UNIVERSITY OF NEWCASTLE UPON TYNE

Version Control in Engineering Design Databases

A Thesis Submitted for the Degree of

Doctor of Philosophy

February 2001

Barry Florida-James

Engineering Design Centre

Faculty of Engineering

Abstract

This thesis is concerned with lifecycle data support for the design of large made to order products. These products have so many complex functions to perform that no one designer will have all of the relevant skills such as in structural design or electrical engineering to produce a comprehensive design. This therefore leads to the utilisation of a team of designers who will not only fulfil logically different design roles but often work at different physical locations.

In such a design environment there may be several local models, represented in local databases.These databases may or may not support versioning either of the data or of the schema which evolves as the product design grows. The interfaces to these databases will be varied as they are intended to suit the local needs of the design agent. This thesis proposes a model for version control in a design environment which does not alter the designers existing view.

A system of distributed co-operating agents is presented whose goal is to manage change and organise version sets in an engineering environment. The agents are designed for full lifecycle support and inter-operation across heterogeneous networks. The agent communication is based on CORBA but an extra messaging layer is developed which utilises a language built in VDM-SL (Vienna Development Method - Specification Language). A version model is presented in two ways informally based on the assumptions on a general design process and formally in VDM-SL.

In order to demonstrate the effectiveness of the version model, two industrial case studies are presented. The first of these is a study of offshore process engineering. The second is a study of conceptual ship design.

Acknowledgements

Firstly I would like to thank my supervisors, Dr. Nick Rossiter at the Department of Computing Science and Professor Bill Hills at the Engineering Design Centre, Newcastle University for their help and advice throughout my research. Sincere thanks also to Dr. Marin Guenov, whose initial guidance was invaluable. Dr. Kuo-Ming Chao, "Danny" and Arthur Guinness for their wisdom and constant support during difficult times.

AMEC Process and Energy Limited for supplying the offshore case study and thanks also to Armstrong Technology Associates, especially Mr. Keith Hutchinson without whose effort the fast ferry case study would have sunk.

I would like to acknowledge the support of the Engineering and Physical Sciences Research Council (EPSRC Grant No. GR/J40272) and the University of Newcastle , for providing funding for this work.

I would also like to thank all of the staff, past and present, at the Engineering Design Centre, in particular Dr. Leo McAlinden, Dr. Peter Norman, Ms. Carol Mee, Ms. M. Hynes, Mr. Chris Forker, Dr. Brent King and Professor Ian Ritchey.

A special thanks to my family espescially my mum and Anthony for reminding me what the really important things in life are. And finally, I would like to thank my wife, Susan, for being the most important thing in my life.

Contents

1 Introduction

Design has often been considered more as a practice and sometimes as an art rather than a science. It was not before this century that design methodologies started to emerge. Not withstanding the development of Information Technology (IT) and Computer Aided Engineering (CAE) most of the methodological problems still remain. Not only this but new problems associated with the communication and cooperation of distributed design agents (Computer Aided Design (CAD) tools and/or human designers) have emerged.

In this thesis it is suggested that an improvement to the design process can be made by better organisation of the data that underpins this process. In order to develop a fundamental understanding of design as a process the following section presents various models of the design process. Subsequently one of the models presented is examined in more detail.

1.1 The Engineering Design Process

There are three main types of design process according to Finger and Dixon (1989) and Evboumwan (1996), namely descriptive models, prescriptive models and computer-based models. Descriptive models are concerned with the actions and activities of the designer and rely on previous experience and knowledge. French's design process can be viewed as a descriptive model (French, 1999). Prescriptive models are described as being concerned with systematic procedural steps which prescribe how the design process should be carried out. The most well known advocates of prescriptive models are Pahl and Beitz (1996). Computer-based models make use of numerical and computational techniques combined with computing technologies. An example of a computer-based design process is that described by Medland (1992).

A comprehensive design system is identified as being able to support various facets including (i) the evolutionary process of design, (ii) the knowledge-based/exploratory aspects of design, (iii) the investigative and search aspects of the design process, (iv) the creative process in design, (v) the logical reasoning process involved in design, (vi) the iterative and interactive process involved in design, (vii) decision making based on judgement, and (viii) the mathematical analysis and computational simulation

processes performed during design (Evbuomwan, 1996).

The ideal design process involves a systematic approach which allows the maximum scope for innovative flare and takes full account of new technologies and changing social trends (Hawkes, 1984). Essential aspects of a systematic approach are identified as (i) the establishment of the primary need for a new product, (ii) detailed design specifications, (iii) logical thought processes, and (iv) the evaluation of a variety of possible solutions to the problem.

Ray (1985) describes the morphology of the design process and the anatomy of the design process. The morphology approach is concerned with the examination of the product life cycle and the definition of each stage involved, namely (i) identifying the problem, (ii) feasibility study, (iii) preliminary design, (iv) detailed design, (v) production, (vi) distribution, and (vii) obsolescence. The anatomy of the design process involves the examination of the design process with reference to the designer's actions, from the initial evaluation to the final solution. Four steps are named as being involved in anatomy of the design process, namely (i) identifying the problem and evaluating the need, (ii) information retrieval and assessment, (iii) evaluating the alternatives, and (iv) communication and implementation.

Ertas (1993) refers to the definition of engineering design given by the Accreditation Board for Engineering and Technology (ABET, 1989) which indicates that the fundamental elements of the engineering design process are the establishment of the objectives and criteria, synthesis, analysis, construction, testing and evaluation. Ertas then proceeds to identify the steps of the engineering design process as the (i) recognition of need, (ii) conceptualisation and creativity, (iii) feasibility assessment, (iv) organisational/work breakdown structure, (v) preliminary design, (vi) detailed design, (vii) production planning and tooling design, production, and (viii) acceptance testing.

Pahl and Beitz (1996) indicate that the engineering design process must be planned carefully and executed systematically. In addition, the design process must be decomposed into phases and then steps, each with its own working methods. The design process is divided into four main phases, namely (i) product planning and clarifying the task, (ii) conceptual design, (iii) embodiment design, and (iv) detail

design.

A five level hierarchical arrangement of the structural parts of the design process is presented by Hubka (1982). These levels are (i) design stages, (ii) design operations, (iii) basic operations, (iv) elementary activities, (v) elementary operations. The design stage is divided into the main areas of conceptual design, layout design, and detail design. The structural parts of the design process identified by Hubka are said to assist in the construction of a procedural model, which is illustrated as a spiral moving through the various stages of design, namely (i) problem assignment, (ii) design specification, (iii) functional structure, (iv) concept, (v) preliminary layout, (vi) dimensional layout, and (vii) detail and assembly drawings.

In the following sections we examine in detail the Pahl and Beitz model of design.

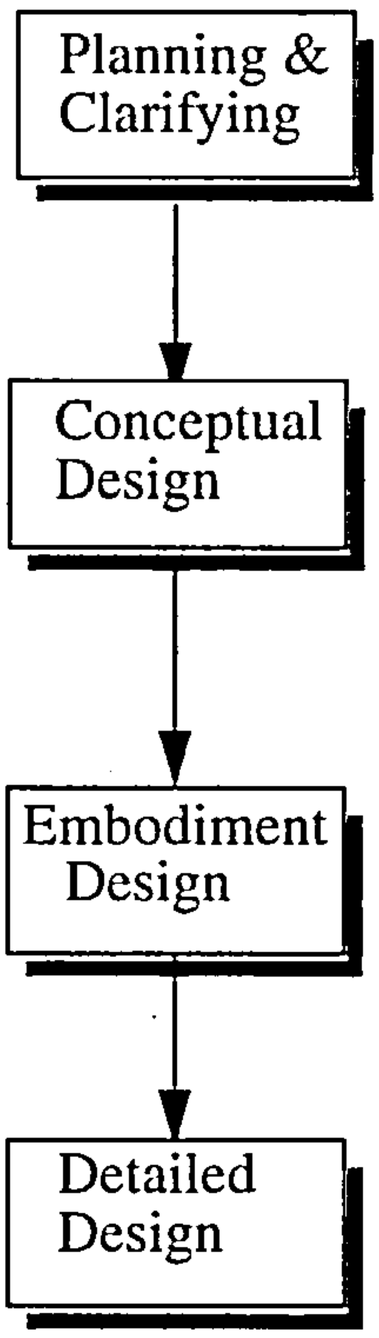


Figure 1.1 The Design Process

1.1.1 Planning & Clarifying the Task

The initial stage, also referred to by Dym (1994) as *preliminary design,* is an attempt to define the design problem so that any problems can be formulated into explicit specifications. The following stages may require further considerations in order to

facilitate the design. The main task in this stage is to process requirements obtained from the clients or engineers. The activities of this stage entail the collection of information and constraints. The diverse requirements may be described in symbolic and/or numerical formats. Thus, the input to this stage of design may be an imprecisely defined specification. The output of this stage could be a construction of a problem domain and a detailed specification.

1.1.2 Conceptual Design

The conceptual phase requires the designer to take only some key requirements and constraints into consideration in order to propose an initial state for design. The general framework for a solution to the original problem is determined. This stage, based on elaborated specifications, should establish functional structures and search for suitable solution principles. The functional structures and solution principles are combined into structural concepts by decomposing functional entities into subentities of decreasing complexity. This division of functional entities must continue until the search for a solution seems promising. The space for the functional entities can be mapped directly onto the space for the structural entities.

1.1.3 Embodiment Design

Embodiment design focuses on the transformation from qualitative solutions to quantitative solutions. The input for this stage is the general solution realised at the conceptual design phase. Embodiment design results in the specification of a layout. The layout may be further developed and components instantiated under technical and economic constraints. Within this stage, the design proposals can be evaluated by using a number of analytical tools. For instance, behavioural performance can be simulated and analysed or financial viability can be assessed.

1.1.4 Detailed Design

In the detailed design phase, the final solution to the design problem is generated. The design determines the overall arrangement which includes the forms, dimensions and surface properties of all the individual components. The materials are also specified.

1.2 Contemporary design environments

In the design phase of large Made-To-Order (MTO) products, recent attempts towards

a greater integration of distributed design agents, has resulted in some progress in tool and data integration. However, the ideal of total concurrent design activity, supported by CAD and other computer based design systems, has not yet been realised. Whilst it is common for designers to have access to a central repository. It is unusual for this central repository to be optimised for performance and complete lifecycle support. In this thesis the view of Cutcosky et al. (1993) is supported. That is, a centralised product model is logically the ideal framework for a more cohesive design process but it does present potential problem when data is physically stored in one location. Even if this system is itself distributed, it may require a huge investment in software and hardware, not to mention human resources, to replace existing systems which currently perform their design functions adequately.

Figure 1.2 below was taken from AMEC Process & Energy Ltd.

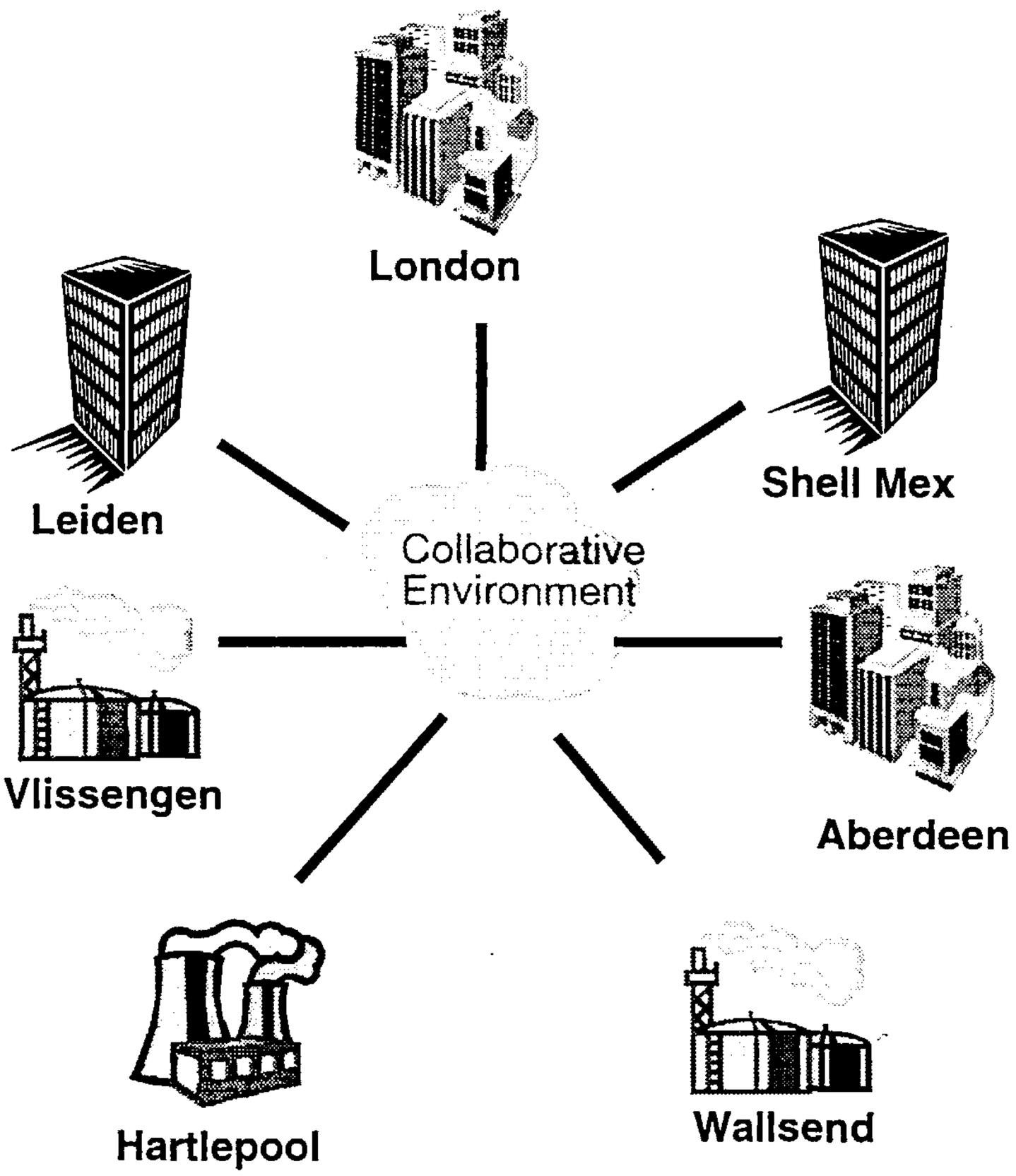


Figure 1.2 Made-To-Order Design Scenario

Figure 1.2 illustrates many aspects of the nature of Made-To-Order product design. The main design office is in London, with some teams based in Leiden, Aberdeen and

another company's London office (Shell Mex). Fabrication of the component parts of the design occurs at three sites, Vlissengen, Hartlepool and Wallsend. The final product is assembled at Wallsend. Product models therefore need to be shared throughout these sites, across geographical and organisational boundaries. The problems associated with this type of distributed information access will be expanded upon in more detail in Section 3.1.

An outcome of this work is that lessons learned from software engineering, where encapsulation, information hiding and data abstraction are keys to the success of a modern software project, should be applied to engineering. Information is shared at appropriately defined or engineered levels, not as *ad hoc* data transactions. Thus designers in traditional engineering disciplines should be hidden from the detail of their collaborative partners and aided in understanding the meaning of the data in the global and, by implication, their local context. In this thesis a system for global configuration and version control is presented which allows designers a consistent view throughout the product life cycle so that they may retain their own product representation and design tools. Integration is achieved dynamically and incrementally with minimum overhead rather than defining a system with static interfaces and parameters. The system is allowed to expand with the product as the requirements of the product lifecycle demand without imposing strict constraints on the base design tools. Such an approach is often the only viable method in a competitive market place.

1.3 The need for a product data model

Real-world engineering design projects require the co-operation of multidisciplinary design teams using a number of sophisticated design and powerful engineering design tools. These tools are styled to aid the particular designer in his or her area of expertise. Hence they contain detailed, domain-specific product data. The individuals or the individual groups of the multidisciplinary design teams work in parallel and independently often for lengthy periods of time. Also, as stated, strategic alliances are forcing designers from more and more remote geographical locations to collaborate on projects. Besides this, at any instant, individual members may be working on different versions of a design and viewing the design from various perspectives, at various levels of detail. In order to co-ordinate the design activities of the various groups, and to guarantee a good co-operation among the different engineering tools, an environment

is required where product data is easily accessible by all participating design agents. Not only must product data be stored amongst varied design agents but updates to this data must be handled and related to all design agents simultaneously.

The product data model is essential to any designer. A model is simply a representation of the product which simulates the required attributes of the product. Therefore a model built from plasticine used to demonstrate a product's physical appearance will not suffice to demonstrate its load bearing properties. This is the essence of the problem: how do we relate a model, designed to demonstrate one attribute of a product such as its load bearing properties, to another which is designed for a completely different purpose.

In a design environment there may be several local models, represented in local databases. These databases may or may not support versioning and the interfaces to these databases will be varied as they are intended to suit the local needs of the design agent. The research to be presented here intends to support a mechanism for versioning in a design environment without altering the design agent's local view. This will require the ability to apply version information to existing database systems externally in order to support a comprehensive product data model. In the following section an examination of the issues involved in describing a suitable model for version management in a modern engineering design environment, is given.

1.4 Version Management

Over the past decade or so the emergence of concurrent engineering technologies attempts to provide an effective infrastructure to increase the performance of design (Molina, 1995). This type of engineering work is characterised by the involvement of collaboration among engineers from many disciplines. Cleetus (1992) gives a definition of concurrent engineering:

"Concurrent Engineering is a systematic approach to the integrated product development, that emphasises response to customer expectations and embodies team values of co-operation, trust, and sharing in such a manner that decision making proceeds with large intervals of parallel working by all life cycle perspectives, synchronised by comparatively brief exchanges to produce consensus."

Each design discipline will have invested significant resources in developing their own design models and tools. These tools can be used within a particular discipline and at a given stage of the product life cycle. The tools cannot be applied in other disciplines or across stages in the product life cycle. The tools are, therefore, local "islands of automation" (Avouris, 1995).

Further, as previously noted, the various engineering design agents (designers, tools or teams of designers) often develop designs in parallel, co-ordinating their decisions through a series of design reviews in order to resolve conflicts. Due to the lack of an integrating framework, the design agents, based on application tools, database or knowledge based systems, cannot effectively communicate with each other. Design agents have to share common information. Using arbitrary terms or vocabularies among design agents increases the problem's complexity. One design agent's output is often the other's input. Apart from this type of relationship, one design agent might have to share the knowledge of the other design agents. This relationship builds a strong interdependency between the design agents. Thus, a mechanism is required to inform the target design agent of any change within the source design agent to ensure consistency between them. It cannot be assumed that the design agents are centrally located. Not only this but each successive change leads to a successive design version. Therefore a mechanism is required which will store and provide version information appropriate to producing a consistent product data model for all the participating design agents. The version strategy presented in this thesis attempts to be generic across Made-To-Order Engineering Design. This will be illustrated in Chapters 8 & 9 with a description of the prototype implementation and industrial case studies.

1.5 Relationship with current technologies

It is inevitable with research of this nature that technology evolves rapidly throughout the course of the project. An example of this is the explosion in the internet since the project's inception and the advent of Java™. In this section the work undertaken is related to current technologies which may offer different implementation options.

The Common Object Request Broker Architecture (CORBA) was chosen as the basis of the implementation strategy. CORBA allows a heterogeneous wide area network to be viewed as a set of distributed object servers, thereby reducing the complexity of

implementation. The growth in the internet is orthogonal to the use of CORBA. This is demonstrated in this research by the use of internet technology including Java and VRML (Virtual Reality Modelling Language) within a CORBA based architecture.

ISO 10303, the Standard for Exchange of Product Data (STEP) is constantly evolving and used in more and more disciplines. This facilitates the use of STEP as an enabling standard for the exchange of product data. Without the base standard, sharing of information would become a much more complex task. The initial case study domain, process engineering, is one where the STEP standard was already established from the outset. The growth of STEP means the application of the methodologies described in this work is also capable of growth.

The use of software agents within this research does not rely on any particular development architecture, for example the Java-based Agent Framework for Multi-Agent Systems (JAFMAS: Chauhan, 1997). Such frameworks are constantly emerging and developing and may in the future offer an alternative implementation of the methodologies described in this research. However, at the commencement of this research project these agent development frameworks did not exist. Their use within this work was therefore precluded.

1.6 Thesis Objectives & Plan

The objectives of this thesis are:

- Produce a suitable model for complete product data management in Made-To-Order Engineering
- Produce a version control and configuration management scheme
- Demonstrate the former objectives being fulfilled with a prototype system

In order to illustrate this the rest of this thesis is organised as follows.

In Chapter 2 previous approaches to the problem of version control in Engineering are discussed. This chapter draws on experiences from a number of different fields, such as software engineering (Rochkind, 1975) and mechanical engineering (Krishnamurthy & Law, 1997). Further, in Chapter 2 the key components required for a method of version control are described. This is used later in the thesis to justify the

methodology proposed. In fulfilling the requirements identified and examining issues with existing systems, it became apparent that the implementation of these demanding requirements would necessitate the use of agent technology.

In Chapter 3 traditional database technology and literature covering areas such as resolving schemata heterogeneity and integrating schemata and views in database systems are discussed. Research in related fields such as active databases is briefly reviewed. Finally an examination of suitable models for describing product data is given. This raises two more specific problems relating to data models: identity and the separation of structure from semantics.

In Chapter 4, the comparatively new area of Agent Oriented Programming is examined. From the literature an attempt is made to define the term *agent.* This definition leads to a description of properties that agents exhibit and also to a classification of different types of agent. Literature describing multi-agent systems is examined, discussing the specific problems relating to these, namely communication, co-ordination and interaction. Ultimately the existing applications of agents in Engineering Design are detailed.

In Chapter 5, a novel product model is presented. Following this, the proposed version scheme is presented. The philosophy behind the approach taken and also the version control from entity management up to total product configuration management is described.

In Chapter 6 the model proposed in Chapter 5 is expanded and clarified. The approaches taken to implement it are described. This description includes a discussion of the use of STEP, CORBA and agent architecture as a means to resolve the problems of version control in a typical engineering environment. Finally the usefulness of the scheme is illustrated with a simple Change Management scenario. Chapters 5 and 6 appear in shortened form in Florida-James et al. (2000).

In Chapter 7 the version model proposed in Chapter 5 and Chapter 6 is formally specified using a suitable formal method.

In Chapter 8 the detailed implementation of the prototype system that has been built and some results from that prototype are given. The practical problems encountered,

are illustrated and the approach taken to address these problems is described. Each type of agent is discussed in terms of its obligations within the whole system and how it fulfils these. A description of how the system would be applied in an existing design environment and the overall system architecture is given.

In Chapter 9 the application of the version scheme and its implementation through two real world case studies is shown. The first study is taken from the offshore oil and gas industry and the second from ship design involving the design of a passenger ferry.

In Chapter 10 conclusions drawn from this research are given, and future developments of the version control mechanism are proposed.

2 Version Models in an Engineering Context

2.1 Introduction

In Chapter 1 the problem domain of engineering design was described. Of course problems of controlling data exchange are not restricted to this domain. Therefore, the problems of product data model versioning have been tackled by many diverse domains. In this chapter work is presented which addresses directly version control in an engineering context, for example Katz (1990) and Dittrich & Lorie (1988).

Firstly, a description of a number of proposed schemes for version control of engineering design data is given. A description of the specific problem of product data models in an engineering environment is given in Section 2.3. Finally, Section 2.4 details the state of the art in software support for the design process.

2.2 Proposed Version Schemes

2.2.1 Commercial Systems

Product version control is well supported by CAD-based commercial Product Data Management tools dealing with design objects that form part of a finished products definition (Quillion, 1995). Conceptual alternatives refer to design objects that are under consideration but may never be embodied in the final design (Blessing, 1994) (Ball et al, 1998). Each alternative concept may be much more vague than a version object and often has abstract functional behavioural parameterisation rather than a precise geometric one. The capture of alternatives as well as versions within a design project provides a more complete picture of the design project.

Kilpi (1997) studies various commercial tools for software version control and configuration management. He concludes that Total Product Management (TPM) can only be achieved by considering that, "version control and configuration management processes have to be regarded as part of developing the whole product management process in a company". Allied with this he states that the most important finding is that there is no *perfect* process and product management processes have to be added

incrementally within existing business process. The word business can easily be substituted by the word design in the context of the work presented in this thesis.

In summary commercial systems tend to offer support for CAD based data. They provide mechanisms for capturing versions but not conceptual alternatives. However, as stated, capture of alternatives is an important feature in understanding the product development. TPM can only be achieved by adding design processes (Kilpi, 1997) within existing processes. This requires a version model to be pro-active as well as re-active. Also CAD based product models may not be the most suitable for TPM.

2.2.2 Dittrich and Lorie

Dittrich and Lorie (1988) propose that "in a design environment, contrary to other uses of copies, versions are associated with a semantic meaning that is known to the user. It is the user who finally controls which version is to be used".

They describe a version model based on the concepts of design, objects, generic references and logical version groupings. They describe a design object as a set of versions with a single distinguishable current version. They provide a mechanism and describe an extended form of SQL that allows design objects to form hierarchical aggregation by referencing each other. These references may be bound to a specific version of a design object or they may be generic, which is to say they refer to a particular design object but not to any of its specific versions. Generic references allow dynamic configurations to be obtained by not resolving the references until the hierarchical relationships are actually traversed. A specific reference is obtained from a generic reference through *environments* which contextualise the reference. Environments are either bound to a specific version of a design object or may bind recursively to other environments.

In addition their model supports the notion of logical version clustering which allows the user to impose more structure on the design versions by aggregating them into arbitrary groups. For example, it is possible to impose on the space of versions a grouping structure that clusters together versions that are revisions of the same alternative. Thus, under the design object node are a group of nodes representing version clusters for individual alternatives. Under these are additional clusters representing revisions of each alternative. Finally associated with each revision cluster

are those versions that participated in the revision. Thus arbitrary hierarchies of version clusters can be formed. A version may appear in more than one structure.

The model proposed by Dittrich and Lorie (1988) is one of the first to support the notion that, "providing some semantics for these versions the system can help the designer manage them more efficiently". However, they do not examine the problems of sharing information amongst collaborative teams of designers. In order to produce a practical version model these issues need to be addressed. A core theme of the version model described in this thesis is its collaborative nature.

2.2.3 Katz Unified Framework

Katz (1990) specifies properties that a version model should have to fulfil for the requirements of Engineering Design. He unifies themes from earlier work (Haskin & Lorie, 1982; Dittrich & Lorie, 1988; Klahhold et al, 1986; Batory & Kim, 1985; Landis, 1986; Chou & Kim, 1986; Ecklund et al, 1987; Rumbaugh, 1988; Vines et al, 1988), some of which have already been discussed. He describes seven basic mechanisms that all version models are required to have:

Version Set Organisation

The key concepts are version history, generic object, ancestor/descendent relationships, main derivation branches and a current version. Version instances are objects in their own right and are uniquely identified to the system. Version instances are related to a generic instance and are related to each other through ancestor/ descendant relationships.

Dynamic Configuration Mechanism

Static references bind to specific versions whereas dynamic references refer to generic objects and must be de-referenced to a specific version for certain operations such as check-in or check-out. The provision of dynamic references allows dynamic configuration.

Hierarchical Composition

The model should be able to support aggregation of objects into composite objects. It should also be able to traverse the hierarchy to produce a flattened configuration for

archiving.

Version Grouping

The logical version clusters described in Section 2.2.2 are an example of version grouping.

Figure 2.1 shows how version grouping allows identifiers to be associated with certain solution types.

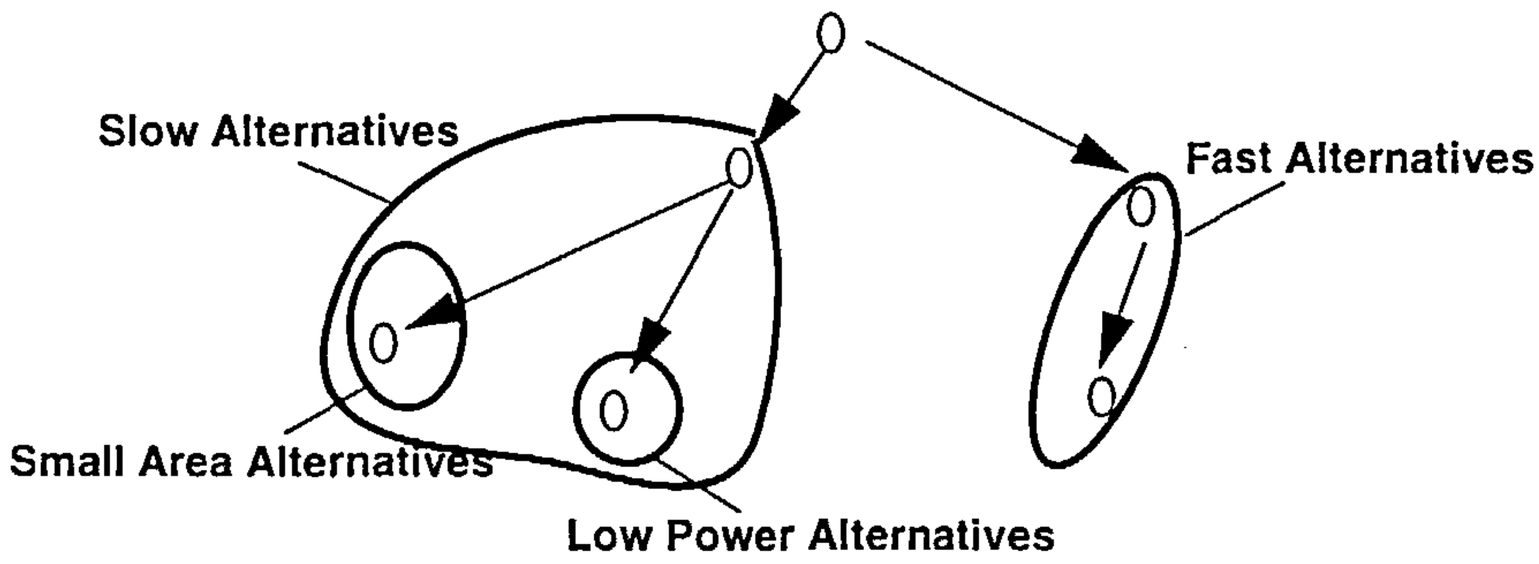


Figure 2.1 Version Grouping Mechanisms (Katz, 1990)

Instances vs Definitions

A distinction needs to be made between an instance hierarchy and a definition hierarchy. That is to say, where particular version instances are defined within the same storage object, for example a file, an instance hierarchy has to contain a separate node for each instance each pointing to the same definition.

Change Notification and Propagation

Change notification allows users to be alerted to changes, whereas propagation implies that new versions are incorporated within the configuration hierarchy automatically. The issue of change propagation is dealt with extensively in Chapters 5 and 6.

Object-Sharing Mechanisms

The basic concepts are private, group/project and public/archive with objects being moved through workspaces with check-in and check-out operations.

The seven mechanisms described by Katz (1990) are a benchmark against which any

version model may be critically compared. It is suggested that the model proposed in Chapter 5 fulfils more of these criteria in the specific context defined than any other proposed or existing model.

2.2.4 Made-To-Order Product Model Configuration

Wooley (1994) proposes a rules based model for product models in ship configuration management. These rules are given as a set of informal state diagrams and there is no suitable implementation scheme, but the research shows a wealth of understanding of the processes involved in design for the modern engineering environment. Wooley defines a Ship Product Item (SPI) as the abstract base entity which is a place holder for any entity in a model whose configuration needs to be managed. Each SPI has one or more Ship Product Item Versions (SPIV). A SPIV is a unique variation of a SPI identified by the concatenation of the SPIs identifier and a specific version.

Wooley's use of rules to describe a version control system is significant. The following example describes one such rule:

The deletion of a SPIV is always considered as a significant change. The following rules apply to the deletion of an SPIV.

A "working" or "preliminary" SPIV can simply be erased unless it has been transferred.

An "issued" SPIV or one that has been transferred must not be erased. A new version must be created and then the new version flagged as deleted.

The model Wooley proposes "provides the structure that a robust configuration management system can be built upon". However, his model does not propose a strategy for implementation of this structure.

2.2.5 A Collaborative Engineering Version Model

Krishnamurthy and Law (1997; 1994) propose a model and implementation which supports collaborative design. Their model addresses issues of storing and managing changes among designers in a multidisciplinary design project. They propose a three-layered model of versions, assemblies and configurations which systematically tracks an evolving project description, relieving designers of the burden of monitoring

individual design changes. Equivalent operations provide the theoretical foundations for operators to store, detect and manage changes at the version level. Moreover, the close coupling of the version, assembly and configuration layers enables computed version changes to be recursively combined to characterise changes at the assembly and configuration levels.

As their model parallels current design practices it supports project co-ordination and management. Their model is however restricted by being linked to a central ORACLE database and the prototype implementation only considers a CAD environment.

2.2.6 Storage Schemes

A number of implementation schemes have been developed for storing changes (deltas) in the software development process, such as Rochkind's (1975) Source Code Control System (SCCS). The SCCS treats each module as a set of related sequences of source code, each member of which represents one version of the module. It deals with storage optimisation, protection and access identification using a stamp and documentation of change.

Rochkind states that:

"The attempt by SCCS to record every version of every module that ever existed is rather ambitious. The system would be impractical unless it used a storage technique and accessing algorithm that allowed many deltas to be kept at a reasonable cost in terms of disk space and time"

There is at present no such system for a complex structured design representation such as a Made-To-Order model. It should be noted that the SCCS does not propose a way to merge existing version descriptions with the new change set.

2.2.7 DESCRIBE project

The DESCRIBE (Carnduff & Gray, 1994)(Kim et al, 1995)(Kim et al, 1996) project (DEsign System to support Concurrent Reuse of data in Building and Engineering Design) has attempted to focus research outcomes by applying an evolving version model to a real world problem. A real world design task under consideration was the conceptual design of a road bridge crossing a dual two-lane highway. Santoydiris et al (1997) defines an object Versioning System to support collaborative design within a

Concurrent Engineering context. This VSSCD has been implemented using ObjectStores (Object Design, 1995) object-oriented database.

Santoydiris et al (1997) state that an engineering design support tool has to model the way that designers actually design. They split the effort of a design project into several subprojects dependent on the targeted product's structural and functional specifications. This division leads to a three-level hierarchy (in section 5.2 it is stated that a three-level aprroach is common in configuration management). They define five possible states that a design artefact may have and provide a mechanism for promotion of object states and version creation, which in Krishnamurthy and Law (1997) is described as an acyclic graph.

The experimentation leads to a number of key conclusions about the version model produced.

- a fixed hierarchy with states is rather inflexible with two particular states seldom used
- the designers did not tend to design in isolation but with members of their subproject
- the configuration management could not be restricted to individual designers as this precluded sharing of finer grained information.

The experiment conducted used a centralised OODBMS as the total product model, this particular issue is discussed in the following section.

2.3 Total Product models

Section 3.2 argues that the choice of a suitable canonical model is critical when attempting to relate heterogeneous schema (Saltor et al, 1991) such as those found in engineering design. The fundamental issue is to make a model general enough to be of use at the enterprise level without inhibiting its use as a local data repository. Satisfying these conflicting requirements is a formidable task in practice. Proposals for general models are difficult if not impossible to implement (Florida-James et al, 1997). Implementation, however, can be made easier by consideration of the data involved in the process.

Clearly in engineering the problem of mismatched domains is very costly, but it

manifests itself with small enough frequency to be considered manageable. Also, much of engineering, particularly in design, is difficult to model in discrete processes and tends to deal with ambiguous data sets. In fact, design engineers particularly pride themselves on the ability to produce a suitable design from inexact information.

Not withstanding this, experience demonstrates that from conceptual design into the detailed design phase much of the designer's time is spent assessing the impact of changes on the design. Most of these changes result from the design process rather than from changes in the specification. It is therefore very important given the heterogeneous nature of a typical design environment that an appropriate enterprise model is produced. In this thesis a model is proposed which brings together work from database theory, software engineering, standards in engineering data exchange and knowledge engineering.

The enterprise model is constrained by three main assumptions. The first is that the designer's local model (database) must remain unaltered, or at worst the impact of the enterprise model should be minimal on the designers normal operation. The second main assumption is that transaction times are long (of the order of days in some cases) and therefore global consistency is of greater priority than efficient processing time. Finally due to the problems of a *bottleneck* effect described in Cutcosky et al (1993) the third assumption is that a centralised product database is not a realistic choice.

Addressing these assumptions in turn a product model can be derived with the following characteristics:

- The product model exists entirely independently of the participating local models
- Translation of data from the local to global schema may be done at run-time
- The product model should be virtual, that is, it should store pointers to data but no actual data.

In Section 5.2 a product data model with these characteristics is defined.

2.4 Software Support for Engineering Design

In this section some of the recent research into the uses of advanced software techniques in supporting engineering design is examined.

2.4.1 Defence Advanced Research Projects Agency - DARPA

Recently the use of agents to solve many problems in Engineering Design has been proposed. Much of this work has come from the DARPA initiative in the United States. In particular, Boeing have carried out extensive research into using agents to support the engineering design process (Jha et al,1998) as part of the RaDEO (Rapid Design Exploration and Optimization) program for a project MADESmart. In this work the agents communicate using the Knowledge Query and Manipulation Language (KQML) (Finin, 1993) and using shared ontologies. MADESmart defines the following classes of agents:

a .User Agents - analogous to interface agents (discussed in Section 4.3.5)

b. Control Agents - handle co-ordination and scheduling of global tasks

c. Wrapper Agents - encapsulate legacy systems

d. Resource Agents - link to external data sources

e. Execution Agents - analogous to autonomous agents (Section 4.3.1) but are specific to design tasks.

This classification is based on the *role* that each type of agent fulfils within the system. Software that fulfils roles rather than performs functions has a number of advantages in supporting processes. Firstly, the role can be defined as above in broad terms, allowing the problem domain to be modelled from the top down. Incomplete information can be used as valid inputs to the system. The agents can interact and respond differently according to the current situation rather than being restricted to one behaviour or function.

2.4.2 Knowledge Reuse and Fusion/Transformation - KRAFT

The KRAFT Project (Gray et al, 1997) involves a number of Universities in the United Kingdom: Aberdeen, Cardiff, and Liverpool. The consortium which also includes BT aims to evolve a combination of database technology and artificial intelligence into a multi-agent system (Pazzaglia & Embury,1998). The overall architecture is similar to that proposed in projects such as the DARPA funded Infosleuth (Bayardo et al, 1998). However, the KRAFT architecture concentrates on constraints to create powerful

problem-solvers at various sites on the network (Gray et al, 1998). The authors argue that constraints are, "a very general form of predicate definition which can be computed using functions. They are effectively recipes for selecting or calculating things; they can be passed between agents and fused or transformed into new recipes." Using constraints as problem solvers has obvious application in collaborative engineering. This is illustrated by Gray et al (1997) through a motivating example taken from engineering.

The system architecture consists of three facilities;

a. Wrappers - provides translation services from local format data to KRAFT format and also provides buffering and scheduling of a request to the local resource if required

b. Facilitators - contain a directory of services and facilities available within the KRAFT domain which allows provision of content based routing whereby messages are delivered to other agents dependent on their content.

c. Mediators - knowledge-level mediators which use ontologies to translate knowledge , specifically constraints, to provide solutions to conflicting design requirements.

2.5 Issues to be Resolved

*I*n this chapter, the required aspects of a version model for engineering design have been presented through research from a number of fields. Katz (1990) in particular categorizes precisely seven basic mechanisms a version model in an engineering context must have. Later work, for example by Krishnamurthy and Law (1997) illustrates an approach which produces a prototype of a version control system for engineering applications.

However, none of the prototypes discussed provide a generalised framework which fits the engineering environment described in Chapter 1. There are a number of issues which remain open, namely;

- a suitable distributed object sharing mechanism - How will objects be released from the designers private model into a group for review and comment? How will these changes then be propagated amongst the designers and into the current total product model? What facilities will support this mechanism?

- a dynamic configuration mechanism which may be applied equally to legacy applications and new applications - Any proposal for a new configuration system has to be equally applicable to existing systems, legacy systems and be flexible enough to support allow systems to be added;
- change notification and propagation in a distributed environment - How are designers alerted that a specification has altered? Is it possible to control negotiation between distriubted teams?;
- consistency of version grouping across isolated 'islands of automation' - Integrating diverse and dsitributed schemata into a single schema is a difficult and challenging issue discussed in Section 3.1. After acheiving this consistent view how do we maintain consistency as the versions evolve?

2.6 Summary

In Chapter 1 the problem domain is defined. In Section 2.2 of this chapter a number of proposed version models are presented. Not only this but seven basic mechanisms that a version scheme must have are identified. In Section 2.5 a clear list of outstanding issues are presented. In Section 2.3 the problem of a suitable product model is discussed briefly, this is also a key issue to be resolved. All these issues will be addressed directly in this thesis. However, before addressing these issues, the thesis continues with two more reflective chapters.

In Chapter 3 a description of research into support for data management in a heterogeneous environment is given. Discussing this earlier work serves two purposes

- it defines clearly the fundamental problems associated with heterogeneous distributed data sources
- it suggests prospective solutions which may be adopted

Section 2.4 identifies the uses of software agents in building distributed systems which are goal oriented rather than algorithmically controlled. Version control gives design engineers the ability to logically group pieces of data in some semantically relevant manner. This **ability** is a complex **goal** which is determined by the **rules** which govern the engineering process as well as the **rules** for data management (transaction

management). Not only this but the nature of the problem, as stated, is composed of distributed autonomous entities. It is clear therefore that software agents offer the most promising implementation strategy for a version model in this problem domain. Hence, in Chapter 4 Agent Oriented Software is defined from literature and its application in this research discussed further.

3 Review of Database Technologies

In Chapter 1 an introduction to the problem domian of this thesis is given. In particular Sections 1.1 and 1.2 describe the requirements of a modern design environment. Exmaining these requirements, in terms of data management, the following objectives can be stated

- Data be shared between distributed design teams
- Data be exported to and from legacy applications
- Data with related schema (not common) be integrated

To meet these requirements, the fundamental issues involved need to be appreciated. In this chapter a review of pertinent research is given. This research is usually classified as database research, hence the title of this chapter.

Firstly, complete integrated database solutions are discussed. Then particular issues are identified namely heterogeneous schema, schema integration, view integration and active systems. In the latter sections of this chapter the focus moves towards discussing the requirement identified in Section 1.3, that of a suitable product model. Sections 3.2 describes the current state of the art in this particular area of research.

3.1 Integrating Distributed Database Systems

3.1.1 Mulitdatabase Systems

Clamen (1994) states that design applications benefit from distribution, giving each user a localised view which is pertinent to them and offering a greater reliability as only local data may be viewed and updated. Problems arise then when it is necessary to integrate geographically separate component databases into a *multidatabase* or *federated* database system. It has been shown that the principles of single database systems may not be applied to the federated level. Litwin (1988) demonstrates this point for homogeneous databases and explains that the problem is compounded immensely over heterogeneous databases.

