Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Federation and tools manifesto

Rob van der Linden, Jane Cameron

Abstract

This paper argues that The way in which we deal with software products is itself a major inhibitor on the road to increased productivity, flexibility and reduced time to market.

The reuse of software components must be encouraged. Component specifications need to be published widely, the components themselves need to be accessible. Tools need to be put in place which facilitate reuse in large distributed environments.

When components are used, the producers must benefit in some way. If this does not happen, then there is no incentive to make components widely available. A use based charging system is proposed as only viable in a distributed system.

NOTE: This paper is pretty rough in places. DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

APM.1100.00.02 **Draft** 26 November 1993

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Federation and tools manifesto

Request for Comments (confidential to ANSA consortium for 2 years)

Federation and tools manifesto

Rob van der Linden, Jane Cameron

APM.1100.00.02

26 November 1993

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

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

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

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

1 Introduction

The trend in the open distributed systems movement is to develop software components as services [ANSA, OMG, ODP, Object Orientation]. This has important economic advantages to those who use the resulting software components to build their systems:

- services can be created from existing component services (*reuse*)
- by replacing one component by another services can be upgraded without upsetting current service users, service properties can be improved or functionality extended for new users (*substitutability*)
- components can be used in many places and at different times, by remote access, or by copying a service into the local domain, either dynamically (at run time) or statically (*configuration*).

All of these allow system builders and service providers to

- reduce their time to market and hit market windows
- customize and change services to meet customer demands for flexibility
- reduce the cost of developmentand help offer systems and services at competitive prices.

Developing software components as reusable services rather than embedded functions has fewer advantages for component producers:

- development may take longer, thus market windows may be missed
- initial costs may be higher
- service component producers may not perceive reusability as important enough to warrant the extra investment and delay.

In todays competitive markets, short term economic benefits are all important. For software component producers the benefits appear decidedly small. Unless the prospects for producers can be improved, they are unlikely to provide reusable components and reconfigurable services in distributed computer systems remain a distant ideal.

Three critical success factors can be distinguished:

- producers should be able to charge for the *use* which is made of the components they produce
- system builders and service providers should be *encouraged* to reuse software components
- tools used to develop and reconfigure services should be *improved*, that is made suitable for use in large heterogeneous distributed systems.

Charging on the basis of *use* fits very well with the notion of service provision by encapsulated software components. Charging can take place at all levels: where software components use other components and where components deliver service to a (human) user. Use-based charging recognises that software (and other information products) can be copied in nanoseconds and

transported at the speed of light [Cox 93]. *Superdistribution* [Mori 90] is based on the idea that it is impossible to control the copy process (because it is external to a software component) but trivial to control its use (because this is intrinsic). Current economic, legal and technical mechanisms prevent the copying of software, protecting the producer who bills for software on the basis of *ownership*. Several technical questions, set out in this document, need to be addressed if use based charging is to become fully practical. Chapter 2 illustrates that distributed systems technology and security mechanisms are the key enabling technology.

System builders and service providers need to be encouraged to (re)use software components. Unfortunately, today's systems and development strategies do not ease exploitation of existing components for integration into large scale systems. It is difficult, or not possible to find out what already exists. It is next to impossible to determine what is being created by others. It is often unclear just what a component does or does not do, just how to access its functionality, and just what side-effects it can create. If it takes more time to solve these problems than to write a new component from scratch, then it will inevitably be rewritten.

Three conditions need to be fullfilled to encourage reuse:

- service specifications should be as complete as possible
- service specifications should be visible
- service implementations should be accessible.

There are two sources of existing technology which enable the visibility of service specifications and the accessibility of implementations:

- distributed systems technologies support service access in large heterogeneous environments but lack the mechanisms for holding and dealing with comprehensive service specifications (Chapter 3)
- system development and programming tools (including class libraries and associated browsers) have mechanisms for storing more comprehensive “service” specifications but do not have the mechanisms to make these specifications accessible in a very large heterogeneous environment (Chapter 4).

A third technology is currently emerging from companies involved in OMG CORBA. Interface and implementation repositories are part of the CORBA specification (Chapter 5).

Finally, the tools which are used to construct services and to configure and reconfigure them need to allow the reuse of system and application components. Traditional compiling and linking tools which are based around a specific intermediate representation (e.g. an abstract syntax tree) support the reuse of components in restricted contexts. More advanced linking and configuration tools are becoming available to support reuse in a wider context. There is little experience in operating such tools in large distributed systems however. Chapter 6 addresses some of the issues.

Chapter 7 contains a summary of the key areas of technical work which will provide the technological enablers in the context of the automated transparency and federation topic groups.

2 Charging and paying for software

Third party software cannot generally be used in a product or service without some sort of explicit legally binding agreement (e.g. a licence). In business terms, such agreements are often expensive to obtain.

This is not true in other engineering disciplines. A hardware component, such as a chip or even a complete board, can be used anywhere and for any purpose once it has been bought. If software is burned into the chips the same applies: buy it and use it anywhere and for any task[1]. The resulting value added product can be sold on for a profit, without the component sources requiring a slice[2]. This state of affairs allows the industry to quickly build on what others produce, and to progress in terms of productivity.

The software industry has not progressed to this state of affairs, perhaps because software represents both *design* and *implementation* (*commodity*). If software is delivered on a tape or disc or via ftp services, it cannot be included in a product or a service, unless the software component provider is in on it. The resulting business structure frequently makes it easier to simply rewrite the required functionality.

Rewriting software is perceived as wasteful: the more complex the software, the more difficult and costly it is to test. Reinventing algorithms also results in a proliferation of components, many of which perform similar functions. Components which perform exactly the same function often do so through different interfaces. There is no sense of substitutionality, an essential aspect of any engineering discipline.

We believe that distributed systems technology can act as an enabler for an environment in which

- the software producer will be rewarded for creating reuseable software components (e.g. by increased revenue and profit)
- the consumer will be rewarded for making use of what is provided (e.g. by shortened lead times and reduced costs).

In this chapter we examine the software production process with respect to the markets in which users are charged for the use of software components.

2.1 Use-based and ownership-based charging

The tradition in an important sector of the software industry is one of producing software, then packaging and selling it. When the package is sold, ownership and control over its use (subject to restrictions) is transferred to the new owner. Revenue is generated directly by the volume of sales and indirectly

1. There may be restrictions such as US Government export restrictions.

2. When a patent applies, the patent holder receives a cut. The purchaser however does not see this.

by the rate of innovation: an improved version of MS-Windows or a C++ compiler can generate a surge in sales.

The tradition in the telecommunications industry is one of providing services and charging for the use of those services. Revenue is generated on the basis of volume of use. Until recently, innovation played a much smaller role: up to the mid 80's the basic telephone functionality had not changed much, its volume of use had increased almost to the point of saturation.

