Dependability and the Management of Large Distributed Systems

Darren R. Hodge

August 1995

Ph.D. Thesis

Department of Computing Science
The University of Newcastle upon Tyne
Kings Walk
Newcastle upon Tyne

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the Faculty of Science of the University of Newcastle upon Tyne.

ABSTRACT

This thesis concerns the design, implementation and application of a dependable management information system to aid in the controlling (monitoring) of large, complex distributed computer systems. Special attention has been given to using a person centred model of an organisation based on the ANSA "Enterprise Projection" and using fault tolerance techniques to provide continued service and recovery in the event of partial sub-system failure.

The information system is accessed using "management workbenches" (implemented in Tcl/Tk) which access managed resources using "probes" (implemented in C++). Existing "legacy systems" are incorporated in the prototype using "integration objects" which "wrap" system software, entry routines, configuration files etc. and enact operation upon logical (physical) resources. Application layer fault tolerance and recovery is implemented using type inheritance whereas remote operations are performed using the Arjuna tool kit.

The prototype information system was used to "manage" several resources including: workstations (running SunOS, Solaris and HP-UX), terminals, printers, disk and tape devices as well as software distributions. A mechanism for re-configuring multiple resources (such as workstation clusters and dedicated devices) based on "dependable change schedules" is presented and applied to change and fault management.

Key Words: Change Schedules, Enterprise Modelling, Fault Tolerance, Legacy Systems, System Management, UNIX®

ACKNOWLEDGEMENTS

I am very grateful for all the support and encouragement which I have received during my research, especially during my recent bereavement and while my (old) flat was repeatedly burgled. In particular, Dr. Lindsay Marshall and Prof. Santosh Shrivastava for their help and suggestions regarding this project, Dr. Adrian Waterworth who has read many drafts of this thesis and Rafhat Iqbal (my "office mate"). Special thanks to Jill Clifford, Annette Krowschewski, Steph and Alan Richardson; my friends from Heaton Baptist Church, particularly Ali Calvert, Dr. Andy Carlisle, Frank and Julia Cosgrove, Alex Grant, Bill Nicholson, Stephi and Rob Wallace, Margaret and Ken Webster, Pat and Joe Wilkes; and my family Kenneth, Valerie and Lisa Hodge.

Many thanks also to our librarian, Shirley Craig; Tim Smith, Steve Varty, Chris Ritson and Trevor Kirby for their technical help; Computing Services for use of their equipment; and members of the Arjuna group for "being there". My research has been part funded by the United Kingdom Science and Engineering Research Council. (Award reference: 8830487X)

The fear of the Lord is the beginning of all wisdom,
All who follow Him have good understanding,
To Him belongs eternal praise.

Psalm 111 v 10 (New International Version)

In Memoriam:
Reg V. Ollis (1907 - 1984)
Gladys F. Ollis (1912 - 1992)

(My Grandparents)

CONTENTS

TABLE OF FIGURES

TABLES

CODE SEGMENTS

Chapter 1

INTRODUCTION

It is a truth universally acknowledged that an organisation in possession of a good fortune must be in want of a large, complex computer system. However little is known of the feelings or views of their systems administrators ... *

The increased availability of sophisticated computer equipment and the corresponding decrease in cost has lead to the widespread use of very large, complex distributed computer systems in many organisations. These distributed computer systems not only comprise many and varied resources but can be geographically dispersed and cross organisational boundaries. Effectively managing such computer systems is very difficult due to their size and complexity. For instance CMU, one of the most computer intensive universities has over three thousand computers attached to its campus (backbone) network via several local area networks. Distributed computer systems are becoming so large that SUN micro systems advertising claims

"The Network *IS* the computer" ®

Many installations have become so computer dependent that their organisation "cannot cope" without a high level of information technology support. They can also incur much disruption (and financial loss) in the event of system failures and major changes to the system configuration. Unfortunately hardware manufacturer's and system software documentation often has little (if anything) to say regarding day to day system administration; worse still, the scale and complexity of such systems cannot simply be removed using standardisation and homogeneity. Even though the effective management of

* Adapted from Pride and Prejudice (Chapter 1) by Jane Austen (1775 - 1817)

the computing resources is so fundamental to the organisation's smooth running, very little research has been performed in this area and even existing research has overlooked several key areas fundamental to effective system management.