Research efforts to produce a practical multidatabase system have yielded some interesting prototype systems (Sheth & Larson, 1990). Motro (1987) proposes a

method whereby mappings are stored rather than actual data but demonstrates only a modest implementation. Motro further states that developing a system which deals with heterogeneous components, is a large and complex engineering task. The metadatabase project at Rensealer (Hsu, 1991) proposes an architecture based on meta data, which is similar to the idea of a virtual database. It goes on to define a rule-based method for producing the required metadata model. However, again there is only a very limited application of the method. *These types of systems, which produce and store mappings, are restircted in application by the complexity of the tasks involved in producing the mappings.*

Pegsasus (Shan, 1995) is a heterogeneous multidatabase management system with which external data sources are registered and import schemas are created to allow data extraction. Object Identifiers (OIDs) are generated for instances of imported types. Currently they are constructed using a prefix associated with the imported type and a suffix from the value set of its generating expression. Imported types are assumed to be disjoint on instances. Therefore, the same instance will have exactly the same OID within each type to which it belongs. The Pegasus system is limited only by the effort required to register external data sources and define import schemas. However, this effort may be prohibitive in certain applications.

Work at Queensland University (Yang, 1995) classifies and organises correspondences between heterogeneous object-oriented schema. This information resides in a knowledge base attached to each local database.The knowledge base allows remote objects to be treated as local data types and also determines which part of a query is local and which is remote. Yang's work demonstrates the usefulness of an approach which combines knowledge with data. However, in order to apply this technique practically knowledge needs to be classified statically before the system runs. In a dynamic environment such an approach may not be fully realisable.

The GARLIC architecture developed by IBM (Roth & Schwarz, 1997; Haas et al, 1997; Haas et al, 1999) exemplifies a middleware / wrapper approach to integrating databases. In this architecture, legacy data is viewed as instances of objects in a unified schema, based closely on the Object Database Management Groups standard. This schema is maintained as metadata and complex objects, which are composed of objects

from a number of sources.The architecture is shown in Figure 3.1

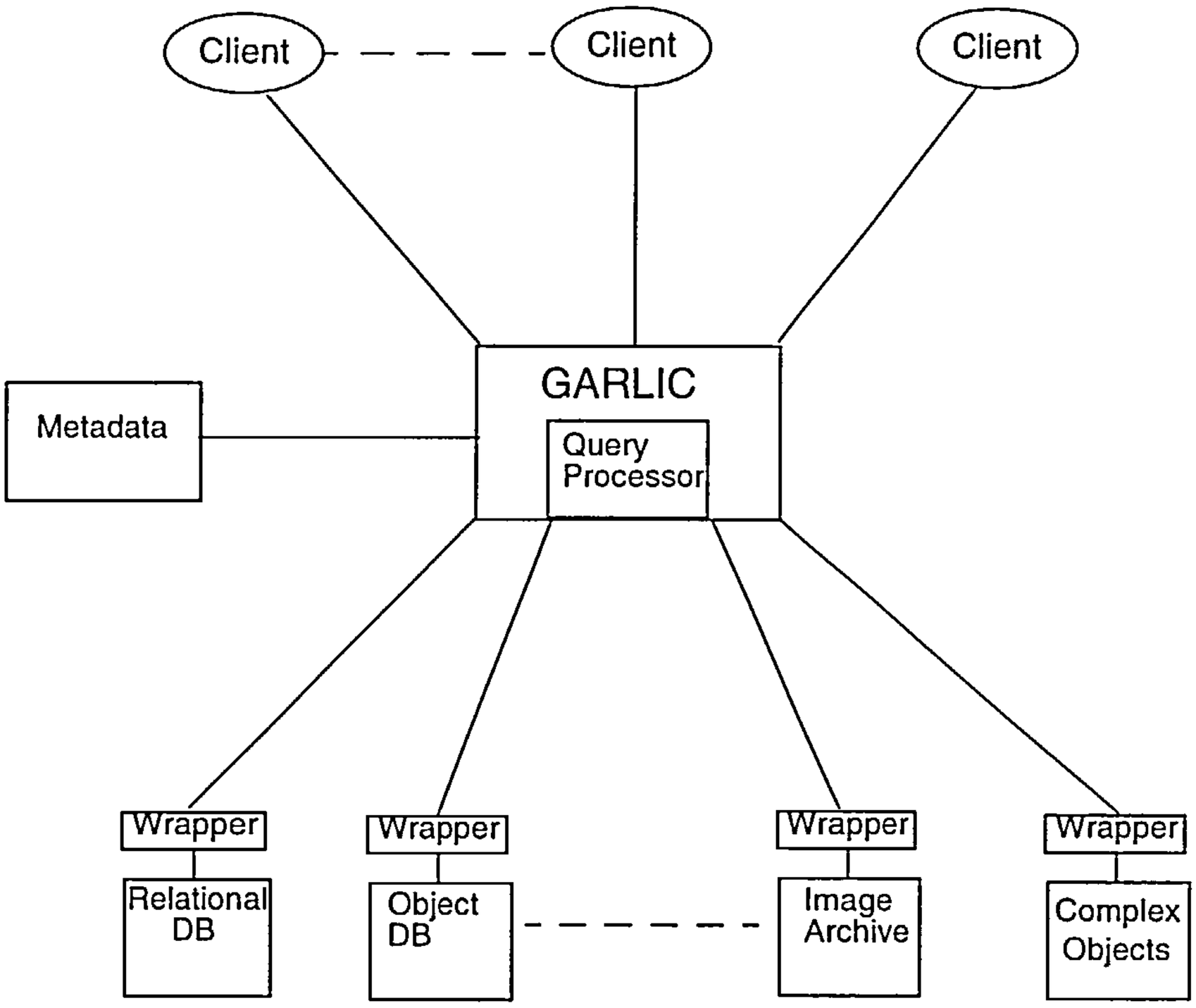


Figure 3.1 The GARLIC Architecture (Roth & Shwarz, 1997)

In this approach SQL, is extended to allow native methods from data sources to be used within queries, facilitating a single straightforward language extension that can support many kinds of specialised search. An optimiser based on rules is used to capture the query engines execution strategy. The key to the success of the wrapper approach is that the wrappers are small, flexible, able to evolve, and lend themselves to query optimisation.

3.1.2 Heterogeneous Schema

The problems and issues concerned with handling heterogeneous schemata are treated at length in Roddick (1995) and will be expanded in the following sections. Suffice it to say that these problems are complex and are a major stumbling block in the integration of design data which will always be represented heterogeneously in a real situation. Urban & Wu (1991) present a framework whereby heterogeneous schema in data models may be uniformly described. Urban & Wu argue that because of the "inherent incompleteness of legacy databases" semantic equivalence must in general

be obtained by the use of additional *a priori* assertions that are external to the representations under consideration.

This point is corroborated by DeMichiel (1989) who uses *partial results* to map values which cannot be ascertained explicitly from different domains. Partial results are developed from earlier work on *maybe results* by Codd (1979) and Biskup (1983). DeMichiel defines *virtual attributes* to denote an entity from a certain domain which contains information about its domain. DeMichiel then describes an extended relational algebra which can be used over virtual attributes and partial values. This allows autonomous databases to remain unaltered but incurs a large overhead when applied practically. The approach is centred around the relational model and is concerned more with mismatch of domains than with mismatched models. Kim & Jungyun (1991) describe a complete framework for enumerating and classifying the types of multidatabase structural and representational discrepancies. The framework is structured according to a relational database schema. However, Kim & Jungyun argue that the results are also applicable to systems which use an object model as the common data model.

Urban & Wu (1991) show how import and export procedures can be combined with global to local mappings to enhance inter-operability of heterogeneous schema. They suggest that the hardest issue to resolve is that of identity of objects in more than one database. Therefore, in order to produce a system of inter-operating schema, for example from legacy systems, resolving object identity is mandatory.

Worboys & Deen (1991) state that there are two main problems when interacting with heterogeneous schemata in distributed geographic databases. Firstly the underlying dichotomy of the model, which may be resolved through the use of a suitable canonical model and secondly in identifying and relating local contexts. Section 3.2 describes in depth the problems of producing a suitable canonical model. The issue of local context will be examined within Section 3.1.3.

3.1.3 Schema Integration

In order to integrate schema one must first define a method of equivalencing attributes from different databases. In Larson et al (1989) attributes are defined by a set of characteristics. These characteristics are used to define a measure of equivalence.

Strong equivalence based on this measure allows us to update and manipulate attributes whereas *weak equivalence* allows only retrieval of attributes across schema. Larson et al (1989) define *disjoint attribute equivalence* to mean attributes which are different but play the same role. They describe two distinct cases for integrating schema: logical database design which produces a conceptual schema and global schema design which produces a single global schema for existing databases. The latter is of more interest from an engineering perspective but the illustrated theory applies equally to both. Duwiari et al (1996) describe a technique for classifying and reusing knowledge accrued from the process of integration

Spaccapietra & Parent (1991) propose a taxonomy of conflicts which may arise from comparison of schema. They extend the notion of correspondence assertions to cope with such conflicts and apply this to interoperable databases.

Litwin et al (1991) suggest that a first order normal form is not good enough to extend queries over multidatabases, as schematic discrepancies often exist. It may therefore be necessary to use a higher order language to process queries in this situation.

Recent work at King's College (Poulovasssilis & McBrien, 1998) introduces a formal framework for schema transformation. This may be applied to a variety of data models and integration methodologies. The advantage of this formalism is that it clearly identifies which transformations apply for any instance of the schema and which only for certain instances.

Qutaishat, Fiddian & Gray (1997) describe the real issues of schema integration design process as:

- the identification of common types
- detection and reconciliation of potential conflicts
- elimination of duplication and redundancy of data

These issues are addressed within their Schema Meta Integration System (SMIS) (Qutaishat et al, 1992; 1996). SMIS is tailored towards integration of logically heterogeneous sets of object-oriented database which have previously been defined in a data definition language (Fiddian et al, 1992). The user of the SMIS is faced with the task of comparing and analysing components of the local schema in order to integrate them. To facilitate this, as a framework for schema integration, they have implemented

a subcomponent of SMIS called the schema meta-visualisation system (SMVS) (Qutaishat et al , 1993 ; 1996). SMVS is capable of producing a wide range of types of visualisation of schema inforamtion. This visualisation allows database users to assimiliate the structure of corresponding data models in a distributed database environment.

The methodologies outlined in this section all incur some overhead in producing the schema tranformations required in order to integrate the disparate schema. The schema examined in testing the methods are necessarily highly heterogeneous. In a more practical situation the degree of heterogenity may be reduced thereby reducing the transformation overhead described.

3.1.4 View Integration

Roddick (1995) states that view integration *aims to facilitate the merging of schemata for update or viewing purposes* and suggests that the area of view integration of temporal systems has had little or no research effort applied to it. Navathe et al (1986a; 1986b) illustrate how database design can incorporate and facilitate the integration of user views. They utilise a data model which extends the entity relationship model to involve categories. They assume a pre-integration phase whereby the equivalence of attributes has been defined. Hence the focus becomes *the relationship integration problem.* They consider the semantics of the relationship set by classification as follows

- Degree of a relationship set - the number of object classes participating in the relationship set. In each relationship, *n* entities participate in a relationship set of degree *n*.
- Roles in a relationship set - is the function played by an object class in a relationship set. Every object class participating in a relationship set is given a distinct role.
- Structural constraints - any specification rules, such as cardinality constraints, supported by the data model to express the constraints between the mapping from one relationship set to another are called structural constraints.

Given this classification, Navathe et al (1986) propose the classification of cases for

relationship integration and the possible outcomes when relationship sets from different views are compared as shown in Figure 3.2.

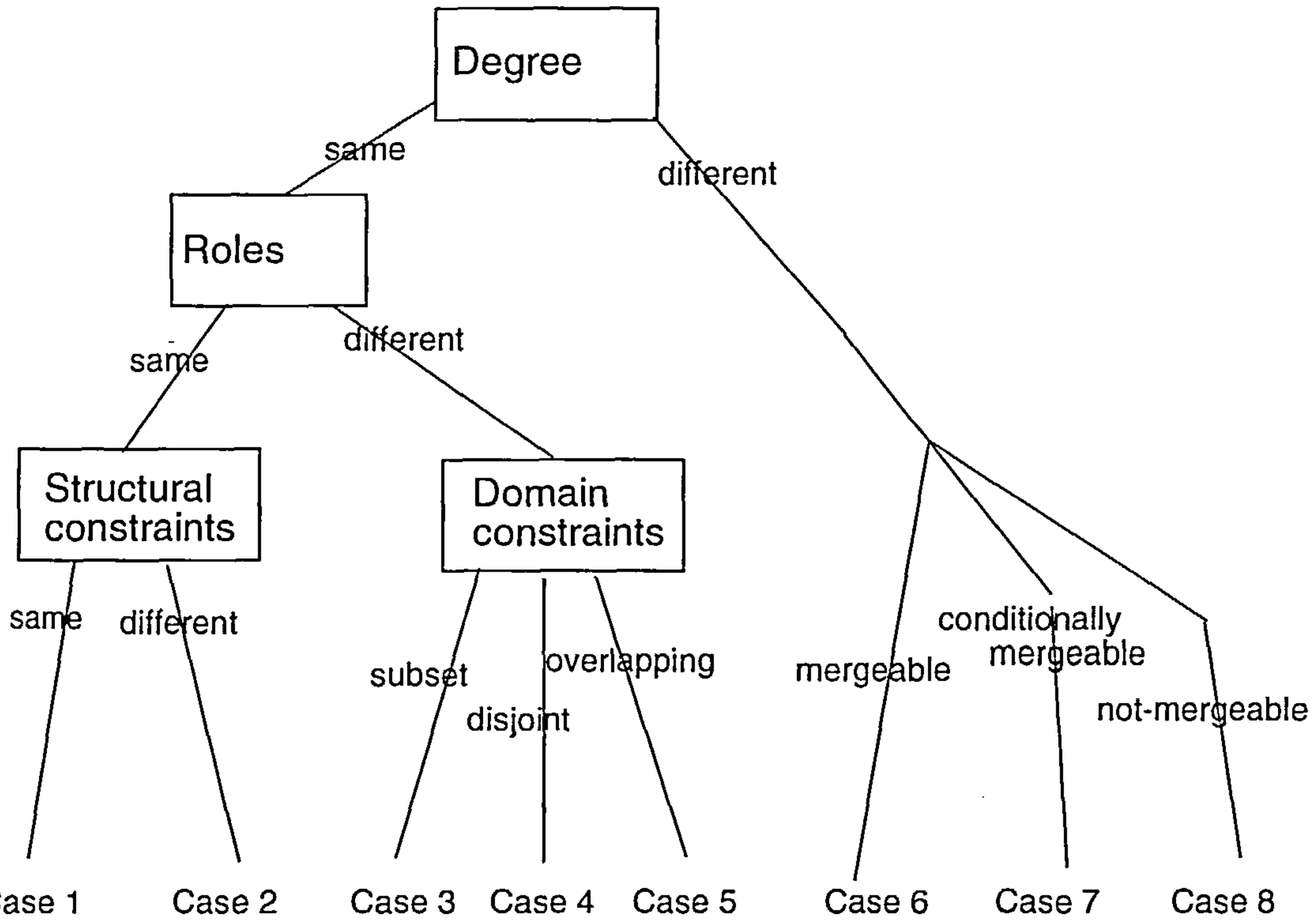


Figure 3.2 Classification of Cases for Relationship Integration of Different Schema(Navathe et al, 1986)

Rather than use the entity-category-relationship model to create views over heterogeneous schema, Qutaishat et al (1997) extend the Object Modelling Technique (OMT) notation to cope with bottom-up database design utilising upward inheritance. In this notation, the view is described as a materialised class in the static model. A new set of symbols is introduced to cope with the five types of upward inheritance required to cover the classes of integration problems described previously.

3.1.5 Active Systems

Temporal databases are discussed in general by Snodgrass & Ahn (1991) and the field of active database theory is introduced by Elmasri (1994). Active database behaviour is achieved through the definition of ECA (Event-Condition-Action) rules as part of

the database. These rules are then associated with objects, making them responsive to a variety of events. Events range from database events (e.g, update in a relational database or a method call in an object-oriented database) to temporal events (e.g, from 18:00:00 every 5 minutes) to general application signals (e.g, on power up of node x). When the event is detected the relevant rules fire. An active database system derives its power from the variety of events to which it can respond and the kind of actions it can perform in response.

Sistla & Woulfson (1995) define a system of constraints on active databases and describe a formal logic Past Temporal logic (PTL) which requires that a history of database transactions be kept. The model is based on *transaction time* and assigns an attribute time to elements in the database. The model may also be applied to *valid time* which is of much more relevance in engineering design. A distinction is made between these two notions. The database is usually a model of the real-world and the time at which changes happen in the real-world, namely the valid time, may precede the time at which these changes are posted in the database, namely the transaction time.

Active Systems offer some interesting insights into the problems of version control which is essentially maintaining consistency given some action. However, the reserch in this field is generally aimed at much smalller transaction times than are considered in this thesis.

3.2 A Canonical Model

As Worboys & Deen (1991) state, one of the main problems when interacting with heterogeneous schemata in distributed geographic databases is the underlying dichotomy of the model, which may be resolved through the use of a suitable canonical model. In this section a number of approaches towards producing a canonical model are described.

Rusinkiewicz et al (1984) suggest that a better approach than modelling constraints on data models is to model dependencies between them. These dependencies, if properly modelled, offer a better view of the world as they contain more about the semantics contained within the models. They include information about the state of the data as well as temporal information. Specification of dependencies is still not a precise process and further work is required to define correctness of a model and develop applications using it.

Motro (1987) develops an object generalisation technique which abstracts local data into a *virtual database*. The virtual database contains information about the mapping from local data to virtual data. Motro proposes, and implements in LISP, a translation system which allows update transactions to occur at the virtual level. This virtual level is synonymous with the enterprise level of the model of Saltor et al (1991). Motro describes a virtual database assuming a homogeneous set of schemas and he proposes that, to extend it to heterogeneous schema, a translation of base schemas be undertaken to a common model.

Saltor et al (1991) suggest a methodology for assessing a model's suitability as a *canonical model* for federated databases. They define four essential and four recommended characteristics that a model should have. They conclude that not all models are equally suitable for the job and that pre-relational models are of little value whereas functional models and some object-oriented models, particularly those supporting views, are better suited. The latter promising approach is considered further in Chapter 5.

3.2.1 Object Identity

The object oriented paradigm relies on a strong sense of identity. It is possible to use this sense of object identity at the federated level to produce transparency between local models. Eliasen & Karlsen (1991) state that the notion of identity at a federated level may only be achieved by relaxing the strict autonomy requirements of component databases.

They describe identity at three levels:

- Value based identity - identity by a key attribute value such as name
- Session object identifiers - object identity which exists during a transaction
- Immutable object identifiers - object identity which exists across transactions

Component databases may be described by models which facilitate varying degrees of identity and hence a suitable global object model is described whereby mappings from local to global objects strengthen the identity of the object at the global level. Eliasan & Karlsen (1991) point out two major issues from this approach:

- There is a large overhead in mapping identities to the global level
- It is not always possible to map a local object to a single entity at the federated level

Eliasan (1995) produces an architecture addressing the issue of identity and finds that computed OID maps are not practical unless frequency of local identity update is low.

Yu et al (1991) provide a method for automatically relating names in heterogeneous database. They define an iterative process, dependent on a knowledge base, which associates keywords with each other. The process described is automated as far as possible but requires user interaction to consolidate the associations.

Kent (1991) shows that problems arise inherently when multidatabase systems are devised because they violate implicit assumptions in the local models causing the information model to always breakdown. Kent suggests that the use of object models at the federated level prevents this breakdown because of the model's strong sense of identity but outlines problems of mapping local identities to global ones. Kent surmises

that in order to produce a full information model for multi-database systems knowledge must be added externally that is otherwise contained implicitly. This view is shared by Ventrone & Heiler (1991), Gangopadhyay & Barsalou (1991) and Siegal & Madnick (1991).

3.2.2 Structure & Semantics

The object model, as described in Rumbaugh et al (1991) and Rumbaugh & Booch (1995), can be viewed as comprising of two parts: the semantic content and the structural content. In Geller et al (1991), this decomposition is carried out to produce a *dual model*. This model supports *semantic relativism* which is defined in Brodie (1984) as the 'ability to view and manipulate data in the way most appropriate for the viewer'. There are widely thought to be two types of semantic relativism, that which enables a data model to be interpreted differently, on a structural level, and that which allows multiple views to be defined. It is obvious, from the latter definition, that the ability to produce a model which decomposes structure and semantics would be useful for integration.

Geller et al (1991) go on to show how classes may be defined in terms of *aspects*. Two defining principles are stated which may be used to classify aspects as either structural or semantic. These principles are then applied to demonstrate how semantic and structural aspects may be modelled practically.

In Frankhauser et al (1991) the importance of incorporating semantics when comparing two data models is demonstrated. An approach is presented which uses terminological knowledge and schema knowledge to produce a better comparison of data models. This approach is utilised in producing schema independent query assignment. The approach consists of three steps

- Collect Semantic Aspects
- Collect Structural aspects
- Determine Semantic similarity.

The approach used is not precise but appears to have great benefits when applied to the comparison of models, as it does not incur large overheads when employed.

3.3 Summary

In this chapter it has been demonstrated, through the wealth of research into integration of heterogeneous database, that there are a wide range of issues to consider and a number of approaches are available for tackling these issues. Whether the database systems to be integrated involve legacy systems or the strategic use of current systems, the fundamental issues as described here are at least to some extent unsolved. Ultimately it can be seen that the choice of a suitable solution depends on the application domain. There are also trade-offs of consistency over availability and efficiency over completeness which will need to be examined.

4 Agent-Oriented Software

Agent-oriented software is a relatively new field in software development. In this chapter the term *agent* is introduced and defined. The history of agents is briefly reviewed although for a more detailed *roadmap* of agent research Jennings, Sycara and Wooldridge (1998) is recommended. Previous research and work in progress which utilise the agent concept is described. Finally, the growth of agents being applied to engineering problems is reviewed.

4.1 Introduction

Shoham (1993) introduced the term agent-oriented programming as a new computational framework. He characterised agents as having a *mental state* consisting of beliefs, choices, capabilities and obligations and critically an ability to communicate this mental state with other agents. Since then many agent-oriented frameworks have been developed based around this notion (JAFMAS, 1997; COBALT,1998; Odyssey, 1997). Methodolgies for agent-oriented software design are still emerging (Wooldridge et al, 1999).

According to Chauhan (1997), the term agent has been used "unsparingly to refer to any software system which has attributes of intelligence". Thus, the definitions of agents have been given for a wide range of applications.

The Foundation for Intelligent Physical Agents (FIPA, 1999) defines an agent as:

"an entity that resides in an environment where it interprets *sensor* data that reflect events in the environment and executes *motor* commands that produce effects in the environment. An agent can be purely software or hardware. In the latter case a considerable amount of software is needed to make the hardware an agent."

Wooldridge and Jennings (1995) define an agent as

"Agents do things, they act: that is why they are called agents"

Maes (1991), from the Software Agents Research Group at Massachusetts Institute of Technology (MIT), gives the definition:

"An agent is a computational system that inhabits a complex, dynamic environment. The agent can sense, and act on, its environment, and has a set of goals or motivations that it tries to achieve through their actions."

As demonstrated above, and discussed at length in Franklin & Graesser (1997), researchers still strive for an absolute definition of an agent but, by examining the sense of the definitions given, some of the properties which define an agent can be described and then a simple classification of agents can be made.

4.2 Properties of Agents

Foner (1993), in examining the question *What is an Agent Anyway?*, suggests certain agent properties. The FIPA (1999) standard also suggests explicit properties that agents should have. From these and other literature the following properties can be defined as belonging to agents.

4.2.1 Autonomous

An agent must be able to behave independently (FIPA, 1999). It should react to and sense its own environment, operating under its own control rather than as directed by a user.

4.2.2 Self-Learning

An agent should have some ability to learn and therefore have some component of memory. Foner (1993) suggests that an ability to reason over more recent propositions is more akin to human behaviour than executing a set of rules defined to achieve long term goals

4.2.3 Continuous

An agent must be a continuous process, running throughout its entire life time (FIPA , 1999). It should not be a one shot program that does some specific task or calculation and then terminates.

4.2.4 Social

An agent must be able to communicate and interact with other agents and possibly humans as peers (Foner, 1993). Genesereth & Ketchpel (1994) propose a language, a grammar and an architecture for allowing agents to communicate in this fashion.

4.2.5 Mobile

An agent may be able to move around a network transferring data as it moves (Telescript, 1996). Mobility, however cannot be granted as an agent right as this raises many security issues.

4.2.6 Pro-Active

An agent should be able to act independently to initiate changes in its environment which make provision of its goal more acheivable.This involves aspects of periodic computation and also of communication in order to inform other agents of its actions.

4.2.7 Reactive

An agent must be able to monitor and react to changes in its environment, performing actions based on its own set of rules (O'Hare & Jennings, 1996).

4.2.8 Adaptive

Agents should be continuously changing state according to the environment (FIPA, 1999).

4.3 A basic classification of agents

As mentioned above, a number of researchers have characterised and developed their own agents for their applications. The following section attempts to classify a number of different agent types.

4.3.1 Autonomous Agents

Agents that inhabit complex, dynamic environments, in which they sense and act autonomously by so doing realise a set of goals or tasks

4.3.2 Entertainment Agents

Agents that inhabit interactive, simulated worlds providing entertainment to a user. These agents serve the purpose of entertainment (for example: games, film/video, production), rather than strictly utilitarian ones (Maes,1994;1995).

4.3.3 Information Agents

Agents that have access to potentially many information sources and are able to collate and manipulate information obtained from these sources to answer queries posed by users and/or agents. Some people refer to these as internet agents, as such agents may roam about the internet in order to collect information (Informant, 1997).

4.3.4 Intelligent Agents

Agents that carry out some set of operations on behalf of a user or another program with some degree of independence (FIPA, 1999).

4.3.5 Interface Agents

Maes (1994), a key proponent of this type of agent, states that the key metaphor underlying interface agents is that of a personal assistant who is collaborating with the user in the same environment. Essentially, interface agents support and provide assistance, typically to a user.

4.3.6 Collaborative Agents

Agents that emphasise autonomy and cooperation (with other agents) in order to perform tasks of their own. Their key attributes include autonomy, social ability, responsiveness and proactiveness. In order to have a coordinated setup of collaborative agents, they may have to negotiate in order to reach mutually acceptable agreements in some matters (Mori & Cutcosky, 1998).

4.3.7 Mobile Agents

Mobile agents are computational software processes capable of roaming wide-area networks, such as the internet, interacting with foreign hosts, gathering information on behalf of their owners and coming back home after having performed the duties set by their users. The attribute of mobility has introduced the concept of remote programming where agents interact as peers and each agent can act as both a client and

server (Telescript, 1996).

4.3.8 Reactive Agents

Reactive agents represent a special category of agents which do not posses internal, symbolic models of their environments; instead they act/respond in a stimulus-response manner to the present state of the environment in which they are embedded.

4.3.9 Hybrid Agents

Agents which brings together some of the strengths of both the deliberative and reactive paradigms. Each type of the above mentioned agents has (or promises) its own strengths and deficiencies. The most relevant technique for a particular purpose, is often a combination of attributes from different agent types. Maes(1991) calls this approach a hybrid approach .

4.4 Multi-Agent Systems

Regarding the issue of implementing a multi-agent system, three key mechanisms should be considered: communication, interaction, and coordination (Luck & d'Inverno, 1995; Shen & Barthes, 1995) :

- **Communication**:How to enable agents to communicate? What communication protocol to use?
- **Interaction**: What language the agents should use to interact with each other and combine their efforts?
- **Coordination**:How to ensure that the agents coordinate with each other to bring about a coherent solution to the problem they are trying to solve?

4.4.1 Communication

In general, there are four communication types that have been used in distributed agent architecture.

Directed Communication:

Directed communication involves establishing direct physical links with other agents

using a protocol such as TCP/IP, which promises safe arrival of message packets by implementing end-to-end acknowledgments. An example of this type of communication is the CORBA Event Service.

Federated System:

When the number of agents in a system becomes very large (for example in a setting like the internet) the cost and processing involved in directed communication is prohibitive. A popular alternative to directed communication that eliminates these difficulties is to organise the set of agents into a federated system (Genesereth & Ketchpel, 1994). Agents do not directly communicate with each other. Instead, they communicate through special facilitator (mediator) agents.

Broadcast communication:

Situations may arise, where a message has to be communicated to all the agents in the environment, or the sender agent does not know who the recipient will be (for example: when it announces a task and has to choose from all possible agents that can perform that task). In such cases, the sender agent can either physically broadcast the message to all the agents in the systems, or it can maintain individual communication links with all the agents in the systems and send each one of them a directed message, using the TCP/IP protocol (Dattola, 1996).

Blackboard-systems:

In Artitficial Intelligence , the blackboard is an often used model of shared memory. It is a repository on which agents write messages, post partial results and obtain information (Ball & Bauert, 1992).

4.4.2 Interaction

An agent needs a common language to deliver its request or result to other agents. An Agent Communication Language (ACL) is a neutral language for agent to agent interactions. An ACL with precisely defined syntax, semantics and pragmatics is the basis of communication between independently designed and developed software agents. The Knowledge Query and Manipulation Language (KQML) (Finnin et al,

1993) is an ACL which is based on Speech-Act Theory. In the multiagent system community; speech-act theory is one of the most common methods used to construct the linguistic layer and formalise the linguistic actions of agents. Speech-act theory uses the concept of performatives to allow an agent to convey its beliefs, desires and intentions. For example, performatives 'assert', 'state', 'affirm' convey a belief, performatives 'ask', 'order', 'enjoin', 'command' convey a wish or a desire, and performatives 'vow', 'pledge', or 'promise' convey an intention.

Currently, however, as Singh & Joshi (1999) point out whilst ACLs have been used for years in proprietary multi-agent systems, agents from different vendors and even different research groups cannot communicate with each other. They propose a conceptual shift from individual agent representations to social intentions, which would allow the definition of a common ACL. All agents would then be able to communicate via this common language.

4.4.3 Coordination

The aim of agents is to collaborate with each other to achieve their common goal. The behaviour coherence of each agent and the way they coordinate with each other have significant impact on the success of the application.

Gasser & Bond (1988) differentiates between 'coherence' and 'coordination' by referring to coherence when discussing how well the entire system behaves while solving the problem and examining the system's behaviour as a whole, and to coordination as the property of interaction among a set of agents performing some collective action.

Many problems cannot be solved by agents working in isolation because they do not possess the necessary expertise, resource or information to solve the entire problem.

In COOL (COOrdination Language), Barbuceanu & Fox (1993) have proposed that the coordination problem in a multiagent system should be studied at an 'organisational level'. The coordination problem can be tackled by having knowledge about the interaction processes taking place among agents. This knowledge is about the problem-solving competence of multiagent systems as opposed to that of the individual agents.

In this approach to coordination, agents usually form a plan which specifies all their future actions and interactions with respect to achieving a particular objective. Within JAFMAS and FIPA agent architectures, an ontology has been used as part of a specification to alleviate the mismatch of terms used in the agents. The ontology to which the agents refer, and the contents that the agents want to deliver can be specified in a speech-act language (for example KQML).

Mori & Cutcosky (1998) consider more specifically the problem domain of agents interacting and co-ordinating the exchange of information. They present a state transition diagram which represents the design process as a series of primitive operations as shown in Figure 4.1

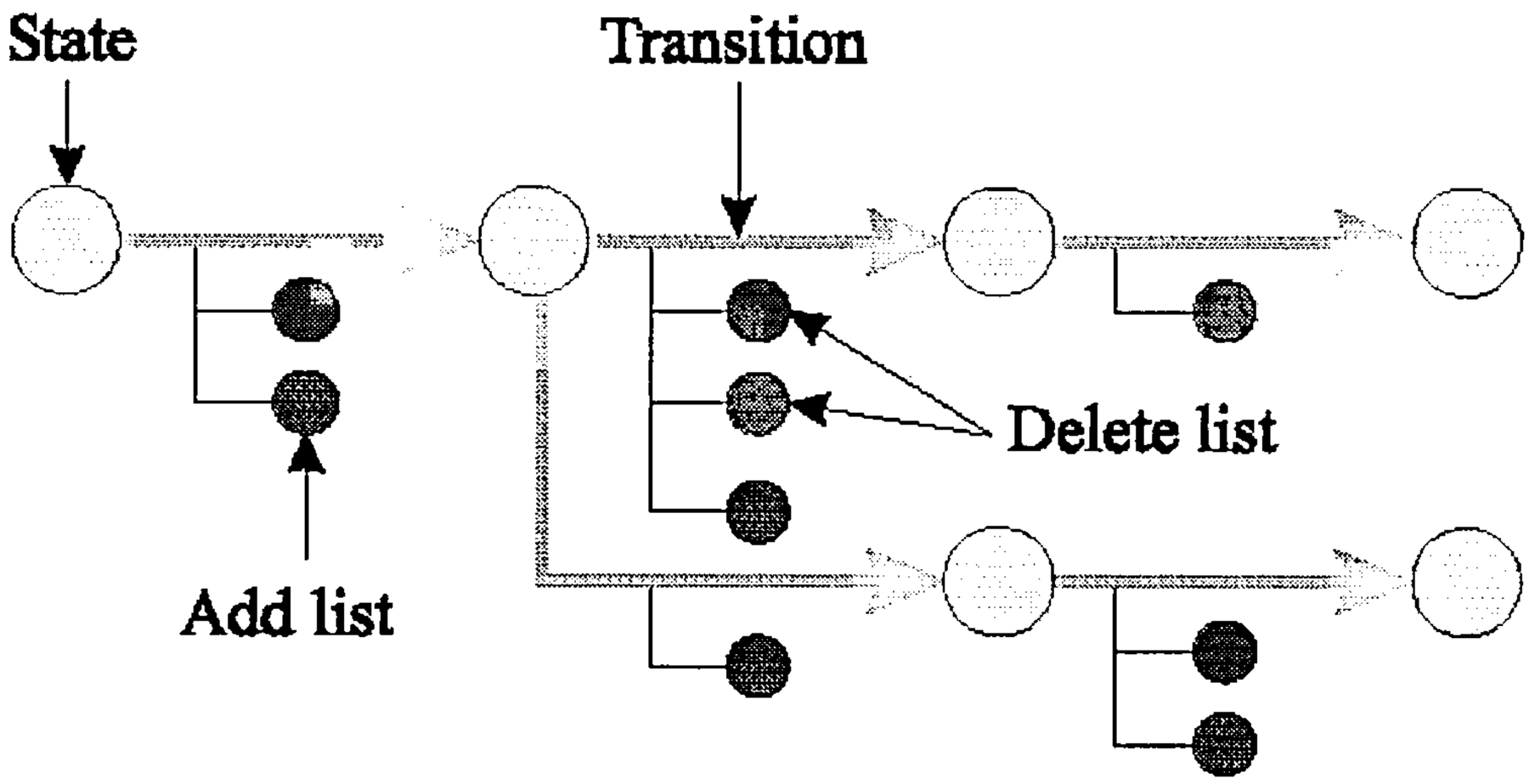


Figure 4.1 Representation of the Design Process as a State Transition Diagram

Each operation is described as a set of three elements;

a pre-condition that must be true to apply the operation,

a delete list to be erased from the design model as a result of the operation,

an add list to be added to the design model.

This representation of the design process is stored in an agent as shown in Figure 4.1, therby allowing the design process to be recorded and co-ordintaed. This concept is applied in a CAD domain to the design of a compact disk player. However, it relies on the assumption that *delete list* and *add list* primitives are commutative which is not valid in all domains.

4.5 Summary

The agent-oriented paradigm offers many exciting new possibilities in many domains such as commerce, information retrieval and not least engineering. The use of agents in engineering design is a field which is expanding rapidly and shows promising results. As Douglas Dyer of DARPA (1999) recently stated:

"If we are successful [*in the use of agents*] we will enable a drastic paradigm shift in software technology toward automation and cost-effective technology".

The complex task of version control in engineering design requires more than traditional software paradigms such as client server or distributed objects. The agent concept appears to offer a viable solution to this issue. As has been shown the use of agents raises many important questions such as the nature and type of the problem and which agent properties are best suited to particular problems. The issues of multiagent systems, communication, co-ordination and coherence also need to be addressed. In the next chapter a version model for engineering design is presented with an initial implementation framework which illustrates the strategy to be followed in the application of agents.

5 A Version Control System for Engineering Design

In Chapter 2 the requirements of a version model which would be suitable for a modern engineering design environment, are clearly identified. There are a number of issues which remain open, namely;an appropriate total product model; a suitable distributed object sharing mechanism; a dynamic configuration mechanism which may be applied equally to legacy applications and new applications; change notification and propagation in a distributed environment; consistency of version grouping across isolated 'islands of automation'.

In this chapter and Chapter 6 a version model is described in detail that attempts to fulfil these demanding requirements. The model presented is distributed, scalable and generic as it makes little assumptions about the design tools involved. It is a system for complete product configuration and version control. It allows designers a consistent view throughout the complete product life-cycle whilst permitting them to retain their own product representation and design tools.

Chapter 5 is organised as follows. In Section 5.1 the total product model proposed in this thesis is described. In Section 5.2 a system of agents is introduced. This multi-agent system is recommended for version control. The agents are described in terms of their roles and responsibilities within a version control system. Sections 5.3 and 5.4 describe the principles of the version management system and the detailed version model presented in this thesis. In these sections all aspects of the model are described, from low-level entity version management to complete product configuration management.

5.1 A Novel Virtual Product Model

In this section the proposed total product model is presented. It has already been argued that the total product model be object-oriented and that it supports views (Section 3.2). To achieve this, objects are stored as pointers to the local *real* objects and relationships as objects in the global schema. This storage is handled by an autonomous software entity called the global agent, as described in Figure 5.1. The application layer talks

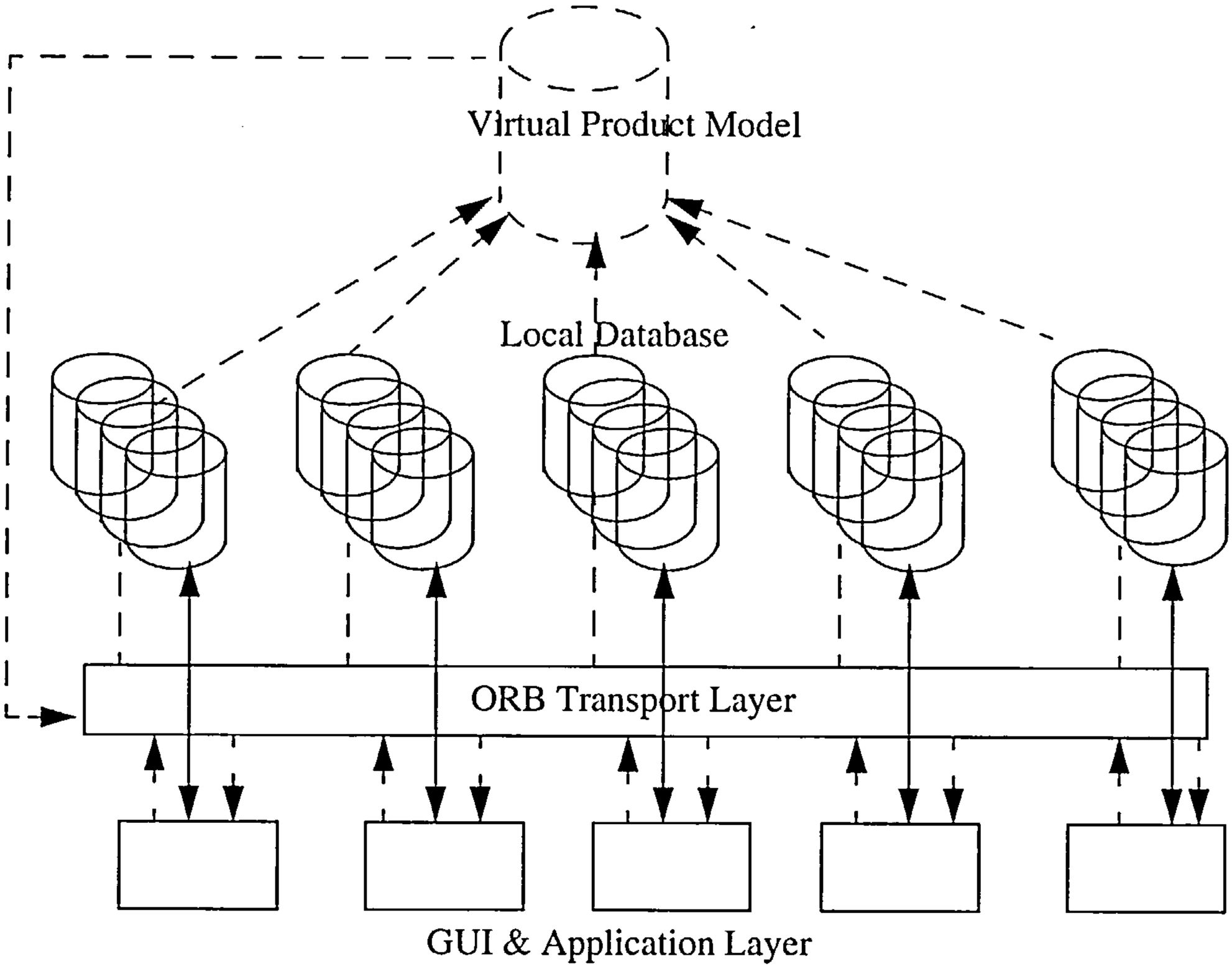
directly to its local database. The databases evolve over time as versions as the design matures. The application layer communicates with other applications and the virtual product model through the ORB (Object Request Broker) transport layer as shown. The ORB layer facilitates the storage of pointers to objects across a distributed network. This is explained fully in Chapter 8 where the implementation of the complete version control system is described.

Figure 5.1 illustrates the conceptual model and the virtual product model as a flat space of objects and relationships. The objects contain data only as pointers to the local object via the local interface. Each object has a unique global identifier within the global schema and therefore there is no ambiguity. The relationships are stored as first-class objects and hence we can view the product model from different perspectives by applying different relationships corresponding to different views. Thus a virtual product model has been created. This allows consistent views of the data to be maintained globally and yet does not force any unnecessary requirements on the underlying data models.

Storage of pointers to distributed data rather than the actual data has previously been described as a *virtual database* (Motro, 1987). Early virtual database were however cumbersome to implement and manage. However, the use of a virtual database (VDB) has recently been supported by Rajaraman and Norvig (1998), of Junglee Corp. They describe their commercial virtual database system which they use to transform the Internet into a database system. They state that VDB technology should be used where applications exist which have one of the following characteristics:

- Large numbers of data sources
- Autonomous data sources (that is, no centralised control)
- Data sources that have a mixture of structured and unstructured data.

Most engineering design environments exhibit one or more of these characteristics, in particular they comprise autonomous data sources, as illustrated in section 1.3 of this thesis.