The telecommunications industry is now more innovative and offers more diverse and flexible services. Flexibility is achieved by increasing the size of the software component in the service on offer. Revenue is still expected from use of diverse services, not the transfer of ownership.

The increased flexibility of services places further constraints on the extent to which software producers must respond in terms of reuse and reconfiguration of their products.

2.2 Transfer of ownwership

The ownership of all commodities, including software, can be transferred from one party to another (through the process of buying and selling or donating and receiving).

Having to own a software component before being able to (re)use it limits the extent to which the component will actually be reused. The challenges related to owning something which can easilly be copied and moved must also be examined more closely (see “protection” below).

2.3 Transfer of use

The potential to use a service can also be transferred.

Note: This needs an example outside the technical sphere of ANSA.

The ANSA Computational Model is based on a parameter and result passing mechanism whereby each parameter and each result represents an interface (a place of service provision). Thus, as a parameter or result is passed, it may be used by the recipient to try and obtain service.

There are obvious requirements to authenticate potential users so that they can be denied access or charged for the service they use.

2.4 Markets

There exist several specific market paradigms for software and services:

- buy and sell paradigm

  Products like MicroSoft Windows, Lotus 123 and FrameMaker are bought and installed on one or several machines. The number of simultaneous users is regulated by a license server and subject to a maximum.

- license paradigm

  Products like ANSAware are subject to a paid for license which restricts the period during which the product may be used (in addition to the machines on which it may be run and the maximum number of users).

- free software paradigm

  Products like Emacs (editor), MH (mail handler), LaTeX (document preparation), and gcc (compiler) are obtained from well known repositories (FTP servers) and can be used subject to very few if any restrictions.

- use paradigm

  Telecommunication services are charged for on a basis of use. Telephone services are charged for in a mixed paradigm, with the use related part being the largest for most people. Database and electronic mail services such as CompuServe and UK Gold are charged for on the basis of usage. The same is true for the IT department (those that still exist) who charge for CPU time or on the basis of particular processing runs.

2.5 Cost, worth, quality, trust and liability

2.5.1 Cost

The cost of software obtained in the free software paradigm is apparently much lower than that obtained in either the buy and sell or license market paradigms. Much of this software is produced by universities or research establishments which are funded by governments. Frequently, a condition of funding is that the resulting software is made publicly available. Thus, tax payers money is used to pay for the “free” software. There is no data on the cost of the use of software.

2.5.2 Worth

The worth of software commodities is not related to the market on which it was obtained. We know of some excellent software, which has been obtained for very little money (to the users) in the free software paradigm. We have also seen some abysmal software obtained for loadsamoney in the buy and sell paradigm. No general judgement on the relative worth of software which is offered for use can be made at this point.

2.5.3 Quality and trust

It is ironic that the more freely available the software, the less we trust its quality. Brad Cox [Cox 93] says that repositories will become garbage dumps in which “... those who don’t mind sifting through other people’s garbage will hunt for Good Stuff Free, wondering why quality conscious engineers’ response to such reuse is “Not in my backyard!” ”. Today the number of “free” products is still relatively small as is the community that decides to use this software. The message about a “good” or “bad” product thus gets around quickly, increasing the use of “good” things whilst constantly enhancing them, leaving the “bad” and the “ugly”well alone. When this community grows to embrace most of the worlds programmers, it will become more difficult to find quality products.

Whenever software is obtained from a source, there is an element of trust, that the software will do what it is supposed to do. Even if your own programmers write the software, you must trust them to do a good job. Trust can be extended by testing products or by accreditation to standards.

2.5.4 Liability

Software providers in the buy and sell paradigm and service providers in the use paradigm (specially telecommunication service providers) attract the greatest liability for any lack of quality or otherwise of their products. In the license paradigm, the license itself may severly limit liability. In the free software paradigm, no liability is accepted on the part of the originator.

2.6 Size and complexity

Software, such as wordprocessing, spreadsheet, database packages, and compilers are generally treated as commodities. Smaller or less complex software components (matrix invertors, sorting algorithms) are not treated as commodities because they are not owned by someone and marketed.

Large components often possess functionality which is not required. The buyer potentially pays for things which are not used.

Smaller components are harder to obtain and are often reimplemented rather than reused. Compilers for object oriented languages such as Smalltalk now include a library of useful components. Libraries encourage reuse of simple software components, but only do so in the small as it were:

- different suppliers offer different libraries, even if the same language is considered
- the use of a library cannot be enforced, even within a single department or company

2.7 Protection

The unique property of software and other information products is that it can be copied in nanoseconds and transported at the speed of light [Cox 93]. For most commodities it takes time, specialised knowledge and tooling to produce them. We are happy to pay those who possess the appropriate skills and equipment to produce products we need. Software can be copied without special knowledge and by almost anyone.

Software producers who want to sell or licence their products protect themselves against immitators who produce “illegal” software copies. They put in place elaborate mechanisms, both legal and electronic. Whilst these measures are often ineffective, (aptly demonstrated by software piracy which is rife to the point of being totally out of control in many countries) they do inhibit reuse, as access to the product by new parties must first be negotiated with the producers or owners of the IPR.

The free software paradigm relies on peoples conscience to pay a sum of money for use of products. Although some may disagree, this is not generally regarded as an acceptable basis on which to do business.

Those who will charge for the use which is made of their software also require protection. There are three parties each with different reuirements:

1. the component producer must receive information which will allow him to charge for use
2. the service provider must receive information on the use of the overall service (built up from several components)

3. the service user must be protected from unfair charging methods

Distributed systems technology can be applied to track places where charging information is to be collected and to send this information to billing agents. From a technological viewpoint,the required protective measures are feasible.

Legal structures are needed to safeguard the interests of the software writer (e.g. under what circumstances is the writer liable for the functioning of a copy?). This aspect is different in nature in that it requires changes in the contracts which govern the way in which software products and services are marketed. Although important, this is outside the scope of ANSA Phase III.

3 Distributed systems environments

Distributed systems environments (OSF/DCE, OMG CORBA, ANSA ANSAware) facilitate access to services on the basis of little descriptive service information.

ANSA supports the exchange of service specification information at build and run time, to allow services to find out about one another and to interwork in a heterogeneous environment. To support this, ANSA has defined:

- a type checker to limit the number of interaction errors which may occur at run time.
- the ANSA Trader to advertise the existence of services and dispense information on how to gain access.
- ANSA binding technology to hide the communications heterogeneity and to allow service providers and users to interconnect.

