORBA::Reference r (&bank,"newAccount") ;
        / insert the argument
        r << "kenw" ;
        // invoke the operation newAccount
        r.invoke() ;
        // extract any results of modify using >>
        CORBA::Object acc ;
        /r >> acc ;
        CORBA::Reference r2 (&acc,"makeDeposit") ;
        r2 << 200 ;
        r2.invoke() ;
}
```

Reference passing for the account is not shown here nor exception handling.