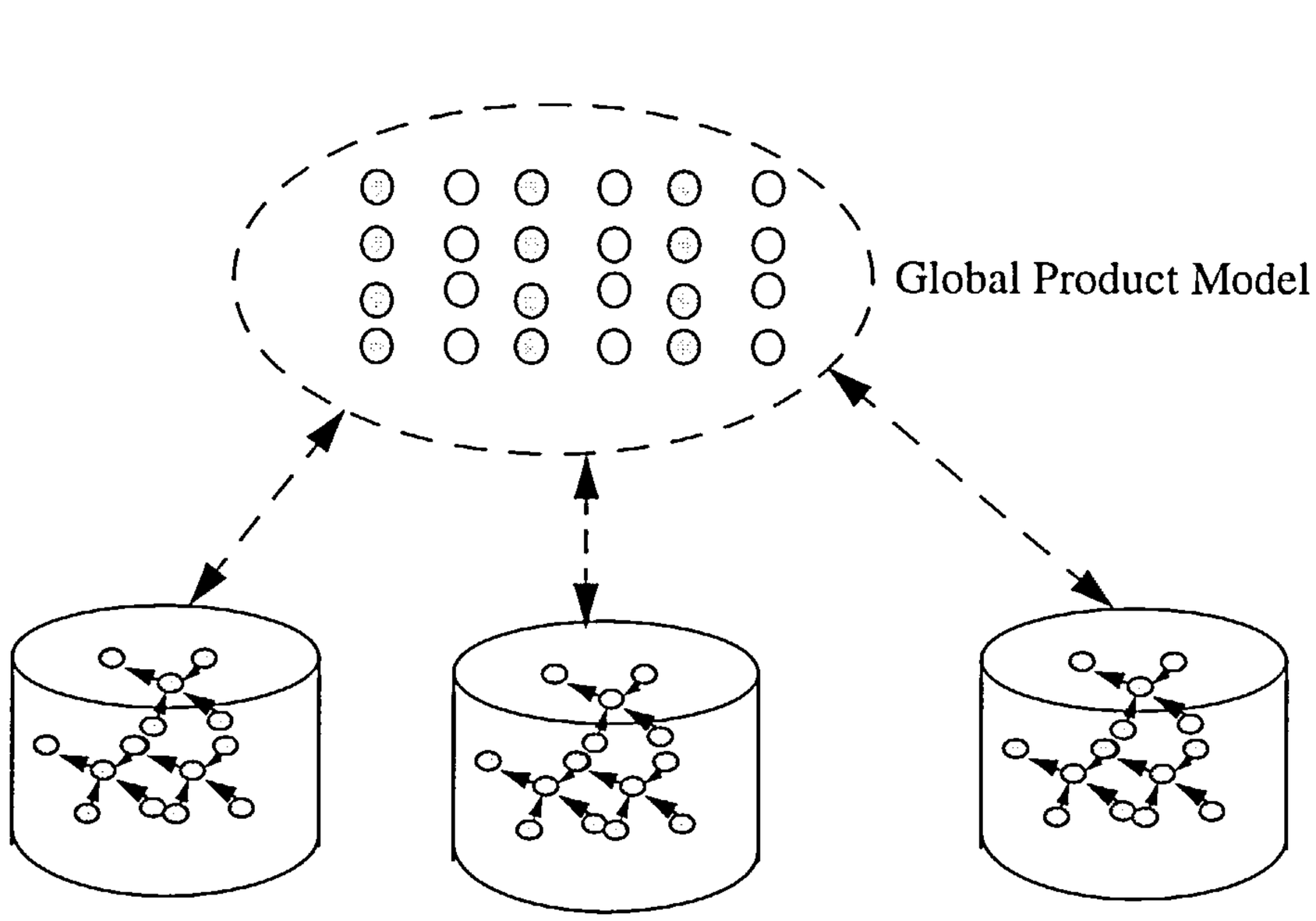


Figure 5.1 Description of the Global Product Model

5.2 A multi-agent system for version control

The discussion of the version model continues with a description of the agent system. Figure 5.2 shows a conceptual description of how two design tools or teams (Design

Agent 'X' and Design Agent 'Y') may be linked by the agents. As shown there are three types of agent namely global, behavioural and resource. The global agent, as introduced in the previous section, manages the total product model. The behavioural agents are the communicating layer and the resource agents control the interactions with the local database. The agents also represent different levels of control within the system.

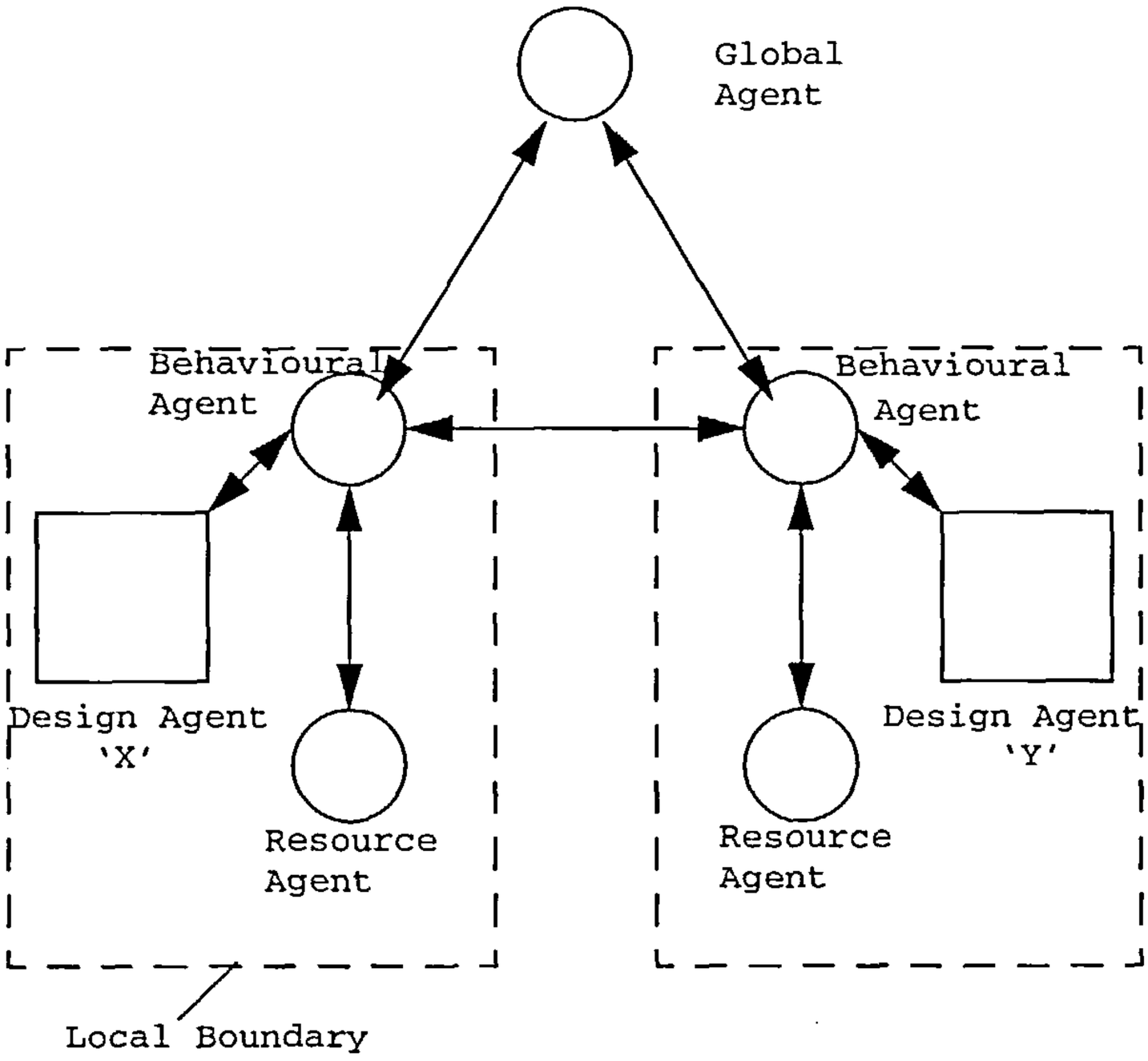


Figure 5.2 Representation of the Agent System

In configuration management, a three-level approach is common. These levels have been called archive, group and private (Chou & Kim, 1986) or more generally released, working and transient. In any event, the three levels equate to different access and viewing controls. The three level approach encompasses the following: the released version is controlled by the global agent, the working version is that which can be examined by the behavioural agents and the transient is that managed by the resource agent.

A fuller description of what each agent does and the knowledge required by each agent is now given.

5.2.1 Resource

The resource agent acts as the interface to the physical resource, that is the data. It has the following responsibilities:

- Stores local information
- Manages a purely structural EXPRESS model of local information
- Appends version information (that is tags or labels)
- Accesses local information

5.2.2 Behavioural

The behavioural agent acts as the designers interface and also as the control layer in the version model. It controls the allowable updates and communications with other designers. It has the following responsibilities:

- Knows how to communicate with other design disciplines
- Resolves conflicts and constraints
- Controls level of access to local model
- Controls user interaction

5.2.3 Global

The global agent represents the semantic or knowledge layer. The knowledge in this case is restricted to the following:

- Knows what resources are where
- Knows the current configuration of the product
- Knows relationships between objects at the global level
- Controls creation and deletion of system objects

These agents will be discussed further later in thisthesis. Their roles and beliefs will be expanded. The application of these agents to version control will be demonstrated and their implementation will be discussed.

5.3 Version Model Definitions

The version model which is presented here has a number of key concepts which separate it from other schemes. Firstly, no unrealistic requirements on the underlying design tools are made. An ability to export data is the only assumption. Secondly, in a more natural representation, tools are described as communicating agents with asynchronous messaging capabilities rather than as remote objects or functions which may be invoked or called. Finally design agents are permitted to collaborate without relaxing any autonomy constraints on the individual design tools.

The description of the model proceeds with a definition of the terms used.

An **Entity** is an item which is considered as a design object in any participating model at some stage in the product lifecycle.

A **Configuration** is a unique set of entities which when logically related describes the complete product model at a given point in time.

A **Version** is a specific instance of a given entity, which may be derived from any previous version through a series of change operations.

Change operations are defined as **create, delete** and **modify.**

Version levels are described as **private, declared** and **recorded.**

It may appear that versions and configurations are synonymous, given that there is no differentiation between complex and simple entities. The important distinction is that a configuration must describe the complete product model. In a logical sense an entity may be complex in one model or version and simple in another. This view is useful when combining models (Florida-James et al., 1997) which may have been developed separately but it is difficult to implement using a traditional procedural or object-oriented programming paradigm. The use of agents allows us to incorporate semantics and activity into the version model. Entities can then be allowed to perform different roles in different local models at different stages in the product lifecycle and let the model respond actively to change.

5.4 Version Management

5.4.1 Entity version management - managed by resource agent

Local models are managed by the local database and the model assumes no control or access to this structure. This allows designers to continue working with the tools they are familiar with and also to introduce the system to legacy applications. The wrapper method (Section 3.1) is frequently used to access these system but in this case the wrapper is contained within the resource agents. The resource agent understands STEP and hence performs the translating of design entities from the local repository to the global repository. Curiously the STEP standard appears to have no versioning mechanism of its own.

The proposed scheme is a forward deltas scheme (Rochkind, 1975), where deltas are stored as a list of entities on which primitive operations have been performed. It is assumed that the three primitive operations described, **modify**, **create** and **delete,** can be used to represent all design actions across all domains.

The labelling scheme applied by the resource agent is external to any local scheme the design tool or database may have. It is of course possible to utilise any available local scheme as long as in the global context a unique version set identifier is obtained. The scheme is adapted from that of Keller & Ullman (1995) but at present there is no implementation of their optimisation processes. Entities are initially **create**d within a version set with a unique identifier corresponding to the EXPRESS model entity name. The entity's storage is physically within the resource agent but rules concerning the creation and deletion of entities and their relationships at the global level are controlled by the global agent. An advantage of this scheme of resource agents is that it is reactive in the sense that changes made in another model are automatically reflected in all local models by the application of a new label. This label is actually designated by the resource agent and is required to be unique in the local context and to tell us which agent caused the change. Rules that govern the resource agent behaviour are given in a later section along with examples. These rules are further formalised in the VDM - SL specification.

As stated earlier there is no distinction between complex and simple entities in the versioning system. Entities are related in terms of hierarchy by the global agent. STEP

however, defines a rigid hierarchical structure in its Application Protocols. The system does not choose to ignore this and indeed this structure is available within the local models where it is very useful. A less rigid definition of an entity for global version control is used whichallows entities to exist at different levels of detail in different models or domains. The reason this is useful is that models based on mismatched domains can be related and merged using hierarchies containing generalisations and specialisations of existing terms. The difficulty is that somewhere a consistent representation of the knowledge of these relations needs to be stored. This is one role of the global agent.

Consider a structural design representation consisting of a Deck entity which contains four subsystems - High pressure, Medium pressure, Low pressure and Power Supplier. In the process design representation the Low Pressure subsystem is further decomposed into two pumps, a compressor and an inspection gear launcher. In this example the domain mismatch can be easily visualised. The process representation is much more detailed than the load bearing concerns of the structural agents. It is also easy to see how the problem may be addressed by hierarchical representation of the relationships between entities. Hence in the global agent there exists a list of entities and a set of relation types *is a part of* or *is equivalent to*.

5.4.2 Configuration management - Global and Behavioural agent

The global and behavioural agents combine to give an overall configuration management system for the complete product life cycle. This system has been developed to support earlier work on change propagation in an integrated design environment (Guenov et al. 1996). In this system the behavioural agents may be described as the logical layer and the global agent as the knowledge layer. The behavioural agents define the rules for co-operation and change management whereas the global agent has knowledge about design entities and their relationships in the global context of the total product.

The first aspect of the configuration management scheme is the labelling of design models. Fundamentally therefore a distinction is made between two types of versions - version histories and version alternatives. Versions are caused by change and hence the process of change is represented within our labelling scheme. When a requirement

for a change is issued a conversation on this change is started between the behavioural agents and the currently declared versions of each model are updated with a new label as shown. If this change is agreed, the label on each version then becomes the timestamp and the other labels are removed, the change issued and delta storage handled by the resource agent. In the second diagram a change is issued but this conflicts with constraints in another model so this agent produces a set of alternatives which compromise both constraints and the label is now composed by adding the agent's alternative. At this point the behavioural agents are in a state of conflict resolution and all subsequent alternatives are labelled in the same way. When the conflicts are resolved the label reverts to the timestamp as in the previous scenario.

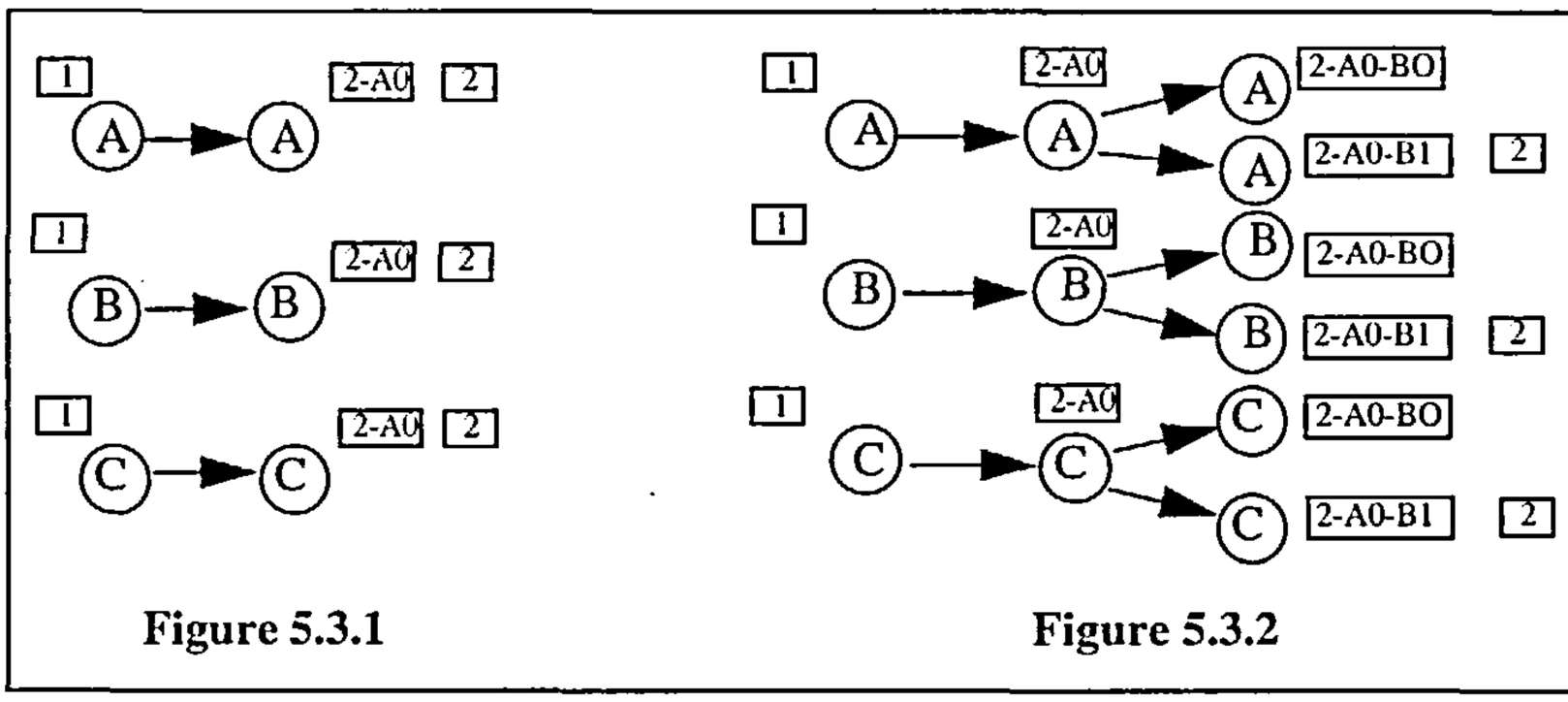


Figure 5.3.1 **Figure 5.3.2**

Figure 5.3 Labelling Scheme Example

In order to cope with the situation where the above resolution fails at this point in time, that is, where parallel design exists, the optimal action appears to be to simply clone (Dattola, 96) the global agent and give it another label based on the same scheme. For example in a case study from the off-shore industry two designs continued for a period whilst a decision on whether a concrete or metal jacket would be produced. Eventually however one design becomes inactive. Experience shows that, whilst cloning the global agent may be expensive in system terms, it allows the model to continue consistently without inhibiting the design process. These clones very rarely stay active for long periods.

5.5 Summary

In the introduction to this chapter a list, taken from the list given in Section 2.4, of open issues is given. Below this list is restated and a brief explanation of how each issue has been addressed within this thesis is illustrated.

•an appropriate product model - In order to produce total product version control a product model is required. This is described in Section 5.2.

•a suitable distributed object sharing mechanism - the three level of agents described in Section 5.2 represent a distributed three level object sharing mechanism

•a dynamic configuration mechanism - Section 5.4.2 describes how the system proposed dynamically manages product configurations throughout the complete lifecycle

•consistency of version grouping across isolated 'islands of automation' - the consistency of version groupings is maintained through the interaction of the agents and the labelling scheme example given in Figure 5.3 illustrates this.

•change notification and propagation - this issue is addressed in Chapter 6, Sections 6.4 to 6.6.

It can be seen, therefore, that in this chapter many of the issues raised have been addressed. In Chapter 6 any remaining issues are discussed and some of the smaller detailed problems are examined. In Chapter 7 a formal specification of the version model is given to remove any ambiguity and also to prove certain aspects of the system. The implementation and application of the model are then demonstrated in Chapters 8 & 9.

6 Collaborative Version Control in Engineering Design

6.1 Introduction

Chapter 5 summarised the arguments for, and then defined the proposed version model. In this chapter more detailed issues which all impact the model are discussed. The model proposed is expanded and clarified with focus on engineering design in particular. The Standard for the Exchange of Product Data (STEP) is briefly described. The issues of version control within this standard and fundamentally any other standard based around a strict data schema are described. The version model proposed in this thesis directly addresses these issues. This is the theme of sections 6.2 and 6.3

Sections 6.4 to 6.6 focus on the ability of the version management scheme to handle change. Within this focus a detailed description of the agent architecture is given. As with all agent architectures the key aspects of the agents' roles and responsibilities are clearly given. The communication between the agents is described. This facilitates the agents working together to produce a practical system of version management. Section 6.6 walks through a typical change management scenario. This demonstrates the agents working together and should aid the reader in understanding the descriptions given earlier.

6.2 STEP

The STEP standard has been growing since its inception in 1984 and includes many discrete parts. The standard is open-ended in the sense that it is extensible to the demands of industry (Owen, 1993). Its structure is illustrated in Figure 6.1. The first nineteen parts of the standard contain the description methods which include the EXPRESS, EXPRESS-G, and EXPRESS-I information modelling languages (ISO, 1995). Later parts of the standard describe techniques for implementing the standard.

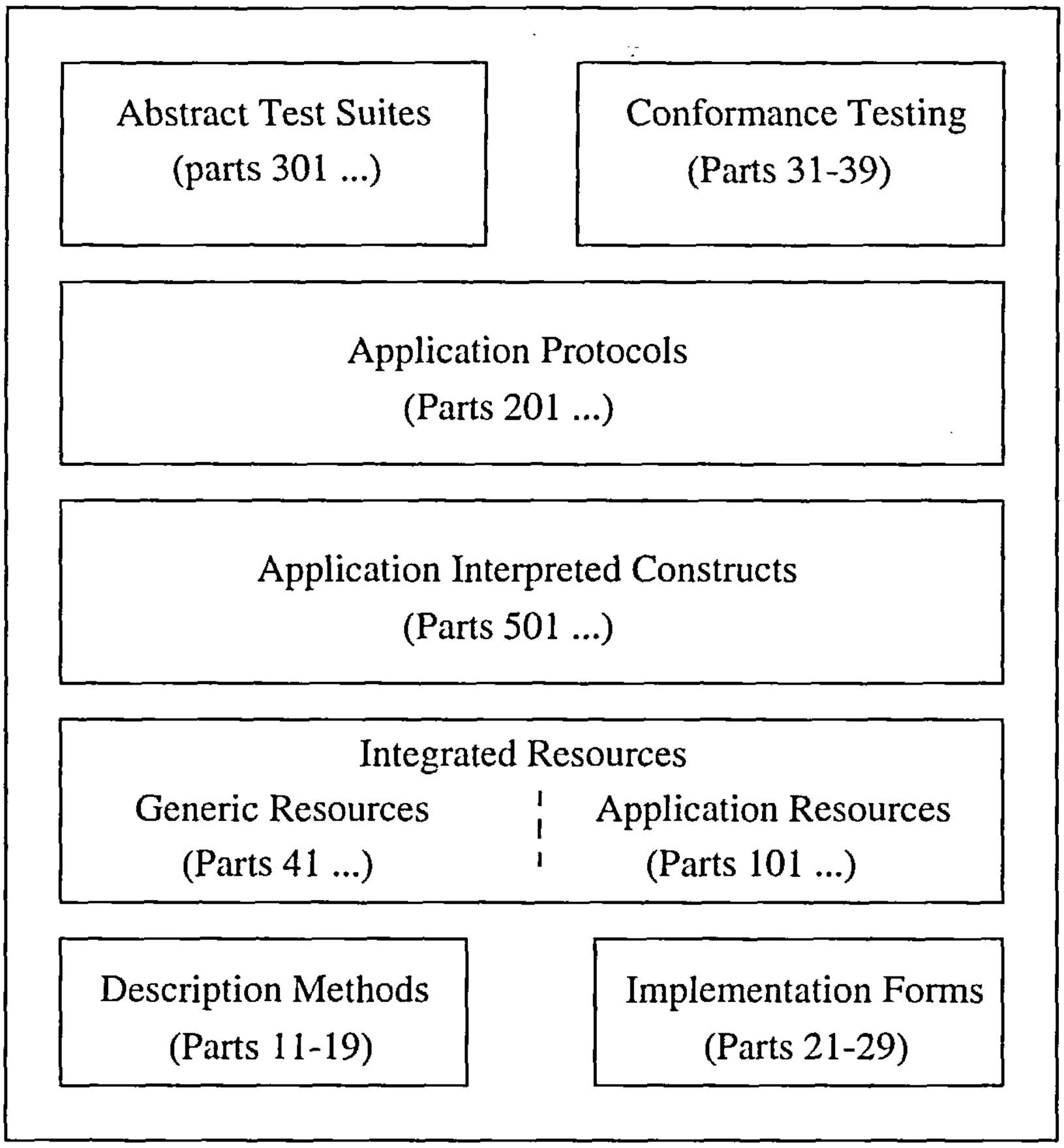


Figure 6.1 Parts structure of the ISO 10303 'STEP' standard

These techniques include the neutral file exchange structure and the application programming interfaces for computer languages such as C, C++, and IDL. The conformance testing methodology which is designed to test STEP implementations is then outlined between parts 31 and 39. A suite of integrated generic resources is outlined from parts 41 to 99 of the standard. These define data model components which are independent of specific use by application domains. Parts 101 to 199 consist of integrated application resources. These use the generic resources to build models which support groups of application domains. Thereafter, the most important parts of the standard are listed, namely the application protocols. These parts, unlike the generic resources, contain specific data models and descriptions, which have been

developed to facilitate information exchange for particular industrial contexts. For example, in the initial release of the STEP standard, application protocols, (APs), were developed for explicit draughting (AP 201) and configuration controlled design (AP 203). Many other APs have been developed since then, including some for the process industries. The abstract test suites contain the details for testing application protocols that have been developed, and are numbered from Part-301 onwards. Where there are common requirements between two or more application protocols, these are published in the Application Interpreted Constructs (AICs). These are listed from Part-501 onwards, and explicitly identify the potential for information sharing between industrial applications.

6.3 Supplementing STEP with a version system

The STEP standard often refers to the need for configuration management and Fowler (1995) highlights the problem with an example from the car manufacturing industry. However, what is actually identified is how the problem is compounded by non-standard data exchange and whilst this may be addressed by STEP, the fundamental data management problems still exist.

Examining the problem in more detail, consider the excerpt shown in Figure 6.2 from the ISO10303 standard -Part 221.

4.2.186 Version_association_between_objects

A Version_association_between_objects is an association between one-object and another that indicates one is a version of the other.

One object is a version of another if it replaces, or is intended to replace the other, where the reason for replacement is either:
- the intended or actual successor object is an improvement on its predecessor; or
- the intended successor object is more completely defined than its predecessor.

NOTE 1 - Usually, if two objects have a Version_association_between_objects between them, then there are many other application objects that are associated with both of them.
Two intended Facility objects with a Version_association_between_objects between them would usually have components in common. These would be components that were not effected by the changes leading to the new version.

EXAMPLES

250 - There can be a predecessor and a successor variant of the intended Facility that is pump P-4506-A in annex L. Both have an Assembly_of_facility association with the Piping_segment S1a.

251 - There can be a predecessor and a successor variant of the intended Facility that is the distillate transfer system in annex L. Both have an Assembly_of_facility association with the pump P-4506-A.

Figure 6.2 Extract from ISO10303 - Part 221

This part of the standard describing a relationship is intended to represent versions in a typical engineering environment, that is process engineering. However, the approach described has a few problems which are now itemised:

- If a supplier or design agent is not STEP compliant then this model fails.

• If it is accepted that all participants are STEP compliant then the versioning information is contained within the data. Rather like the problems which Kent (1991) highlights about primary keys, this data is open to corruption and therefore may be unreliable.

• If it is stated that the Version_association_between_objects can represent two different cases, yet only one association is available, then the semantics of the relationship are lost. That is there is no way of knowing whether a successor is a better alternative solution or a more detailed description.

• If a structure on the local databases has been imposed then the size of the data involved may be increased.

It is clear that version control and configuration management is better supported by a system separate to the data. It should be free of unnecessary constraints on design tools. Hence a system is proposed which utilises the STEP standard and yet provides a mechanism for version control which is orthogonal to that standard.

6.4 Version Management

The scheme for version management is agent based. The goal of the agent architecture is global consistency of data and the ability to reflect change in all models of a disparate design process. An agent communication language (ACL) is implemented on top of CORBA but does not use the CORBA event services as this is too limited (Somers, 1997). In its present form the ACL that is used is specified in VDM and employs a very limited vocabulary but it is demonstrated that it is complex enough to convey all the necessary semantics in a configuration management system.

The version mechanism is session-based, that is updates are made at the end of a user session. The mechanism is intended to be used in full lifecycle support and also as a history of the design process. In order to achieve this an agent architecture has been developed. This architecture consists of three layers which may be considered as the physical layer, the logical layer and the knowledge layer. Each layer is represented by a separate type of agent - resource, behavioural or global.

To understand the role of each type of agent an informal description of the version model based on assumptions for a general design process, is presented. The role of each agent in fulfilling that model is then described. In the next chapter formal

specifications are presented in VDM-SL describing the logic controlling the agent behaviour. An example of change management from a case study is also given in section 6.6.

6.5 Agent Descriptions

6.5.1 Messaging Design

The messaging system supports communication in both broadcast and directed modes. The implementation chooses how to send messages depending on the circumstances in which the agents communicate. Messages may be sent to all, some or just one specific agent depending on its content. The system is supported by CORBA remote object calls and so the diversity of what can be sent is almost unlimited, in fact it would be possible for agents themselves to be transferred. It is assumed that at the agent level, no messages are lost in the system. The architecture ensures this through resending and message queuing. Tests of this initial system with thousands of concurrent messages being sent and received have demonstrated that the system is robust. A centralised post office which is controlled by the global agent determines the assignment of agents to subjects and conversations in the system.

6.5.2 Resource Agent.

The following state-based model shows the function of the resource agent, in its role as the version control. The resource agent also handles physical storage optimisation and conversion from local repositories to the STEP data standard.

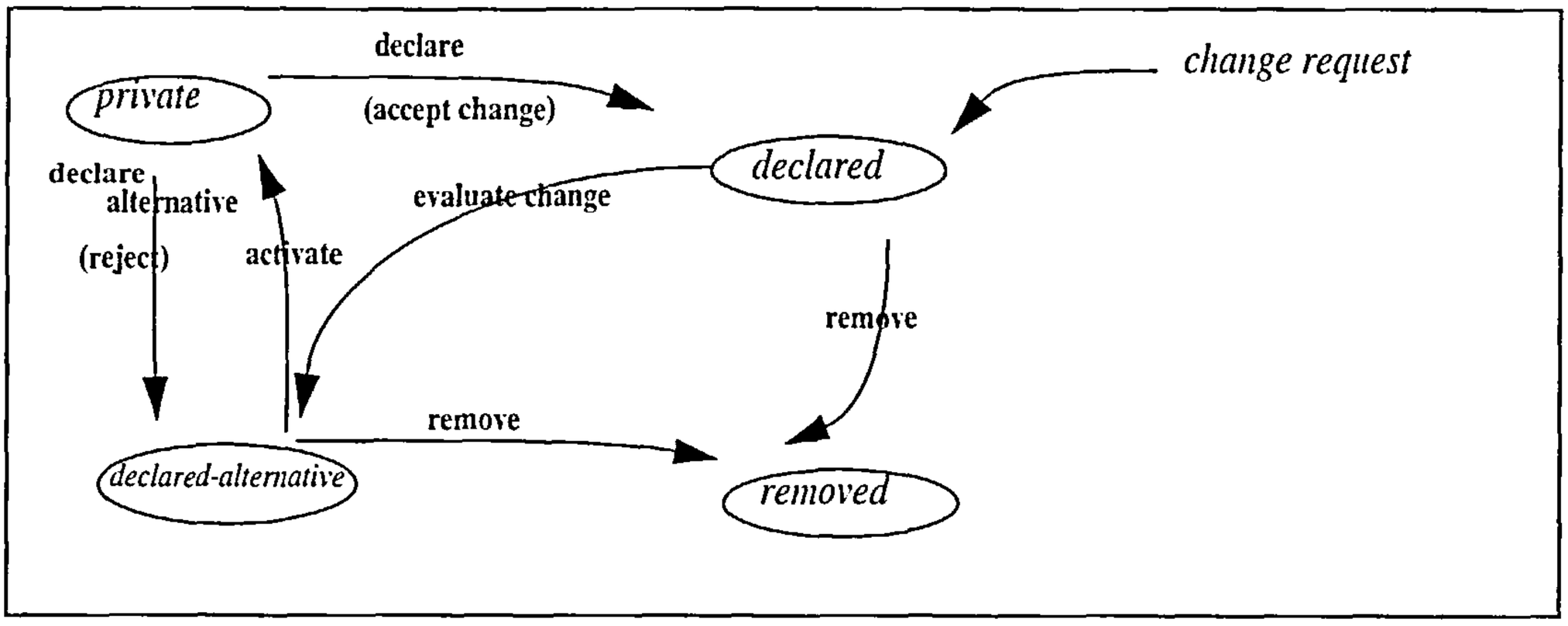


Figure 6.3 State Chart of Resource Agent Behaviour

Figure 6.3 is adapted from Krishnamurthy & Law (1997), but significantly external messages are allowed to influence the internal state messages. This is illustrated by the *change request* arrow. This message causes an alternative to be declared. This alternative now has only one route to being declared by being activated and incorporated into the private model. This process is analogous to an approval process and this is unique to the model proposed in this thesis. The VDM specification detailed later shows these state changes in the context of communicating agents.

6.5.3 Behavioural Agent

The behavioural agent is the representative of each design discipline in the agent structure. It contains rules about co-operation and negotiation with other behavioural agents and uses these to operate version control over its own resources. The behaviour is controlled by a number of state variables which represent change activity and design activity. Design activity is designated by two states:

1) **active,** it is safe for design to continue as normal or

2) **frozen,** legitimate design activity is currently postponed due to some global inconsistency or constraint violation.

Change management is represented by a unique change identifier and two sets of

variables activator, the agent which originated the change or responder agents acting in response to a change. Each agent may only be in either a frozen or active state but may be acting as either an activator or responder in numerous change conversations. An activator may be in the following states **pending(evaluation/wish)**, **pending(requirement)**, **resolving** or **recording**. A responder may be **evaluating** or **conflicting**. These states represent the following conditions,

- Pending - awaiting one or more responses to a change request
- Resolving - awaiting the result of a management decision process
- Recording - a change has been accepted and the details are being recorded; all updates are being made
- Evaluating - currently assessing the change
- Conflicting - the change violates a local design constraint

The rules for controlling these states are described in the change management examples but are formally represented in VDM-SL. Changes are described at three levels **required, evaluation** or **wish** and these levels have different effects on the state variables and different message passing priorities.

6.6 Change Management

In this section a typical design problem encountered when producing an offshore oil platform is described. An initial design has been established. This involves a process design, a plan of equipment layout, an electrical systems design and an estimation of cost. However, the original specification on which these designs are based changes fundamentally; the required export pressure of the oil being produced is changed.

Here it can be seen how the system of agents cope in a concurrent engineering environment. There are four agents in the system process design, layout design, electrical systems and cost. The process engineer is required to change the export pressure of a pump. This increases the pump size and causes the generator size to be increased. These can no longer fit in the existing deck size so a constraint is violated within the layout design. Figure 6.4 shows the model changes and the version numbering.

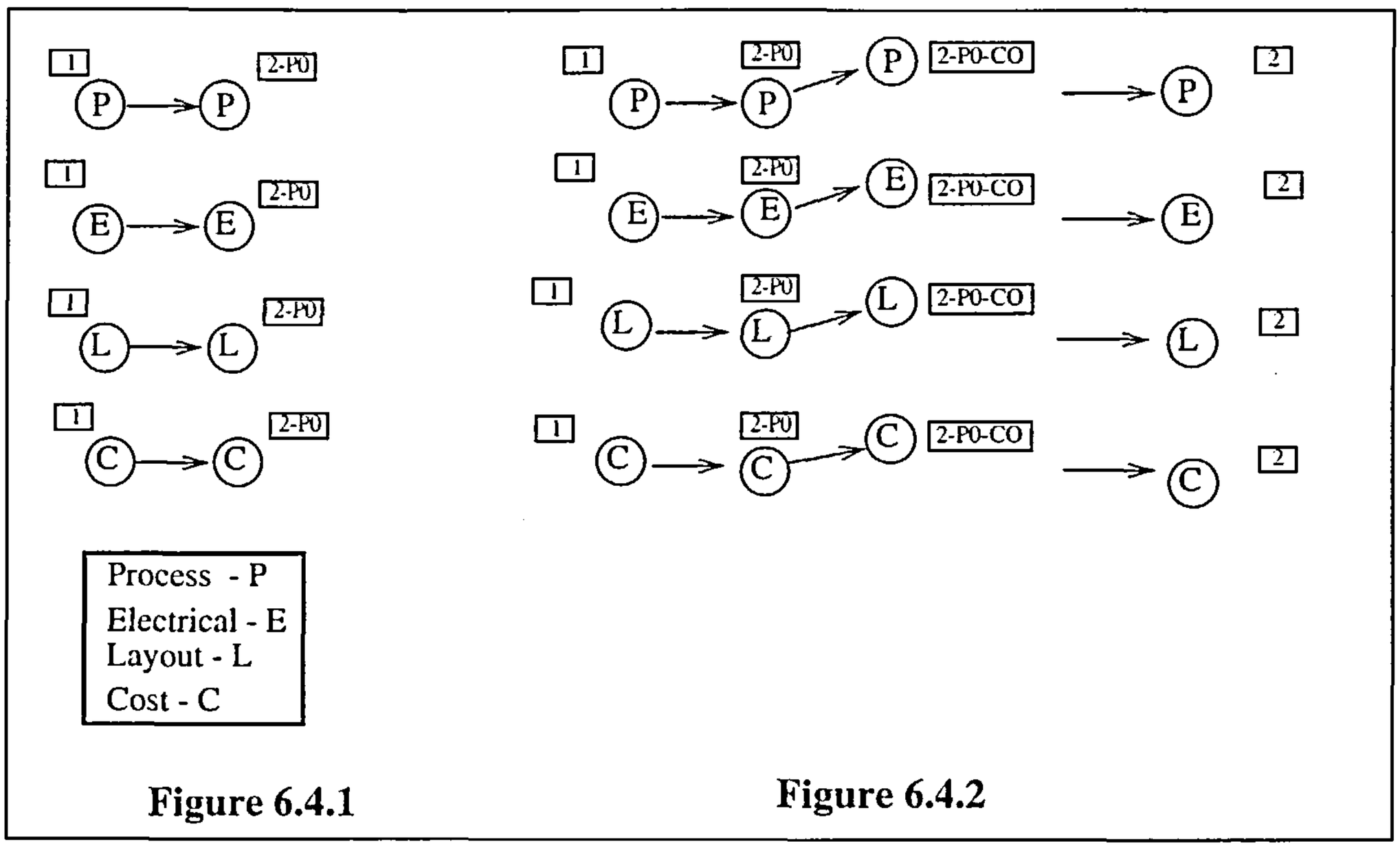


Figure 6.4.1 **Figure 6.4.2**

Figure 6.4 Labelling Scheme for a Design Conflict

Figure 6.4.1 illustrates the state of labels after the process engineer has issued his initial change. The layout agent has replied to the change request with a conflict. The activator invokes a stage of resolution and design is frozen by all agents. At this point the cost engineer produces an alternative generator which is more expensive but smaller. This situation is shown in Figure 6.4.2. All agents evaluate and respond that this change is *acceptable*. If this were not true then a management process would be invoked and a new global agent created.

Figure 6.5 below shows the various changes in the state variables at the agents involved.

Agent State	1	2	3	4	5	6
Proc - Activity	A	F	F	F	F	A
Activator	N	PR	PR	RE	RE	N
Respondor	N	N	N	N	E	N
Electrical- Activity	A	F	F	F	F	A
Activator	N	N	N	N	N	N
Respondor	N	E	N	C	E	N
Layout - Activity	A	F	F	F	F	A
Activator	N	N	N	N	N	N
Respondor	N	E	C	C	E	N
Cost - Activity	A	F	F	F	F	A
Activator	N	N	N	PR	PR	RC
Respondor	N	E	N	C	N	N

Key: A - Active; F - Frozen;
N - Normal; E - Evaluating; C-Conflict;
PR - Pending-Requirement; RE - Resolving; RC - Recorded

Figure 6.5 Table of State Variables at Four Agents

Figure 6.5 shows stage 1 with all agents active and activator and respondor states normal. After the requirement for the pump change is issued by the process designs behavioural agent, all design activity is frozen and the process activator state is set to Pending-Requirement whilst the other agents' respondor states are evaluating. The layout respondor is then set to conflict due to the constraint violation on the deck size. The activator replies to this by applying resolution and all respondors are now set to conflict. Concurrently, the cost engineer activator is set to *Pending-Requirement* as the alternative generator is selected and the process activator is now set to *Evaluating*. As the final step, all design activity is now returned and the respondor and activator states are set to normal with the cost engineer recording the decision.

7 Application of VDM Modelling to the Proposed Agent Systems

7.1 Introduction

In Chapters 5 and 6 the methodology for version control in a typical engineering design environment has been introduced. The agent architecture which will be used to implement this version control has been established. Thus far however these descriptions have been informal. In this chapter the specification of agents is formalised using the model oriented specification language VDM-SL (ISO95, 1996; Larsen & Plat , 1996a;b; Larsen & Pawlowski; 1995) .VDM-SL was chosen rather than VDM++ as it was the more mature of the two languages. The use of VDM-SL to specify the system is justified in two complimentary ways; (a) it is a model oriented specification language and (b) it is a formal specification language. The model oriented approach allows the system to be prototyped and the model of version control examined thoroughly using suitable test cases. It could be argued that these same benefits could be achieved by developing a specification in a protoyping language with the same facilities as VDM-SL but only informal semantics. However, as Larsen et al (1996) conclude, use of formal specifications is appropriate in certain circumstances, one of those being when " complex functionality is involved, when there are many choices to be made or many exceptional conditions arise". This is the case with the proposed version model. In more detailed terms the use of formality in our specification produced a number of key advantages:

i. Agent behaviour was explicitly stated as a set of rules which could be verified. The agents themselves could have been implemented directly from these rules had a suitable expert system shell been available at the time of development.

ii. The logic governing the version control needed to be detailed with extremely rigorous semantics, so any initial ambiguity was removed before the implementation phase.

iii. Agent interactions were explicitly stated and defined. These interactions stated as separate rules could be examined and verified in their own right. A prototype system may have hidden certain complexities within the communicating system.

Having determined VDM-SL as a suitable specification language, the rest of this chapter will be as follows. Section 7.2 discusses the fundamental concepts behind modelling version control as a formal specification. Section 7.3 presents critical parts of the VDM-SL specification. Section 7.4 examines issues raised by the use of a formal language. Section 7.5 describes how the specification led naturally to the implementation.

7.2 Using Formal Methods to Describe Version Control

Version control gives design engineers the ability to logically group pieces of data in some semantically relevant manner. Kilpi (1997) states that "concepts of release project, release planning and release approval need to be formalised" in order to enhance version models in software engineering. The same formalism of concepts is applied to general engineering design within this thesis.

Typically, the semantics of versions can be described by two categories; versions evolved through time and versions as alternative solutions. This can be represented simply by the directed graph shown in Figure 7.1.