The trading function uses the interface type signature (ANSAware just uses a type name), the binding function uses the interface reference, currently with communications protocol, relocation and replication information. There is no support for:

- designers to access and browse service specifications
- implementers to re-use code or data
- maintenance personnel to upgrade services in situ in a type safe manner
- analysis of compatibility between services with specific transaction, dependability, and performance characteristics (all dealt with in ANSA Phase III), or other characteristics, such as security
- expressing aspects of service specifications other than the signature and the protocols which are supported
- analysis of compatibility between other service properties (e.g. to detect unwanted interference, known as “feature interaction” in the telecommunications community) .

The ANSA trading system (ANSAware)

- has been designed to operate in the run-time environment, not to support a design or maintenance activity
- has no support for emitting service specifications to a designer, only type and property names are supported

In OSF/DCE service specifications are restricted through the use of name servers: neither the Cell Directory Service, nor the Directory Service Agents in DCE can support any form of type checking.

4 Specification languages and technology

In this discussion, a distinction is made between the description (specification) of a system and the realization (implementation) of that system. This distinction is not out of line with current practices; however, it is not always clear precisely what is description and what is realization. This section addresses issues surrounding system descriptions; it does not address the fundamental issue of how to ensure that system descriptions and system realizations actually agree.

To specify a software system, four types of questions must be answered:

1. How is the system to be described, in order to define its purpose? What aspects of the system need to be included?
2. How are the descriptions to be structured and organized? How is the system to be structured and organized?
3. Who (or what) needs access to the specification? Who will create the specification? How will the specification be used? How will access to the specification be provided?
4. How are the specifications to be represented, i.e. what languages are to be used? What tools are available to support these languages?

These concerns are somewhat interrelated. However, separating them provides a framework to address software specification. Existing software specification techniques address parts of some of these concerns. Examples include: ODP's five projections (enterprise, information, computation, engineering, and technology) address the purpose of a system. Layering and Object Oriented design are ways to structure systems. People involved in design, marketing, testing, development, and maintenaince must all have some understanding of what the system does. Both natural and special purpose languages can be used to represent system specifications. Special purpose languages may be textual or graphical, vary from informal to formal, and may or may not be supported by computer based tools. Special purpose languages include entity relationship diagrams, flow-charts, LOTOS, Z, VDM, SDL, data-flow diagrams, statecharts, Backnus Naur form, relational algebra, Petri nets, CCS, CSP, attribute grammars, IDL, first order predicate logic, graphs, charts, finite state automata, Horn clauses, ACTA, GDMO, and so on.

To encourage software reuse software specifications must be as complete as possible, and visable or available from the system. Using the above framework, "complete" and "visible" must be interpreted quite broadly. However, existing techniques and tools have typically focused on particular parts of this space. For example: Entity-Relationship diagrams are designed to capture the relationships amoung data elements in information models. Backus-Naur form is designed to describe the syntax of computer languages; attribute grammars to capture information needed to define semantics. IDL to describe interfaces in an open distributed system. CCS and CSP to formally model some types of concurrency from a process algebraic viewpoint. Flow

charts and graphical SDL represent the control logic of procedural languages, and typically ignore data representation. ACTA is a formal framework for describing and reasoning about the semantics of transaction models. Special purpose languages of this nature do not exist to describe information needed to charge for a components use in a distributed system, nor to describe such control notions as “who can do what to whom”, nor to describe various naming criteria and models, nor to describe work flows, nor responsibilities of one agent to another, and so on. Today, typically in a system specification some of its aspects are described by special purpose languages, others by natural languages, and still others are not described at all.

Some special purpose languages are supported by computer based tools. Often, individual tools are based on their own particular way of capturing and representing information needed by the targeted application. Thus, the heterogenity of both the application domains and tool implementations severely limits their interoperability. Hence, while these tools may effectively support a particular application in isolation, they cannot do so when it is one of many interdependent pieces of a complex evolving software system.

Several approaches exist to creating special purpose languages. One is to create a language specifically to suit the needs of an application, another is to use a mathematically precise language or a subset of one, and a third is to use a subset of a natural language and provide it with a more precise semantics.

Note: <Put in arguments here -- Rob do we really want this?> One could argue that mathematics is just a subset of natural languages, and so are application specific languages. <Rob, I’m digging a hole, can you see a way out?>

Mathematically precise special purpose languages, such as predicate logic, can be used to describe many aspects of a software system, but only by highly trained specialists. Because of their precision these languages provide a basis to carefully reason about about properties of the specified system. Thus, such specifications can be formally analyzed to determine whether or not they satisify particular criteria. Unfortunately, proving particular properties is tedious and computationally intensive, and the underlying proof theory is computationally intractable. To provide automated inference engines, subsets of these mathematical systems are designed for a variety of special purposes. Specialized front-ends are also added to make them accessible to people who are not highly trained specialists. (Horn clauses are a subset of predicate logic, and prolog is an assessible implementation of them -- I think, I’m checking with an expert on this.) Creating tools this way also leads to special purpose tools that don’t easily interoperate.

Natural languages are rich; they can be used to describe most aspects of most systems, including their parts and the relationships among these parts, at various levels of detail and from various points of view. People who know them find them easy to use. (They are easy to learn, provided one is young enough, but are quite difficult to learn later on.) They are often imprecise, and definitely computationally intractible. However, some specification tools are based on natural languages. Some specification techniques extract a somewhat semantically precise subset, and then support it with tools similar to those used to support natural language documents. Also, a natural language and a document control system could simply be used. This approach lacks the formality necessary to support sophisticated computer aided analysis. However, it requires the least specialized training to use. Unfortunately, as with other tools, interoperability is a major concern for this strategy as well.

Today, the most widely used tools are those based around paper; specification documents are prepared, and distributed using the existing paper based infrastructure. < Footnote here, this infrastructure includes, document preparation systems, document libraries, document control systems, cataloging systems, surface mail, FAX, education systems to train people to read natural languages, etc> Before, system specifications can constructed around a software paradigm, the necessary software infrastructure must be in place. Some peices of this infrastructure exist, but others are non-existant. Whatsmore creating a coherent usable infrastructure from these diverse peices will require considerable effort.

Object Oriented design is becoming quite popular. as it has developed a more comprehensive language for describing the organizational structure of a system. Perhaps the widest variety of description techniques exist for representing various aspects of software systems. Such techniques include data modeling/description techniques, control flow diagrams, concurrency control descriptions such as CSP and CCS, interface definition languages, languages to describe causality, diagrams or languages to represent timing information, etc

Note: <clean this up>. It is worth noting that

Many techniques exist for describing the behavior and environment of software systems. Approaches to software specification include a wide range of techniques, from the informal to the formal. One reasonable categorization is:

1. ad hoc discussions of issues written in a natural language,
2. systematic design development represented in a variety of diagrams together with natural language discussion, (DFD, EAR, SDL, etc -- operational sematics)
3. rigorous discriptions written in any number of specification languages, (Z, VDM, etc -- denotational sematics)
4. mathematically precise formal descriptions (horne clauses, predicate logic, etc -- axiomatic semantics)

Most software specification is done using (1) and (2), sometimes (3) and rarely (4). Typically, natural language descriptions require the least training to use, whereas the mathematically precise languages require extensive training. Unfortunately, the less mathematically precise description techniques do not lend themselve to machine assisted analysis, but the more mathematically precise enable a wide variety of machine assisted analysis techniques. (Note: Object-oriented methods exist for this entire range of methods, OO is a technique for organizing and structuring software systems not a language for describing them.)