When designing a management information system to aid human system administrators in controlling and monitoring a very large distributed computer system, it is important to consider the role of the people within the organisation. Some system designs have been criticised for failing properly to consider people, as ANSA [ANSA90a] explain:—

- "some system design approaches suffer from a strong technological bias which treat the user as a peripheral object."
- "the influence of special groups ... has tended to make the design process concentrate on technical issues."
- "human concerns were simply ignored" partly because "of the lack of the rigorous theory or accepted guidelines ... and the complexity of the problems involved."
- "purchasers often respond to technological advances."

Therefore, the concerns of people within social groups (and the organisation) must be incorporated into the system model. Whilst it is almost certainly impossible to model every aspect of human needs, concerns, aspirations etc., ANSA have described a taxonomy of people and systems from a designer's viewpoint. This taxonomy models several groups of individuals within the organisation and comprises the following groups:—

- Users — these include "Service Users" such as managers, clerks and sales assistants; "Service Providers" which include system architects designers, and "System Engineers". System engineers are further subdivided into "Hardware Engineers", "Operating Systems Engineers", "Programmers and Operators" etc.

- Non-Users — which include "owners", "purchaser and lessee", "vendor", "organisation administrator" and "customer". These non-users do not interact directly with the computer system but nevertheless maintain an interest in its progress.

ANSA deliberately consider human concerns which include "people as individuals", "individuals as members of a social group", "individuals as members of an organisation" and "individuals as users of a system". These human issues concern a mixture of psychological, physiological and ergonomic factors which must be considered not only when designing the user interface but the whole system.

1.1 MANAGING AN ORGANISATION

Before considering the role of dependability in managing very large computer systems, let us first address two fundamental issues: what do we mean by management and where does management fit in to the organisation?

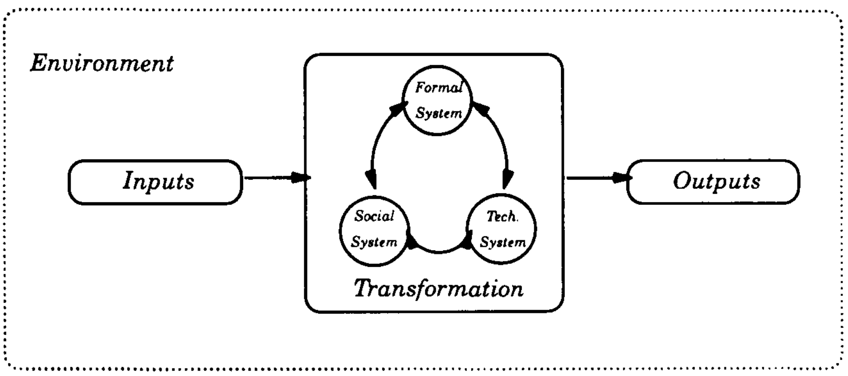
An organisation [Duncan81a] is a "collection of interacting and interdependent individuals who work together towards common goals and whose relationships are determined according to a common structure." This definition leads to several key points:—

- organisations are collections of people.
- people who work together within the organisation *interact*, and cooperate towards goals. These goals or policies need not be explicit at all levels but nevertheless members of the organisation are aware of the policies' existence.

- the organisation is structured. This structure can be *both* formal (organisational segmentation) and informal i.e. social groupings of employees. These informal groups can play an important role in the organisation.

It is possible to describe organisations using "systems theory." Organisations interact with their environment:— transforming *inputs* of goods and services, producing outputs, and are therefore "open systems."* This transformation process can be further subdivided in terms of a "formal system" (reporting responsibilities), "technology"(production system) and a "social system" (informal groupings) as shown in the following diagram:—

The Organisation viewed as an Open System.



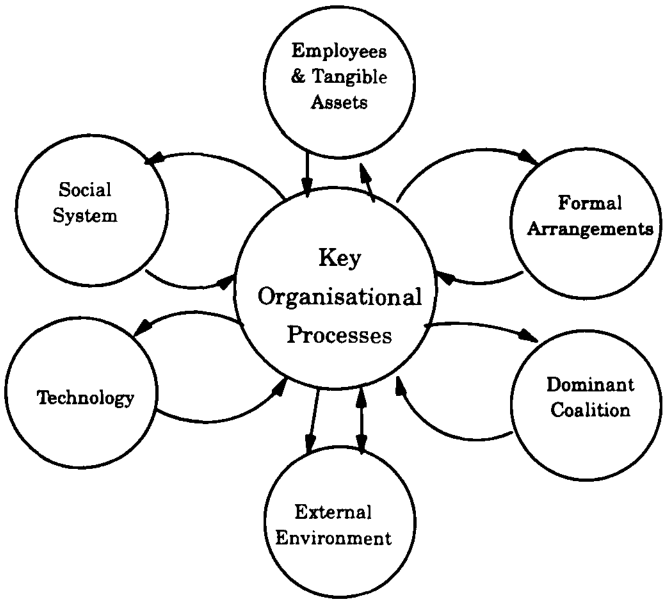
Objectives and strategies are important in order to manage an organisation. These policies shape the organisation's behaviour and give the organisation its character. Individual managers require *information* in order to implement policies; and this information can be processed, stored, retrieved and communicated to other managers.