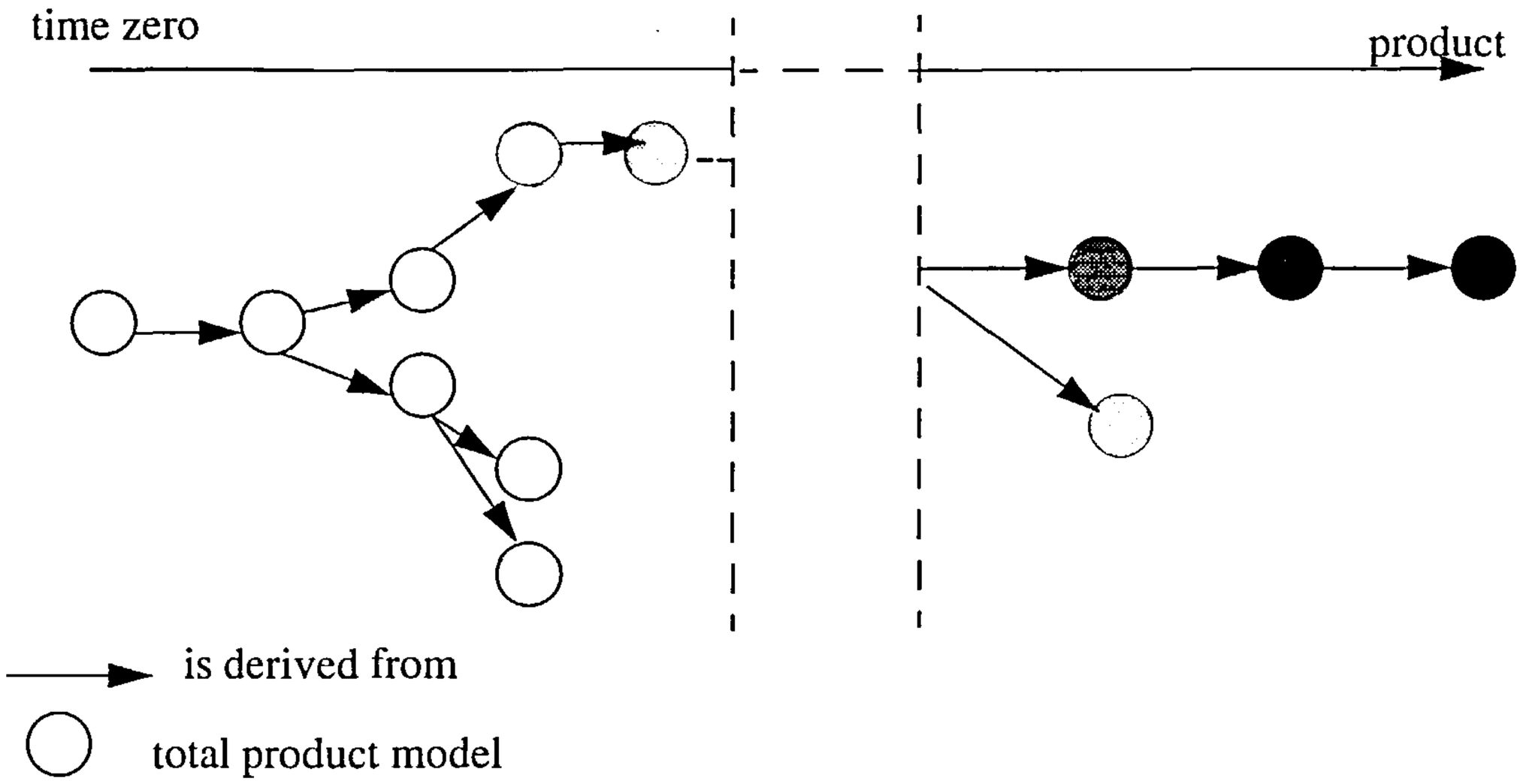


Figure 7.1 Graph representation of Version derivation

The graph shows how the total product model evolves from previous models through time and how various alternatives may be created from an original design solution. The graph also shows that a single original solution maps to a single product model representing the completed product. Hence there is a single unique path through the design version graph from time zero to the finished product.

In order to create a version mechanism which is able to replicate this version graph the rules and constraints on change in the design environment need to clearly defined. These rules need to be expressed to represent design behaviour as it occurs at the data transaction level. Definition of rules is in essence what formal modelling involves and this suggests the benefit of a formal specification of a versioning mechanism.

7.3 VDM-SL specification

In this section the application of VDM modelling is illustrated by giving selected extracts from the VDM-SL specification. VDM-SL models the system's internal state using a mathematical model in which data types represent the classes of input and output values. System functionality is modelled by functions and operations working on values of these types. The VDM-SL specification was developed using the IFAD VDM Toolbox (IFAD, 1994; Lassen, 1993; Elmstrom et al, 1994; Agerholm et al;

1998) which supports modular definition of sepcifications. Hence a description of the modules used in the specification will be introduced and then the discussion will continue by describing the basic types defined. Finally some of the critical function and operation specifications on the defined types are examined.

7.3.1 Description of Modules

The VDM-SL specification is composed of eight modules, six of which are executable and combine to give the overall behavioural agent implementation. The second two modules contain separate specifications for the resource agent and global agent. The six executable modules are defined as follows:

- Agent Version Control (AVC) -This module contains the major type definitions used in the specification, e.g. that of a DesignAgent and a VersionControl. Functions contained here are mainly invariant clauses but also contain `Record` and `UpdateAgent` which operate on the types described above.
- Messaging (MSG)-This module contains the specification of the Agent Communication. Language used in the version control system.It also contains the functions `Send` and `PostMessage` which recursively distribute messages to their recipients.
- Process Messages (PMG) -This module contains the logical specification of what happens at each agent when it receives a certain message.
- Support Functions (SFN) - The functions specified in this module are somewhat auxiliary to the main specification. They include aspects like sorting functions. They are nevertheless critical to the executable specification
- Testing (TST) - Functions to run the model against some specified test data.
- Change Control (CCTRL) -This module contains the type descriptions for any 'change' related types. The functions specified here represent a change being issued by a designer.

7.3.2 Basic Types

```
Version = <private> | <declared> | <alternative> | <recorded>;
VersionLabel:: time: nat
       alternative: AlternativeLabel;
```

```
Label :: id : AgentID
              number : nat;
AlternativeLabel = seq of Label;

DesignActivity = <Active> | <Frozen>;
ActivatorStates = <Normal> | <PendingEW> | <PendingR> | <Resolution>
| <Recording>;
RespondorStates = <Normal> | <Evaluating> | <Conflict> ;
Change = <Conflict> | <OK>;
ChangeRequest = <Required> | <Evaluation> | <Wish>;
ChangeID = token;
ReplyID :: change : ChangeID
agent: AgentID;
ReplyInfo = map ReplyID to Reply;

DesignAgent :: version : VersionLabel
          activityState : DesignActivity
          activators : map ChangeID to ActivatorStates
          respondors : map ChangeID to RespondorStates
          alternatives: map ChangeID to nat
          replies: ReplyInfo
          vState : Version;
```

The types shown above are the basis of the VDM representation. The various states that describe the version model, such as `RespondorStates`, can be easily identified. The representation of a design agent stores all current states:

`vState` represents the version access level.

`activators` and `respondors` represent the ability to play different roles in conversations about different changes.

`alternatives` represents the number of alternatives this agent has created in a given conversation.

`replies` represent agents' responses to a change request using a mapping of `AgentIDs` and `ChangeIDs` .

```
VersionControl ::
  timestamp : nat
  labels : map ChangeID to AlternativeLabel
  AgentInfo : map AgentID to DesignAgent
  ChangeDetails : map ChangeID to Change
  MessageBox : map AgentID to seq of OrderedMessage
  inv mk_VersionControl(time,labels,agents,changes, messageBox) ==
      ChangesUnique(changes)
  and AgentsUnique(agents)
  and LabelsConsistent(labels, changes)
  and AgentsConsistent(agents, changes)
  and BoxConsistent(agents, messageBox);
```

The type `VersionControl` represents the complete system state:

`timestamp` represents the current global design time.

`labels` maps each design change to a unique prefix which is used in the labelling scheme.

`AgentInfo` maps a unique agent identifier to a design agent.

`ChangeDetails` again uses a mapping to represent the design changes.

`MessageBox` holds a sequence of ordered messages for each unique `AgentID`. The invariant on this type records the following restrictions:

`ChangesUnique` - each `ChangeID` is unique in the domain of `ChangeDetails`

`AgentsUnique` -each `AgentID` is unique in the domain of `AgentInfo`

`LabelsConsistent` - the domain of labels is a subset of the domain of `ChangeDetails`

`AgentsConsistent` - the domains of `activators`, `respondors`, `alternatives` of each agent in the domain of `AgentInfo` are all subsets of the domain of `ChangeDetails`

`BoxConsistent` - the domain of `MessageBox` matches exactly the domain of `AgentInfo`

7.3.3 Language of Messages

```
IssueChange:: agent : AgentID
              change : ChangeID
              type : ChangeRequest;

ReplyOK::     agent : AgentID
              change : ChangeID;

ReplyConflict::agent : AgentID
              change : ChangeID;

ApplyResolution::agent: AgentID
                change:ChangeID;

ResolveConflict::agent : AgentID
                change : ChangeID;

ChangeRecorded::agent : AVC`AgentID
                change : CCTRL`ChangeID;

ReactivateChange:: change : CCTRL`ChangeID;


--Definiton of vocabulary
```

```
Messages = IssueChange | ReplyOK | ReplyConflict | ResolveConflict |
                ApplyResolution | ChangeRecorded | ReactivateChange;
```

The message type denotes the content of the message. Each type contains the referred `ChangeID` and the name of the originating agent. The message `IssueChange` also contains the `ChangeRequest` type representing the meaning of the change.

```
MessageType = nat
inv mt == mt <= 3 and mt >= 1;

OrderedMessage :: priority : MessageType
content : Messages;

Send : MessageType * Recipients * Messages * map AgentID to seq of
OrderedMessage  ->
 map AgentID to seq of OrderedMessage
Send(type, toMsg, msg, mbox) ==
 let agent in set toMsg in
PostMessage( agent , msg, type, mbox) ;
```

The `OrderedMessage` type allows messages to have different priorities. This priority is represented as a natural number between 1 and 3 in order to simplify the ordering functions. The `Send` function posts a message with a priority to all the agents specified in the `Recipients` term. This message is queued in order at each agent.

```
ProcessCommand: MSG`Messages * AVC`AgentID * AVC`VersionControl ->
                  AVC`VersionControl
ProcessCommand(msg, processingAgent, vc) ==
 cases msg :
   mk_MSG`IssueChange(agent, change, type) ->
        CreateVersion(SFN`Prioritise(type),agent,change,
                      processingAgent,vc),
   mk_MSG`ReplyOK(agent, change, type) ->
        ReplyYes(type, agent, change, processingAgent,vc),
   mk_MSG`ReplyConflict(agent,change,type)->
        ReplyNo(type, agent, change, processingAgent,vc),
   mk_MSG`ApplyResolution(agent,change) ->
        ApplyRes(agent, change, processingAgent,vc),
   mk_MSG`ResolveConflict(agent,change) ->
        Resolve(agent,change,vc),
   mk_MSG`ChangeRecorded(agent,change) ->
        RecordChange(agent,change,vc),
   mk_MSG`ReactivateChange(change) -> Reactivate(change, vc),
   others ->vc
 end;
```

`ProcessCommand` shows how a message gets interpreted by an agent. The interpreting agent is represented by the term `processingAgent`. The cases statement decides, based upon the message type, what action should be taken by calling the appropriate function. These functions update the agents' state variables and depending on the message, the change and the processing agents state respond accordingly. The next section gives an example of one of these processing functions and the rules it applies.

7.3.4 Implementation of Version Control

The function described below `ApplyRes` is invoked when an agent receives a message from an activator telling it that conflict resolution has been applied.

```
ApplyRes:AVC`AgentID*CCTRL`ChangeID*AVC`AgentID*AVC`VersionControl -
>
AVC`VersionControl
ApplyRes(agent, change, thisAgent, vc) ==
let conflict: AVC`RespondorStates = <Conflict>,
  a = vc.AgentInfo(thisAgent) in
  mk_AVC`VersionControl(vc.timestamp,vc.labels,vc.AgentInfo++
   {thisAgent |-> mk_AVC`DesignAgent(a.version,a.activityState,
   a.activators, a.respondors++{change|->conflict},
   a.alternatives,a.replies,a.vState)},vc.ChangeDetails,
  vc.RequiredChanges, vc.totalAlternatives,vc.MessageBox);
```

In this case the action is fairly straight forward, all responder states are set to `conflict` and design activity is `frozen`. No other changes to the version control model is made and no messages issued to other agents.

The following excerpt is from the function `CreateVersion` which is called as a response to an `IssueChange` message

```
if vc.ChangeDetails(change) = <Conflict> then
if type = 1 then
  mk_AVC`VersionControl(vc.timestamp, vc.labels,
  vc.AgentInfo  ++ { thisAgent |-> mk_AVC`DesignAgent(vers2,frozen,
  da.activators++{change|->norm}, da.respondors++{change|->eval},
  da.alternatives++{change |-> alt}, da.replies,alternative)},
  vc.ChangeDetails, vc.RequiredChanges, vc.totalAlternatives,
  MSG`Send(type,recip,mk_MSG`ReplyConflict(thisAgent, change, type),
          vc.MessageBox))
```

In this excerpt `thisAgent` is evaluating a change which causes a conflict. The local state variables are set accordingly, that is, `respondor` is set to evaluating as shown and `activity` is frozen as the change is considered. Concurrently a `ReplyConflict` message is sent to the `activator` on this change, informing it of the effect of this change on this agents local model. It should be noted that the priority of the reply is the same as the original change, shown by the variable `type` in the example.

The following function `Record` shows an agreed change being recorded and the agent states being updated.

```
Record: MSG`MessageType * AgentID * VersionLabel *CCTRL`ChangeID *
          VersionControl -> VersionControl
Record(type, agent, version,change,vc)  ==
let     da = vc.AgentInfo(agent),recip = dom vc.AgentInfo  {agent} in
if type = 1 then
        mk_VersionControl( vc.timestamp, vc.labels,vc.AgentInfo++
        {agent|->mk_DesignAgent (da.version,da.activityState,
        da.activators++ {change |-> <Recording>},da.respondors,
        da.alternatives,da.replies,<recorded>)},
        vc.ChangeDetails,vc.RequiredChanges,vc.totalAlternatives,
        MSG`Send(type,recip,mk_MSG`ChangeRecorded(agent,change),
        vc.MessageBox))
    else
        mk_VersionControl( vc.timestamp, vc.labels,vc.AgentInfo,
        vc.ChangeDetails,vc.RequiredChanges,vc.totalAlternatives,
        MSG`Send(type,recip, mk_MSG`ChangeRecorded(agent,change),
        vc.MessageBox))

 pre {vc.AgentInfo(a).vState | a in set dom vc.AgentInfo} =
                                                {<declared>};
```

The activator on the change `agent` has its state changed to `Recording` with the current version label. It also informs all other agents that this change has now been recorded.

The `pre` condition states that an agent must be in the state `declared` before it can be `recorded`.

The following function `Resolve` represents the outcome of a conflict resolution process.

```
Resolve: AVC`AgentID * CCTRL`ChangeID * AVC`VersionControl ->
        AVC`VersionControl
Resolve (agent, change ,vc) ==
let version= mk_AVC`VersionLabel(vc.timestamp,
                SFN`ChooseAlternative(vc.labels(change))),
                type :MSG`MessageType =1
in
        AVC`Record(type,agent, version, change, vc);
```

This function simply states that a solution from the current set (`change`) must be chosen. This is controlled by `ChooseAlternative` and then this design decision is recorded

7.3.5 Resource Agent

The resource agent model demonstrates the internal state changes and the external messages that influence it. The complexity of queues and priorities is not included in this state based model.

```
Messages = NewVersion | Retrieve | EvaluateChange | Resolve;

state ResourceAgent  of
        version: VersionLabel
        objects : set of ObjectID
        changes: map ChangeID to Deltas
        modelStates: map VersionLabel to VersionState
        expressmapping : map UniqueObject to Express
inv mk_ResourceAgent(version, objects, changes,states,express ==
        ObjectsUnique(objects)
        and ChangesUnique(changes)
        and Consitent(changes, states)
init ra == ra = mk_ResourceAgent(<a0>, {},{\->}, {|->}, {|->})
end;
```

The resource agent consists of a version, a list of objects, a mapping between change identifiers and the changes, a mapping to represent the version states as mentioned earlier and a mapping representing the conversion from local object to EXPRESS model. The invariant functions ensure that objects and changes have unique identifiers and that the domain of `modelStates` is a subset of `version`.

```
StoreDeltas(chng : ChangeID, delta:Deltas)
ext wr objects
      wr changes
post objects = objects~ union ProcessDelta(deltas)
                  and changes = changes~ munion {chng |-> deltas};
```

The function `StoreDeltas` processes changes and stores them. The `post` condition states that the set of objectIDs after the change is equivalent to those before with the application of the changes and also that the value of changes now includes the old value plus the new set of changes.

```
RetrieveObject(o:ObjectID, version: VersionLabel) obj: Express
ext rd expressMapping
post obj = expressMapping(mk_ObjectMarker(o, version));
```

`RetrieveObject` returns an EXPRESS model of the given object selected from the version set by the identifier `version`.

```
Activate(version:VersionLabel)
ext wr modelStates
pre modelStates(VersionLabel) = <alternative>
post modelStates(VersionLabel) = <private>;
```

`Activate` demonstrates an internal state change caused by an alternative being activated. The `pre` and `post` conditions control the state change.

7.3.6 Global Agent

The global agent model is the most straight forward as it models simply the agents ability to manage the consistency of global objects and relationships between them.

```
state GlobalAgent of
        configuration : VersionLabel
        objects : map ObjectID to Object
        relationships : map RelationID to Relation
        agents : set of Agent
inv mk_GlobalAgent(configuration,objects,relationships,agents) ==
        ObjectsUnique(objects) and
        RelationShipsUnique(relationships)  and
        ObjectRelationsConsistent(objects, relationships)
init ga == ga = mk_GlobalAgent(<a0>, {|->} , { |->}, {})
```

The global agent stores the global configuration, the list of global objects and the relationships between these objects. It also stores a directory of the participating agents in the variable `agents`. The invariants make sure that id's are unique within the global agent and that the cross referencing between objects and relationships are consistent.

```
CreateObject(oid:ObjectID) ==
        (objects := objects munion {oid |-> object})
ext wr objects
pre ObjectID not in set dom objects;

DeleteObject(oid:ObjectID) == (objects := {oid} <-: objects)
ext wr objects
pre oid in set dom objects;
```

`DeleteObject` and `CreateObject` show the operation of the Global Agent, similar functions exist for the relationships in the global agent. The `pre` conditions ensure that the consistency is maintained by not allowing duplication of object identifiers or removal of object identifiers not in the domain.

7.4 Representing concurrency in VDM-SL

A major issue in the development of the VDM -SL specification was the representation of concurrency. Concurrent design as a feature of modern engineering, needs careful consideration. It was decided that a time-slicing algorithm would be used to unwind the concurrency. This would then be represented within the VDM-SL specification by the message processing module in the manner described:

- Messages are sent and received as discrete events.
- Messages are queued at the receiving agent where message priority determines where messages are placed in the queue.
- The function `ProcessMailboxes` then cycles through each agent reading the top message and acting on it. This occurs recursively.

Whilst the time-slicing algorithm described adequately models concurrency it raises issues when verifying the model. Not only do rules controlling agent behaviour need to be verified but also the scheduling of events needs to be verified as a suitable representation of the real system. Given this problem it may be suggested that a suitable temporal logic could have been used such as one of those described by Fisher (1996). However, there are three reasons why these logics were not used:

- The relative immaturity of the *proof theory* would have added unnecessary complications into model verification
- The mathematical complexity of the language would have precluded many design engineers from discussions about the model, a necessary part of the model development.
- A lack of tool support for such logics.

Another temporal aspect of the system is that of real design time rather than system time. This is modelled by the timestamp variable in the verisonControl type. Referring to the graph representation of the model shown in Figure 5.3, this variable could be described as an edge between two sequential nodes on the horizontal time line.

7.5 Specification to implementation

In order to take the Agent specifications and convert them into code written in C++ a common framework between VDM-SL and C++ must be established (Fröhlich & Larsen, 1996). The problem is that the values in these two worlds have different representations and in order for them to communicate it is necessary to convert values from the specification world into the code world. The IFAD VDM-SL Toolbox defines such a common framework and supports automatic code generation. However, the code genrated could not be used directly for two reasons;

- The real system is distributed
- The real system runs on different platforms

Hence, the code could have been used to produce a simulation of the real world system but not the real system itself. However, the mapping from VDM-SL to C++ will be illustrated by using examples of the C++ code and comparing it with the VDM-SL specifcation. It is intended to demonstrate informally that the VDM-SL specification leads obviously to the final implementation, and supports the initial argument for the use of VDM.

Figure 7.1 shows the C++ header file for the implementation of the ResourceAgent along with extracts from the VDM-SL specification. It is evident in figure 7.1 that the VDM-SL specifcation mirrors the C++ header almost exactly. For example consider the VDM-SL operation `EvaluateChange(chng:ChangeID, vers:VersionLabel)`. In C++ the function signature is almost identical with the exception of an extra parameter of type `ChangeRequest`. This parameter is added as it contains the real detail of the design change, an aspect which did not need to be considered in the formal model. The use of abstract functions in the C++ code gives an indication that the implementation is not as important as the interface. That is an explanation of why the ResourceAgent specifcition module is non-executable.

In Figure 7.2 the particular implementation of one function, `CreateChange`, is considered. Unlike the previous example this function is part of the executable specification. It can be easily seen how the VDM-SL rules map well into the procedural `if...then` constructs of the C++ language. However, it can be seen that

the `pre` condition represented in VDM-SL is not easily represented in C++. It may be possible to represent such conditions via the exception mechanism in C++. However in this particular case it can be demonstrated satisfactorily that the `pre` condition is never contradicted through simulating the model with the VDM toolbox.

7.6 Summary

The VDM specification presented in this chapter represents two aspects of the version model

- the formalism of the model
- a prototype for testing the model

The formal aspect is dealt with sufficiently by the VDM specification. However, as a prototype there are certain deficiencies. As discussed VDM does not model with complete certainty the temporal or distributed nature of the problem domain. Not only this, but it is also necessary to investigate how well particular abstractions in the model may be applied to real engineering tools. These issues are resolved in the following chapter where this formal specification is developed into a fully operational prototype system.

```
#ifndef RES_AGENT_HH
#define RES_AGENT_HH

#include "agent_i.hh"
#include "delta.hh"
#include "ChangeRequest.hh"

typedef int ChangeID;
typedef char* ObjectID;
typedef char* VersionLabel;

const int ARRAY_SIZE = 100;
typedef char  DeltaRoot[ARRAY_SIZE];


struct VersionState
{
VersionLabel version;
enum {private_ =0, declared =1 , alternative =2, removed=3 } state;
};


//class ResAgent: public virtual Agent_i
class ResAgent
{
public:
ResAgent( const char* name, const char* data);
~ResAgent();
void setCurrentVersion(VersionLabel currentVersion_);
 virtual void CreateVersion(ChangeID cid, VersionLabel version) =0;
virtual void RetrieveObject(ObjectID oid, VersionLabel version) =0;
virtual void EvaluateChange(ChangeID cid, ChangeRequest change
, VersionLabel version) =0;
virtual void ResolveChange(ChangeID cid, VersionLabel version)
=0;



void Activate(VersionLabel version);
void Declare(VersionLabel version);
void Remove (VersionLabel version);
void DeclareAlternative(VersionLabel version);
void Evaluate(VersionLabel version);



protected:
virtual void StoreDeltas(ChangeID cid, VersionLabel version) =0;

ObjectID objects[ARRAY_SIZE];
VersionLabel currentVersion;
DeltaRoot changes[ARRAY_SIZE];
VersionState modelStates[ARRAY_SIZE];
 int versionCount;
enum {private_ =0, declared =1 , alternative =2, removed=3 };
};

#endif;
```

```
module RESOURCE_AGENT
exports all
definitions

types
ObjectID = token;
ChangeID = token;
VersionLabel = token;
VersionStates = <private> I <declared> I <alternative> I
<removed>;
```

```
CreateVersion(chng : ChangeID, vers : VersionLabel,
deltas : Deltas)
ext wr version
     wr objects
     wr changes
post version = vers and Declare(vers) and StoreDel-
tas(chng, deltas);
```

```
RetrieveObject(o:ObjectID, version: VersionLabel) obj:
Express
ext rd expressmapping
post obj = expressmapping(mk_UniqueObject(o, ver-
sion));
```

```
EvaluateChange(chng: ChangeID, vers : VersionLabel)
ext wr version
     wr objects
    wr changes
post version = vers and Evaluate(vers);
```

```
ResolveChange(chng: ChangeID, vers : VersionLabel)
ext wr version
     wr objects
    wr changes
post version = vers ;
```

```
--Operations which state the allowable state transitions
Activate(vers:VersionLabel)
ext wr modelStates
pre modelStates(vers) = <alternative>
post modelStates = modelStates~ ++ {vers I-> <private>};
```

```
R
DeclareAlternative(vers:VersionLabel)
ext wr modelStates
pre modelStates(vers) = <private>
post modelStates = modelStates~ ++ {vers I-> <alterna-
tive>};
```

```
state ResourceAgent  of
version: VersionLabel
objects : set of ObjectID
changes: map ChangeID to Deltas
modelStates: map VersionLabel to VersionStates
```

```
-Auxiliary operations
StoreDeltas(chng : ChangeID, deltas: Deltas) result:[bool]
ext wr objects
     wr changes
post objects = objects~ union ProcessDeltas(deltas)
 and changes = changes~ munion {chng I-> deltas};
```

Figure 7.1 ResourceAgent C++ Header file with VDM-SL specifcation.

```
CreateChange: AVC'AgentID * ChangeID * ChangeRequest* Change * AVC'VersionControl -> AVC'VersionControl
CreateChange(agent, change ,type, details, vc) ==
  if ChangeExists(change, vc.ChangeDetails) then   -- First Determine if change already exists
   let alts = vc.totalAlternatives(change) in
    let derChange = SFN'DeriveChange(change, alts),   label = vc.labels(change), newAlts = alts+1,
        da = vc.AgentInfo(agent), alt = vc.AgentInfo(agent).alternatives(change) in
      let newVc = mk_AVC'VersionControl(vc.timestamp,vc.labels++ {derChange |->label},
                    SFN'InitialiseChange(derChange,details,vc,   dom vc.AgentInfo, agent),vc.ChangeDetails,
                vc.RequiredChanges,  vc.totalAlternatives ++ {change |-> newAlts} ++ {derChange |-> 0}, vc.MessageBox)
       in
        let newVC2 = mk_AVC'VersionControl(newVc.timestamp, newVc.labels , newVc.AgentInfo++ {agent |->
                      mk_AVC'DesignAgent(da.version, da.activityState, da.activators,da.respondors, da.alternatives
                      ++{ change |->alt+1},da.replies, da.vState)},  newVc.ChangeDetails,   newVc.RequiredChanges,
                      newVc.totalAlternatives,newVc.MessageBox) in
            CreateResponse(agent,derChange,type,details, alt, newVC2)
  else
   let newVc = mk_AVC'VersionControl(vc.timestamp,vc.labels, SFN'InitialiseChange(change, details, vc,
                dom vc.AgentInfo,agent), vc.ChangeDetails, vc.RequiredChanges, vc.totalAlternatives++
                {change |->1}, vc.MessageBox)   in
     CreateNewChange(agent, change, type ,details, newVc)

pre forall c in set dom vc.ChangeDetails &   card GetActivators(vc.AgentInfo, c) = 1;
```

```
void BehaviouralAgent::CreateChange(ChangeID cid, ChangeRequest change,ChangeType typeRequest)
{
if (cid <= CIDcount) // ChangeExists?
{
      int derivedCID = DeriveChange(cid);
designAlternatives[cid]++;
//derive change gets old label
//derivechange alternatives =0;
designAlternatives[derivedCID] =0;
CreateResponse(derivedCID, designAlternatives[cid], change);
}
else
{designAlternatives[cid] = 0;
CreateNewChange(cid, change, typeRequest);
}
}
```

Figure 7.2 : Comparison of C++ function definition and VDM-SL specifcation.

8 Description of Prototype system

8.1 Introduction

This chapter describes the detailed implementation of the prototype system which has been built and some results from that prototype. In Section 8.2 how an object-based client server architecture was adapted into a logic processing message-based system is described. In the following sections the agent architecture introduced in Chapter 5 and formally specified in Chapter 7 will be expanded upon. The practical problems encountered are illustrated, the approach to these problems described and finally related to a real engineering environment. Each type of agent is discussed in terms of its obligations within the whole system and how it fulfils these. In Section 7.4 it is shown how the system would be applied in an existing design environment. Finally the overall system architecture is given. Chapter 9 describes in more detail the output from the prototype in terms of two case studies.

8.2 Asynchronous Messaging Architecture

As described in Chapter 4, a key feature of an agent is the ability to communicate its intentions. This communication should be asynchronous and based on messages which may be interpreted. However, CORBA is a distributed object server architecture. Clients request information via object method calls on remote objects. The information passing between clients and server objects is neither asynchronous or intelligent.

The CORBA standard specifies a *service* for passing messages. This is called the Event Service. However, it was decided not to use this service for implementing the messaging. The reasons for this decision were quite straightforward. Firstly, when the architecture was initially proposed, the CORBA implementation being used (Orbix 1.3.5) did not have available a full implementation of the Event Service. Secondly whilst OrbixTalk^{TM} is now widely available it is still less readily available than the basic CORBA standard. Thirdly, it is more difficult to reconcile different ORBs via the Event Services. For example the Java ORB with JDK 1.2 does not support the Event

Services. Finally and most critically, the FIPA standard proposes only a function signature for messages using CORBA and this does not include use of the Event Service.

The messaging architecture is designed to fulfil two main requirements

- All messages arrive at their destination
- Message sending does not cause any redundancy within the sending agent

In order to achieve message passing, clients must act as senders and servers as receiving agents. This is shown in Figure 8.1

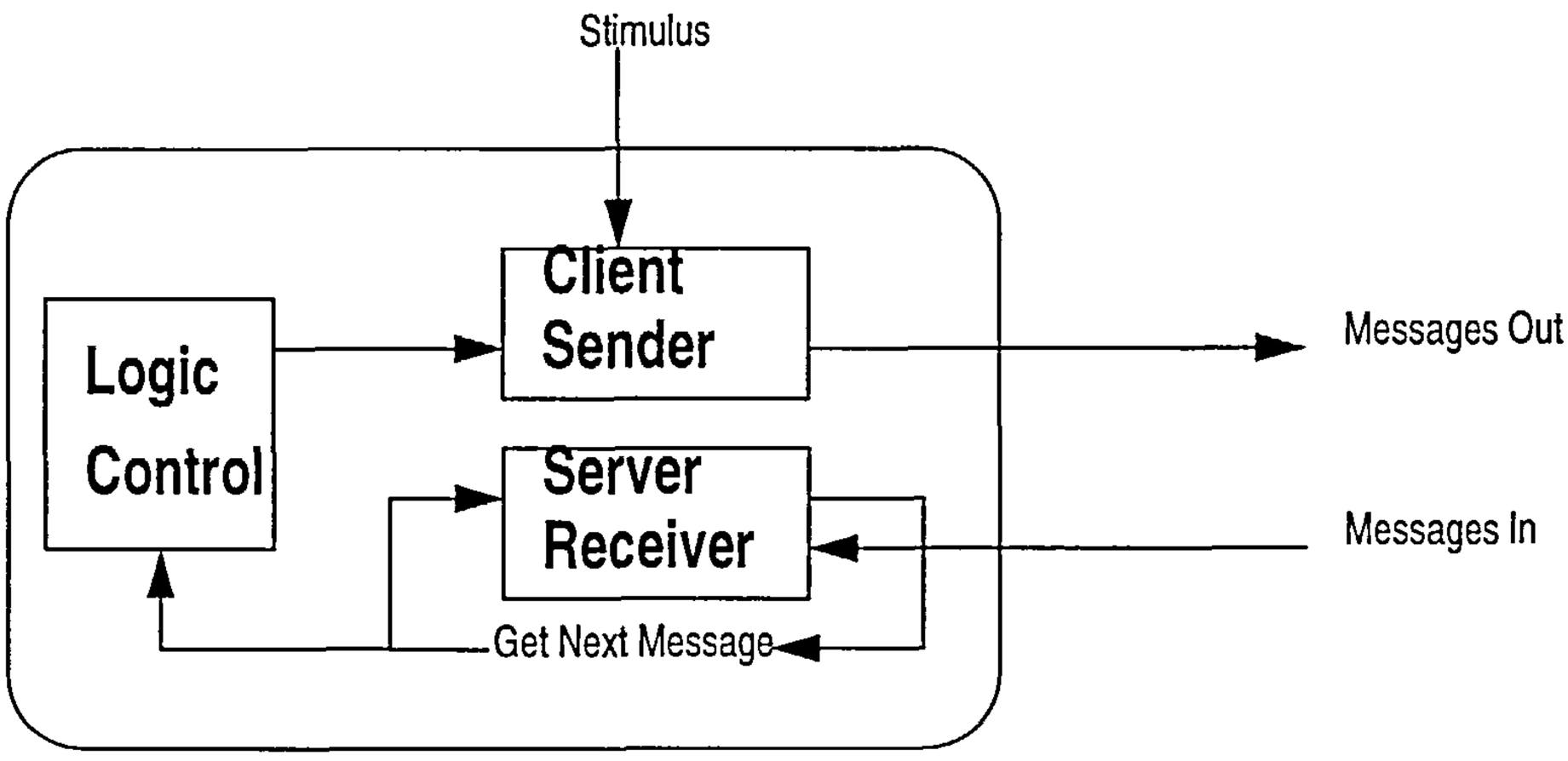


Figure 8.1: Agent Implementation using CORBA

Each agent contains a mailbox into which incoming messages are queued in order of priority. The receiving agent reads each message in turn as shown. The controlling logic deals with this message and if necessary, sends an appropriate response message. The *Stimulus* arrow represents a source, other than the messaging system, acting on the agent

To avoid the possibility of deadlocks, clients do not directly bind to the receiving agents server and leave the message. An intermediate server called the post office is utilised. This server also knows the server addresses of each agent thereby simplifying

the communication protocol. In order for agent A to send a message to agent B the following protocol occurs. Agent A first binds to the post office and using a predefined syntax sends a message and specifies the recipient Agent B. The post office server creates a separate process for each client binding and this process remains alive until the message is placed within the recipients mailbox. Therefore the work of re-sending messages is undertaken by the *postoffice* process and not by the client agent.

8.3 Presenting the Agent System

In Chapter 5, Figure 5.2 illustrates the layers of agents in the system. The top layer is the knowledge layer where the global agent resides. In the middle layer is the logical control behaviour and at the bottom is the physical layer. Each of these layers is represented by a different type of agent as described previously. The physical implementation of these agent types starting with the lowest layer - the resource agent, is now given.

8.3.1 Resource Agent (RA)

The resource agent acts as a wrapper to the local design data store whether that is a design tool or design database. In Section 5.2.1 the obligations (Shoham, 1993) that the resource agent has to the overall version control mechanism are stated. Each of these are now addressed in turn, detailing how the resource agent fulfils each of the obligations.

- *Manages a purely structural EXPRESS model of local information*

This obligation essentially states that the schema used for **data** transactions will be STEP compliant. This is achieved in the following manner. The appropriate Application Protocol (AP) is chosen and the Application Interpreted Model (AIM) is acquired which gives the ASCII text EXPRESS representation of the AP. ST-Developer is then used to generate the SDAI CORBA/IDL bindings. This binding is an 'early-bound' interface. This means that the syntax of the data structures is defined at compile time. An impact of this is that a change in the EXPRESS model could require a complete recompilation. This is a restriciton of the toolset.

The IDL produced from the tool is then implemented via the Orbix/IDL binding essentially mapping from EXPRESS to CORBA. The resource agent is linked with the IDL stub object code at run time ensuring all transactions are STEP compliant.

- *Appends version information (that is tags or labels)*
- *Stores local information*
- *Accesses local information*

To show how the resource agent fulfils these three obligations lets examine the implementation of a typical resource agent, the *Layout Resource Agent*. The layout resource agent wraps a tool developed in the Engineering Design Centre for automatically generating layouts of complex products(Smith, 1998; Hills & Smith, 1997; Smith et al, 1996). This tool takes a set of dimensions and co-ordinates as input along with a 2-d frame and some knowledge about the equipment, and produces an optimised layout based upon simulating annealing. Inputs and outputs are held in flat UNIX files. In order to store the local information and versioning information there exists an indexing structure within the agent. This stores a version label and a filename as a tuple within the agent. Optimisation of storage is done by storing changes between files as flat files from which later/earlier versions may be generated. Retrieval of an object binds with the EXPRESS AP as described previously and produces a CORBA object which is in STEP compliant format. Figure 8.2 below shows the static model

(UML) of the Resource Agent Architecture.

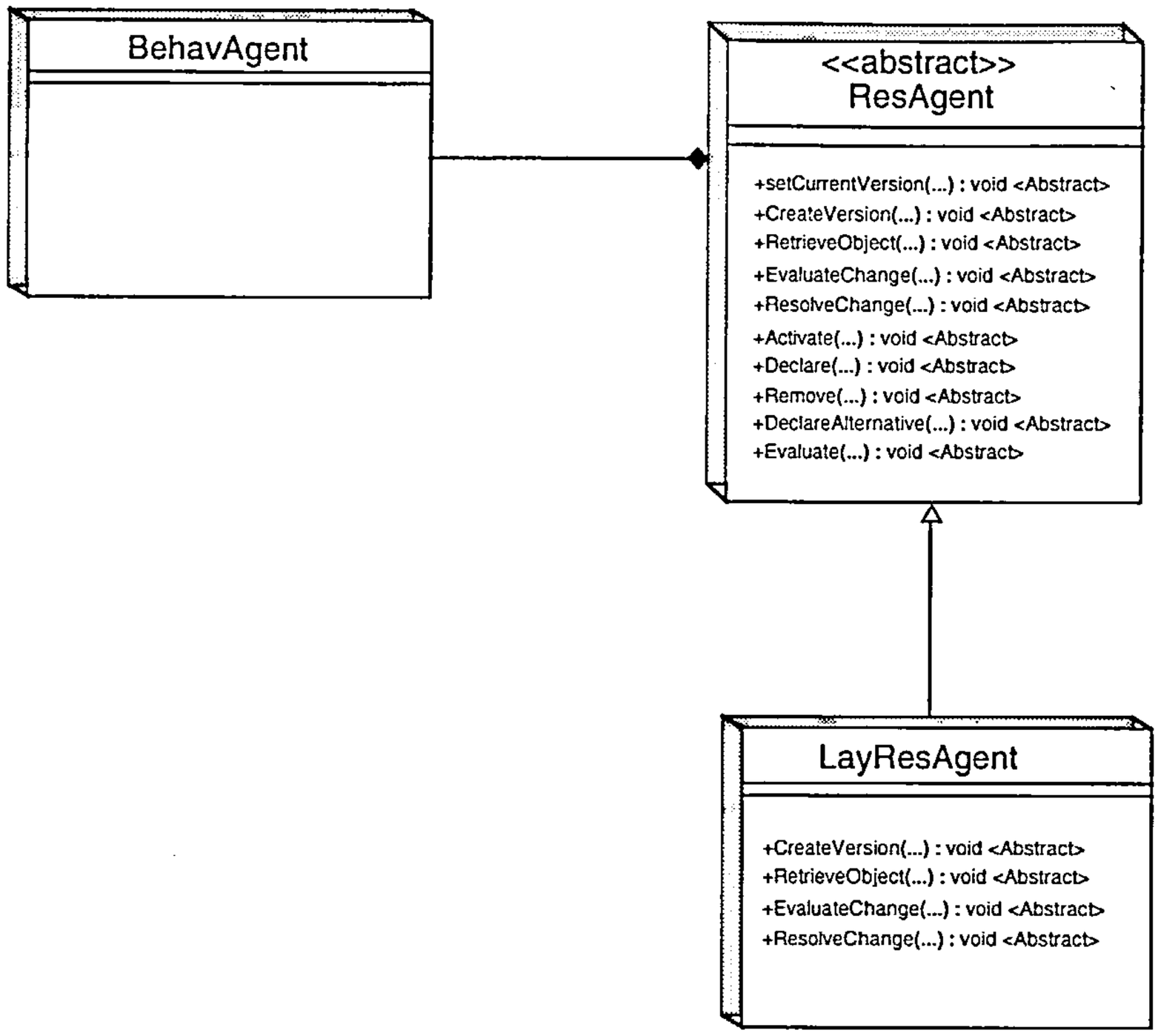


Figure 8.2 UML Static model of the Resource Agent

8.3.2 Behavioural Agent (BA)

The Behavioural Agent implements the logical behaviour specified in the VDM modules ProcessMessages.vdm, VersionControl.vdm and ChangeControl.vdm. This behaviour is implemented in C++ in the following manner:

- *Communication with other design disciplines*

The implementation is described in Section 9.2 and the relationship of this implementation to the BA is shown below.