5 Specification repositories

OMG CORBA introduces two repositories:

- an Interface Repository
- an Implementation Repository

The Interface Repository combines the visibility of interface definitions (in IDL) with distributed systems technology (the ORB). The CORBA specification indicates that the repository can be used at times other than run time. Opposition to the Interface Repository centres round its use at run-time, where stub code could be generated on the fly after consultation of the Interface Repository. The obvious problems of not being able to type check any interactions that may take place worries many people. The Interface Repository is also limited to storing IDL specifications. As the IDL is not sufficient to characterise a service, this must be seen as an important limitation.

The Implementation Repository is not well defined yet: there is no interface description and only a hint as to the possible contents of the repository. The CORBA specification and the two implementations we have seen (DEC ACAS and Iona Orbix) suggest it should be possible to store information *about* implementations. This may be, or include, configuration information, a pointer to binary code or a shell script, debugging information, other policy information about who may use the implementation etc. Lack of agreement in this area will be a handicap for some time to come.

6 Programming and (re)configuration tools

There are many tools which can assist in the production of software based products. In this chapter we will mainly consider the programming environment and the tools a programmer uses to write and construct applications. The software tools market shows a clear trend towards constructing software rather than writing it from scratch. The clearest indications come from object oriented programming languages. Most OO compilers now come with reasonably comprehensive class libraries. The resulting reuse of components remains limited to projects or departments, because current tools do not possess support for the distribution of the design and programming process on a wider scale.

The construction of services which consist of smaller components is thus becoming an increasingly important part of the programming task. We will refer to this constructive activity as configuration when it relates to the first time a particular structure is imposed. The term reconfiguration is used when an existing structure is altered to suit a new set of circumstances or requirements.

We distinguish two kinds of configuration:

- horizontal configuration or reconfiguration in which several application level functions or services are combined or recombined to form a more comprehensive function or service.
- vertical configuration or reconfiguration in which an application component is combined with mechanisms which ensure that the original application function or service is delivered with predefined or redefined properties or quality of service.

Horizontal (re)configuration is supported in OO languages, for instance when inheriting from two classes to form a new class. In most other programming environments there is less support, but if components are reused in some way we can consider that as a case of horizontal configuration. Advanced configuration management can automatically reconfigure a system in response to certain stimuli: a database may switch to a hot stand-by after the primary database fails to respond for instance.

Vertical (re)configuration is used to combine application code with engineering mechanisms which deliver some required properties. Some of these engineering mechanisms are referred to as transparencies or transparency mechansisms. In ANSA Phase III, we are concerned with transparencies for performance and dependability. The work on performance and dependability [refs] is primarily concerned with:

- the statement of performance and dependability properties
- the identification (and where necessary construction and illustration) of a library of engineering mechanisms

- the derivation of a set of rules which give guidance to vertical configuration (which engineering mechansism should be combined to achieve particular application properties).

The library of engineering mechanisms and the rules for their composition together form the engineering model. Desirable application properties can be stated in different ways. This is an important aspect of programming models. Tools must be employed to help construct applications with specified properties. Such tools must be able to:

- interpret statements of required application properties
- have knowledge of libraries of engineering components
- interpret rules for engineering mechansism composition

This chapter looks at existing tools to find out whether they can be used for (re)configuration of application and engineering components in distributed heterogeneous computer systems. Because both engineering and programming models for performance and dependability are still under consideration, the examination of existing tools and tooling environments is based on trends, rather than specific product capabilities.

6.1 Configuration epochs

The ability to modify system configuration is very important if the resulting system and services are to flexibly respond to new demands which will constantly be placed on them. This points to late configuration, an extension of late binding. As a general rule, the later the configuration takes place:

- the larger the flexibility of the resulting system
- the more reusable will the components need to be if they are to be employed in many different configurations
- the less impact on the language in which the components are specified
- the more need for externalised configuration information
- the more complicated the tools

The configuration epoch is intimately related to the way in which the configuration is specified. Three epochs for system configuration can be distinguished: programming time, link time, and run time configuration.

6.1.1 Programming time configuration

Early configuration leads to configuration information being embedded in the programming language. Programming languages have excellent facilities for statically specifiying horizontal configurations in the small (e.g. function calls within a module). The support for vertical configuration is very limited with programmers often having to deal with low level system code to give their applications the correct performance and dependability properties. This task is becoming far too complicated in distributed systems and tool support has been researched extensively [refs: ansa, comandos, amadeus, etc].

The following is a taxonomy of ways in which a programmer can specify application properties at programming time and at source code level:

- programmer specifies by inserting appropriate code

- programmer declares requirements using macro-statements; a preprocessor inserts appropriate code [ANSAware]
- programmer declares requirements using *attributes*; extended compiler inserts appropriate object code (code inserts, library calls) [AMADEUS]
- programmer uses specialised tightly scoped programming language; scope boundaries can be used as *distribution guidelines*; attributes are used for application properties [DPL]

Use of a specialised tightly scoped language has the added advantage that tools can analyse application semantics. This is necessary for atomicity transparency for instance [AR.004]. Introducing a “new” language can prove very difficult however.

Configurations which are fixed at programming time can only be reconfigured by reprogramming and recompiling. Fixing configurations early leads to more efficient but less flexible implementations. One way to afford more flexibility is to build facilities into the program or to add it by means of tools (e.g. the addition of stubs and run-time binding facilities allows changes of bindings at run time).

6.1.2 Link time configuration

Configuration at link time is most common. The programmer sets particular link options to direct the linker to select appropriate software modules and libraries. This information is often lost after linking, although it may be retained in makefiles or other compiler/linker instruction files. The run time component does not carry any configuration information with it, so that any maintenance can be quite hard.

Configuring a system at this time allows the inclusion of components whose implementation cannot be controlled by the programmer (other than by selecting a different library) but whose specification has to be well known (to have passed the compiler stage).

Reconfiguration is achieved by re-linking with fresh linking instructions. A new implementation results. If it is to replace an existing implementation, then extra mechanisms are needed to deal with existing users if service is to be provided continuously (similar arguments apply to configurations fixed at programming time).

6.1.3 Run time configuration

Most flexibility is achieved when systems and services can be reconfigured at run time. For this to be possible, configuration information needs to be present in the run time system.