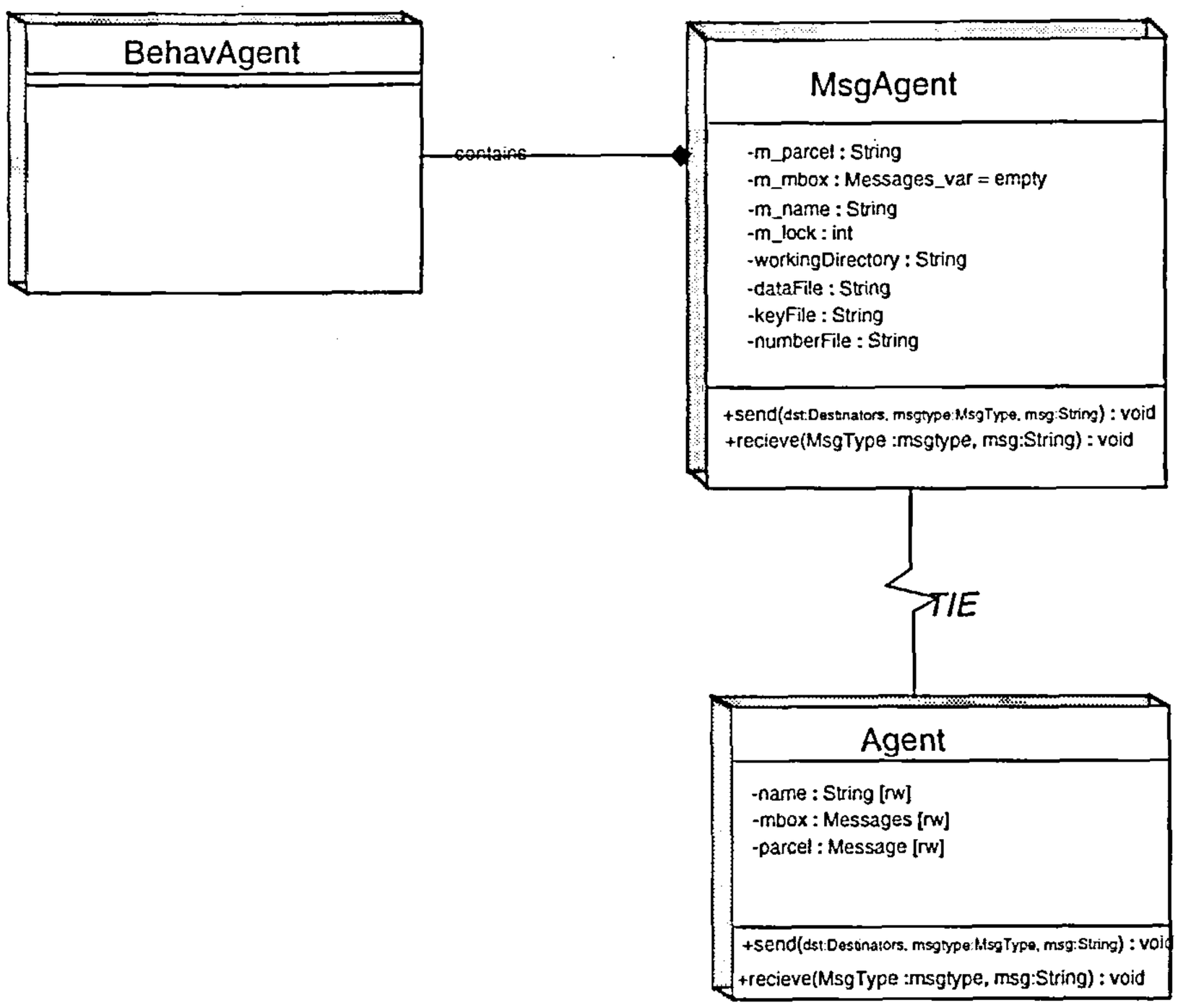


Figure 8.3 UML Static Model showing MsgAgent Class

A BA is associated with a msgAgent which contains the CORBA implementation. The operation of the communication is shown in the Event Sequence diagram below.

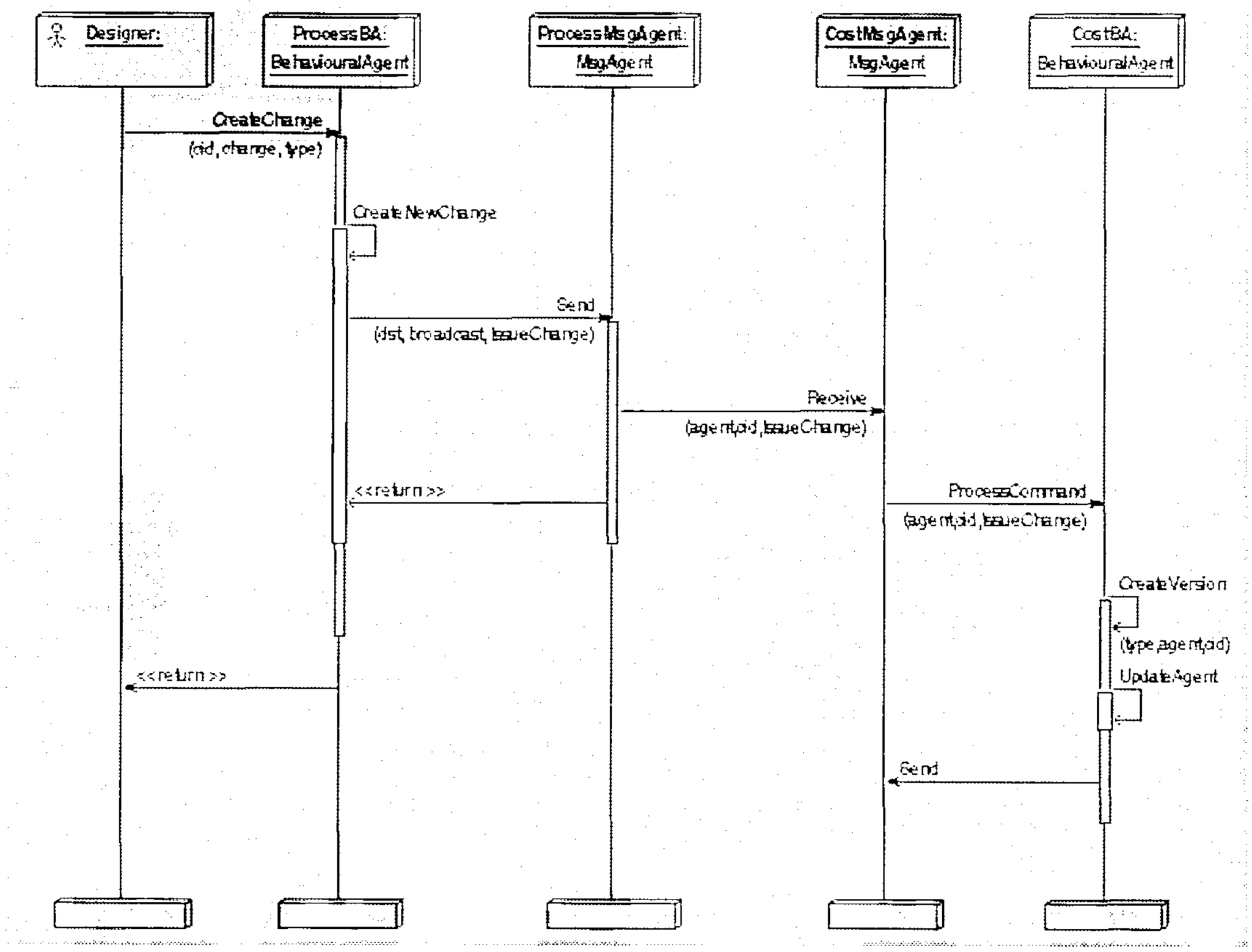


Figure 8.4 Event Sequence Diagram illustrating Communication between Agents

- *Resolution of conflicts and constraints*

Each BA follows the same set of rules in this regard which can be stated simply: if a constraint in a local model is violated then conflict resolution is imposed. What the BAs actually do is to impose semantics on the local data model allowing each alternative proposed to be uniquely identified. Therefore at each design agent each model is given the same global label as the corresponding model at a different site. Having specified this behaviour in VDM-SL the implementation in C++ is straightforward as described in Section 7.4.

- *Control of level of access to local model*

As described previously the version model contains three access levels, private declared and recorded. Private models are not an issue as this is normal operation of a local model. However once a model is declared then it may be viewed by all other agents. The BAs role in this to ensure that requests for change are incorporated into the declared model before the designer examines it, makes it private and then accepts or rejects the change. It also monitors the local users behaviour in order to inform other remote BAs of changes in the declared model.

- *Control of user interaction*

The BA controls user interaction in terms of version control of the local model. By only allowing certain actions to take place on the data by 'freezing' design decisions. Changes to the global design model can only be made through the behavioural agents. Presently the administration of this mechanism is up to the user but it is envisaged that a particular management strategy or design strategy would be adopted to manage the agents. A method which unduly burdens the design engineer is not proposed. The assumption currently is that all transactions are monitored by the BAs, which is not necessarily the case.

8.3.3 Global Agent (GA)

The global agent has the responsibility for maintaining consistency across the information sources. The assumption is that there is only one active configuration, that is, design moves from one state to another single state until its completion. This may be assumed given that in the overall system, conflict resolution and control management procedures exist to ensure this. It may be required in some instances to have two versions of GAs existing in parallel and this is done simply by the agent replicating itself. However, ultimately one will be discarded.

Consider a typical situation where two GAs may be required. An industrial partner, AMEC Process & Energy plc, was uncertain as to whether an offshore platform it was designing would be manufactured from steel or concrete so two alternatives were progressed until this issue was resolved.

The GA has the following commitments or obligations[Sho93]:

- Knowing the location of resources
- Knowing the current configuration of the product
- Controlling creation and deletion of global objects
- Knowing relationships between objects at the global level

Global objects and global relations can only be created and destroyed by the GA. These global IDs are maintained across transactions and access to global objects and relations is only via the GA. As well as knowing the current configuration the GA maintains a history of the product evolution.The following section describes a typical example of the agent behaviour.

8.3.4 Example Behaviour

The GA has got a message from one of the BAs indicating that a change has been made in the local model:

- The BA presents the change by first talking to the local RA (e.g. a comparison algorithm)
- The GA examines its list of current entities and either

  Versions the current configuration and creates a new one

  Or

  Says I don't recognise this entity and sends a query to the BA: Is this a part of or equivalent to something already existing?

- The BA makes this decision by user intervention (see browse procedure below)
- The BA makes the appropriate request to the GA based on this decision

8.3.5 Browse Procedure

If the GA encounters a name that it cannot resolve it follows a simple given procedure. The GA has a root object and from this root, other objects are displayed in a hierarchical manner. The design engineer browses this tree until the component or sub-component concerned is found. Any new object must either be a part of or equivalent to some other component. This is illustrated in Figure 8.5. The agent responds to this by either making a new mapping to another local object at the global level or creating a new object with a unique id and a new relationship which places it within the hierarchy. The worst case in this instance is that all objects exist

one level below the root object, however, it is felt that design inherently lends itself to this hierarchical model and it is an intuitive procedure for design engineers to use. Thus a simple but effective procedure for resolving the issue of semantic equivalence has been described.

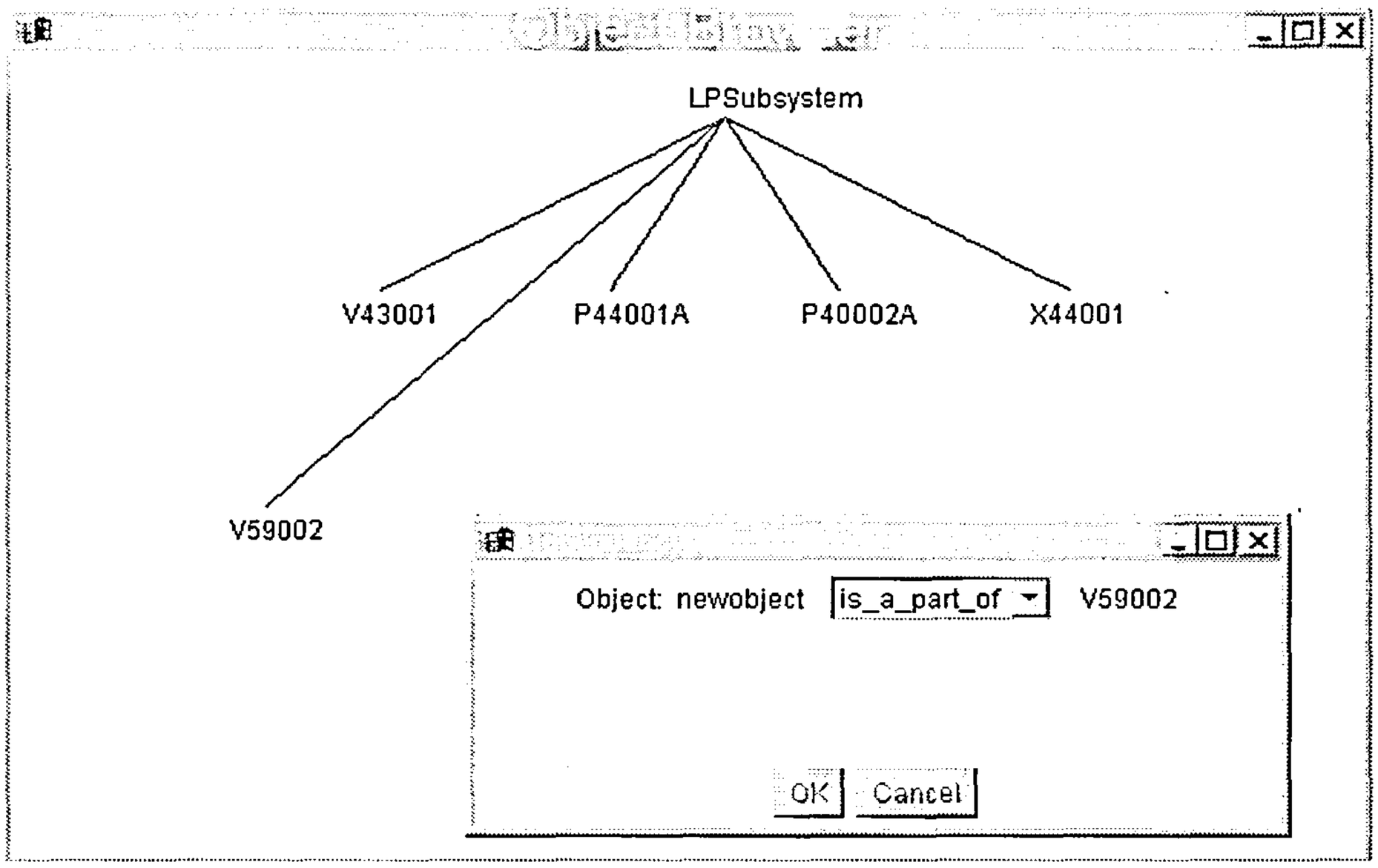


Figure 8.5 : Screen Shot of Browse Procedure

8.4 Application of the system

In the previous section the implementation of our agent architecture is described. The practicability of this architecture is now demonstrated by describing the steps required to apply the architecture in a typical design environment.

8.4.1 Take a design tool and wrap it up

Referring to the static model described in Figure 8.1 it can be seen that in order to wrap a tool up the following decisions have to be made.

- *Decide on the appropriate STEP AP.*
- *Examine the sets of inputs and outputs from the tool*
- *Determine the inherent query capabilities of the tool*

Having analysed the tool and answered these questions, simply inherit from the ResAgent class shown and implement the methods described:

+ CreateVersion(cid : ChangeID, version : VersionLabel) : void

This method assigns a label to the new model and requires that a method of comparing the models and storing changes be available. The changes are stored as *Deltas,* in which design changes are represented on each object as a series of ADD, DELETE or MODIFIED operations.

+ RetrieveObject(oid : ObjectID, version : VersionLabel) : void

This function uses the STEP AP to map the local model to the global model. Essentially this requires a CORBA wrapper to be placed around the individual data objects, where the IDL defined for the wrapper is STEP compliant.

+ EvaluateChange(cid : ChangeID, change : ChangeRequest, version : VersionLabel) : boolean

This function returns an evaluation of a design change in terms of whether it causes a conflict on the agent's local database. The change and version is indicated by the parameter list. The implementation of this function depends on the inference capabilities of the underlying design tool. In the worst case the agent will illicit a YES /NO decision from the design engineer.

+ ResolveChange(cid : ChangeID, version : VersionLabel) : void

The resolve change function specifies that the version given in the parameter list is now the current version for the given change, also in the parameter list.

8.4.2 Build Behavioural Agent

In order to build a behavioural agent the steps are less involved than that of implementing the resource agent as in fact each behavioural agent behaves the same as every other behavioural agent. In fact the only thing that need be done is to give the behavioural agent an identifier and then register this identifier within the global agent.

In terms of code a constructor is created which links the BA to an appropriate RA and

then given an identifier integer which is unique within their system. In a more dynamic environment this identifier would be assigned at runtime rather than at compile time as is currently implemented.

8.5 Demonstration architecture

Figure 8.6 illustrates the integration schema, which was used for the reconstruction of the case study shown in Chapter 9a.Each box represents a design agent.

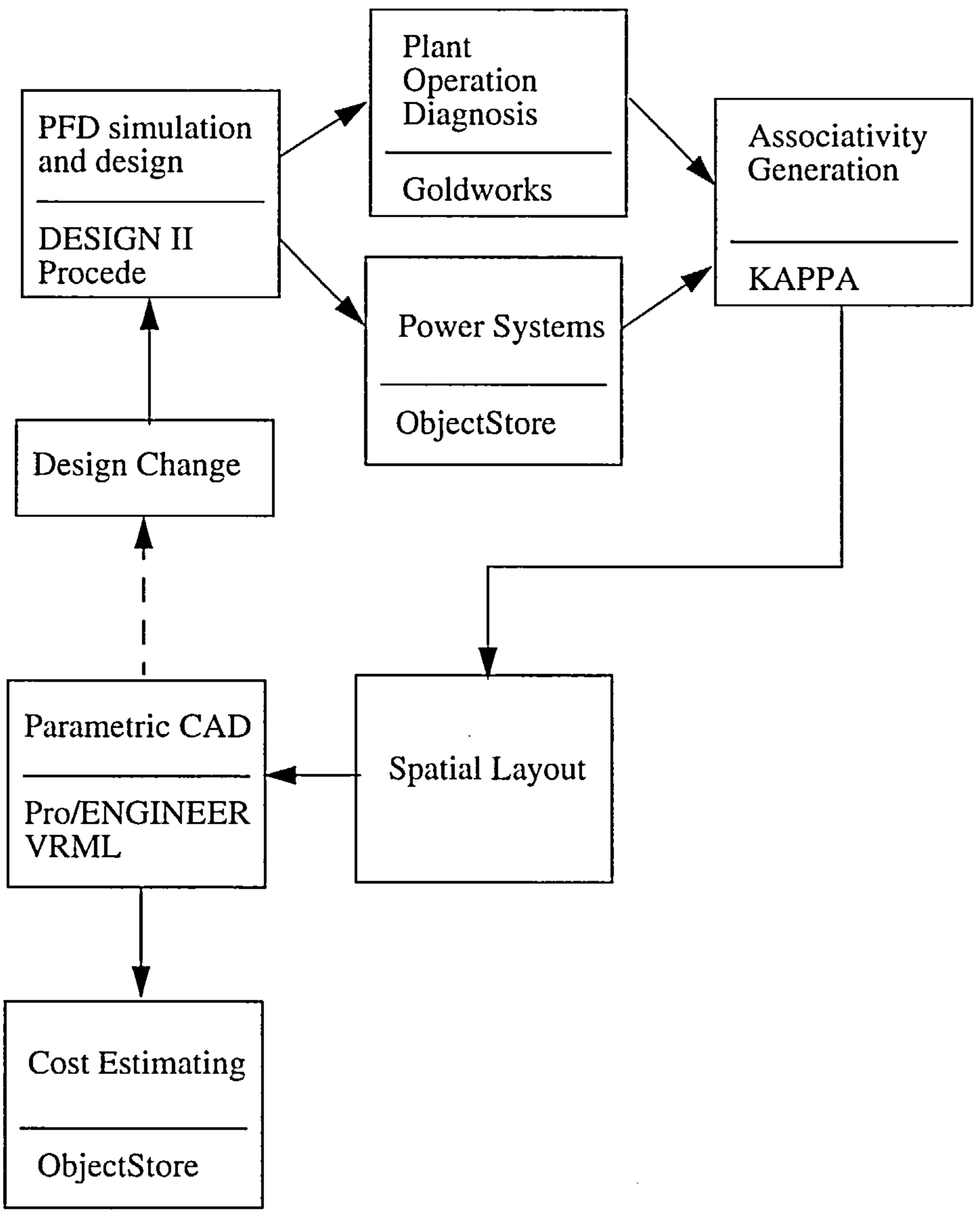


Figure 8.6 Demonstration Architecture for Case Study A

Where commercial software is incorporated in an agent, it is listed in the bottom half of the box.

The PFD agent is responsible for the design of the gas-condensate separation process. It consists of two proprietary design tools, a process simulation package called Design II for Windows and a flowsheet design package Procede. The actual design model which is versioned is stored within Procede for the purposes of the case study. This is because the simulation package produces suitable parameter values for the design equipment, whereas these values are actually stored within the flowsheet model in Procede. These packages both run under Windows NT on an INTEL based PC.

The Power Systems agent calculates the power demands of the design. This power calculation is then used to size the gas turbine power generators. This agent was developed in the EDC and relies on an Objectstore database to select the correct size of generator based on a simple lookup table.

The Plant Operation agent extracts knowledge from the PFD agent and generates some diagnostic rules for the plant operation (King et al , 1995a; King, 1995b). Equipment and connectivity information from the Power Systems and the Plant Operation agents is transported to the Associativity generation agent, which assesses the strength of relation between any two pieces of equipment (including aspects such as connectivity and safety and so on). These agents again reside on a PC with the Associativity Generation agent actually residing at a different geographic location in Sunderland. These knowledge based tools generates the input for the Spatial Layout agent.

The Spatial Layout Tool considers all the input parameters and produces a complete product layout. The product layout data is versioned and stored at the layout agent, in a system which is external to the layout tool. This is incorporated into the Layout Resource Agent.

The model can then be displayed in a Parametric CAD system such as ProE. The Parametric CAD system is linked to a Cost Estimating database, so that a change in the design cost can be estimated. The cost estimation database was developed as part of an existing project within the Engineering Design Centre (Buxton & Softley, 1996; Buxton & Bull, 1996) and runs on UNIX and utilises an Objectstore database.

9 Evaluation Case Studies

9.1 Introduction

In this chapter two real world scenarios are presented as case studies illustrating the practicability of the version model described in this thesis. Case Study A is taken from the offshore petro-chemical industry. It involves a number of disparate design domains, namely, process engineering, structural engineering, electrical engineering and layout design. Each of these highly specialised domains is represented by a design tool. These tools range from interactive design packages running on Windows NT to highly customised analysis tools, developed at the Engineering Design Centre (University of Newcastle Upon Tyne), running on UNIX. Case Study A concentrates on the detailed design phase.

In contrast Case Study B is at the conceptual stage of the design process. The product under consideration is a modern ship design. Again the disparate design disciplines are represented by tools; although the range is not as great as in Case Study A. Both case studies emphasise different characteristics of the version model as will be demonstrated throughout the remainder of this chapter.

9.2 Case Study A: Offshore Process Engineering

In this section a case study is presented taken from the offshore petro-chemical industry. It involves a sequence of design changes that occurred in the manufacture of an offshore oil production platform. Firstly, these changes in the design and their impact on the process are described. The application of the version model to this scenario is then discussed. Describing how it controls version creation and illustrating how it logically stores the different versions.

Figure 9.1 shows an extract of the full process flow diagram (PFD) of an offshore oil production platform. At this stage the oil reserve from the sea bed has been separated into condensate and gas. The condensate stream is not shown but will eventually become the oil for export. The dry gas feed input shown is the gaseous portion of this production stream.

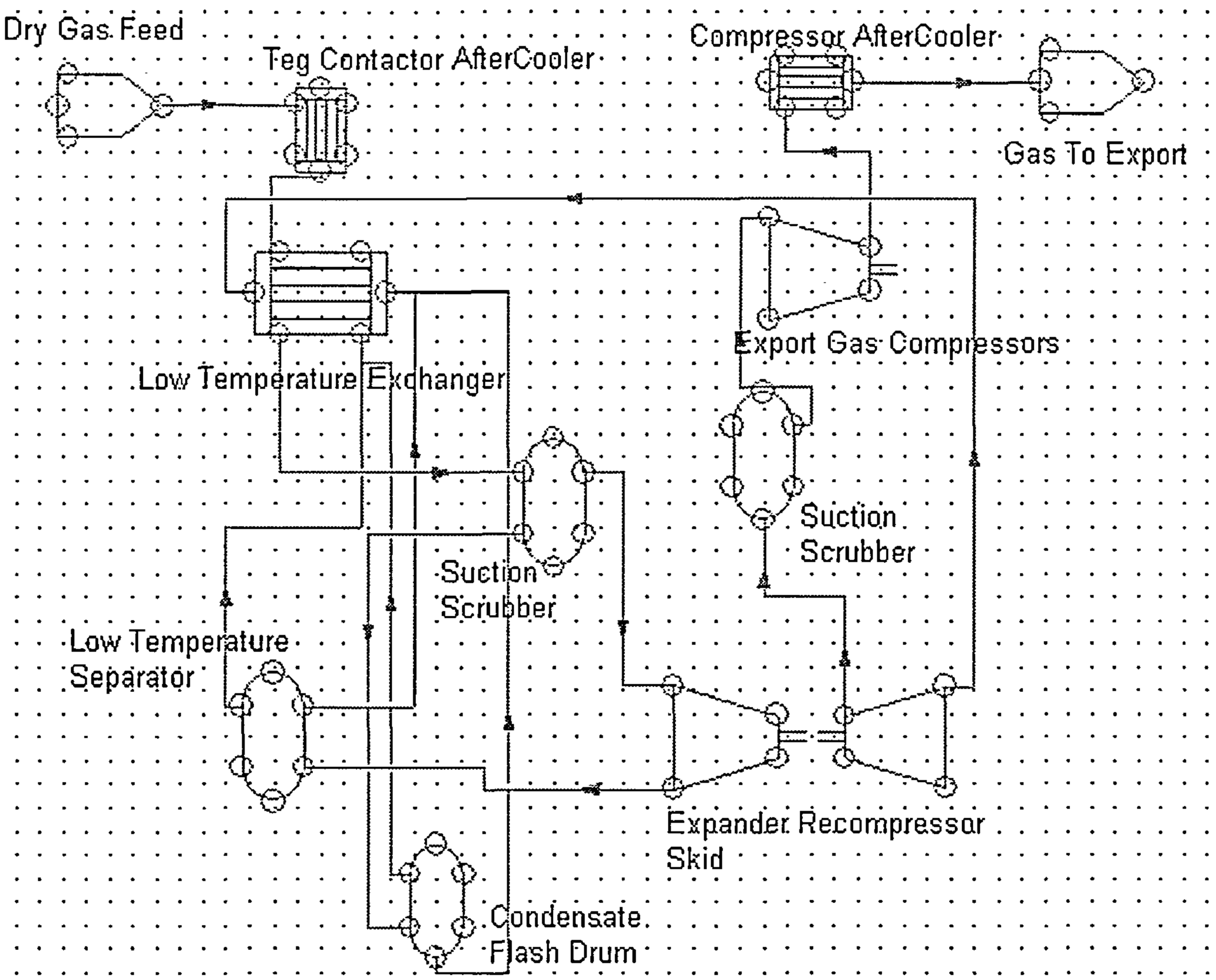


Figure 9.1 Low Temperature Process of OffShore Oil Production

The export gas has to fulfil very strict regulatory criteria. Therefore the process shown, known as the Low Temperature process, 'cleans' the dry gas and compresses it for export along a pipeline. The Teg Contactor AfterCooler and the Low Temperature exchanger cool the gas to the required temperature for separation, to get rid of any of the heavier hydrocarbons in the stream. The Condensate Flash drum collects any of these. The waste streams are not shown on this diagram. The series of suction scrubbers separated by the Expander Recompressor skid further refines the gas. Finally the compressor and the compressor AfterCooler on the export side increase its pressure and decrease its temperature ready for export.

9.2.1 The Design Scenario

The following scenario describes a series of events which occurred during the design of the Low Temperature process described in Figure 9.1. These events involved four active design agents, namely, process design, layout design, electrical systems design and cost estimation. In the actual case each of these design agents (teams of designers) shared information using a traditional paper based system of memos and change requests. As stated these agents are recreated using analysis tools in order to demonstrate the version model.

The process engineer is required to change the operating temperature drop across the Low Temperature Exchanger and so adjusts the flowsheet parameters accordingly. The new equipment is selected and the change is passed to the other design agents. The layout and electrical systems report that this does not conflict with their existing models. They accept these changes and update their models accordingly. However, the cost engineer decides that this exchanger configuration would be cheaper if replaced by a set of two exchangers. The process engineer responds to this suggestion, accepting it and changes his model accordingly. The electrical engineer also accepts this change. However, the extra space required for the two exchanger configuration cannot be accommodated in the existing area. In model terms, constraints in the layout design have now been violated and the layout engineer cannot accept this change. In response to this conflict the electrical systems engineer produces a new design for the platforms power supply. This alternative uses a generator set up with similar power output but which is smaller. However this generator configuration is more expensive. The cost engineer states this violates a global project constraint.

At this moment there are three separate conflicting design configurations under review. Each has advantages but also each requires at least one of the design agents to compromise their local design constraints. It is now that a management decision is made which selects the original configuration (after the initial design change) as the one to progress.

9.2.2 Graph of version history

Figure 9.2 represents the global version history described in Section 9.1 as a directed graph, with labels showing the various version organisations.

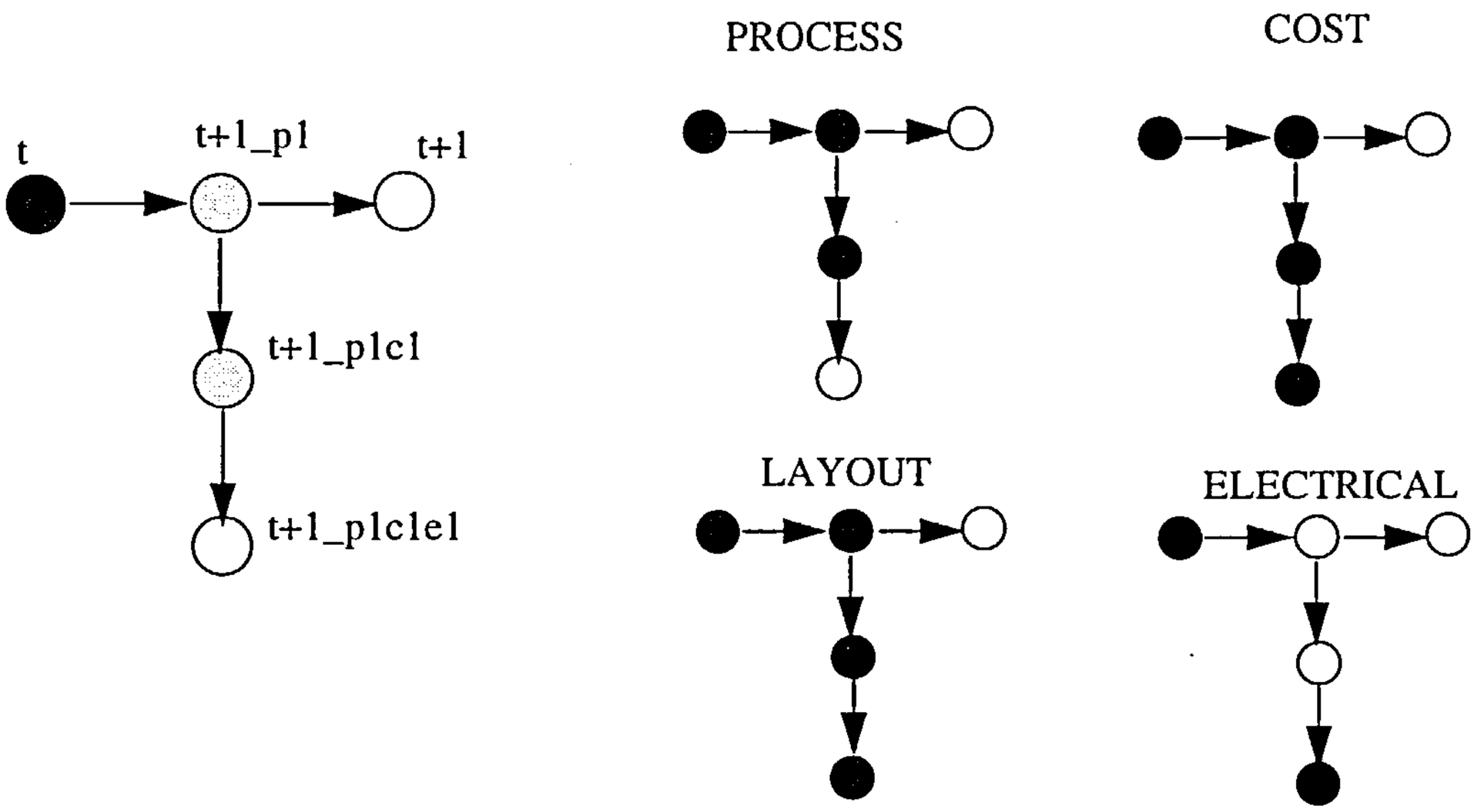


Figure 9.2 Graph representation of Version derivation

The larger graph to the left is intended to show the global model changes and the labelling system applied. t+1 is derived from t through an intermediate step where t+1_p1 is a change which has yet to be verified. In fact t+1_p1 and t+1 are identical and this is represented by the latter being an empty circle. The four other graphs show when a model is actually updated and when the same model receives a new label but the model is actually unaffected by this change. For example when t+1_p1c1 -> t+1_p1c1e1 all models are changed apart from process as it contains no information about generators and so is unaffected.

9.2.3 State Changes at Agents

The sequence of events described above causes a total of over eighty individual state changes in the agents so a suitable subset of these is given to describe typical agent behaviour. Figure 9.3 shows stage 1 with all agents active and activator and respondor states normal. After the requirement for the operational change on the heat exchanger is issued by the process designs behavioural agent, all design activity is frozen.The process activator state is set to Pending-Requirement whilst the other agents' respondor states are evaluating.

Agent State	1	2	3	4	.....	n-1	n
Proc - Activity	A	F	F	F	.....	F	A
Activator	N	PR	PR	RE	.....	RE	RC
Respondor	N	N	N	N	.....	C	N
Electrical- Activity	A	F	F	F	.....	F	A
Activator	N	N	N	N	.....	RE	N
Respondor	N	E	N	C	.....	C	N
Layout - Activity	A	F	F	F	.....	F	A
Activator	N	N	N	N	.....	N	N
Respondor	N	E	N	C	.....	C	N
Cost - Activity	A	F	F	F	.....	F	A
Activator	N	N	N	PR	.....	RE	N
Respondor	N	E	C	C	.....	C	N

Key

A - Active
N - Normal

F - Frozen
E - Evaluating Change
RC - Recording Change

C - Model in Conflict
PR- Pending Requirement
RE - Conflict Resolution

(Note: Section 6.5.3 Describes in detail the meaning of these states)

Figure 9.3 State Changes at Design Agents

The cost respondor is then set to conflict due to the constraint violation on the cost model. The activator replies to this by applying resolution and all respondors are now set to conflict. Concurrently, the cost engineer activator is set to *Pending-Requirement* as the alternative generator is selected. The state changes continue in a similar fashion until we reach time frame n-1. It is at this stage that a ResolveChange message invoked by a management process is sent to all agents and the original change that is t+1_p1 from the graph representation is chosen as the new recorded t+1. This is represented by the RC variable in Figure 9.3.

9.2.4 Comparison of Product Models

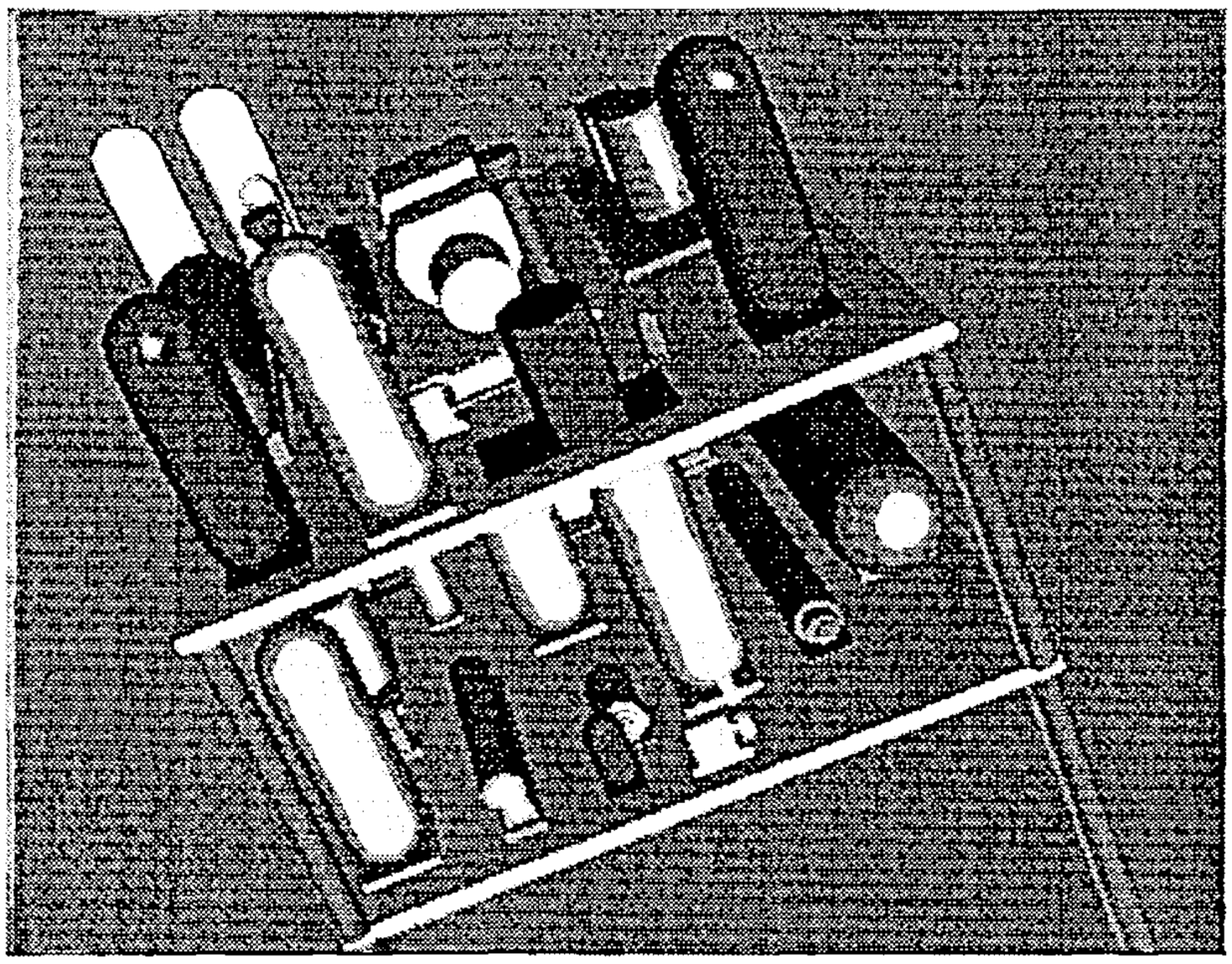


Figure 9.4 3D Product Representation before Design Change

Figures 9.4 and 9.5 show the output of the parametric 3d CAD package before and after the initial design change. This clearly shows the violation of the layout constraints. There are three major pieces of equipment no longer within the permitted deck area. It is at this stage that the electrical engineer produces a smaller generator set up.

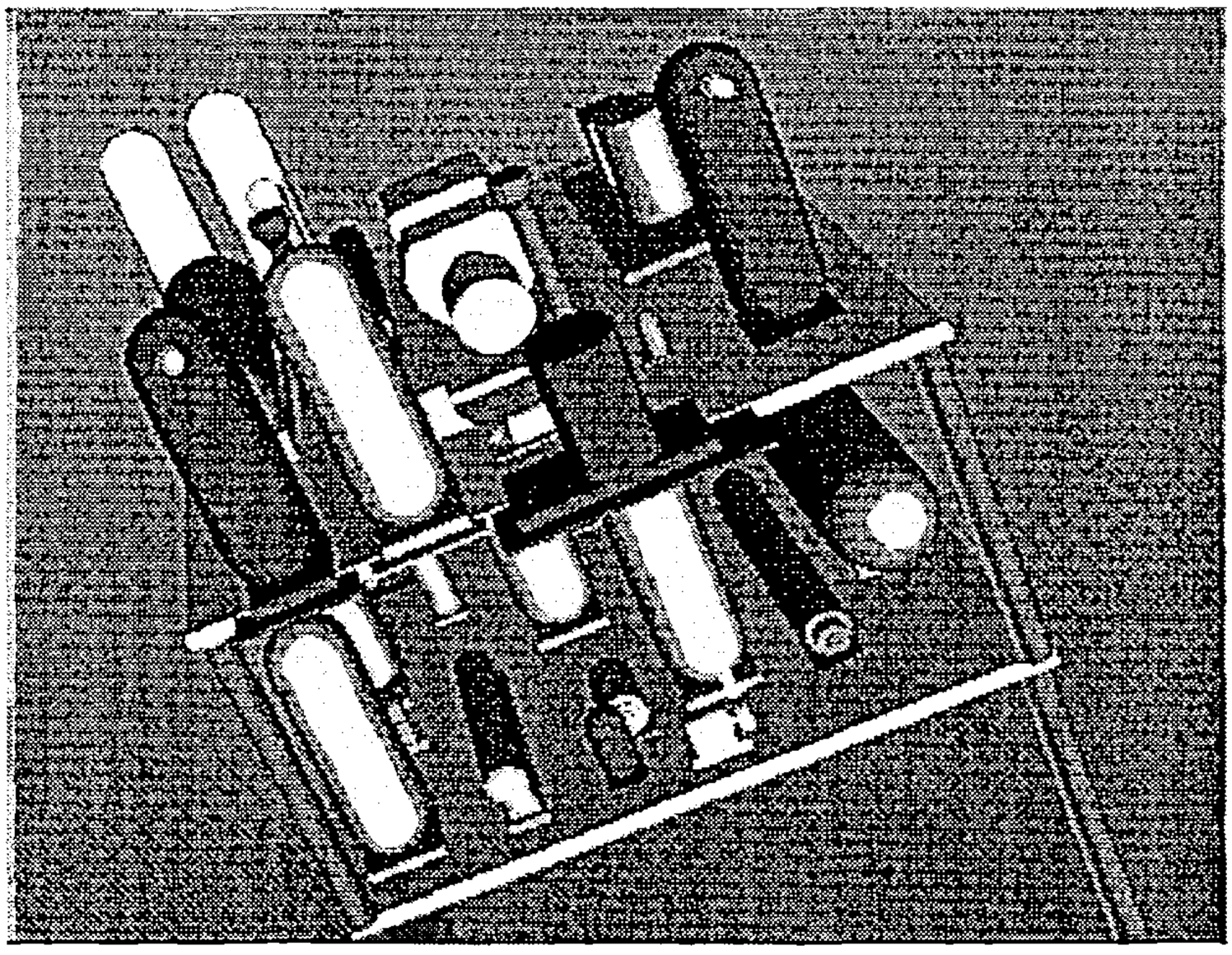


Figure 9.5 3D Product Representation After Design Change

9.3 Case Study B: Fast Ferry Concept Design

In this section a case study supplied from a ship design consultancy Armstrong Technology Associates is presented (Hutchinson et al, 1998). The Made To Order product under consideration is a mono hull fast ferry and it is the concept design phase of the process which is examined. The case study begins with what the company calls a *basis* ship from which design solutions are evolved. The basis ship is actually a design specification of an existing sea-going ship with similar design criteria. as the ship to be built. Firstly the basis ship is described. Then two consequent revisions to the specification are presented. The effects of these revisions on the design are described and the effects on the concept design process are detailed. Finally, it is shown how the version model supports and aids this process.

9.3.1 Basis Ship

The business rationale for the basis ship is stated as follows. The basis ship will operate on an existing route and run in parallel with an existing monohull displacement ferry service. Therefore improving service to attract mainly non-freight cargo from other routes and modes of transport.