One important consideration is that the overall functionality of the system or service remains available whilst service components are modified or exchanged for others. It is therefore not possible to bring down the service or system for recompilation and/or relinking with replacement components.

Reconfiguration decisions can be taken in the system itself, in response to certain measurements for instance. Two components which are exchanging large amounts of data with timing restrictions may measure the throughput of the underlying network and if it falls below a certain limit open up another channel perhaps over another network, in order to preserve service properties refered to in the contract of cooperation.

Expert systems can be employed to implement the configuration rules and conditions. Measurements made in the system can be used as triggers for reconfiguration.

Several mechanisms which influence configuration at run time are already in place:

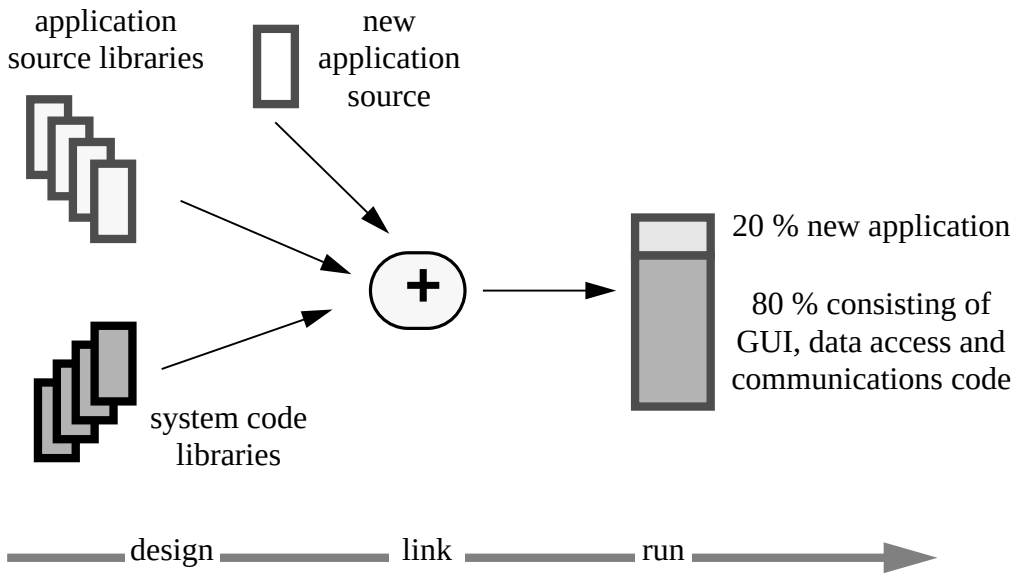
- trading policies determine service selection (horizontal configuration)
- binding information determines the calling of appropriate stub code (interworking vertical configuration)
- factories read from templates (configuration files)
- object management implements configuration with respect to underlying rsources (passivation and activation)
- relocators can cope with moving objects (but we have no mechanisms to move them yet).

We propose to take a fresh look at these mechansism from the point of system management and configuration. The relationship between trading, binding, factories, object management and relocation on the one hand and system configuration tools which enable the “management engine” on the other must also be examined.

6.1.4 Tooling perspective

It is generally known that no more than 20% of an application consists of code which is directly related to solving a particular problem or providing a particular functionality desired by an end-user. The other 80% is concerned with complexities which are independent of the particular application in hand. Figure 6.1 illustrates how a traditional tool chain links application sources and system code from libraries to deliver object code.

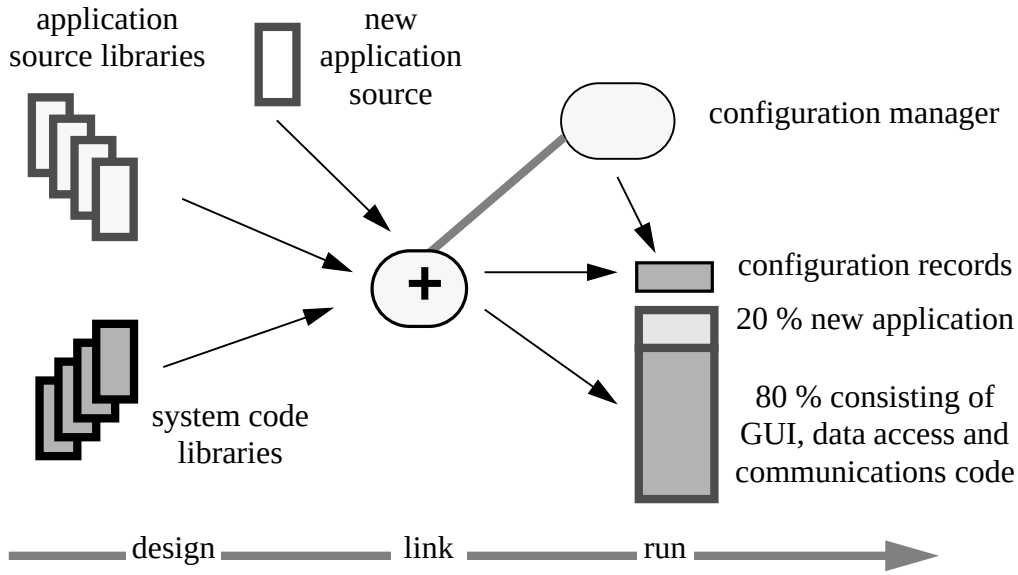
Figure 6.1: traditional tool chain



Examples of “systems code” are code for inter-process communication, to drive a graphical user interface (GUI), code to access data in a database or file system, code to deal with different levels of functionality offered by operating systems, code to interface devices, such as printers, hand-held scanners, etc. A common characteristic is that such code is not owned or controlled in any way by the application programmer who merely uses it.

When run time (re)configuration is desired, it is necessary to deliver the configuration records for use at run time. In addition there is a need for a configuration manager that can act on both the configuration records and the application which it describes. Figure 6.2 illustrates the new arrangement as an extension of the traditional tool chain.

Figure 6.2: run time reconfiguration



The configuration records must specify the components (or refer to such specifications) and include information on the relationships between the components. Most tool chains already have a shared internal representation for such information. It is never made available in the runtime system for access by runtime tools.

6.2 Type checking and binding

Two ANSA distributed systems principles are of particular relevance to configuration:

1. check types of interacting components as early as possible
2. bind interacting components as late as possible

Both have to take place before interaction between the components takes place. The principles are traditionally made in the context of horizontal configuration. The also apply in vertical configuration [ref. dependability work on expectation matching]

6.2.1 Type checking

Early type checking is accepted as good practice in most programming environments. There is a trend to use expert systems to assist not only in syntax directed editing but also in type checking as the program is constructed by the software engineer.