From this rationale the following conceptual design requirements are derived:

Crew	15	Vtrial (Vt)	40 Knots
Passengers	600	Vservice (Vs)	37 Knots
Lane Length	685m	Vcruising (Vc)	25 Knots
Number of Cars	136	Endurance (Es)	400 Nautical miles
Number of Coaches	6	End. Days	1
Articulated Vehicles	5		

In order to give the reader a concrete notion of the vessel concerned. Figure 9.6 shows a profile of the ship and in Figure 9.7 some actual designs known as the General Arrangement are presented.

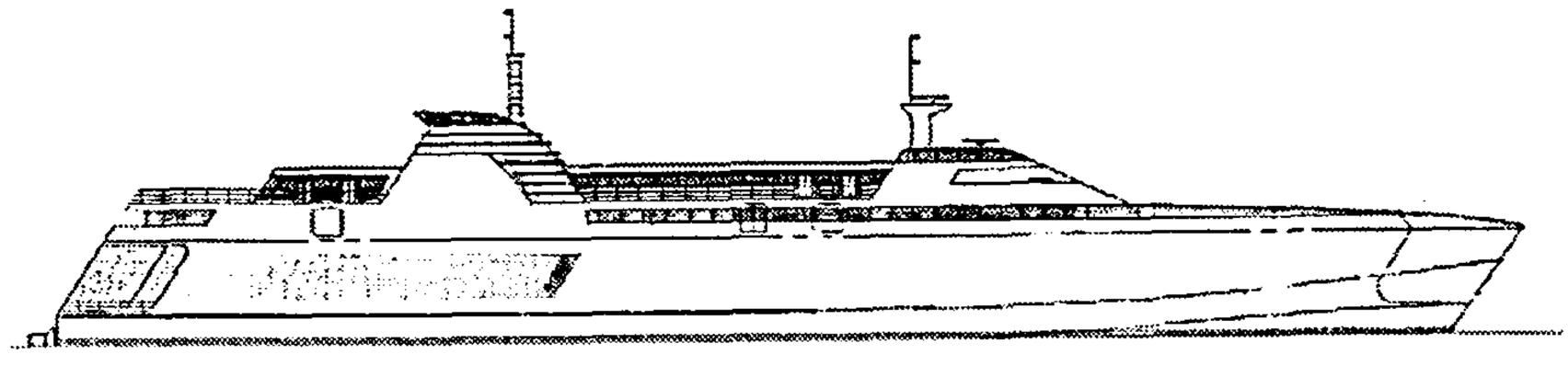


Figure 9.6 Profile of Mono Hull FastFerry

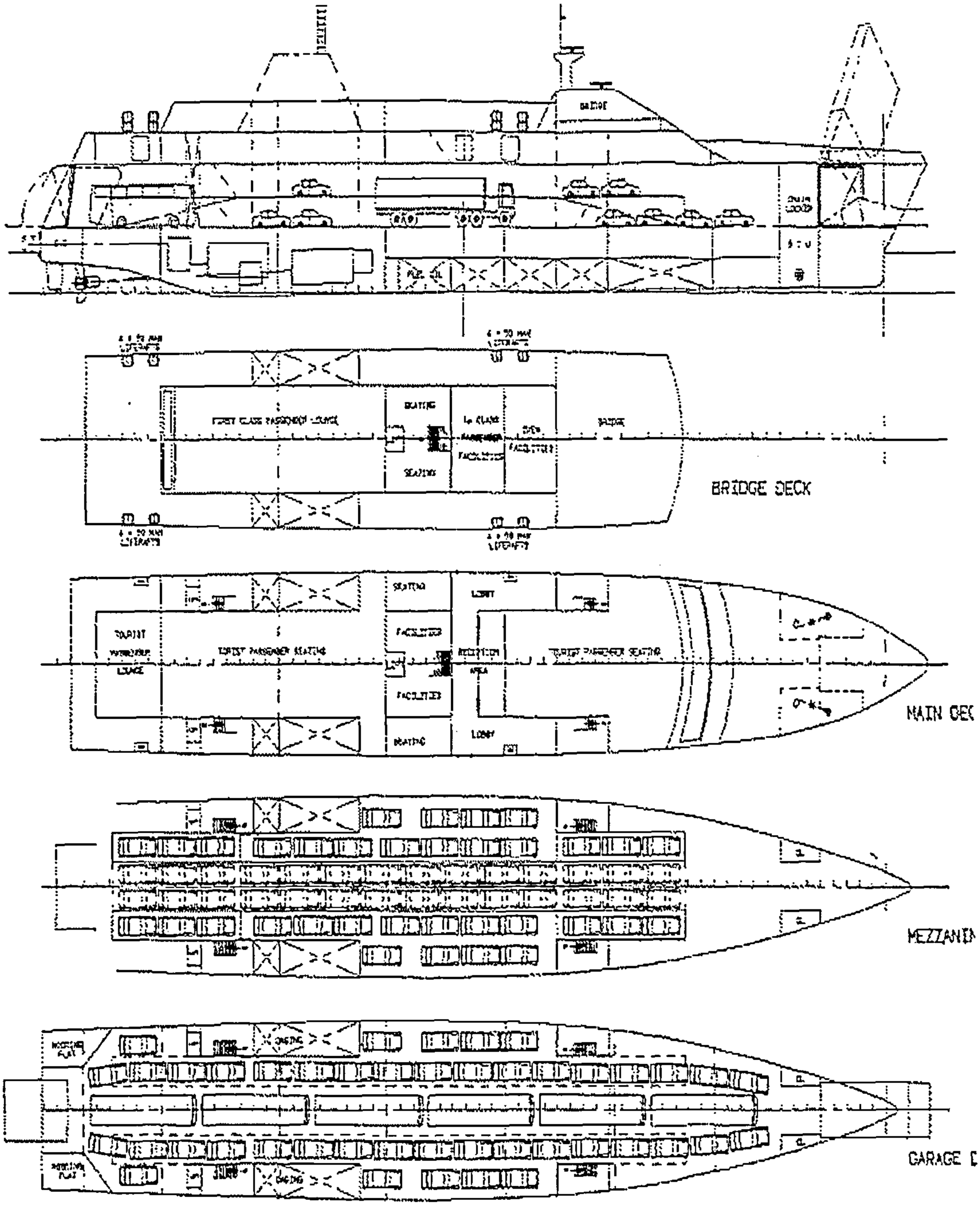


Figure 9.7 General Arrangement of the Ship

9.3.2 Route Data for the Basis Ship

This section describes what is called the *route data* for the basis design ship. As illustrated this covers the parameters of the potential operating route of the ship which impact on the ship design.

Distance	45 miles
Sailing Time	45/37 = 1 hr 13 mins Berthing = 15 mins Loading/Unloading = 30 mins Total = 1 hr 58 mins
Sailing Intervals	2hrs
Sailings per day	8

From this data the following timetable can be derived which leads to a time in service of 16 hours per day and therefore 8 hours down time.

PortA		PortB	
Arrive	Depart	Arrive	Depart
-	07:00	08:30	9:00
10:30	11:00	12:30	13:00
14:30	15:00	16:30	17:00
18:30	19:00	20:30	21:00
22:30	-	-	-

Figure 9.8 Timetable for the Basis Ship Design

9.4 Description of Design Changes

In this section two consecutive revisions to the basis design are presented. The business drivers for these changes are indicated. The consequent changes to the ships specification are then derived.

9.4.1 Revision 1

From experience with a similar route operated by the company, the introduction of a fast ferry service generated a greater demand than that accounted for in the basis ship design. As demand is not at the start or end of the day it was not thought that an extra sailing would alleviate the potential problem. Therefore the cargo (passengers and cars) capacity needed to be increased by 25%. The result of this change in capacity is reflected in an increased required lane length in the garage deck of 856 metres.

9.4.2 Revision 2

The company has recently taken the corporate decision to inaugurate a new route currently served by a conventional monohull ferry service (passengers and freight) and a fast ferry (passenger only). Therefore there is potential to develop a new modern service giving modern facilities and reduced sailing times. This specification requires a design to cope with these additional requirements giving the vessel flexibility and interchangeability over the companies routes.

New Route Data

Distance	73 miles
Sailings per day	8
Sailing Time	Must give a 2.5 hr turn around time
	Berthing = 15 mins
	Loading/Unloading = 30 mins

Hence sailing time is 1 hr 45 mins which means a speed of

Vs = 73/1.75 = 41.71 = 42 kts

Also Es = 73 *8 = 584 + (margin) 75 = 660. Hence Vt = 45 and Vs=42.

PortC		PortD	
Arrive	Depart	Arrive	Depart
-	06:00	08:00	8:30
10:30	11:00	13:00	13:30
16:00	16:30	18:30	19:00
21:00	21:30	23:30	00:00
02:00	-	-	-

Figure 9.9 Revised TimeTable for Ship Design

In the next section an overview of the conceptual ship design process is given.

9.4.3 Conceptual Ship Design

The conceptual ship design environment is described in Figure 9.10. This conceptual design environment is recreated within the Engineering Design Centre by a number of analysis tools, each of which manage their own input and output data. This interaction is represented in Figure 9.11. The data flow diagram represents the distributed design environment on top of which the agent structure previously described sits.

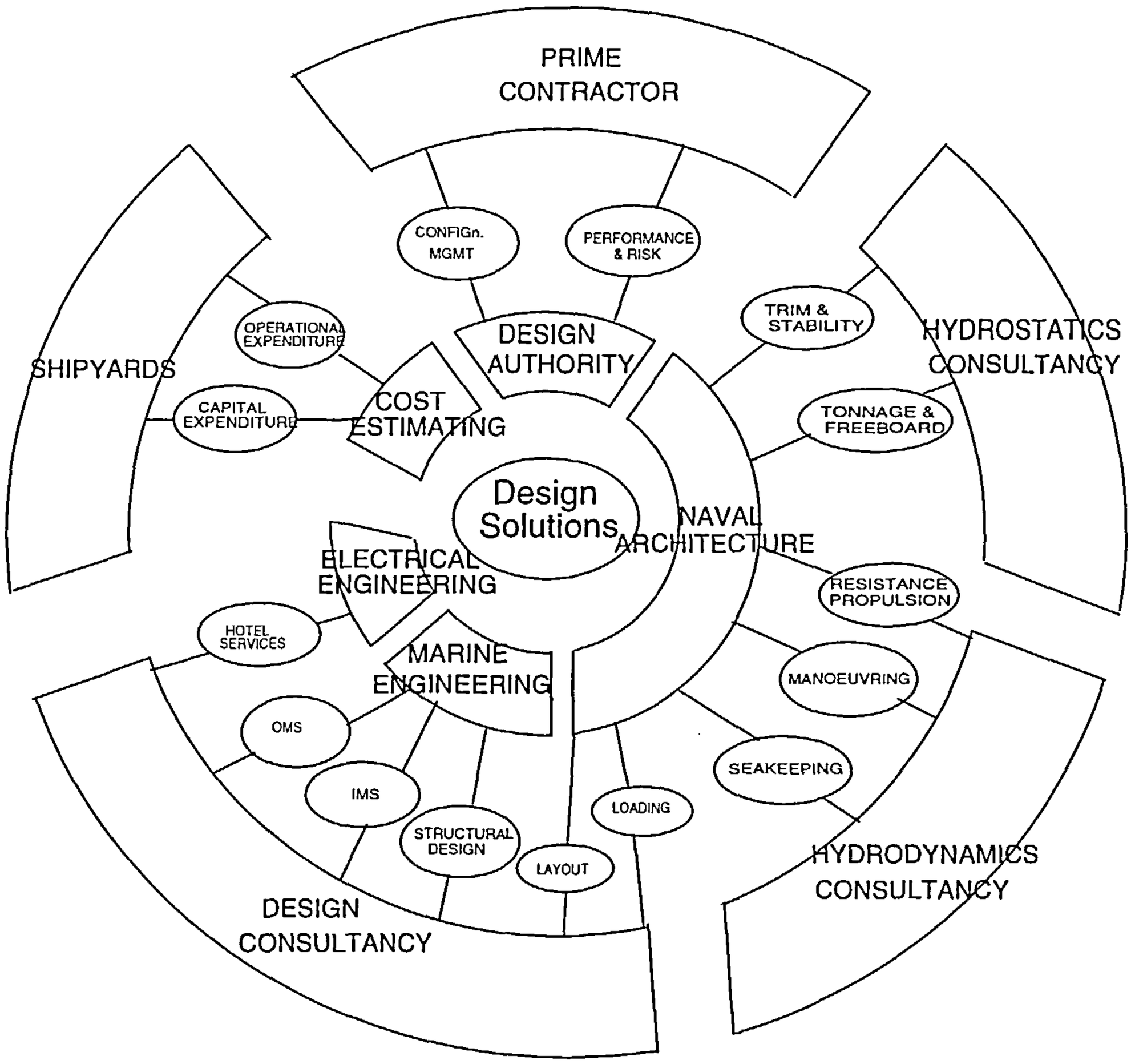


Figure 9.10 The Conceptual Ship Design Process

Figure 9.11 illustrates how a conceptual ship design environment is recreated at the Engineering Design Centre. It also shows the information flows between the isolated design tools. The Prime Contractor issues a specification to the design consultancy (Flow 1). The design consultancy perform some preliminary analysis producing output which is sent concurrently to the hydrostatic consultancy (Flow 2a), hydrodynamic consultancy (Flow 2b) and shipyards (Flow 2c). The hydrodynamic consultancy require that the hydrostatic consultancy have completed (Flow 3) before they can produce their design analysis. The final information flow is back to the prime

contractor (Dashed Lines).

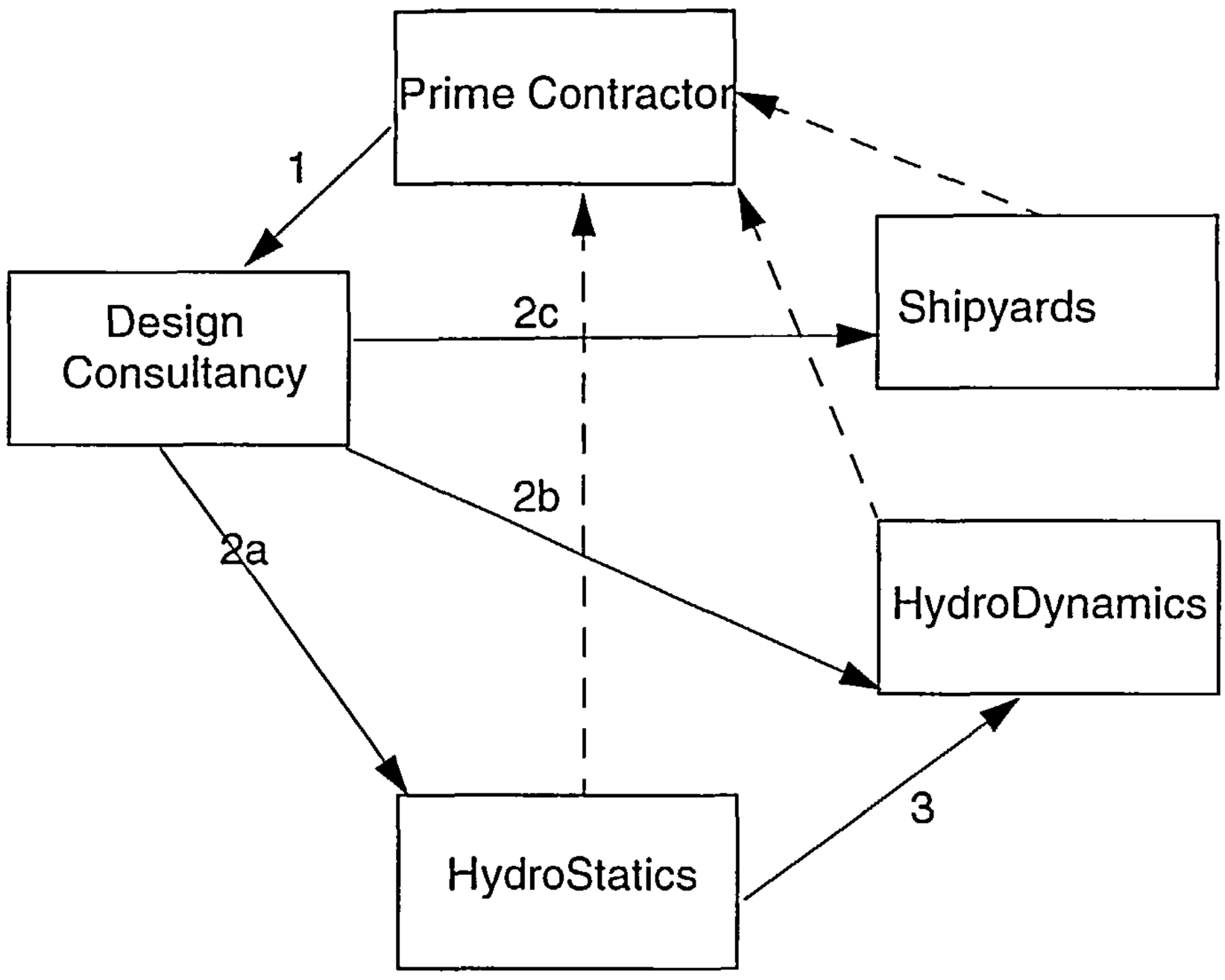


Figure 9.11 Data Flows Between Design Teams

9.5 Case Study Details

In this case study the first revision to the *basis* design requires an increase in the number of passengers. To achieve this the prime contractor decides to increase the length of the ship. A range of three designs are initially evaluated which have an increased length of 100 metres (Design A), 105 metres (Design B) and 110 metres (Design C). These designs all cause a violation within the hydrostatics consultancy and the first two violate the cargo constraints at the design consultancy. At this point the hydrostatics consultancy suggest an increase in freeboard (Fb) and two alternatives are now tried 2.5m (Design D) and 3 m (Design E) . The hydrostatics consultancy state that the second of these designs now produces the required stability. However, the design consultancy states that it feels that the Length over Breadth ratio is too high. Hence the overall breadth is increased to two new values 14.5 metres (Design F) and 15.5 metres (Design G). At 14.5 metres the power requirements of the ship are too high

and again the design consultancy raises a conflict in its model. To compensate for this the design consultancy changes the shape of the ship by altering the midship coefficient (Cm). This is reduced from 0.68 to 0.65 (Design H). However this violates passenger comfort requirements (hydrodynamics), so Cm is increased slightly to 0.66 (Design I). At this stage we have 3 designs which are valid and it is chosen to progress only 2 of these, the two cheaper options, which we now refer to as version a (Design E) and version b (Design G). Table 9.1 shows the set of designs so far evaluated. Note: Draft (T) is constant at 3.3 m.

Design	Length	Breadth	Cm	Fb	Valid Design
A	100	13.7	0.68	2	no
B	105	13.7	0.68	2	no
C	110	13.7	0.68	2	no
D	110	13.7	0.68	2.5	no
E (a)	110	13.7	0.68	3	yes
F	110	14.5	0.68	3	no
G (b)	110	15.5	0.68	3	yes
H	110	14.5	0.65	3	no
I	110	14.5	0.66	3	yes

Table 9.1

9.5.1 Impact of Route Change

From revision 1 there are now two live versions under consideration. In this section the impact of the second revision, that is the adaptation of the design to a new route, is described.

Version a

The design consultancy informs the prime contractor that the increase in speed requires a larger set of engines which can be accommodated but this reduces the number of cars which can be carried by the ship causing a design violation. To compensate for this the breadth of the ship is increased to 16.5 metres (Design J) which compensates for the bigger engines but now the hydrodynamics state this causes a conflict in their model.

Two alternatives are now proposed- a further increase in breadth (Design K) or an increase in the draft (Design L) to 4.3m; which is allowable on this route. The first alternative again causes a conflict in the hydrodynamics consultancy's model. The increase in draught causes no violations in any models

Version b

This design was originally wider than Version a and can accommodate the bigger engines but these again reduce the number of cars that can be carried. To compensate the hydrodynamics consultancy suggest an increase in draft and freeboard (Design M) to 3.5m but this causes another violation in the hydrodynamics model, and hence the draft is reduced and the shape of the ship is changed by altering the midship coefficient. This produces a stable design.

At this stage version b is chosen as the most suitable design to progress. The overall characteristics of this design are perceived by the design consultancy to give better performance.

Design	Length	Breadth	T	Cm	Fb	Valid Design
E (a)	110	13.7	3.3	0.68	3	yes
J	110	16.5	3.3	0.68	3	no
K	110	17	3.3	0.68	3	no
L	110	16.5	4.3	0.68	3	yes
G (b)	110	15.5	3.3	0.66	3	yes
M	110	15.5	4.6	0.66	3.5	no
N	110	15.5	3.3	0.72	3	yes

Table 9.2

Examining the complex set of design interactions and solutions occurring in the case study. The version history graph looks as shown in Figure 9.10 where revision one and two are clearly indicated.

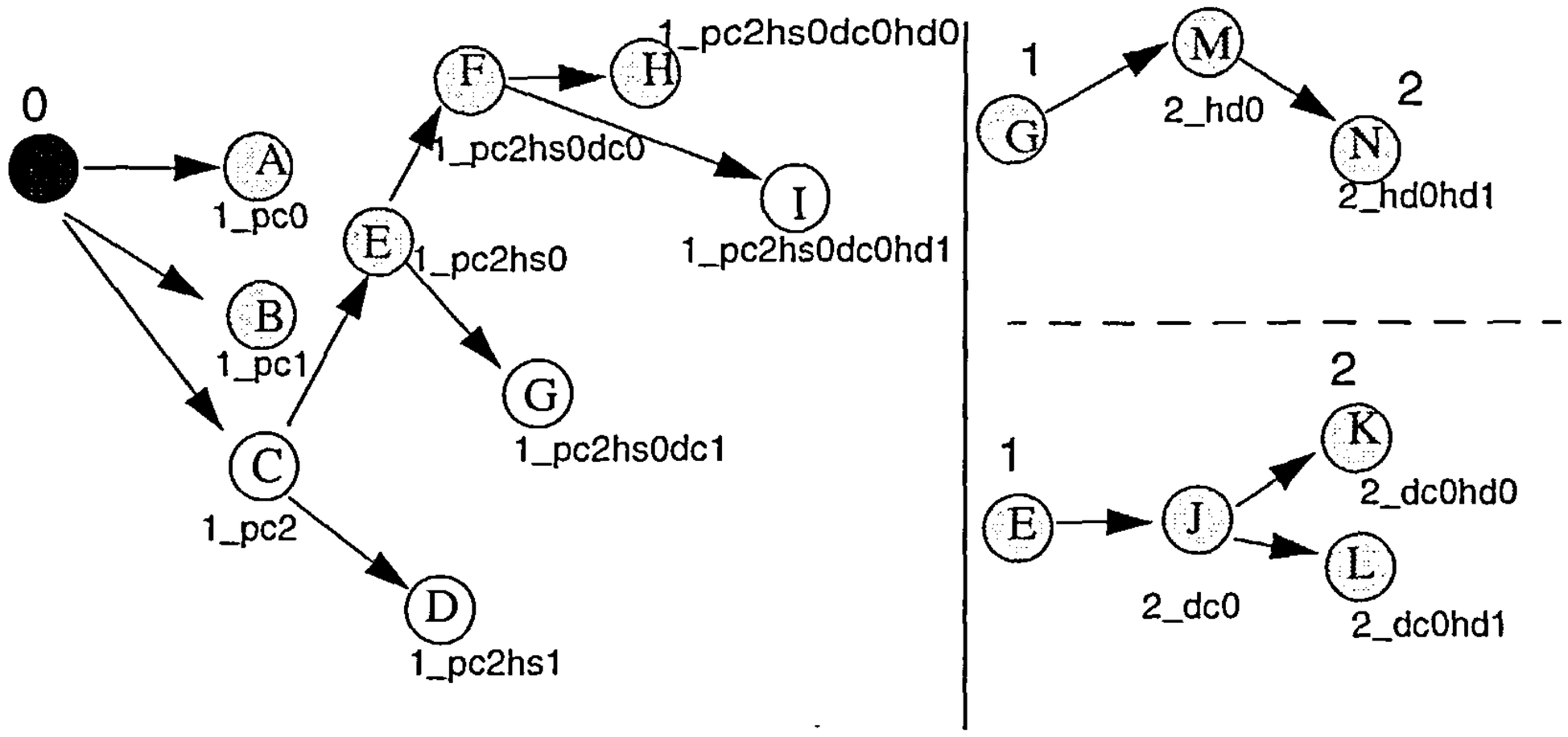


Figure 9.12 Graph of Version History

The key to the version model at this stage in the design process is really in the grouping and monitoring of the derivative solutions and the recording of the design rationale as the model is not too detailed. It can be seen from Figure 9.12 that the labelling scheme applied clearly identifies the solutions and keeps track on the alternative solutions and propositions chosen. Examining the labels derived from the *basis* design, which are 1_pc0, 1_pc1, 1_pc2. The labels indicate three alternative solutions, pc1, pc2, pc3 to the first design revision (1_) from the *basis* design.

Figure 9.13 shows how we use these labels to group similar derivative solutions. Whilst appearing cumbersome the lexical ordering of the label is useful (Keller & Ullman, 1995). This is used by the agent architecture rather than the human designers. A more meaningful label may then be applied at the user interface, such as low cost solutions. This information is clearly important in any knowledge acquisition process and also can be used at later stages in the design to indicate weaknesses that may have been avoided and could be avoided in the future.

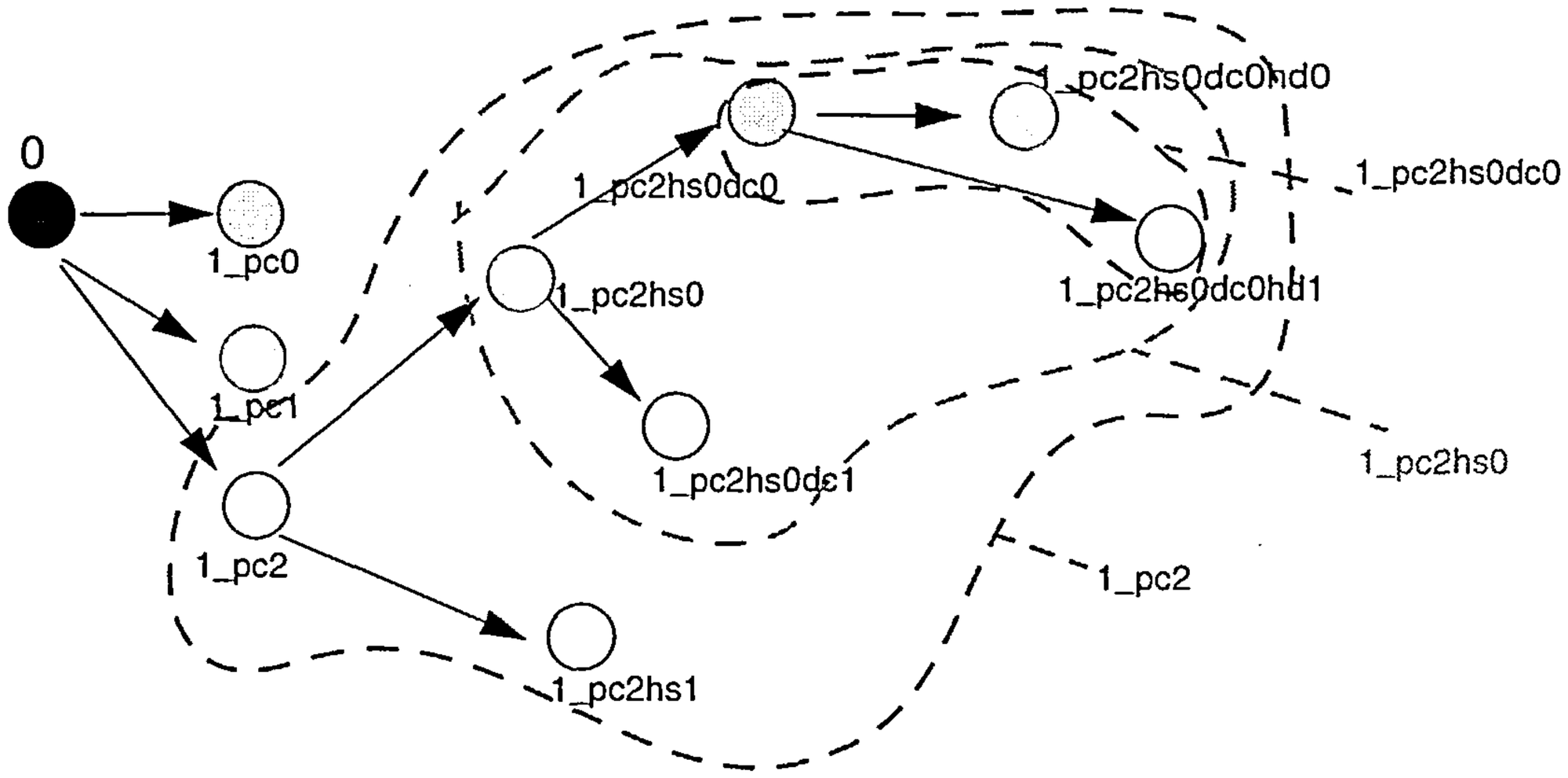


Figure 9.13 Version Grouping

9.6 Summary

The case studies presented in this chapter emphasise the approach taken throughout this thesis of producing a realistic methodology which may be easily applied to existing design scenarios. They also help to test the methodology in a real context and stimulate new ideas for future research. These ideas will be further discussed in Chapter 10 in the conclusions to the thesis.

10 Discussion and Conclusion

10.1 Introduction

That building a version control mechanism for engineering design is a complex task is identified in this thesis by the depth of earlier research presented. From Cutcosky et al (1993), who described the Palo Alto Collaborative Testbed (PACT), research has been driven towards building a completely integrated design system. Work since then has involved projects such as n-dim (Levy et al, 1993) and DOME at MIT (Pahng, 1998) which have examined the building of a concurrent product model. Through to projects such as DESCRIBE in the UK (Kim et al, 1995), DISCO in France (Rodin, 1999) and initiatives such as Infosleuth (Bayardo et al, 1998) or MADEFast (1999) in the USA.

As demonstrated by the nature of these projects, building an integrated design environment requires state-of the art technology in many areas such as database, internet, knowledge sharing, co-ordination and management of information. In this thesis an investigation into the state of the art in database technology is presented in Chapter 3. This is followed, in Chapter 4, by a description from literature of software agents whose promise is to populate the internet (and other global networks) facilitating information management and allowing knowledge sharing.

The earlier projects presented in this thesis highlight specific areas where continued research is needed. In Chapter 2 characteristics of these issues are identified. In presenting the methodology for version control, in Chapters 5 to 7, there is a clear focus on how it attempts to resolve or partially resolve these issues. In Chapters 8 and 9 it is sought to illustrate the application of the methodology through a description of its implementation and by presenting two real world case studies. The benefit of these case studies is discussed further in Section 10.3. The remainder of this chapter discusses the most important findings from this research and as an outcome further work which may be undertaken.

10.2 Findings

The primary objective of this research project was to propose a generic mechanism for version control and configuration management for MTO product design. Design of this

nature typically involves data transactions over a wide area network between tools running highly specialised domain applications which are therefore generally not running on the same platform. The mechanism introduced in the preceding chapters addresses the specific needs of such an environment. It is also proven to be practicable by its application to a real world problem. The deliverables from this project include:

- A fuller understanding of the application of version control and configuration management to the design of large made to order products
- A software framework to support version control and configuration management over a distributed heterogeneous network
- Two real world case studies to demonstrate the general applicability of the framework

Apart from these direct outputs the Engineering Design Centre (EDC) has seen major benefits from the results of this research and two new research proposals, influenced directly by this work, are being prepared.

Problems with Unifying Data Standards

Data exchange requires common interfaces to be devised and agreed upon. However, in MTO product design these interfaces cannot be static as it is required to allow them to develop as project tools mature. The latency in the standardisation process means that once a set of interfaces has been finalised the requirements for information exchange between them will have changed. Thus, until a definitive map of the engineering design process is created, standards such as STEP will always lag behind the requirements driving them. An approach to this issue is to augment a standard data description with a semantic layer which allows more flexible representations of data exchange. Recently XML (Extensible Markup Language) has demonstrated how this may be deployed successfully (Sundsted, 1999; Johnson, 1999). In the version control model presented in this thesis a restricted semantic layer is built which guarantees that all data involved in the process can be mapped on to all other data, however it is represented.

Freedom to evolve - local autonomy

An achievement of the research is to control the way in which data is updated without

placing unrealistic demands on the designers or tools used in the process. A designer simply has to publish changes made by him at appropriate stages, according to his own judgement. It is assumed that the designer is best placed to assess when significant alterations have been made.

In most existing engineering environments designers have autonomous control over their design models. The system presented in this thesis can be applied incrementally as it allows designers to retain this freedom. If the granularity of changes tracked needs to be increased this is controlled by design management processes. The version control system can easily be influenced by these processes through changes to the rules specified in the VDM-SL model. This leads to another significant achievement: the production of a set of rules for distributed version control in the form of a VDM specification.

A Formal Specification for Distributed Version Control

The term *control* suggests a set of rules. An explicit statement of rules is contained within a formal specification. Hence it was decided to model the version mechanism in a formal language. VDM-SL was chosen as this language as it provides the ability to define an executable model. The strengths of using such an approach are:

- The rules governing version control are explicitly stated.
- The rules can be validated against real data to verify the model.
- Any ambiguities in the model can be discussed and resolved.
- Any mistakes in the version model are highlighted before a full implementation.

Emphasising the last point, the version model that is presented in this thesis, altered significantly throughout the duration of the project. Many of the developments that were made resulted directly from simulation of the VDM specification. Had an executable, formal model not been developed these weaknesses would not have been found until the prototype system had been implemented.

Use of an Agent Architecture

The agent architecture produced was particularly suited to the problem domain for two key reasons;

- message passing
- delegation of responsibility.

These are in addition to all the general advantages of agent technology. In this domain the ability to send messages asynchronously frees up resource and allows designers to work without any delays. This must be supported by the system's ability to process and deal with most of these messages intelligently, through delegation of responsibility. The three-layer approach presented facilitates version control in two ways; firstly each layer represents the different access levels in the version model and secondly each layer represents a different aspect of the information exchange, that is, physical, control and knowledge.

The role of a version management system is the ability to represent the product lifecycle data in a semantically relevant manner. In today's complex design environments, traditional storage mechanisms are inadequate (Florida-James et al, 1997). The use of agents, giving the ability to incorporate predicates, knowledge and data models in one software component addresses this issue. The architecture presented gives the ability not only to store data, but to manage global change and record design decisions meaningfully.

10.3 Case Studies

In order to evaluate the usefulness of the version control mechanism for engineering design, two case studies were undertaken. These illustrate two general aspects of the version methodology:

- the mechanism is generally applicable
- the mechanism is suitable for use with real design tools in a real design environment.

The offshore engineering case study demonstrates the mechanism enabling updates of design changes to be made across a wide area heterogeneous network. In this case study four active design agents are considered, each running a highly specialised domain application. This case study demonstrates in particular how the following

issues, are dealt with:

- heterogeneity of operating systems networks and data storage.
- the use of STEP with an appropriate Application Protocol
- the ability to communicate efficiently across platforms and programming languages
- consistency may be maintained at this stage in the product lifecycle

The second case study illustrates application in a separate engineering domain namely shipbuilding. Apart from developing the model in a different domain this case study examines the following features of our mechanism

- the number of participating models is increased
- the version model is equally applicable at an earlier design stage
- grouping of alternative solutions

The case studies presented in Chapter 9 serve to further the evaluation of the model developed throughout this thesis. Together with the VDM-SL specification and the prototype implementation a strong argument for the advancement of this model is developed.

10.4 Lessons Learned and Further Work

Pessimistic

The version model produced is, in distributed database terms, *pessimistic* (Bell & Grimson, 1994). Pessimistic protocols aim for consistency rather than availability. Recovery from failed transactions using this approach is much more straight forward, since updates would have been confined to a single site. Optimistic protocols, however, choose availability at the expense of consistency. On recovery when sites are reconnected inconsistencies are likely. It would be useful to apply more optimistic methods of concurrency control but as Bell points out these are "based on the premise that conflict is rare and that the best approach is to allow transactions to proceed

unimpeded by complex synchronization methods". It is clearly not appropriate to make this assumption. However, all constraints are considered as being of the same degree when in fact some constraints may be relaxed given certain conditions. By introducing the notion of degree, usually referred to in terms of *hardness* a more descriptive model of the local constraints could provided. Hence, a more optimistic protocol could be devised based on what is known about the hardness of the constraints.

Improved Version Clustering

Katz indicates that a version model should support version grouping and through the labelling scheme this is certainly possible. However a more sophisticated scheme could be produced whereby the groupings mechanism was orthogonal to the labelling scheme. The purpose of our labelling scheme is to differentiate between evolving versions over time and version alternatives. This is the main classification of versions dealt with in this thesis. However, more subtle version groupings such as those discussed in Section 2.2.2 should also be permitted. That being said systems which allow such partitioning are at best rudimentary and again restrict the semantics of what is being versioned.

Granularity of Changes

One of the key problems with the methodology, previously mentioned, is that of the granularity of changes. The assumption made is that the designer will submit his work after making a suitable number of changes. The issue of when a version should be created has only been dealt with superficially. In order to make the system more pragmatic a strategy addressing this issue needs to be adopted. Fundamentally, the problem is this. A version of the product model could be created every time there is a change to even the smallest component within the product. For Example when a bolt is replaced on an oil platform. Clearly this is neither necessary or desirable. However, the versioning scheme usually employed by the directorate of an engineering company or consortium - Design for Tender, 2 or 3 revisions and a final design, is also clearly not detailed enough. The answer to the question 'At what level of detail do we want to track changes?' is however not a generic one applicable to all MTO industries. Further research into the life cycle processes of an industry would need to be done in order to answer this trade off between the overhead of storing many designs and the ability to

accurately trace the design process.

Advancement of the Global Model

The global model chosen allows the system to control the creation and deletion of objects. The system also tracks version information and organises the global set through global relations which are used to define views. It is hoped to extend these relations to enable representation of more complex information in order that the occurrence of design conflicts may be identified earlier. The explicit definition of objects and relations at one site means that there is no ambiguity in the final system. Although it does take some effort to produce this global model this expenditure will be far outweighed by the benefit produced.The architecture presented gives the ability not only to store data but to manage global change and record design decisions meaningfully

10.5 Further Application of Work

The application of this work to certain other areas is already being investigated within the EDC. In this section two such proposals are described.

10.5.1 "Design Decision Tracking"

Design rework often requires designers to return to a previous stage in the design due to some new constraints beyond the control of the design process. For instance a new supplier may occur or a budget may be cut. This design rework is usually achieved by discarding all work done since a certain point in time and continuing with the design process in the usual manner. In order to improve this costly and wasteful process, it would be useful to be able to backtrack through the design and assess the impact of the new constraints at each stage. By applying a suitable version model throughout the design process, an intelligent system is then employed to perform intelligent design backtracking.

10.5.2 "Naive Designer"

Experienced engineers have the deep knowledge in a specific domain and also have common sense knowledge in related disciplines. It takes years to build up the knowledge that they possess. Training naive engineers to retain the deep domain knowledge requires considerable resources, time, and costs, let alone the effort needed to acquire common sense. This project aims to devise a mechanism that assists 'naive' engineers in improving their common sense knowledge in other design disciplines. This will be done by producing an arbitrary Design with naive constraints. This design will be interrogated by separate domains involved in the design process. Cased Based Reasoning (CBR) will be used to assess conflicts by deciding dependability of designs. The CBR tool will require versions and alternative design solutions to be stored in order to iterate over a series of designs to assess dependability. This storage will be handled assisted by the version control mechanism described in this thesis.

Note: The proposal described in 10.5.1 has consequently lead to the following research publication:

Laing, C.D., Florida-James, B.O., & Chao K.M., Life-cycle knowledge management in the design of large made-to-order products, in Industrial Knowledge Management edited by R. Roy, published by Springer-Verlag (London) Ltd, to be published

10.6 Conclusion

A novel system of version control has been presented which uses agents to fulfil its responsibilities. These agents address problems in three distinct layers: physical, logical and knowledge. Hence there are three distinct types of agent: resource, behavioural and global.

In order to represent the agents in a declarative manner and also to verify the version control it was chosen to first specify the agent system in VDM-SL. The advantages of doing this were:

1 An executable model was available before system implementation on which test cases could be run.

2 The rules governing version control were explicitly stated and therefore could be:

- verified using formal techniques; and
- revised prior to a costly implementation.

It is believed that the formal model has been successful for the reasons stated and also because version control is in essence a rule based exercise rather than an algorithmic one. Therefore it is much more intuitive to devise a set of rules and compare these with design practice than to verify a complex algorithm.

Finally, the ability to describe product data in a meaningful manner throughout the complete lifecycle is critical in the successful engineering of large made-to-order products. This is the role of a version management system. In today's complex design environments, traditional storage mechanisms are inadequate. The use of predicates, knowledge and data models in one software component, addresses this issue. The architecture presented gives the ability not only to store data but also to manage global change and record design decisions meaningfully.

Glossary

ABET Accreditation Board for Engineering and Technology.

ACL Agent Communication Language. The language that a system of software agents use to communicate.

Agent Oriented Software - A software paradigm composed of autonomous and partially intelligent entities.

AIM Application Interpreted Model. An Application Protocol is represented by its AIM.

AP Application Protocol. A division of the STEP standard, for example a given industry such as Process Engineering will have its own AP.

BA Behavioural Agent as defined in this thesis

CAD Computer Aided Design. A term used to describe the use of computers in traditional design roles, usually refers to parametric three dimensional modelling tools.

COOL Co-ordination Language A language defined by Barbuccenau & Fox (1993) which is used to co-ordinate agent behaviour in a multiagent system.

CORBA The Common Object Request Broker Architecture. A standard architecture for distributed programming.

DARPA Defence Advanced Research Projects Agency

DESCRIBE Design System to support Concurrent Reuse of data in Building and Engineering Design. Introduced in a paper by Carnduff & Gray (1994).

ECA Event-Condition-Action. A methodology for defining rules which are responsive

EXPRESS Modelling language used to represent schema in the STEP standard.

FIPA Foundation for Intelligent Physical Agents. A body which is attempting to standardise agent technology.

GA Global Agent as defined in this thesis

IDL Interface Definition Language. The language of the CORBA standard.

IMS Institute of Marine Sciences. From Figure 8.7. This represents the statuatory regulations that the ship must adhere to.

IT Information Technology.

JAFMAS - Java Agent Framework for MultiAgent Systems. A research framework developed in the United States for prototyping multiagent systems

JDK Java Development Kit. The standard libraries that make up the Java programming language.

KQML Knowledge Query and Manipulation Language. A standard language proposed for knowledge sharing.

KRAFT Knowledge Reuse and Fusion/Transformation. A large research project involving a number of research establishments in the United Kingdom

MIT Massachussets Institute of Technology

MTO Made To Order . Used to describe one off products designed for a single specific purpose rather than mass produced goods for example a ship.

OID Object Identity. The notion of identity from object oriented software development.

OMS Office of Mobile Sources. From Figure 8.7 this represents the regulatory requirements on a vessel pertaining to its sea going emissions.

OMT Object Modelling Technique. A standard notation for modelling object oriented design.

OODBMS Object Oriented Database Management System

ORB Object Request Broker. A component of the **CORBA** standard.

PFD Process Flow Diagram. A schematic diagram used in process engineering.

PTL Past Temporal Logic. A formal logic which allows temporal constraints to be modelled

RA Resource Agent as defined in this thesis.

SCCS Source Code Control System. A version system for software engineering described by Rochkind (1975).

SDAI A division of the STEP standard that specifies a standard interface for database access.

SMIS Schema Meta Integration System. A system to aid schema integration proposed in Qutaishat et al, 1992.

SMVS Schema Meta Visualisation System. An extension of **SMIS** that allows visualisation of disparate schema.

SPIV Ship Product Item Versions. A term introduced by Wooley (1994).

SQL Standard Query Language. The standard language of database systems.

STEP - Standard for the Exchange of Product Data , ISO 10303

TPM Total Product Management. The ability to manage the complete lifecycle of a product from inception to abandonment.

UML Unified Modelling Technique. The accepted standard notation for modelling object oriented design.

VDM Vienna Development Method. A formal methodology devised originally by IBM.

VDM++ An object oriented version of VDM-SL. Its name is derived for the object-oriented programming language C++.

VDM-SL Specification language for the Vienna Development Method

VRML Virtual Reality Modelling Language. A 3 dimensional modelling language used predominantly on the internet.

VSSCD Versioning System to Support Collaborative Design (Santoyridis et al, 1997).

Annexes

The following annexes contain information which is supplemental to the main thesis but provide useful reference material.

Annex A presents a brief review of relational and object-oriented database systems.

Annex B introduces the Common Object Request Broker Architecture (CORBA) which is a key technology used within this thesis.

ANNEX A - Relational & Object-Oriented Database

A database system is essentially nothing more than a computerised record-keeping system. The database itself can be regarded as a kind of electronic filing cabinet (Date, 1995). A database system is not only a repository for a collection of computerised data files, but also it provides a number of functions allowing the users to operate on the system such as adding new data, retrieving data, updating data, deleting data, removing the file etc. A number of researchers in database (Date, 1995; Khoshafian 1993; Codd, 1970; Elmasri 1994) have defined the aims of a Database Management System (DBMS), but they more or less share similar views. These are summarised as follows:

Links between data: A DBMS must be based on a data model whose specific aim is to define the way data items represented in the system are structured and the links that can be established between data items.

Data Consistency: The store data must be consistent with reality. A DBMS must allow users to define rules for maintaining the consistency of the database.

Ease of access to data: The system must allow data to be accessed using high-level declarative languages called query languages.

Data security: A DBMS must be capable of protecting the data it manages against any external aggression.

Data Sharing: A DBMS must provide the means for managing data sharing and for detecting any access conflicts that may arise between several users and applications and provide the tools for resolving them.

Data independence: A DBMS should allow applications to be written without the programmer having to worry about the physical structure of the data and the associated access method.

Performance: A DBMS must be capable of managing a large volume of data and providing users with reasonable access times.

Administration and control: A DBMS should provide a mechanism to allow the system administrators to manage all aspects of the DBMS that are not automated and must be transparent at user level.

Four main models have been used in existing database management systems. They are the hierarchical, network, relational and object-oriented models (Date, 1995). In the hierarchical model data is organised in a tree structure. Early version of IBM IMS (Information Management System) is a typical hierarchical data model. The network model is an extension of the hierarchical model in which the graph of objects is not limited. CODASYL (Conference On Data Systems and Languages) and the Time-Shared Database Management System use the model for their DBMS. The relational model is based on the mathematical idea of a relation. It allows data to be represented in a form of tables whose size is predefined. The function of a data model is to represent the real-world inside the system. In most applications the model must represent entities and the associations between those entities. IBM DB2, ORACLE, and Microsoft Access use relational data model as the core data structure. The object-oriented data model is based on the object-oriented paradigm such as object identification, inheritance, instantiation, and encapsulation. It tries to provide a way to allow the users to model the real world as closely as possible. However, the area of object-oriented database system is vast and diverse because there are at least six approaches towards object-oriented database systems that are identified. The detailed descriptions of these approaches are in Khoshafian (1993).

A DBMS should provide, at external level, the concept of view (or sub-schema) which allows users to be shown the part of conceptual schema that corresponds to their needs (or their access right). Khoshafian (1993) argues that both the hierarchical and network data models do not have physical data independence, because they are primarily navigational. The users' view of the navigational and hierarchical database reflected the way the data is organised, stored, and accessed from the underlying physical storage media. In some cases, the users or system administrator need to specify details of record placement, storage areas, record ordering, record location, and so on. The object-oriented database system is described as next generation of database system by relational database vendors and some new software companies are incorporating object-oriented features into their products. However, the diversity of system architecture and the complexity of the language supported by the system hinder its popularity in the DBMS technologies. Some excellent discussions on the limitations of OODB are in Kim (1995).

ANNEX B - Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) (Object Management Group, 1991; 1995; 1997) is based on distributed object technology and is different from any currently provided classical client/server systems (Orfali, 1996). CORBA defines a set of services to help users to develop their objects without any concern for system services, which can operate beyond the boundary of operating systems, network communications, and programming languages. The applications can be built upon the users' requirements, and then the developer can mix the original component with any combination of CORBA services to create the needed function in a distributed manner. CORBA object services provide a unique approach for creating distributed applications (Mowbray, 1995).

CORBA, defined by the OMG (Object Management Group, 1991; 1995; 1997), is a specification of an architecture and interface which allows client applications to make requests to server implementations in a transparent manner, regardless of programming language, operating system, or hardware platform. In the ORB (Object Request Broker) architecture, the clients and the servers can be located at different sites across a wide area network. Thus, CORBA is an architecture for the management of a wide variety of distributed objects, based on a client/server concept (Ben-Nation, 1995).

Elements in CORBA

Clients can gain access to the services provided by servers via a well-defined interface called IDL (Interface Definition Language) without being concerned with the communication mechanism. The IDL compiler, included in ORB (see Figure A2.1), generates the stub codes for the client, and skeleton codes for the server, to be invoked during run-time. The clients can also dynamically invoke predefined IDL services in an interface repository by using the Dynamic Invocation Interface (DII). Either Static Skeleton Interface (SSI) or Dynamic Skeleton Interface (DSI) code in the server can be invoked by a request in the form of method invocations after ORB core and Object Adapter (e.g. Basic Object Adapter) locate the actual object implementation address. Object Adapter often provides ORB with the following services:

- Generation and interpretation of object references,
- Method invocation,
- Security of interaction,
- Object and implementation activation and deactivation,
- Mapping object reference to implementation, and
- Registration of implementation.

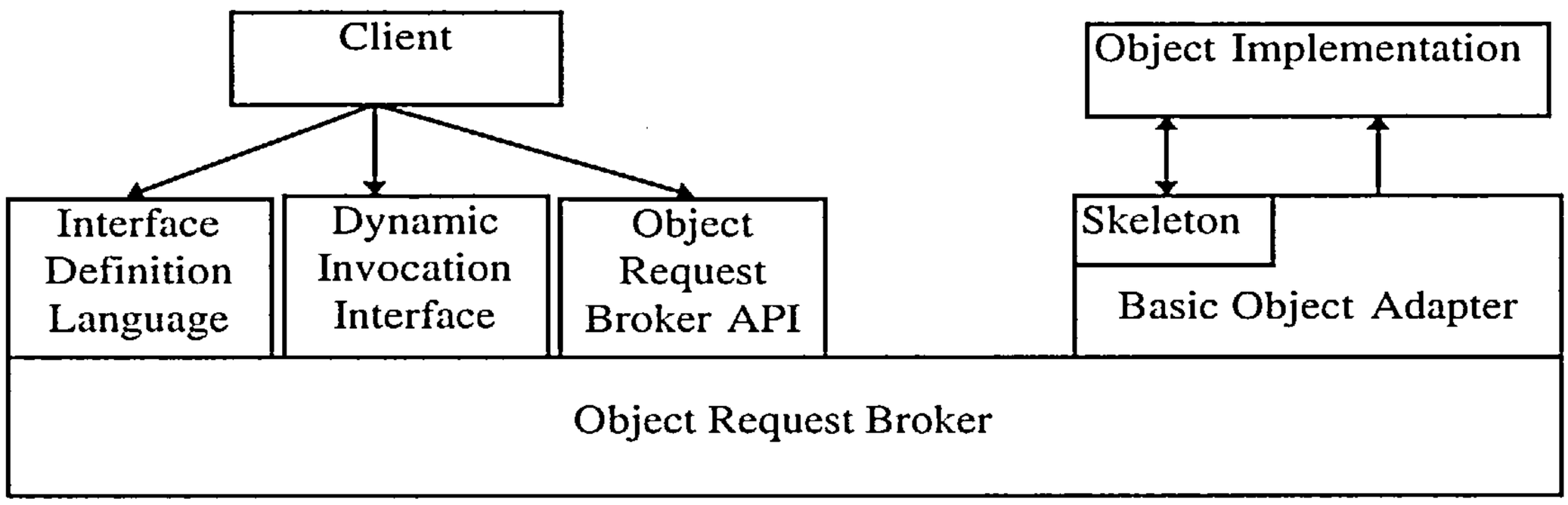


Figure B.1 Object Request Broker Architecture (CORBA)

DSI is an interface to allow dynamic handling of object invocations. DSI in the server is analogous to the client side's DII which does not have to be compiled until the code is invoked. After the server has received the request from the client (via the stubs) the object implementations are executed and these return the results to the client (Object Management Group, 1997).

The interface repository, one of the specifications in ORB, provides persistent storage for the interface definitions. Basically, it manages and provides access to a collection of object definitions specified in IDL. DII relies on this repository to trigger the distributed objects without pre-compiling the resource code into an executable program. One of the major characteristics of CORBA is to allow interoperation between different object systems and ORBs. In order to do so, the IDL-defined object-oriented invocation is used as a higher-level model to span the differences between systems. IDL is a common language to all ORBs, so it is

defined independently of any ORB. The clients and object implementations complying with the specifications defined in IDL are also built independently of ORB. As a result of these features, it is possible for a particular request to pass through multiple ORBs, preserving the invocation semantics transparent to clients and implementations.

The ORBs have already provided a number of IDL compilers to link with other programming languages, for example, C, C++, LISP, COBOL, SmallTalk, Java etc., for the implementation of distributed applications. These languages are available for a number of operating systems; for example SUN Solaris 2.x , Hewlett Packard HP/UX 10 MT, IBM OS/2, DEC Alpha OSF/1 3.2b MT (Digital UNIX), Windows 95, NT, 3.*, etc. Thus ORB compliant applications built upon these operating systems can inter-operate remotely with each other through the communication network (e.g. the Internet).

Advantages of using CORBA

The advantages of adopting the CORBA architecture are:

- ORB reduces the necessity for implementers to know anything about the communication mechanism and an object's actual address. The IDL compiler automatically generates the communication link for the client and the server after the implementations have defined the interface.

- ORB is a communication framework which enables developers to develop, deploy, and maintain distributed object-oriented applications in an easy way. IDL does not only provide an interface mechanism for the users, but also includes a compiler that could map IDL to other programming languages such as C, C++, or LISP. This leads to a reduction in the gap between programming and communication mechanisms.

- A particular ORB implementation interoperates with other brokers provided that they comply to the CORBA specifications. The implementations of the ORB specification can be portable across different vendors' hardware and software architectures.

Appendices

Appendix I contains a list of the author's publication.

Appendix II contains a full listing of the VDM specification presented in Chapter 6.

APPENDIX I - List of Publications

2000

- Florida-James B. O., Rossiter B.N. & Chao, K-M., "An Agent System for Collaborative Version Control in Engineering", Special Issue - Journal of Integrated Manufacturing System (The International Journal of Manufacturing Technology Management), Vol. 11 No. 4, 2000.

1999

- Florida-James B.O., Stoyell J.L., Chao K-M., Norman P. & Ritchey I., Integrating Environmental Knowledge in a Distributed Design Process, First International Conference on Advanced Engineering Design, Prague, Czech Republic, 31 May- 2 June, 1999.

1998

- Florida-James B. O., Rossiter B.N., Norman P., Hills W., & Chao, K-M., "An Agent Mechanism for Version Support in Engineering Design", Proceedings of ASME Design Engineering Technical Conference, Atlanta USA, 13-16 Sept. 1998.
- McAlinden L.P., Florida-James B. O., Chao, K-M, Norman P., W. Hills, & Smith P., "Information and Knowledge Sharing for Distributed Design Agents", Conference Proceedings of AI in DESIGN,98 ,Lisbon, Portugal, 1998.
- Chao K-M, Florida-James B. O., Norman P., Smith P., Hills W., "Use of Virtual Reality and Agents in Engineering Design", Proceedings of EXEPERSYS-98, Virginia, USA,16-17 Nov. 1998
- Chao K-M., Smith P., Hills W., Florida-James B. O., Norman P., "Knowledge Sharing and Reuse for Engineering Design Integration", Journal of Expert System with Applications, Vol. 14, 1998.

1997

- Florida-James B.O., Hills W., Rossiter B.N., 'Semantic Equivalence in Engineering Design Databases',4th Workshop on Knowledge Representation Meets Databases at VLDB '97, Athens Greece
- Florida-James B.O., Chao K-M, Norman P., "Tracing the Effects of Design Changes across Distributed Process Design Agents", 75th IchemE, University of Newcastle-upon-Tyne, 1997
- Chao, K-M, Hills, W., Florida-James, B.O., & Norman, P., "A Methodology for Knowledge Modelling",EXPERSYS-97, 15-16 Oct., University of Sunderland, UK, 1997.

1996

- Guenov M, Florida-James B.O., Chao K-M, Smith N, Hills W, Buxton Ian, 'Tracing the Effects of Design Changes across Distributed Design Agents' 2nd World Conference on Integrated Design & Process Technology, Austin Texas, Dec 1996

APPENDIX II - VDM - SL Listing

A EXECUTABLE SPECIFICATION

The main project is agents.prj which contains the following files:
ChangeControl.vdm
ProcessMessages.vdm
SupportFunctions.vdm
versionControl.vdm
Testing.vdm
Messaging.vdm

BY Module Name:
Agent Version Control (AVC) -This module contains the major type definitions used in the specification eg that of a DesignAgent, and a VersionControl. Functions contained here are mainly invariant clauses but also contains. Record and UpdateAgent which operate on the types described above.

Messaging (MSG)-This module contains the specification of the Agent Communication. Language used in the version control system.It also contains the functions Send and PostMessage which recursively distribute messages to their recipients.

Process Messages (PMG) -This module contains the logical specification of what happens at each agent when it receives a certain message.

Support Functions (SFN) - The functions specified in this module are somewhat auxiliary to the main specification they include things like sorting functions. They are nevertheless critical to the executable specification

Testing (TST) - Functions to run the model against some specified test data.

Change Control (CCTRL) -This module contains the type descriptions for any change related types. The functions specified here represent a change being issued by a designer.

```
--MODULE AVC - AGENT_VERSION_CONTROL
--This module contains the major type definitions used in the specification
--eg that of a DesignAgent, and a VersionControl.
--Functions contained here are mainly invariant clauses but also contains
--Record and UpdateAgent which operate on the types described above.
--Last Edit: 9th July 1998

module AVC

imports from CCTRL types
        ChangeID;
        Change;
        ChangeRequest,
from MSG  all

exports all

definitions

functions

--Record archives a design decision against a certain version after
--checking that this is allowable given the local model state
  Record: MSG`MessageType * AgentID * VersionLabel *CCTRL`ChangeID *
        VersionControl -> VersionControl
  Record(type, agent, version,change,vc)  ==
  let da = vc.AgentInfo(agent),
     recip = dom vc.AgentInfo \ {agent} in
  if type = 1 then
    mk_VersionControl( vc.timestamp, vc.labels,vc.AgentInfo++
    {agent |-> mk_DesignAgent(da.version, da.activityState,  da.activators++
    {change |-> <Recording>},da.respondors, da.alternatives,da.replies,
    <recorded>)}, vc.ChangeDetails, vc.RequiredChanges,vc.totalAlternatives,
    MSG`Send(type,recip, mk_MSG`ChangeRecorded(agent,change),vc.MessageBox))
  else
    mk_VersionControl( vc.timestamp, vc.labels,vc.AgentInfo,vc.ChangeDetails,
    vc.RequiredChanges,vc.totalAlternatives,
    MSG`Send(type,recip, mk_MSG`ChangeRecorded(agent,change),vc.MessageBox))

pre  {vc.AgentInfo(a).vState | a in set dom vc.AgentInfo} = {<declared>};

ResetAgents: map  AgentID to DesignAgent -> map AgentID to DesignAgent
ResetAgents(agents) ==
 { a |-> mk_DesignAgent (mk_VersionLabel(agents(a).version.time, []),
 <Active>,agents(a).activators,agents(a).respondors,agents(a).alternatives,
 agents(a).replies, <private>) | a in set dom agents};


AgentsConsistent: map AgentID to DesignAgent
          * map CCTRL`ChangeID to CCTRL`Change -> bool
AgentsConsistent(agents, changes) ==
  forall a in set rng agents  & (dom  a.activators subset dom changes) and
  (dom a.respondors subset dom changes) and
  ( dom a.alternatives subset dom changes);
 LabelsConsistent: map CCTRL`ChangeID to AlternativeLabel
    * map CCTRL`ChangeID to CCTRL`Change -> bool
 LabelsConsistent(labels, change) ==
  dom labels subset dom change;

BoxConsistent: map AgentID to DesignAgent
* map AgentID to seq of MSG`OrderedMessage -> bool
 BoxConsistent(agents, messageBox) ==
 dom agents = dom messageBox
```

```
types
-- The types section here describes the basis of our abstraction from
-- the real world design process to the VDM-SL specification
-- DesignActivity, ActivatorStates & RespondorStates are state variables
-- contained at each design agent, these control agent behaviour.

 DesignActivity = <Active> | <Frozen>;
 ActivatorStates = <Normal> | <PendingEW> | <PendingR> |
                   <Resolution> | <Recording>;
 RespondorStates = <Normal> | <Evaluating> | <Conflict> ;

-- AgentID specifies the permissible agents in our test system
-- ReplyID and ReplyInfo are used to store the messages an
-- individual agent receives.

 AgentID = <process> | <layout> | <electrical> | <cost>;
 ReplyID ::
    change : CCTRL`ChangeID
    agent: AgentID;
 ReplyInfo = map ReplyID to MSG`Reply;

--DesignAgent has a current version and a current design activity.
--the state variables described above are stored for each separate change
--by mappings and the number of alternatives formed on a given change is also
--stored. Replies from agents are stored temporarily as is the local model state.

 DesignAgent ::
    version : VersionLabel
    activityState : DesignActivity
    activators : map CCTRL`ChangeID to ActivatorStates
    respondors : map CCTRL`ChangeID to RespondorStates
    alternatives: map CCTRL`ChangeID to nat
    replies: ReplyInfo
    vState : Version;

 Version = <private> | <declared> | <alternative> | <recorded>;
 VersionLabel::
    time: nat
    alternative: AlternativeLabel;

 Label ::
      id : AgentID
      number : nat;
 AlternativeLabel = seq of Label;

--VersionControl represents a simualtion of the complete design system
--timestamp represents progression of design time

 VersionControl ::
        timestamp : nat
        labels : map CCTRL`ChangeID to AlternativeLabel
        AgentInfo : map AgentID to DesignAgent
        ChangeDetails : map CCTRL`ChangeID to CCTRL`Change
        RequiredChanges : seq of CCTRL`ChangeID
        totalAlternatives : map CCTRL`ChangeID to nat
        MessageBox : map AgentID to seq of MSG`OrderedMessage
 inv mk_VersionControl(time,labels,agents,changes, requirements, alternatives,
                                                   messageBox)
 ==LabelsConsistent(labels, changes) and AgentsConsistent(agents, changes) and
   BoxConsistent(agents, messageBox)

end AVC
```

```
--MODULE CCTRL -  CHANGE_CONTROL
--This module contains the type descriptions for any 'change' related types
--The functions specified here represent a change being issued by a designer
--Last Edit: 9th July 1998

module CCTRL
imports from AVC all,
from SFN all,
from MSG all
exports all
definitions
types

--Simple model of change in a collaborative environment
--Either it causes a conflict in my local model or it doesn't
  Change = <Conflict> | <OK>;
--Designers may wish to simply experiment with a change rather
--than make it a hard requirement
  ChangeRequest = <Required> | <Evaluation> | <Wish>;
  ChangeID = seq of char;
--Simplfiy typing
  Agents = map AVC`AgentID to AVC`DesignAgent

functions

          --Createchange simulates a designer introducing any kind of
  --change to the design
  CreateChange: AVC`AgentID * ChangeID * ChangeRequest* Change
              * AVC`VersionControl -> AVC`VersionControl
  CreateChange(agent, change ,type, details, vc) ==
   -- First Determine if change already exists
    if ChangeExists(change, vc.ChangeDetails) then
      let alts = vc.totalAlternatives(change) in
     let derChange = SFN`DeriveChange(change, alts),  label = vc.labels(change),
       newAlts = alts+1, da = vc.AgentInfo(agent),
       alt = vc.AgentInfo(agent).alternatives(change) in
        let newVc = mk_AVC`VersionControl(vc.timestamp,vc.labels++
                    {derChange |->label},SFN`InitialiseChange(derChange,vc,
                     dom vc.AgentInfo, agent),vc.ChangeDetails,vc.RequiredChanges,
                   vc.totalAlternatives ++ {change |-> newAlts} ++  {derChange |-> 0}, vc.MessageBox)
        in
         let newVC2 = mk_AVC`VersionControl(newVc.timestamp, newVc.labels ,
                      newVc.AgentInfo++ {agent |-> mk_AVC`DesignAgent(da.version,
                      da.activityState, da.activators,da.respondors, da.alternatives
                      ++{ change |->alt+1},da.replies, da.vState)},    newVc.ChangeDetails,
                     newVc.RequiredChanges,newVc.totalAlternatives,newVc.MessageBox)
         in
                     CreateResponse(agent,derChange,type,details, alt, newVC2)
else
  let newVc = mk_AVC`VersionControl(vc.timestamp,vc.labels,
       SFN`InitialiseChange(change,  vc, dom vc.AgentInfo,agent),
         vc.ChangeDetails, vc.RequiredChanges, vc.totalAlternatives++ {change |->1}, vc.MessageBox)
     in
           CreateNewChange(agent, change, type ,details, newVc)
  pre forall c in set dom vc.ChangeDetails &
     card { a| a in set dom vc.AgentInfo & not( vc.AgentInfo(a).activators(c) = <Normal>)} <= 1;


--GetActivators added after discussion with John Fitzgerald - September 1998
  --Returns the number Activators on a given change
  GetActivators:  map AVC`AgentID to AVC`DesignAgent * ChangeID -> set of AVC`AgentID
  GetActivators(m, c) ==
     {a| a in set dom m & not( m(a).activators(c) = <Normal>)};
```

```
--CreateNewChange simulates that the change is a  new requiremnt on the design
 CreateNewChange: AVC`AgentID * ChangeID * ChangeRequest* Change
              * AVC`VersionControl ->  AVC`VersionControl
 CreateNewChange(agent, change ,type, details, vc) ==
  let recip =  dom vc. AgentInfo \ {agent}, frozen:AVC`DesignActivity = <Frozen>,
    vState:AVC`Version = <declared>, pr:AVC`ActivatorStates = <PendingR>,
     active:AVC`DesignActivity = <Active>,    pe: AVC`ActivatorStates = <PendingEW>,
    msg = mk_MSG`IssueChange(agent,change, type), alt = vc.AgentInfo(agent).alternatives(change),
     altlabel = [mk_AVC`Label(agent, alt+1)], da = vc.AgentInfo(agent),
     vers:AVC`VersionLabel = mk_AVC`VersionLabel(vc.timestamp+1,[mk_AVC`Label(agent, alt)]),
    vers2:AVC`VersionLabel = mk_AVC`VersionLabel(vc.timestamp,  []^[ mk_AVC`Label(agent,alt+1)])
  in
   if SFN`Prioritise(type) =1 then
    mk_AVC`VersionControl(vc.timestamp+1, vc.labels++ {change |->
    [mk_AVC`Label(agent, alt)]}, vc.AgentInfo++{ agent |-> mk_AVC`DesignAgent( vers,frozen,
    da.activators++{change|->pr},da.respondors, da.alternatives++{change |-> alt+1},
    da.replies,vState)}, vc.ChangeDetails munion {change |-> details}, vc.RequiredChanges^
    [change], vc.totalAlternatives, MSG`Send(SFN`Prioritise(type),recip,msg, vc.MessageBox))
   else
    mk_AVC`VersionControl(vc.timestamp, vc.labels++ {change |-> altlabel},
      vc.AgentInfo ++{ agent |-> mk_AVC`DesignAgent( vers2,active,da.activators++{change|->pe} ,
    da.respondors, da.alternatives++{change |-> alt+1}, da.replies,vState)},
    vc.ChangeDetails munion {change |-> details}, vc.RequiredChanges,
    vc.totalAlternatives, MSG`Send(SFN`Prioritise(type),recip,msg,  vc.MessageBox));

--CreateResponse simualtes a change which is a reflection of a previous changes
 CreateResponse : AVC`AgentID  *ChangeID * ChangeRequest * Change * nat*
            AVC`VersionControl -> AVC`VersionControl
 CreateResponse ( agent, change, type, details,alt, vc) ==
  let  recip =  dom vc. AgentInfo \ {agent},frozen:AVC`DesignActivity = <Frozen>,
     pr:AVC`ActivatorStates = <PendingR>, msg = mk_MSG`IssueChange(agent,change, type),
   thisAlt=0, vState:AVC`Version = <declared>, altlabel = vc.labels(change), da = vc.AgentInfo(agent),
     vers2:AVC`VersionLabel=
                  mk_AVC`VersionLabel(vc.timestamp, altlabel^[mk_AVC`Label(agent,alt)])
  in
    mk_AVC`VersionControl(vc.timestamp,vc.labels++{change |-> altlabel^[mk_AVC`Label(agent,alt)]},
      vc.AgentInfo ++ { agent |-> mk_AVC`DesignAgent( vers2,frozen,da.activators++{change|->pr} ,
      da.respondors, da.alternatives++{change |-> thisAlt}, da.replies,vState)},
       vc.ChangeDetails munion {change |-> details}, vc.RequiredChanges,
       vc.totalAlternatives, MSG`Send(1,recip,msg, vc.MessageBox));


--Test whether the given ChangeID is already in existence
 ChangeExists:ChangeID * map ChangeID to Change -> bool
 ChangeExists(c1, changedetails) ==
  c1 in set dom changedetails

end CCTRL
```

```
--MODULE MSG - MeSsaGe types and sending functions
--This module contains the specification of the Agent Communication
--Lanugauge used in the version control system.
--It also contains the functions Send and PostMessage which recursively
--distribute messages to their recipients.
--Last Edit: 9th July 1998

module MSG

imports from CCTRL all,
from AVC all,
from SFN functions DoSort

exports all

definitions
types

  Recipients = set of AVC`AgentID;


  --Language of Messages
  Reply = <OK> | <Conflict> | <NotDefined>;

  IssueChange :: agent : AVC`AgentID
     change : CCTRL`ChangeID
     type : CCTRL`ChangeRequest;

  ReplyOK :: agent : AVC`AgentID
          change : CCTRL`ChangeID
          type: MessageType;

  ReplyConflict :: agent : AVC`AgentID
        change : CCTRL`ChangeID
       type : MessageType;

ApplyResolution :: agent: AVC`AgentID
        change:CCTRL`ChangeID;

ResolveConflict :: agent : AVC`AgentID
         change : CCTRL`ChangeID;

ChangeRecorded :: agent : AVC`AgentID
        change : CCTRL`ChangeID;

ReactivateChange :: change : CCTRL`ChangeID;


--Definiton of vocabulary
 Messages = IssueChange | ReplyOK | ReplyConflict | ResolveConflict |
          ApplyResolution | ChangeRecorded | ReactivateChange;

  --Priority Assignment
MessageType = nat
inv mt == mt <= 3 and mt >= 1;

OrderedMessage :: priority : MessageType
         content : Messages

functions

  Send : MessageType * Recipients * Messages * map AVC`AgentID to
       seq of OrderedMessage  -> map AVC`AgentID to seq of OrderedMessage
```

```
Send(type, toMsg, msg, mbox) ==
  let agent in set toMsg in
  let newtoMsg = toMsg \ {agent} in
  let newmbox = mbox ++ PostMessage(agent, msg, type, mbox) in
  if newtoMsg = {}
  then
    newmbox
  else
    mbox ++ Send(type, newtoMsg, msg, newmbox)
pre toMsg subset  dom mbox;


 PostMessage: AVC`AgentID * Messages * MessageType * map AVC`AgentID to
seq of OrderedMessage   -> map AVC`AgentID to seq of OrderedMessage
  PostMessage(agent, msg, p, mbox) ==
  let newmsg = [mk_OrderedMessage(p, msg)],
      sortedM = SFN`DoSort(mbox(agent)^newmsg) in
    mbox ++  {agent |-> sortedM}
  --place msg in message box in correct location
  pre agent in set dom mbox



end MSG
```

```
--MODULE PMG - Process_MessaGe
--This module contains the logical specification of what happens at each
--agent when it recieves a certain message.
--Last Edit: 9th July 1998

module PMG

imports from AVC all,
from MSG all,
from CCTRL all,
from SFN all

exports all

definitions


functions

--ProcessCommand represents the language parser of the agent, it is
--important to note that processingAgent represents the agent receiving
-- the message where as agent represents the agent that originated it.

  ProcessCommand: MSG`Messages * AVC`AgentID * AVC`VersionControl ->
                AVC`VersionControl
  ProcessCommand(msg, processingAgent, vc) ==
    cases msg :
      mk_MSG`IssueChange(agent, change, type) ->
        CreateVersion(SFN`Prioritise(type),agent, change, processingAgent,vc),
      mk_MSG`ReplyOK(agent, change, type) -> ReplyYes(type, agent, change,
                                              processingAgent,vc),
      mk_MSG`ReplyConflict(agent, change, type) -> ReplyNo(type, agent, change,
                                              processingAgent,vc),
      mk_MSG`ApplyResolution(agent, change) -> ApplyRes(agent, change,
                                              processingAgent,vc),
      mk_MSG`ResolveConflict(agent,change) -> Resolve(agent,change,vc),
      mk_MSG`ChangeRecorded(agent,change) -> RecordChange( agent, change, vc),
      mk_MSG`ReactivateChange(change) -> Reactivate(change, vc),
      others ->vc
    end;

--The following five functions are the specific functions which deal with our
--current language of five messages
--CreateVersion is a response to an IssueChange message etc.
--September 1998: vState should actually go from dec -> d-a ->private->d or da
-- but because we lose concurrency can only demonstrate final state

  CreateVersion: MSG`MessageType*AVC`AgentID * CCTRL`ChangeID * AVC`AgentID *
        AVC`VersionControl -> AVC`VersionControl
  CreateVersion(type, agent, change, thisAgent, vc ) ==
    let  recip = {agent},
      frozen:AVC`DesignActivity = <Frozen>, norm:AVC`ActivatorStates = <Normal>,
      eval: AVC`RespondorStates = <Evaluating>,
      active:AVC`DesignActivity = <Active>,
      alternative:AVC`Version = <alternative> , declared:AVC`Version = <declared>,
      alt = vc.AgentInfo(thisAgent).alternatives(change),
      altlabel = vc.labels(change),
      da = vc.AgentInfo(thisAgent),
      vers2:AVC`VersionLabel = mk_AVC`VersionLabel(vc.timestamp, altlabel)
      in
        if vc.ChangeDetails(change) = <Conflict> then
  if type = 1 then
    mk_AVC`VersionControl(vc.timestamp, vc.labels,
```

```
  vc.AgentInfo  ++ { thisAgent |-> mk_AVC`DesignAgent(vers2,frozen,
        da.activators++{change|->norm}, da.respondors++{change|->eval},
        da.alternatives++{change |-> alt}, da.replies,alternative)},
        vc.ChangeDetails, vc.RequiredChanges, vc.totalAlternatives,
  MSG`Send(type,recip,mk_MSG`ReplyConflict(thisAgent, change, type),
        vc.MessageBox))
else
  mk_AVC`VersionControl(vc.timestamp, vc.labels,vc.AgentInfo ++
       {thisAgent |->mk_AVC`DesignAgent( vers2,active,da.activators ++
       {change|->norm},da.respondors++{change|->eval}, da.alternatives ++
       {change |-> alt},da.replies,alternative)},vc.ChangeDetails ,
        vc.RequiredChanges, vc.totalAlternatives, MSG`Send(type,recip,
        mk_MSG`ReplyConflict(thisAgent, change, type),vc.MessageBox))
    else
if type = 1 then
  mk_AVC`VersionControl(vc.timestamp, vc.labels, vc.AgentInfo ++
       {thisAgent |-> mk_AVC`DesignAgent(vers2,frozen,  da.activators ++
       {change|->norm}, da.respondors++{change|->eval}, da.alternatives++
       {change |-> alt}, da.replies,declared)},vc.ChangeDetails,
        vc.RequiredChanges,vc.totalAlternatives,  MSG`Send(type,recip,
        mk_MSG`ReplyOK(thisAgent, change, type), vc.MessageBox))
else
  mk_AVC`VersionControl(vc.timestamp, vc.labels, vc.AgentInfo ++
       {thisAgent |-> mk_AVC`DesignAgent( vers2,active,da.activators++
       {change|->norm},da.respondors++{change|->eval}, da.alternatives++
       {change |-> alt}, da.replies,declared)}, vc.ChangeDetails ,
  vc.RequiredChanges, vc.totalAlternatives, MSG`Send(type,recip,
        mk_MSG`ReplyOK(thisAgent, change, type), vc.MessageBox))
pre {agent, thisAgent}  subset dom vc.AgentInfo;



  ReplyYes:MSG`MessageType * AVC`AgentID * CCTRL`ChangeID * AVC`AgentID *
        AVC`VersionControl -> AVC`VersionControl
  ReplyYes(type, agent, change, thisAgent, vc) ==
   let totalReplies = vc.AgentInfo(thisAgent).replies ++
   {mk_AVC`ReplyID(change,  agent) |-> <OK>} ,
   versLabel = vc.AgentInfo(thisAgent).version  in
    let repSet = { r | r in set  dom totalReplies &  r.change = change} in
     let replies = { r |-> totalReplies(r) | r in set repSet} in
      if (forall reps in set  rng replies & reps  = <OK> ) then
    AVC`Record(type, thisAgent,  versLabel,change,vc)
  else
   let da = vc.AgentInfo(thisAgent),
       a = vc.AgentInfo(agent) in
    mk_AVC`VersionControl(vc.timestamp, vc.labels, vc.AgentInfo++
          {agent |->mk_AVC`DesignAgent (a.version, a.activityState,
          a.activators, a.respondors++{change|-><Normal>}, a.alternatives ,
          a.replies, <declared>)}++ {thisAgent |-> mk_AVC`DesignAgent
          (da.version, da.activityState,da.activators, da.respondors,
          da.alternatives, da.replies++replies,da.vState)},vc.ChangeDetails,
          vc.RequiredChanges,vc.totalAlternatives, vc.MessageBox)
  pre {agent, thisAgent}  subset dom vc.AgentInfo;



  ReplyNo:MSG`MessageType * AVC`AgentID * CCTRL`ChangeID * AVC`AgentID *
        AVC`VersionControl -> AVC`VersionControl
  ReplyNo(type, agent, change, thisAgent, vc) ==
    let  replies= vc.AgentInfo(thisAgent).replies++
        {mk_AVC`ReplyID(change,  agent) |-> <Conflict>},
  frozen:AVC`DesignActivity = <Frozen>, resolution:AVC`ActivatorStates
        = <Resolution>,
  norm: AVC`RespondorStates = <Normal> in
```

```
      let da = vc.AgentInfo(thisAgent) in
       let agentInfo = {thisAgent |->mk_AVC`DesignAgent (da.version, frozen,
       da.activators++{change |-> resolution},da.respondors++{change |-> norm},
       da.alternatives, replies, da.vState) },
msg = mk_MSG`ApplyResolution(thisAgent, change),
recip = dom vc.AgentInfo \ {thisAgent} in
 if type = 1 then
   if  da.activators(change) <> <Resolution> then
    mk_AVC`VersionControl(vc.timestamp, vc.labels, vc.AgentInfo ++
          agentInfo, vc.ChangeDetails, vc.RequiredChanges,
          vc.totalAlternatives,MSG`Send(type, recip, msg, vc.MessageBox))
         else
    mk_AVC`VersionControl(vc.timestamp, vc.labels,  vc.AgentInfo ++
          agentInfo, vc.ChangeDetails, vc.RequiredChanges,
          vc.totalAlternatives, vc.MessageBox)
        else
   mk_AVC`VersionControl(vc.timestamp, vc.labels,  vc.AgentInfo ++
          {thisAgent |-> mk_AVC`DesignAgent (da.version, da.activityState,
          da.activators, da.respondors, da.alternatives, replies, da.vState)}
          ,vc.ChangeDetails, vc.RequiredChanges,
          vc.totalAlternatives, vc.MessageBox)
 pre {agent, thisAgent}  subset dom vc.AgentInfo;

 ApplyRes: AVC`AgentID * CCTRL`ChangeID * AVC`AgentID * AVC`VersionControl ->
                AVC`VersionControl
 ApplyRes(agent, change, thisAgent, vc) ==
  let conflict: AVC`RespondorStates = <Conflict>,
    a = vc.AgentInfo(thisAgent) in
    mk_AVC`VersionControl(vc.timestamp, vc.labels, vc.AgentInfo++
    {thisAgent|-> mk_AVC`DesignAgent(a.version,a.activityState,a.activators,
    a.respondors ++ {change |-> conflict},a.alternatives,a.replies, a.vState)}
   ,vc.ChangeDetails,vc.RequiredChanges,vc.totalAlternatives, vc.MessageBox)
pre {agent, thisAgent}  subset dom vc.AgentInfo;


 Resolve: AVC`AgentID * CCTRL`ChangeID * AVC`VersionControl ->
       AVC`VersionControl
 Resolve (agent, change ,vc) ==
  let version= mk_AVC`VersionLabel(vc.timestamp,
   SFN`ChooseAlternative(vc.labels(change))), type :MSG`MessageType =1
  in
     AVC`Record(type,agent, version, change, vc)
pre agent  in set dom vc.AgentInfo;


 RecordChange:AVC`AgentID * CCTRL`ChangeID * AVC`VersionControl ->
                                    AVC`VersionControl
 RecordChange (agent, change ,vc) ==
 let agents = vc.AgentInfo++AVC`ResetAgents(vc.AgentInfo),
  changes = GetAllDerived(SFN`Ancestor(change), dom vc.ChangeDetails),
  required = UpdateSequence( vc.RequiredChanges,SFN`Ancestor(change)) in
   if required = []  then
    mk_AVC`VersionControl( vc.timestamp, vc.labels,
    AgentRemoveChange(agents,changes), changes <-: vc.ChangeDetails ,
    [], vc.totalAlternatives,  vc.MessageBox)
   else
    let msg = mk_MSG`ReactivateChange( Latest(hd required,
     vc.totalAlternatives(hd required) )) in
     mk_AVC`VersionControl( vc.timestamp, vc.labels,
     AgentRemoveChange(agents,changes), changes <-: vc.ChangeDetails ,
     required,vc.totalAlternatives, MSG`Send(1,dom agents,msg,vc.MessageBox))    pre agent   in set
dom vc.AgentInfo;
```

```
Latest: CCTRL`ChangeID * nat -> CCTRL`ChangeID
Latest(change, derivations) ==
  if (derivations >1 ) then
     SFN`DeriveChange(change, derivations -1)
  else
     change
pre derivations >=0;

Reactivate: CCTRL`ChangeID * AVC`VersionControl -> AVC`VersionControl
Reactivate(change, vc) ==
let agents = vc.AgentInfo,
    originator =  vc.labels(change)(len vc.labels(change)),
     at = vc.AgentInfo(originator.id)
  in
    let newAgents = {a |-> mk_AVC`DesignAgent
        (mk_AVC`VersionLabel(at.version.time,vc.labels(change)),<Frozen>,
        agents(a).activators, agents(a).respondors, agents(a).alternatives,
        agents(a).replies, <alternative>) | a in set dom agents},
orig = { originator.id |-> mk_AVC`DesignAgent(mk_AVC`VersionLabel
        (at.version.time,vc.labels(change)), <Frozen>, at.activators,
        at.respondors,at.alternatives, at.replies, <declared>)}
    in
mk_AVC`VersionControl( vc.timestamp, vc.labels, newAgents++orig,
        vc.ChangeDetails,vc.RequiredChanges,vc.totalAlternatives,vc.MessageBox)
pre {v.vState | v in set rng(vc.AgentInfo++AVC`ResetAgents(vc.AgentInfo)) }
        = {<private>};


  AgentRemoveChange: map AVC`AgentID to AVC`DesignAgent * set of CCTRL`ChangeID
                -> map AVC`AgentID to AVC`DesignAgent
  AgentRemoveChange(agents,cid ) ==
   { a |-> mk_AVC`DesignAgent( agents(a).version, agents(a).activityState,
    cid <-: agents(a).activators,  cid <-: agents(a).respondors,
    cid <-:agents(a).alternatives , RemoveReplies(cid,agents(a).replies),
    agents(a).vState) | a in set dom agents};


RemoveReplies: set of CCTRL`ChangeID  * AVC`ReplyInfo ->
            AVC`ReplyInfo
 RemoveReplies(cid,  replies) ==
 let reps = { mk_AVC`ReplyID(c, agt.agent) | agt in set dom replies,
           c in set cid}
 in  reps <-: replies ;

 GetAllDerived: CCTRL`ChangeID * set of CCTRL`ChangeID -> set of CCTRL`ChangeID
 GetAllDerived(cid, changes) ==
 { c | c in set changes & SFN`Occurs(cid, c)};

 UpdateSequence: seq of CCTRL`ChangeID * CCTRL`ChangeID ->
              seq of CCTRL`ChangeID
  UpdateSequence(changes, change) ==
  if changes = []
  then []
  else if hd changes = change
     then if (tl changes) = [] then [] else [ hd (tl changes)]
     else [hd changes]

end PMG
```

```
-- Model SFN - Support_FuNctions
-- The functions specified in this module are somewhat auxiliary
-- to the main specification they include things like sorting algorithms
-- They are nevertheless critical to the executable specification
--Last Edit : August 10th 1998


module SFN

imports from  CCTRL all,
from AVC all,
from MSG all
exports all
definitions

functions
Prioritise: CCTRL`ChangeRequest -> MSG`MessageType
Prioritise(change) ==
cases change :
<Required> -> 1,
<Wish> -> 2,
<Evaluation> -> 3
end;

DeriveChange: CCTRL`ChangeID * nat -> CCTRL`ChangeID
DeriveChange(change,i)  ==
    change^['-', 'D' ,NatToChar(i)];

Ancestor: CCTRL`ChangeID -> CCTRL`ChangeID
Ancestor(change) ==
let i = Position(change, '-') in
        change (0,...,i);

Position: CCTRL`ChangeID * char -> nat
Position( s, c) ==
      if s= [] then 0 else
if hd s = c
then 0
else  1 + Position (tl s, c);


NatToChar: nat -> char
NatToChar(i) ==
cases i :
    0 -> '0',
    1 -> '1',
    2 -> '2',
    3 -> '3',
    4 -> '4',
    5 -> '5',
    6 -> '6',
    7 -> '7',
    8 -> '8',
    9 -> '9'
end
pre i < 10;

Occurs: seq of char * seq of char -> bool
Occurs (substr, str) ==
exists i,j in set inds str & substr = str(i,...,j);
```

```
LoopCount: nat * nat -> nat
LoopCount(i, limit) ==
 if ( i+1 > limit) then
     1
else
     i+1;

InitialiseModel: nat * set of AVC`AgentID -> AVC`VersionControl
InitialiseModel( start, agents)  ==
mk_AVC`VersionControl(start,{|->},
              CreateAgents(start, agents)
              , { |->},[], {|->},
              {a |->[] | a in set agents});


CreateAgents: nat * set of AVC`AgentID ->
                              map AVC`AgentID to AVC`DesignAgent
CreateAgents(t,agts) ==
{ a |-> mk_AVC`DesignAgent(mk_AVC`VersionLabel(t,[]),
<Active>,{|->},{|->},{|->},{|->},<private>) | a in set agts };



InitialiseChange : CCTRL`ChangeID  * AVC`VersionControl *
 set of AVC`AgentID * AVC`AgentID -> map AVC`AgentID to AVC`DesignAgent
InitialiseChange(change,  vc, agts, ag) ==
  { a |-> mk_AVC`DesignAgent(vc.AgentInfo(a).version,
  vc.AgentInfo(a).activityState ,vc.AgentInfo(a).activators++ {change
  |-><Normal>},vc.AgentInfo(a).respondors ++{change|-><Normal>} ,
  vc.AgentInfo(a).alternatives++{change|->0}, vc.AgentInfo(a).replies
  ++SetReplies(<NotDefined>,change,agts, a, ag ),
  vc.AgentInfo(a).vState) | a in set agts  };


SetReplies:  MSG`Reply * CCTRL`ChangeID * set of AVC`AgentID *
 AVC`AgentID * AVC`AgentID-> AVC`ReplyInfo
SetReplies(r,cid,agents, a, ag) ==
let agts = agents \{a} in
  if a = ag then { mk_AVC`ReplyID(cid, a) |->r | a in set agts }
  else {|->};

ChooseAlternative: AVC`AlternativeLabel -> AVC`AlternativeLabel
ChooseAlternative(alternatives) == alternatives;

DoSort: seq of MSG`OrderedMessage-> seq of MSG`OrderedMessage
DoSort(l) ==
  if l = [] then []
  else
     let sorted = DoSort (tl l) in
InsertSorted (hd l, sorted);

  InsertSorted: MSG`OrderedMessage* seq of MSG`OrderedMessage
              -> seq of MSG`OrderedMessage
  InsertSorted(i,l) ==
  cases true :
    (l = [])    -> [i],
    (i.priority <= (hd l).priority) -> [i] ^ l,
    others      -> [hd l] ^ InsertSorted(i,tl l)
  end
end SFN
```

```
module TST
imports from  CCTRL all,
from AVC all,
from SFN all,
from PMG all,
from MSG all

exports all

definitions

functions

--RunModel is a simple change entered at runtime
 RunModel: AVC`AgentID * CCTRL`ChangeID * CCTRL`ChangeRequest *
      CCTRL`Change * seq of AVC`AgentID -> AVC`VersionControl
 RunModel(agent, change, priority, details, agts) ==
 let  vc = SFN`InitialiseModel(0, elems agts) in
 let  vc2 =
  mk_AVC`VersionControl(vc.timestamp, vc.labels,
  SFN`InitialiseChange ( change, vc, elems agts, agent),
  vc.ChangeDetails, vc.RequiredChanges, vc.totalAlternatives, vc.MessageBox)
  in ProcessMailBoxes(
    CCTRL`CreateChange(agent,change,priority,details, vc2), 1, agts);

--RunModel2 is two successive changes specified here in the model

 RunModel2 : AVC`AgentID * CCTRL`ChangeRequest * CCTRL`Change ->
        AVC`VersionControl
 RunModel2(agent,priority, details) ==
 let agent1:AVC`AgentID = <process>,
    change1:CCTRL`ChangeID = "change1",change2:CCTRL`ChangeID = "change2",
     type:CCTRL`ChangeRequest = <Required>, detail:CCTRL`Change = <OK>,
     agents = [<process>, <cost>, <electrical>, <layout>] in
   let vc = RunModel(agent1, change1, type, detail, agents) in
    ProcessMailBoxes(CCTRL`CreateChange(agent, change2,
       priority, details, vc),1, agents);

--RunModel3  is four successive changes specified in the model
 RunModel3 : AVC`AgentID  -> AVC`VersionControl
 RunModel3(agent) ==
 let
   ag1:AVC`AgentID = <process>, ch1:CCTRL`ChangeID = "change1",
    type1:CCTRL`ChangeRequest = <Required>, detail1:CCTRL`Change =<Conflict>,
   ag2:AVC`AgentID = <cost>, ch2:CCTRL`ChangeID = "change1",
    type2:CCTRL`ChangeRequest = <Required>, detail2:CCTRL`Change =<Conflict>,
   ag3:AVC`AgentID = <electrical>, ch3:CCTRL`ChangeID = "change2",
    type3:CCTRL`ChangeRequest = <Evaluation>, detail3:CCTRL`Change =<Conflict>,
   ag4:AVC`AgentID = <cost>, ch4:CCTRL`ChangeID = "change3",
    type4:CCTRL`ChangeRequest = <Evaluation>, detail4:CCTRL`Change =<OK>,
     agents = [<process>, <cost>, <electrical>, <layout>] in
  let vc = RunModel(ag1, ch1,type1, detail1, agents) in
  let vc2 =
    ProcessMailBoxes(CCTRL`CreateChange(ag2,ch2,type2,detail2,vc),1, agents) in
  let vc3 =
    ProcessMailBoxes(CCTRL`CreateChange(ag3,ch3,type3,detail3,vc2),1,agents) in
    ProcessMailBoxes(CCTRL`CreateChange(ag4,ch4,type4,detail4, vc3),1, agents);
--ProcessMailBoxes effectively runs the model  by cycling through each agents
--mail box and picking off the top messsage. This is how we unwind the
--concurrency which is evident in the real system

 ProcessMailBoxes: AVC`VersionControl * nat * seq of AVC`AgentID ->
```

```
  AVC`VersionControl
  ProcessMailBoxes(vc, i, designAgents) ==
    if (vc.MessageBox(designAgents(i)) = []) then
      if  (rng vc.MessageBox = {[]}) then
vc
      else
ProcessMailBoxes(vc, SFN`LoopCount (i, len designAgents), designAgents)
    else
      let  j = SFN`LoopCount(i, len designAgents),
      newvc =mk_AVC`VersionControl( vc.timestamp, vc.labels,
      vc.AgentInfo , vc.ChangeDetails, vc.RequiredChanges,
      vc.totalAlternatives,
      vc.MessageBox ++ {designAgents(i) |-> tl vc.MessageBox(designAgents(i))}),
      msg = (hd vc.MessageBox(designAgents(i))).content in
          ProcessMailBoxes(PMG`ProcessCommand(msg, designAgents(i), newvc),
                           j, designAgents)


end TST
```