Type checking can only be performed within a particular predefined context. For instance, the context in which a service or function will be used needs to be known before a type checker can indicate that interaction errors at runtime are unlikely. Reusable components are written without knowledge of the precise context in which they will be employed. The type check will therefore have to be postponed until the context is known. This is often not until just before deployment, when the final configuration is determined. When reconfiguration is required, new components may replace old ones. If reconfiguration takes place in the live system then compile time type checking is not appropriate. The asserted type relationships must now be seen to impose a constraint on the possible configurations. Configuration tools in this epoch are directed by constraint specifications. We look in the direction of rule and constraint based programming [Giariatano ..] for tool support at this stage.

A problem with the approach to type checking within an AST [ref. ANSA DPL] is that the context of the type check is limited to the AST itself. It can never deal with components which are supplied by others, which are under the control of other design authorities, and which may have been written in a different language. Type checking within an AST should not be dismissed, but seen as tackling part of the problem of type safe binding.

Note: <Beyond here things look more ropey: more ike a dump of text with little editing being done. Some of the text should be stripped down to concentrate on making the points.>

6.2.2 Late binding

Note: text from section 2.2 in APM.1058:

The idea of not binding application specific code and “system” code until configuration (or system building) time clearly encourages re-use and increases productivity.

Advanced application development systems allow application development staff to specify the application specific aspects in detail, whilst tools automatically deal with the systems related aspects. For the GUI, for instance, a visual prototyping environment is provided, within which the right interface can be constructed from pre-defined components, by pointing, clicking, dragging and dropping. Binding to a particular GUI environment is only done just before system deployment. The same approach is adopted for database management systems. A range of database management system interfaces are provided and will be linked in once it has been decided which database management system will be used in a particular configuration.

In all cases binding occurs at deployment time. ANSA allows binding of communications resources to occur even even later and dynamically.

6.3 Rule and constraint based programming

Advanced tool sets all come with an expert system of some sort.

Note: text from 2.4 of APM.1058:

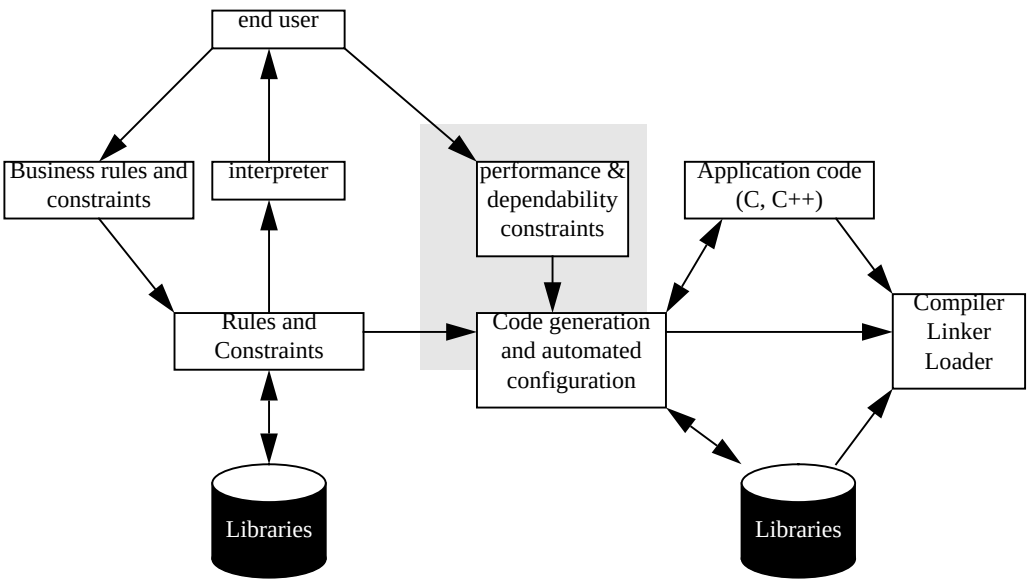
6.3.0.1 *Reduce (eliminate) 3rd generation programming effort*

It is accepted that productivity gains can be achieved by programming in a 4th generation programming language rather than a 3GL. Many of the tools which we examined offer the opportunity to program in a rule based environment supported by an expert system (Art*Enterprise, Magic CAPtm, OOA/RD, Ptech, SNAP). This offers three important advantages:

- programming proceeds in close cooperation with the problem owner, who sees his problem translated in recognisable and comprehensible rules in the system;
- rapid prototyping, interspersed with many consultations with the end-users, is a reality, resulting in more satisfactory solutions;
- tools are used to turn rule based specifications into imperative programs, which can then be compiled to achieve specific performance characteristics, delivering results more quickly.

Figure 6.3 illustrates the rapid prototyping cycle supported by rule and constraint based programming. It also shows the back end through which more efficient code can be generated. We suggest that some of the rules can relate to the required quality of service aspects of an application or service. The grey area in the figure is not found in any existing tools, and would represent new work.

Figure 6.3: Advanced Tool Environment



6.4 Interoperability and federation

Note: Issues: Tools presentation slide 13; text from APM.1058:

From the tool user perspective, tool diversification offers increased choice but of course has a downside: where multiple tools are in use the usual and inevitable questions arise. This time it is about multiple tool sets in a distributed environment. Three classes of problems can arise. The first two are related to the realisation that in large systems, the design process is itself a distributed application, possibly involving many tools. The last issue is related

to interworking problems between the systems which are created using different tools.

6.4.1 The tool as a distributed application

For the problems associated with the way a tool is implemented in a distributed environment we can use the Architecture to suggest design choices: it is as if the tool is an application which is to be distributed. This can ensure that the tool is portable, can be distributed, meets certain reliability requirements, scales, can be configured etc.

ANSA need not address this issue as special since it is covered by our concern about system integration and federation in general. However, it is clear that all of the tools we looked at have limitations in terms of the size of the design they can manage and the extent to which the tool can be used in a large distributed design process.

6.4.2 Exchanging designs between tools

Issues relating to exchanging designs between tools are more akin to multi-database problems, where schema integration / conversion is appropriate. The design itself is a complex of information held by the design tool. The information is organised according to the "data model" which the tool builder considers the heart of the tool (and that which he believes gives it the competitive edge). The reluctance to be open about this leads to issues which are are perhaps more difficult to sort out. Specific questions include:

- can a design developed in one tool be further developed in another?
- is there any consistency checking of my design across multiple tools which all support control integration (clicking on an element in one tool opens a window in another tool)
- what if my company who use tool set A merge with a company who use tool set B?
- what if I want to rationalise the use of tools in my company?

In ANSA we advocated addressing this problem by choosing CORBA IDL as the common interface definition language at the scenario implementation level. Where multiple IDL's are in use we suggest the use of Abstract Data Types (ADT) [TR.042]. By adopting a three stage process, consisting of a front end (FE), an abstract syntax tree (AST) and a back end (BE) we hope that mappings between other tools which adopt the same approach will be simplified.

There is little evidence however, that the tools market is either adopting IDL or ADT style service definitions. Current work in the area of federation further suggests that the information carried by an IDL is insufficient to decide whether a service will meet a particular need. The use of an AST as the "internal" representation of a design is common in principle. Of course many such ASTs exist: they are often secret as they are thought to give market edge.

6.4.3 Interworking between systems which are built with different tools

We have to recognise that the tool which assists designers build distributed applications will transparently insert engineering mechanisms to make the distribution come true. The nature of the distribution technology is hidden from the application programmer. This is deemed to be "a good thing" because

it releases the programmer from having to worry about it. A tool builder may not support many engineering functions or options. In fact it is likely that most tools will initially be brought to market with just one fixed set (e.g. a tool which can generate applications on DCE 1.01 and OSF/1[1], c.f. SNAP on IBM RS6000).

The hidden problem is that application programmers no longer have any sight of the “engineering” and therefore cannot solve interoperability problems themselves. If the market behaves properly, then there may well be a push for common interconnection engineering or engineering which can cope with multiple technologies. This will not happen in the short term: it will take time for the end-user or tool user to realise what are the significance of engineering issues. The Architecture can be used to point out the issues early rather than later and hopefully induce a willingness to agree before it is too late.

6.4.4 Are you ready for automation?

There have been several reports of organisations who have introduced CASE tools to find a reduction in productivity of their staff and a reduction of the quality of the software they produce [INFO 93]. CASE users say that without careful preparation you end up with “an automated mess”, a “faster disaster”, or “paralysis by analysis”. The preparations concern the human and organisational dimensions within which the tools will be deployed.

Most complaints are said to come from CASE tools which encompass the analysis and design activities. In many cases it was the inflexibility of the tools with respect to methodology enforcement which affronted and insulted the users. The design of a class library and its use are of crucial importance to the success of object technology. Productivity gains were reported more often from so called programming workbenches.

1. Current support for distribution is however typically limited to inclusion of drivers for TCP/IP for instance.

7 The way forward

The previous chapters suggested what technology can be used to enable reuse of service specifications and implementations. The extent to which service specifications and implementations will *actually* be reused (be “re-useful” [Adams 93]) will depend on the incentives which derive from reuse. The creation of the right incentives are outside the scope of the project. This chapter therefore focusses on the creation of the proper technical environment in which

- service *use* can be charged for
- service *reuse* is the rule rather than the exception
- service flexibility is achieved by *(re)configuration*.

7.1 Use based charging

Use based charging needs to protect all parties involved:

1. investigate security sealing mechanisms which will prevent software components be tampered with

   Whoever receives a copy of the component must not be able to switch off the charging facility or change the functionality. It must also be protected from tampering at run time. The strength of the encapsulation is important and mechanisms in the security domain can help achieve the sealing capabilities.

2. investigate what provisions are needed to allow charging for service use in a distributed, heterogeneous environment

   Distributed systems technologies can be employed to convey both charging and payment information. The location transparency of the charging information source with respect to the billing authority should be further examined:

   (i) a copied component, in a new environment must be able to find out where to send charging information

   (ii) the charging agent needs to know where to send the bill or where to obtain (possibly electronic) payment in pay as you go schemes.

   The policy which a service provider adheres to is internal to the provider and must be protected. For instance:

   (i) service may be witheld until a charging address is known

   (ii) a diminished service may be offered until a charging address is known.

7.2 Service visibility and accessibility

We feel that the facilities offered by current distributed systems technologies should be combined and integrated with facilities which support the exchange of more comprehensive service specification information. This exchange should be possible at any time during the evolution of a service in a large system. Therefore, it seems obvious at this point to:

3. express the provision of service specifications as a service
4. use ANSA (CORBA) technology to support the distribution of this service (the service specification repository) as any other service
5. use existing database technology to implement the distributed repository and examine how ANSA principles can be used to achieve the distribution
6. decide WHAT information should comprise the service specification (the logical information model)
7. examine HOW this information is to be represented
8. determine WHO or WHAT should use the information and WHEN
9. examine how to deal with differences in the semantics and syntax of the information (federation issues)
10. advise OMG of our findings w.r.t. multiple object and interface repositories
11. determine what levels of compatibility checking can be achieved with the information on service specifications in their potentially many forms
12. examine what tools can be provided to automate the creation of applications in the distributed system and how these tools would use the information in the service specification repository (automated transparency issues)

This agenda of work clearly links the activities of the automated transparency group and the federation topic group. It also provides a test bed for the integration of database technologies.

In deciding the WHAT, HOW, WHO and WHEN questions w.r.t. the service specification repository, we need input from the work on programming models in the dependability and performance topic groups. I would expect them to start examining appropriate programming strategies, rather than details of how to extend Orbix with attributes for example. The current work on transactions is already taking a strategic approach.

7.3 Tool chains

AST with nodes as ADT solves part of the problem; does not cover reuse issues; this needs investigation

Extend the Computational Model AST to include QoS issues? Can it be done?

Need for several AST like mechansisms & at differnet levels of abstraction? How to map designs; many questions!

8 remaining text

Note: The text in this section has been cut from a previous version of the paper.

8.0.1 Technical enablers

A complex system is modelled as a set of components which offer services to one another. Systems are reconfigured by renegotiating service contracts. Good services will be (re)used. Bad services will be ignored. This looks like a desirable market paradigm. Unfortunately, software component producers cannot yet economically benefit from the use which is made of products which are made available in an open distributed system. Unless there is a way in which providers can charge?? users, there will be no service provision

Making charging distribution transparent. That is, being able to charge for a service independently from where the service is delivered and where the service provider resides.

Moving service providers around a large heterogeneous distributed system will then become easier. This will encourage the reuse of components and their reconfiguration.

There is a clear market structure: the good stuff will get paid for andpeople select from what is known (encouraging advertising). For example we would pay a tiny sum to the writer of the postscript filters we use every time we print a document or we would switch to a different package on economic grounds.

8.0.2 Context

Rather than providing a few simple standardised interconnection services to all customers, telecommunications service providers now have to provide many complex specialised services to small groups or individuals. In a fiercely competitive market, customers demand:

- the latest personalised service, e.g. video on demand
- service provision at competitive rates, e.g. similar to video cassette rental
- control over the service configuration, e.g. no X-rated movies for kids
- some or full integration amongst services offered by different suppliers, e.g. access to several video libraries, using the same controller.

Providing and evolving often highly personalised services, at competitive rates, in a widespread heterogeneous environment will squeeze profit margins, and reduce investment in new technologies and procedures.