B NON EXECUTABLE SPECIFICATIONS

The non executable specification consists of two modules:

GLOBAL_AGENT
RESOURCE_AGENT

```
--module GLOBAL_AGENT

--exports all

--definitions
types
ObjectID = token;
RelationID = token;
VersionLabel = token;
Agent = <Process> | <Electrical> | <Layout> | <Cost>;

state GlobalAgent of
configuration : VersionLabel
objects : map ObjectID to Object
relationships : map RelationID to Relation
agents : set of Agent
inv mk_GlobalAgent(configuration,objects,relationships,agents) ==
ObjectsUnique(objects) and
RelationShipsUnique(relationships)  and
ObjectRelationsConsistent(objects, relationships)

init ga == ga = mk_GlobalAgent(<a0>, {|->} , { |->}, {})

end


operations

CreateObject(oid:ObjectID) == (objects := objects munion {oid |-> object})
ext wr objects
pre ObjectID not in set dom objects;

CreateRelation(rid:RelationID) == ( relationships := relationships munion {rid |-> relation})
ext wr relationships
pre rid not in set dom relationships;

DeleteObject(oid:ObjectID) == (objects := {oid} <-: objects)
ext wr objects
pre oid in set dom objects;

DeleteRelation(rid:RelationID) == (relationships := {rid} <-: relationships)
ext wr relationships
pre rid in set dom relationships;


--end GLOBAL_AGENT


-- Resource Agent
-- November 24th 1997
-- Part of agent transition diagram project
-- Represents the interface that a resource agent displays
```

```
-- Last Edit: June 18th 1998

module RESOURCE_AGENT

exports all

definitions
    types
        ObjectID = token;
        ChangeID = token;
        VersionLabel = token;
        VersionStates = <private> | <declared> | <alternative> | <removed>;

        UniqueObject :: object : ObjectID
                version : VersionLabel;
Express = token;

Add :: object :ObjectID
        model : Express;

Delete :: object : ObjectID;

Modify :: object : ObjectID
      model : Express;

ChangeTypes = Add| Delete | Modify;
Deltas = seq of ChangeTypes;

NewVersion :: change : ChangeID
          version : VersionLabel
          deltas : Deltas;

Retrieve:: object : ObjectID
      version : VersionLabel;

EvaluateC:: change : ChangeID
              version: VersionLabel;
Resolve:: change:ChangeID
      version: VersionLabel;

Messages = NewVersion | Retrieve | EvaluateC | Resolve;


state ResourceAgent  of
      version: VersionLabel
      objects : set of ObjectID
      changes: map ChangeID to Deltas
      modelStates: map VersionLabel to VersionStates
      expressmapping : map UniqueObject to Express

end

        operations
-- ProcessMessage is the message interpreter at the agent
        ProcessMessage(msg : Messages)
        == cases msg :
        mk_NewVersion( change, vers, deltas) -> CreateVersion(change, vers, deltas),
        mk_Retrieve(object, vers) -> RetrieveObject(object, vers),
        mk_EvaluateC(change, vers) -> EvaluateChange(change, vers),
        mk_Resolve(change, vers) -> ResolveChange(change, vers),
        others -> ResourceAgent := ResourceAgent
end
```

```
ext wr version
   wr objects
        wr changes;

--Controlling logic of behaviour

CreateVersion(chng : ChangeID, vers : VersionLabel, deltas : Deltas)
ext wr version
     wr objects
     wr changes
post version = vers and Declare(vers) and StoreDeltas(chng, deltas);

EvaluateChange(chng: ChangeID, vers : VersionLabel)
ext wr version
     wr objects
    wr changes
post version = vers and Evaluate(vers);

ResolveChange(chng: ChangeID, vers : VersionLabel)
ext wr version
     wr objects
    wr changes
post version = vers ;
--should also remove any alternative versions but
--this is controlled by the Behavioural Agent calling the Remove operation


RetrieveObject(o:ObjectID, version: VersionLabel) obj: Express
ext rd expressmapping
post obj = expressmapping(mk_UniqueObject(o, version));

--Auxiliary operations

StoreDeltas(chng : ChangeID, deltas: Deltas) result:[bool]
ext wr objects
     wr changes
post objects = objects~ union ProcessDeltas(deltas)
              and changes = changes~ munion {chng |-> deltas};


--Operations which state the allowable state transitions

Activate(vers:VersionLabel)
ext wr modelStates
pre modelStates(vers) = <alternative>
post modelStates = modelStates~ ++ {vers |-> <private>};

DeclareAlternative(vers:VersionLabel)
ext wr modelStates
pre modelStates(vers) = <private>
post modelStates = modelStates~ ++ {vers |-> <alternative>};

Evaluate(vers:VersionLabel) result:[bool]
ext wr modelStates
pre modelStates(vers) = <declared>
post modelStates = modelStates~ ++ {vers |-> <alternative>} and result = true;

Declare(vers:VersionLabel)  result:[bool]
ext wr modelStates
pre modelStates(vers) = <private>
post modelStates = modelStates~ ++ {vers |-> <declared>} and result = true;
```

```
Remove(vers:VersionLabel)
ext wr modelStates
pre modelStates(vers) = <declared> or modelStates(vers) = <alternative>
post modelStates = modelStates~ ++ {vers |-> <removed>} ;


functions

ProcessDeltas: Deltas -> set of ObjectID
ProcessDeltas(deltas) ==

if deltas = []
then {}
else  ProcessDeltas(tl deltas) union
{ProcessSingleDelta(hd deltas)}



ProcessSingleDelta:ChangeTypes  -> ObjectID
ProcessSingleDelta(change) ==

cases change:
mk_Add(object, model) -> object,
mk_Delete(object) -> object,
mk_Modify(object, model) ->object
end
;
end RESOURCE_AGENT
```

References

Accreditation Board for Engineering and Technology, Inc. Annual Report for the year ending September 30, 1988, New York, 1988.

Agerholm, S., Lecoeur, P-J., and Reichert, E.,"Formal Specification and Validation at Work: A Case Study using VDM-SL", *Proceedings of Second Workshop on Formal Methods in Software Practice*, Florida, ACM, March 1998.

Ahmed, R., De Smedt, P., Du, W., Kent, W., Ketabchi, M., Litwin, W., Rafii, A. & Shan,M-C., "The Pegasus Heterogeneous Multidatabase System", *Computer*, Vol 24 No 12 pp 19-27, 1991.

Ananthanarayanan, R., Gottemukkala, V., Kaefer, W., Lehman, T.J. & Pirahesh, H., "Using the Co-existence Approach to Acheive Combined Functionality of Object-Oriented and Relational Systems", *SIGMOD RECORD*, pp 109-117, 1993.

Avouris, N. M., "Cooperating Knowledge-Based Systems for Environmental Decision Support", *Knowledge-Based System*, Vol 8, No 1, pp 39-54, 1995.

Batory, D. S. & Kim, W. "Modeling concepts for VLSI CAD objects", *ACM Transactions on Database Systems*, Vol 10, No 3, pp 322-346, 1985.

Ball, N., Mayyhews, P., Wallace, K., "Managing Conceptual Design Objects", *in* J. S. Gero & F. Sudweeks (eds.), *Artificial Intelligence in Design '98*, Kluwer, Dordrecht, The Netherlands, pp. 67-86, 1998.

Ball, N. & Bauert, F., "The integrated design framework: Supporting the design process using a blackboard system. *in* J. S. Gero (ed.), *Artificial Intelligence in Design '92*, Kluwer, Dordrecht, The Netherlands, pp. 327-348, 1992.

Barbuceanu, M & Fox, M.S., "*Capturing and Modeling Coordination Knowledge for Multi-Agent Systems*," http://www.ie.utoronto.ca/EIL/ABS-page/ijicis-feb96.html, 1996.

Bayardo, R., Bohrer, W., Brice, R., Cichocki, A., Fowler, G., Helal, A., Kashyap, V., Ksiezyk, T., Martin, G., Nodine, M., Rashid, M., Rusinkiewicz, M., Shea, R., Unnikrishnan, C., Unruh, A., Woelk, D., " InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments.", URL *http://www.mcc.com/projects/infosleuth*, 1998.

Benech, D., Desprats, T. & Raynaud, Y., COBALT: An Architecture for Intelligent Agent-based Management, *http://www.irit.fr/~Dominique.Benech/docs/noms98/index.htm*, 1998.

Ben-Nation, R., "*CORBA: A Guide to Common Object Request Broker Architecture*", McGraw-Hill, 1995.

Bell, D. & Grimson, J., "*Distributed Database Systems*", Addison-Wesley Pub., 1994.

Biskup,J, "A foundation of Codd's relational maybe operations", *ACM Transactions of Database Systems*, Vol 8 pp 608-636, 1983.

Blessing, L, "*A Process-Based Approach to Computer Supported Engineering Design*, Ph.D. Thesis, University of Twente, Netherlands, 1994.

Brodie, M, "On the Development of Data Models" , *in* Brodie, Mylopoulos & Schmidt (eds): *On Conceptual Modelling*, Springer Verlag, 1984.

Buxton, I.L. & Softley J., "A cost estimating tool for designers at the concept stage of made-to-order product devlopment.", *Proceedings of the 1st Annual International Conference on Industrial Engineering Applications and Practice*, 4-7 Dec 1996.

Buxton, I.L.& Bull, R.J., " An interactive cost estimating tool for the early stage of made-to-order product design.", *11th international conference; Applications of Artificial Intelligence in Engineering*; Clearwater, Fl, USA. Wessex Institute of Technology; University of South Florida (organisers), 11-13 Sept 1996.

Carnduff, T. & Gray, A., "Functional materialization in object-oriented databases", *Proceedings of International Conference on Object-Oriented Information Systems (OOIS)*, Springer-Verlag, pp 292-305, 1994.

Cellary, W. & Jomier, G., "Consistency of Versions in Object-Oriented Databases", *Proceedings of the 16th International Conference on VLDB*, 1990.

Chauhan, D. , "*JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*", Thesis , ECECS Department, University of Cincinnati, 1997.

Chao, K-M., "*Knowledge Sharing and Reuse For Engineering Design*", Ph.D. Thesis 1997.

Chao, K-M, Guenov, M., Hills, B., Smith, P., Buxton, I., & Tsai, C-F, "An Expert System to Generate Associativity Data for Layout Design", *Journal of AI in Engineerin*g, Vol 11 No 2, pp. 191-196, ISSN 0954-1810, 1997.

Chao, K-M, Guenov, M., Smith, P., & Husein, S., "Using CORBA to Achieve Knowledge Sharing in the Design Process of Made-To-Order Products", *Proceedings of Polymodel Conference 16,*. University of Sunderland, UK, 1995.

Chou, H-T. and Kim, W., " A Unifying Framework for Version Control in a CAD Environment",*Proceedings of the 12th International Conference on VLDB*, Kyoto, Japan, 1986.

Clamen, S.M., "Schema Evolution and Integration", *Distributed and Parallel Databases*, Vol 2 No 1, pp 101-126, 1994.

Cleetus, K.J., "*Definition of Concurrent Engineering*", CERC Technical Report Series, CERC-TR-RN-92-003, 1992.

Codd, E. F., "A Relational Data Model for Large Shared Data Banks", *Communications of the ACM*, Vol 13 No 6, 1970.

Codd, E. F., "Extending the database relational model to capture more meaning", *ACM Trans. Database Sys*, Vol 4 pp 397-434, 1979.

Cutcosky, M. R., Engelmore, R. S., Fikes, R., Genesereth, M. R., Gruber, T. R., Mark, W. S., Tenenbaum, J. M., & Weber, J.C., "PACT: An Experiment in Integrating Concurrent Engineering Systems", *IEEE Computer*, 1993.

Date, C. J., "*An introduction to database systems*" ,6th ed., Addison-Wesley Pub. , 1995.

Dattola, A., "Collaborative Version Control in an Agent-Based Hypertext Environment", *Information Systems Journal*, Vol 20 No 4, pp 337-359, 1996.

Decker, K., and Lesser, V., "Generalizing the Partial Global Planning Algorithms", I*nternational Journal of Intelligent and Cooperative Information system*s, June, 1992.

Demichiel,L. G., "Resolving Database Incompatibility : An Approach to Performing Relational Operations over Mismatched Domains", *IEEE Transactions on Knowledge and Data Engineering*, Vol 1 No 4 pp 484-493, 1989.

Dittrich, K.R. and Lorie, R.A. , "Version Support for Engineering Database Systems", *IEEE Transactions on Software Engineering*, Vol 14 No 4, pp. 429-437, 1988.

Duwairi, R.M., Fiddian, N.J., & Gray, W.A., "Schema Integration Meta-knowledge Classification and Reuse", *Proceedings 14th British National Conference on Databases (BNCOD 14)*, Edinburgh, 1996.

Dyer, D. E., "Multiagent Systems and DARPA", *Communications of the ACM*, Vol 42 No 3 ,pp 53-55, 1999.

Dym, C.L., "*Engineering Design: A Synthesis of Views*", Cambridge University Press, 1994.

Ecklund, D.J., Ecklund, E.F., Eifrig, R.O., Tonge, F.M., "DVSS: A distributed version storage server in CAD applications.", *Proceedings of the 13th VLDB Conference*, Brighton, UK, 1987.

Eliassen, F. & Karlsen R., "Interoperability and Object Identity", *SIGMOD RECORD*, Vol 20 No 4 pp 25-29, 1991.

Eliassen, F. , "Managing Identity in Global Object Views", *Proceedings RIDE-DOM '95* , Taiwan pp 70-77, 1995.

Elmasri, R., "*Fundamentals of Database Systems*", Benjamin/ Cummings Pub., 2nd Edition, 1994.

Elmstrøm, R., Larsen,P.G., Lassen, P.B., "The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications", *ACM SIGPLAN Notices*, Summer 1994.

Ertas A. and Jones J.C., "*The Engineering Design Process*", John Wiley and Sons, ISBN 0-471-51796-8, 1993.

Evbuomwan, N.F.O., Sivaloganathan, S. and Jebb, A., "A survey of design philosophies, models, methods and systems", *Proceedings of the Institution of Mechanical Engineers*, Vol. 210, 1996.

Fiddian N.J.,Gray W.A., Ramfos, A., Cooke, A., *"Database meta-translation technology: Integration, status and application,* Database Technology , 1992.

Fisher, M., "Introduction to executable temporal logics ", *Knowledge Engineering Review,* Vol 11 No 1, 1996.

Finin, T., "*Specification of KQML Agent-Communication Language*", Computer Science Department, Stanford University, 1993.

FIPA , "*FIPA 99 Specification*", Foundation for Intelligent Physical Agents, Geneva, Switzerland, 1999.

Florida-James, B., Hills, W. and Rossiter, B.N. , "Semantic Equivalence in Engineering Design Databases", *Proceedings, 4th International Workshop on Knowledge Representation meets Databases*, Athens, Greece, 1997.

Florida-James B. O., Rossiter B.N. & Chao, K-M., "An Agent System for Collaborative Version Control in Engineering", *Special Issue - Journal of Integrated Manufacturing System (The International Journal of Manufacturing Technology Management)*, Vol 11 No 4, 2000.

Foner, L., "What is an Agent Anyway? A Sociological Case Study," *Agents Memo 93-01*, MIT Media Lab, Cambridge, MA, 1993.

Fowler, J., "*STEP for Data Management Exchange and Sharing*", Technology Appraisals, Twickenham, 1995.

Frankhauser, P., Kracker, M. & Neuhold, E., " Semantic vs. Structural Resemblance of Classes" , *SIGMOD RECORD*, Vol 20 No 4, pp 59-63, 1991

Franklin, S. & Graesser, A., "Is it an agent, or just a program?", in Muller, J.P., Wooldridge, & Jennings, N.R. (eds.), *Intelligent Agents III,* pp 21-36, Springer-Verlag, Berlin, Germany, 1997.

French, M., "*Conceptual Design for Engineers*", Third Edition, Springer, ISBN 1-85233-027-9, 1999.

Frohlich B. & Larsen, P.G., " Combining VDM-SL specifications with C++ code." In James Woodcock Marie-Claude Gaudel, editor, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of LNCS, pages 179-194. Springer, March 1996.

Gangopadhyay, D. & Barsalou, T., "On the semantic equivalence of Heterogeneous Representations in Multimodel Multidatabase Systems", *SIGMOD RECORD*, Vol 20 No 4, pp 35-39, 1991.

Gasser, L. & Bond, A. H. (eds.), "*Readings in Distributed Artificial Intelligence*", San Mateo, CA: Morgan Kaufmann, 1988.

Geller,J and Perl,Y and Neuhold,E.J, "Structure and Semantics in OODB Class Specifications", *SIGMOD RECORD*, Vol 20 No 4 pp 40-43, 1991.

Genesereth, M.R. & Ketchpel, S.P. ,"Software Agents", *Communications of the ACM*, Vol 37, pp 48-53, 1994.

Gray, W.A., Wikramanayake, G.N. & Fiddian, N.J., "Assisting Legacy Database Migration", *IEE Colloquium*, 1994.

Gray, P. M. D.; Preece, A.; Fiddian, N. J.; Gray, W. A.,KRAFT: Knowledge Fusion from Distributed Databases and Knowledge Bases, International workshop; *8th Database and expert systems applications (DEXA)*, IEEE Computer Society, Toulouse, France, 1997.

Gray, P.M.D., Cui, Z., Embury, S.M., Gray, W.A. Hui, K. & Preece, A., "An Agent-Based System for Handling Design Constraints", *Workshop on Agent-Based Manufacturing at Agents'98*, Minneapolis, USA, 1998.

Guenov M., Florida-James B., Chao K-M., Smith N., Hills W. and Buxton I., "Tracing the Effects of Design Changes across Distributed Design Agents", *2nd World Conference on Integrated Design & Process Technology*, Austin Texas, 1996.

Haas, L. M., Kossmann, D., Wimmers, E. L., Yang, J., "Optimizing Queries Across Diverse Data Sources", *Proceedings of the Twenty Third international Conference on Very Large Databases*, Athens, Greece, 1997.

Haas, L. M., Miller, R. J., Niswonger, B., Roth, M.T., Schwarz, P.Wimmers, & E. L., "Transforming Heterogeneous Data with Database Middleware: Beyond Integration", *Data Engineering Bulletin*, 1999.

Haskin, R.L., & Lorie, R.A., "On Extending the functions of a reltaional database system", *Proceedings of the ACM SIGMOD Conference*, ACM, New York, pp 207-212, 1982.

Hawkes B. and Abinett R., "*The Engineering Design Process*", Longman Scientific & Technical, ISBN 0-582-99471-3, 1984.

Hills, W. & Smith, N., "A New Approach to Spatial Layout Design in Complex Engineered Products", *Proceedings of the International Conference on Engineering Design (ICED 97)*, Tampere, Finland, August 19-21, 1997.

Hsu, Cheng, "The Metadatabase Project at Rensselaer", *SIGMOD RECORD*, Vol 20 No 4, pp 83-90, 1991.

Hubka V., "*Principles of Engineering Design*", Butterworth, ISBN 0-408-01105-X, 1982.

Hutchinson, K.W., Todd, D.S., Sen, P., "An Evolutionary Multiple Objective Strategy for the Optimisation of Made-To-Order Products with special reference to the conceptual design of high speed mono hull Roll-On/Roll-Off Passenger Ferries", *Proceedings International Conference of Royal Insitution of Naval Architects AUSMARINE '98*, Australia, 1998.

IFAD, The VDM-SL Tool Group, "*The IFAD VDM-SL Language.*", Technical Report IFAD-VDM-1, The Institute of Applied Computer Science, 1994.

Informant, "The Informant", *http://informant.dartmouth.edu*, 1997.

ISO95 International Organisation for Standardisation , "*Information technology - Programming languages, their environments and system software interfaces - Vienna Development Method – Specification Language - Part 1: Base language*", International Standard ISO/IEC 13817-1, 1996.

ISO, International Organisation for Standardisation , "*Requirements for the second edition of EXPRESS*", International Standard TC184/SC4/WG5 - N252 (P2) , Dec. 27, 1995.

ISO, International Organisation for Standardisation , "*AP 231 - Release Draft*", International Standard TC184/SC4/WG5: ISO 10303: Part 231 , 1996.

Jennings, N. R. & Sycara, K., Wooldridge, M., "A Roadmap of Agent Research and Development", *Int Journal of Autonomous Agents and Multi-Agent Systems* Vol 1 No 1, pp 7-38, 1998.

Jha, K. N., Morris, A., Mytych, E., Sperring, J., "Agent Support For Design of Aircraft Parts", *1998 ASME Design Engineering Technical Conferences*, Atlanta, Georgia, 1998.

Johnson, M., "XML for the absolute beginner", *JavaWorld, at http:// www.javaworld.com/javaworld/jw-04-1999/jw-04-xml.html*, April 1999.

Katz, R., "Toward a Unified Framework for Version Modelling in Engineering Databases", *ACM Computing Surveys*, Vol 22, No 4, pp 375-408, 1990.

Keller A.M. & Ullman, J.D., " A version numbering scheme with a useful lexicographical order", *Proceedings of the IEEE Data Engineering Conference*, Taipei, Taiwan, 240-248, 1995.

Kent, W.,"The Breakdown of the Information Model in Multi-Database Systems", *SIGMOD RECORD*, Vol 20 No 4 pp 10-15, 1991.

Kim, I., Carnduff, T., Gray, A., Miles, J., "DESCRIBE: An Object-Oriented Desgin System to Support Concurrent Reuse of Data in Building and Engineering Design", *The Second International Conference on Object-Oriented information systems, OOIS '95*, Dublin, Ireland, 1995.

Kim, I., Carnduff, T., Gray, A., Miles, J., "An Information System for Concurrent Reuse of Construction Data in Design", *Information Processing in Civil and Structural Engineering Design*, pp 35-37, Inverleith-Spottiswoode, 1996.

Khoshafian, S., "*Object-Oriented Databses*", John Wiley & Sons Inc., 1993.

Kilpi, T. ,"New Challenges for Version Control and Configuration Management: a Framework and Evaluation", *Proceedings of EuroMicro Conference on Software Maintenance and Re-engineering*, Finland, 1997.

Kim, W. & Jungyun, S., "Classifying schematic and data heterogeneity in multidatabase systems", *Computer*, Vol 24 No 12, pp 12-18, 1991.

Kim, W.(ed.), "*Modern Database Systems*: *The Object Model Interoperability and Beyond*", Addison-Wesley Publishing Company, 1995.

King, B., Steward, A.P. & Tait, J.I., "Toward Automated Knowledge Acquisition for Process Plant Diagnosis", *IEE Colloquium on Knowledge Discovery in Databases*, IEE Professional Group C4(AI), Digest No 1995/021(b), London, 1995a.

King, B., "*Automatic Extraction of Knowledge from Design Data*", Ph.D. Thesis, University of Sunderland, 1995b.

Klahold, P., Schlageter, G. & Wilkes, W., "General model for version management in databases " *Proceedings of Twelflth International Conference on Very Large Data Bases (VLDB '86)*,Kyoto, Japan, 1986.

Krishnamurthy, K. & Law, K.H., (1994), "Towards A Formal Model of Version and Configuration Management For Collaborative Engineering", *Proceedings of ASME Database Symposium*, ASCE, NY, pp 21-32, 1994.

Krishnamurthy, K. & Law, K.H., (1997), "A Data Management Model for Collaborative Design in a CAD Environment", *Engineering with Computers*, Vol 13 No 2, pp 65-86.

Landis, Gordon S., " Design evolution and history in an object-oriented cad/cam database", *Proceedings of the IEEE Computer Society International Conference*, 1986.

Larsen, P.G., Fitzgerald, J., & Brookes, T., "Applying Formal Specification in Industry", *IEEE Software*, May 1996.

Larsen, P.G., Plat. N.,"An Overview of the ISO/VDM-SL Standard" ,*ACM SIGPLAN Notices*, Vol 27 No 8, August 1992a.

Larsen, P.G., Plat. N., "Standards for Non-Executable Specification Languages" , *BCS Computer Journal*, Vol 35 No 6, December 1992b.

Larsen, P.G., Pawlowski, W., " The formal Semantics of ISO VDM-SL", *Computer Standards and Interfaces*, Vol 17 No 5-6, 1995.

Larson,J., Navathe,S. and Elmasri,R., "A Theory of Attribute Equivalence in Databases with Application to Schema Integration", *IEEE Transactions on Software Engineering*, Vol 15 No 4 pp449-463, 1989.

Lassen, P.B., "IFAD VDM-SL Toolbox", *Proceedings of FME'93: Industrial-Strength Formal Methods*, Springer-Verlag, Odense April 1993.

Levy, S. , Subrahmanian, E., Konda, S., Coyne, R., Westerberg, A., Reich, Y., *"An overview of the n-Dim environment"*, EDRC Technical Report, 05-65-93, Carnegie-Mellon University, 1993.

Litwin, W. and Ketabchi, M. and Krishnamurthy,R., "First Order Normal Form for Relational Databases and Multidatabases", *SIGMOD RECORD*, Vol 20 No 4 pp 74-76, 1991.

Litwin, W., "From Database Systems to Multidatabase systems: Why and How", *Proceedings 6th National Conference on Databases*, pp 161-187, 1988.

Luck, M. & D'Inverno, M., "A Formal Framework for Agency and Autonomy", *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS 95)*, AAAI, San Francisco, CA, 1995.

MADEFast, at ,*http://www.madefast.stanford.edu*, 1999.

Maes, P., "Agents that Reduce Work and Information Overload", *Communications of the ACM*, Vol 37 No 7, pp 31-40, 1994.

Maes, P., "Aritificial Intelligence meets Entertainment: Lifelike Autonomous Agents," *Communications of the ACM*, Vol 38 No 11, pp 108-114, 1995.

Maes, P., "*Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*", London, The MIT press, 1991.

Medland A.J., "*The Computer-Based Design Process*", Second Edition, Chapman & Hall, ISBN 0-412-44780-0, 1992.

Molina, A., Al-Ashaab, A.H., Ellis, T.I.A., Young, R.I.M., & Bell, R., "A Review of Computer-Aided Simultaneous Engineering Systems", *Journal of Research in Engineering Design*, Vol 7, pp 38-63, 1995.

Monk, S "*A Model for Schema Evolution in Object-Oriented Database Systems*", PhD Thesis, Dept of Computing Science, Lancaster University, 1993.

Mori, T. & Cutcosky, M.R., "Agent-Based Collaborative Design of Parts in Assembly", *1998 ASME Design Engineering Technical Conferences*, Atlanta, Georgia, 1998.

Motro, A., "Superviews : Virtual Integration of Multiple databases", *IEEE Transactions on Software Enginering*, Vol SE-13 No 7 pp 785-798, 1987.

Mowbray, T. J. & Zahavi, R., "*The Essential CORBA: Systems Integration Using Distributed Object*", Object Management Group & John Wiley & Sons Inc., 1995.

Navathe, S.B., Batini, C., & Lenzerini, M., "A Comparative Analysis of Methodologies for Database Schema Integration", *ACM Computing Surveys*, Vol 18 No 4, pp 323-364, 1986a.

Navathe, S. , Elmasri, R. & Larson, J, "Integrating User Views in Database Design", *Computer*, Vol 19 No 1 pp 50-62, 1986b.

Object Design Inc., "*ObjectStore C++ API User Guide Release 4*", June 1995.

Object Management Group (OMG), "*The Common Object Request Broker: Architecture and Specification*", OMG & x/Open, 1991.

Object Management Group (OMG), "*The Common Object Broker: Architecture and Specification : Release 2 *", OMG, June 1995.

Object Management Group (OMG), "*The Common Object Broker: Architecture and Specification* : Revision 2.1 ", OMG, August 1997.

Odyssey, "Odyssey Information",*http://www.genmagic.com/technology/odyssey.html,* General Magic Inc., 1997.

O'Hare, G. M. P., & Jennings, N. R., editors, *Foundations of Distributed Artificial Intelligence*, Wiley-Interscience: New York, 1996.

Orfali, R., Harkey, D., & Edwards, J., "*The Essential Distributed Objects Survival Guide*", John Wiley & Sons, New York, 1996.

Owen,J., "*STEP An Introduction*", Information Geometers Ltd, 1993.

Pahl, G. and Beitz, W., *Engineering Design - A Systematic Approach (2nd Edition)*, K Wallace (ed.), Springer Verlag, Berlin, 1996.(1996)

Pahng, K. F., Senin, N. and Wallace, D. R., 1998, "Distributed modelling and evaluation of product design problems", *Computer Aided Design*, May 1998, v 30, n 6, p 411-423.

Pazzaglia, J-C.R. & Embury, S.M., "Bottom-up Integration of Ontologies in a Database Context ", *KRDB'98 Workshop on Innovative Application Programming and Query Interfaces*, Seattle, WA, USA, 1998.

Poulovassilis, A. & McBrien, P., "A General Formal Framework for Schema Transformation" , *Data & Knowledge Engineering*, Vol 28 No 1, pp 47-71,1998.

Quillion Systems Limited: *PETS Product Data Management*, 1995.

Qutaishat M.A., Fiddian N.J. & Gray W.A.," Extending OMT to support bottom-up design modelling in a distributed database environment", *Data & Knowledge Engineering* 22, pp 191-205, 1997.

Qutaishat M.A., Fiddian N.J. & Gray W.A.,"Association merging in a schema meta-integration system for heterogeneous object-oriented database environment", *Proceedings 10th British National Conference on Databases*, Sprigner-Verlag, 1992.

Qutaishat M.A., Gray W.A. & Fiddian N.J., "A Highly Customisable Schema Meta-Visualisation System for Object-Oriented Database Schemas - OverView", *Proceedings 4th International Conference on Database and Expert Systems (DEXA)*, Springer-Verlag, 1993.

Qutaishat M.A., Gray W.A. & Fiddian N.J., "Review and Potential of Meta-Programmed Expert Systems in a Heterogeneous Distributed Database Environemnt", P*roceedings Symposium on Advances in Database and Expert Systems,* 1996.

Ray M.S., *"Elements of Engineering Design: An Integrated Approach"*, Prentice Hall International, ISBN 0-13-264177, 1985.

Rajaraman, A., & Norvig, P., "Virtual Database Technology: Transforming the Internet into a Database", *IEEE Internet Computing*, pp 55-58, 1998.

Rochkind, M. , "The Source Code Control System", *IEEE Transactions on Software Engineering*, Vol SE- 1 No 4, pp 364-370, 1975.

Roddick, J.F., "Survey of Schema versioning issues for database system", *Information and Software Technology*, Vol 37 No. 7 pp 383-393, 1995.

Rodin, *"DISCO (Distributed Information Search Component)"*, at http:// rodin.inria.fr/disco/, 1999.

Roth, M.T. & Schwarz, P., "Don't Scrap It, Wrap It!" , *Proceedings of the Twenty Third international Conference on Very Large Databases*, Athens, Greece, 1997.

Rumbaugh, J., "Controlling propagation of operations using attributes on relations, *Proceedings of the OOPSLA '88 Conference*, ACM, New York, pp 285-296, 1988.

Rumbaugh,J., Blaha,M., Premerlani,W., Eddy,F., Lorensen,W., *"Object-Oriented Modelling and Design"*, Prentice-Hall International, 1991.

Rumbaugh, J. & Booch, G., "*Unified Method*", Rational Software, 1995.

Rusinkiewicz, M., Sheth, A. & Karabatis, G., "Specifying Interdatabase Dependencies in a Multidatabase Environment", *Computer*, Vol 24 No 12 pp 46-53, 1984.

Saltor,F. and Castellanos,M. and Garcia-Solaco,M., "Suitability of data models as canonical models for federated databases", *SIGMOD RECORD*, Vol 20 No 4 pp 44-48, 1991.

Santoyridis, I., Carnduff, T.W., Gray, W.A., Miles, J.C., "An Object Versioning System to Support Collaborative Design within a Concurrent Engineering Context",*Proceedings 15th British National Conference on Databases (BNCOD 15)*, 1997.

Shan, M-C., Ahmed, R., Davis, J., Du W. & Kent, W. , "Pegasus: A Heterogeneous Information management System", in *Modern Database Systems* pp 664-682, 1995.

Shen, W. and Barthes, J.P., "DIDE: A Multi-Agent Environment for Engineering Design", *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS 95),*AAAI, San Francisco, USA, 1995.

Sheth, A., & Larson, J., "Federated Database Systems for Managing Distributed Heterogeneous, and Automated Databases", *ACM Computing Surveys*, Vol 22 No 3, 1990.

Shoham, Y. , "Agent-Oriented Programming", *Artificial Intelligence*, Vol 60, pp 51-92, 1993.

Siegal, M. & Madnick, S., "Context Interchange : Sharing the Meaning of Data", *SIGMOD RECORD*, Vol 20 No 4 pp 77-78, 1991.

Singh, M. P., Joshi, A., "Multiagent Systems on the Net", *Communications of the ACM*, Vol 42 No 3,pp 38-40, 1999.

Sistla, A. & Woulfson, O., "Temporal Conditions and Integrity Constraints in Active Database Systems", *SIGMOD RECORD*, pp 269-280, 1995.

Smith, N., Hills, W. & Cleland, G. A Layout Design system for Complex Made-to-order Products, *Journal of Engineering Design*, 1996, vol. 7, no. 4, p.363-375.

Smith, N., *A concurrent approach to component layout.* Ph.D. Thesis 1998.

Snodgrass, R. & Ahn, I., "Temporal databases", *IEEE Computer*, Vol 19 pp 35-42, 1986.

Somers, F., "*KQML and CORBA*", at http://www-ksl.stanford.edu/e-mail-archives/ kqml.messages/327, 1997.

Spaccapietra, S. and Parent, C., "Conflicts and Correspondence Assertions in Interoperable Databases", *SIGMOD RECORD*, Vol 20 No 4 pp 49-54, 1991.

Sundsted, T., "XML and Java tackle enterprise application", *JavaWorld, at http:// www.javaworld.com/javaworld/jw-06-1999/jw-06-howto.html,* June 1999.

Telescript, Telescript Technology: The Foundation for the Electronic Marketplace, *http://www.genmagic.com/Telescript/Whitepapers/wp1/whitepaper-1.html*, General Magic Inc., 1996.

Urban, S. D. & Wu, J., "Resolving Semantic Heterogeneity Through the Explicit Representation of Data Model Semantics", *SIGMOD RECORD*, Vol 20 No 4 pp 55-58, 1991.

Ventrone, V. & Heiler,S., "Semantic Heterogenity as a Result of Domain Evolution", *SIGMOD RECORD*, Vol 20 No 4 pp 16-20, 1991.

Vines D., & King, T., "Gaia: AN Object-Oriented Framework for an Ada EnvironmenA", *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, June, 1988.

Wooldridge, M. & Jennings, N. R., "Intelligent Agents: Theory and Practice," *The Knowledge Engineering Review,* Vol 10 No 2, pp 115-152, 1995.

Wooldridge, M., Jennings, N. R. & Kinny, D., "A Methodology for Agent-Oriented Analysis and Design," *Proceedings of the 3rd International Conference on Autonomous Agents (Agents, 99),* Seattle, WA, 1999.

Wooley, D., "Configuration Management of a Ship Product Model", *Proceedings International Conference on Computer Applications in Shipbuilding* (ICCAS), Bremen, 1994.

Worboys, M. F., & Deen, S. M., "Semantic Heterogeneity in distributed Geographic Databases", *SIGMOD RECORD*, Vol 20 No 4 pp30-34, 1991.

Yang, J. and Papazoglou, M., "A Configurable Approach for Object Sharing among Multidatabase Systems", *4th International Conference on Information and Knowledge Management*, Baltimore, USA , pp 129-136, 1995.

Yu, C., Jia, B., Sun, W. & Doa, S., "Determining Relationships among Names in Heterogenoeus Databases", *SIGMOD RECORD*, Vol 20 No 4 pp 79-80, 1991.