The services we concentrate on in this paper consist of two main components. Telecommunications equipment is dominant in the delivery of services to customers. Computing equipment is dominant in service support and provisioning, required by the telecommunications service provider.

Note: We could add the willingness to pay picture here and explain that the software component is on the increase. As a result we ought to think about how to deliver that component more cheaply. (see TC presentation slide 4)

Competitive advantage may be gained by those who adopt an efficient and consistent strategy for service deployment and maintenance; a strategy which assigns equal importance to the telecommunications and computing components involved in the provision of advanced information and multi-media services.

Note: Rob's old text:

Large distributed systems consist of many interacting components. The relationship amongst them can be characterised as components providing services to one another via interfaces.

The benefits of properly specified interfaces is well understood: from small well defined components, more complex components can be constructed, whilst implementation detail remains hidden behind component interfaces.

Generally, development of new services needs to take place in the context of what already exists. Adding to what exists rather than starting afresh results in increased productivity and better integration between new and old.

We believe that there are three necessary conditions which enable and encourage people to build on things which are already provided:

- service specifications should be visible
- service implementations should be accessible
- the economic circumstances must be such that use of existing specifications and implementations inevitably becomes good practice.

8.0.3 What is a service?

In ANSA we have used the concept of service to be synonymous with some functionality which can be provided by an object. The way in which the service can be obtained and is delivered is governed by the ANSA computational model, which prescribes a particular style of interaction between service providers and consumers. A type signature characterises the allowable interactions. (Amongst all existing distributed computing environments - ANSA, DCE, CORBA - ANSA does the most to negotiate compatibility between service providers and users.)

It is now clear that the traditional type signature and connection/binding information is an insufficient basis for negotiation and that many other aspects of a service also play a role. The contractual relationship between provider and consumer, the quality of the service, and the way in which the consumer may be billed for its use of the service are examples of such extensions.

Note: I would like to add the picture here, which describes the relationship between telecomms and sftware components (see TC presentation slide 5)

8.0.4 The ANSA Phase III scenario

The scenario fits in with the agenda set out above, as we are clearly working towards a situation in which development systems remains linked to the operational systems, to perform maintenance and configuration tasks, and to evolve the system.

try a mapping from boxes to micro scenarios and vv.

Note: Old text:

We will examine the existing economic structure in which software is provisioned as a commodity. We will analyse what aspects of the current economic situation inhibit reuse.

9 BPR and CE paper

Note: This chapter holds all the text from previous attempts at putting BPR and CE stuff in with the tools, reuse and config. mgmt stuff. It has been dumped here so a separate paper can be brewed out of this.

BPR points to be made:

- producing software is a business process
- lead times are too long
- producing from scratch (innovation) but using existing components
- producing copies is not what we address
- waterfall model has too many hand-overs
- Bellcore example: Jane's story of adding a feature in the phone system

The saviour: Business Process Reengineering applied to the software production process:

- reduce handovers
- build from what you have (restructure)
- OO helps:
- class libraries
- rapid prototyping
- incremental development: growing the software

also see [First Class Nov '93]

Can we tie in the tools paper?
It will have to be re-worked:
Tools are needed to support the software production process. Many existing tools are supporting the "old" production model. What is needed for the "new" model?

Other things:

DFD is based on the transformation paradigm; has many hand-overs

9.1 BPR stuff from previous attempt:

Note: The stuff below is all mixed up with CE and reuse as well.

Note: Jane's first para with a few edits:

Functionality in large distributed systems is typically enhanced by adding new software components to an installed base. A large system is rarely written afresh. Often, several new components are being created concurrently, by independent developers. In such large systems, the boundary between what is new development and what is maintenance or reconfiguration is blurring.

Note: the previous paragraph touches on:
concurrent engineering
life cycle of s/w component within a system with a much longer life span
It does not yet pick up reuse

Adding software to an installed base often is a risky exercise. The software components in the installed base are generally ill defined. This can lead to the addition of components which perform the same or a very similar task to an already installed component. Because the system can also be large it is not generally known by any one individual. In most cases, configuration information is not an integral part of the system either. The complexity of the task of enhancing or changing an existing installation requires tool support.

Note: Jane's second para:

Unfortunately, today's systems and development strategies do not ease exploitation of existing components for integration into large scale systems. It is difficult, or not possible to find out what already exists. It is next to impossible to determine what is being created by others. It is often unclear just what a component does ordoes not do, just how to access its functionality, and just what side-effects it can create. Hence, it is often unfeasible to combine components other's have built.

Note: Other things I want in:
reengineering the s/w development process
Object orientation as the enabler (incremental development, prototyping)
current methodologies as the show stopper (too many hand-overs)
process/data split and DFD is transformational paradigm stuff (more hand-overs!!)

Note: And then we can say what technologies are appropriate:
we have already got them &they need combining
OO
classes,
browsers
distribution/interworking
database technology
compilers, linkers, loaders, libraries, interpretors, trading and binding technologies
etc. etc.

Note: Then we need to touch on the legal/economical framework:
bill/charge for use not ownership brings technical issues
most are solved:
security aspects to enable encapsulation
operating systems which enforce separation (no side effects)
mechanisms for accounting and billing (charging if you like) in distr. system

9.2 Concurrent engineering

Concurrent engineering is an emerging discipline which uses computer technology in support of distributed design. E.g. in motor car design several parts are engineeried concurrently and checkpoints are made to ensure the whole thing will fit together at the end. By making design and engineering steps concurrent you can obviously speed up the engineering of large complex things.

Concurrent engineering has been applied to the software design process (see IEEE Computer Jan '93).

It is obvious that the issues of concurrency and distribution which we are experts on emerge in this field too, but more application driven than technology driven.

CE is fast becoming an important application area for distributed computing.

Note: Chris Mayers writes:

If you are interested in finding out more, you might like to borrow “Concurrent Engineering: The Product Development Environment for the 1990s”.

This book is 'sponsored' by Mentor Graphics. But I found it a handy introduction; and it has good coverage of process and organizaation issues.

9.2.1 Feature interaction

Once there are mechanisms which can manipulate service specifications, it may be possible to find mechanisms which can determine the kinds of object incompatibility which lead a class of problems known as “feature interaction”.

References

[Cox 93]

Brad Cox on Developing Software for Large-Scale Reuse, OOPSLA 93, p142-143

[Mori 90]

R. Mori, M. Kawahara, “Superdistribution: The Concept and the Architecture”, Transactions of the IEICE; Vol. E-73#7 July 1990; Special Issue on Cryptography and Information Security.

[Hammer 93]

M. Hammer and J. Champy, “Reengineering the Corporation”, Harper Business, Harper Collins Publishers, New York, 1993.

Note: I have not got [Mori 90]. Can anyone help me here?