Organisations comprise a formal structure, technology, environment and social system. Kotter[Kotter78a] has integrated these (and other aspects) into an overall framework for examining organisations, as shown in the following

* The term "open systems" is used within "systems theory" to denote systems which interact with their environment and should not therefore be confused with the ISO "Open Systems" definition.

diagram:—

Major Elements of Organisational Dynamics



- *Key Organisational Processes* — the major information gathering, communication, decision-making, matter/energy converting and transportation actions of the employees and machines.
- *External Environment* — including suppliers (labour, raw materials etc.), markets, competition and other factors related to the organisation.
- *Employees and Other Tangible Assets* — including employees, plan, offices, equipment, tools etc.
- *Formal Organisational Arrangements* — such as formal systems, procedures and regulations.
- *Social System* — this includes the employees "culture" and social (i.e. informal) relationships.
- *Technology* — the major techniques used by employees while engaging in organisational processes. (As contrasted with the actual equipment used by the employees.)

- *The Dominant Coalition* — concerns the organisation's "policy makers" and their objectives, strategies etc.

Kotter therefore provides a systematic checklist of interactions within an organisation. The model does *not* attempt to provide a complete picture of *every* possible element within an organisation, *nor* does the model attempt to understand or explain every possible interconnection and relationship. Nevertheless, this model identifies the wide variety of interactions within the organisation which must be analysed to maximise adaptability[Schein80a].

1.2 MANAGING LARGE COMPUTER SYSTEMS

Having briefly considered organisations and several aspects of managing an organisation, let us now turn our attention to using very large computer systems within an organisation. Very large (distributed) computer systems raise particular problems for system administrators, due to their size, complexity and the sheer volume (and variety) of interconnected resources. Worse still, the *traditional* approach of allocating *one* system administrator per machine is now impractical due to the availability of small but powerful units. Their scale, breadth of support and controlling technological improvements has grown beyond day-to-day managers, single departments or sub-departmental groups.

Organisation wide computer systems are managed by organisation wide management structures with access to senior executives. More than half of the (university) organisations considered in Arms's survey of "Campus Networking Strategies" created a responsible "executive" director in order to harness and focus their information technology and organisation's policies. This included budgetary and operational controls, effective use of resources and co-ordination between organisation departments. Controlling computer and networking technology are therefore an institutional priority. [Arms88a] Information technology "investment strategies" must be carefully planned in order to keep pace with rapid technological developments, standards and vendor reliability.

At CMU for example, their campus networking expertise is distributed between several departments. These include:—[Arms88b]

- Academic Services division — responsible for campus network, telephone and data services on campus.
- Vice-Provost for Research Computing — who leads a very strong team providing computing support for their major research groups.
- Associate-Provost for Scientific Computing — with particular expertise in national computer networks.
- Individual Departments and College Centres.

Even though these groups may have their own sets of priority, they recognised the value of sharing their expertise and resources. Day-to-day network management is performed by a central team of technicians and engineers responsible for both the *operational* and *evolutionary* aspects of the campus network. "Operational management" concerns "real time" issues, such as fault management, monitoring and policing the network etc.; whereas "evolution management", performed by their most experienced staff, concerns adapting the network based on traffic load and profiles.

Hence, the organisation's "policy makers" determine the organisation's aims, objectives and strategies. These policy's affect the organisation's management structure, formal communication channels and technological requirements. High level policies are refined and implemented by middle management and performed by operational staff. Managing CMU's large, complex network relies less upon their formal organisational structure than the good working relationships (i.e. social system) between key individuals and groups working in harmony towards a common goal.

1.3 OPERATIONAL MANAGEMENT

Whether computers are actually managed by an organisation's "computer departments" or by individual departments' staff, computer system management is a highly technical and specialised activity requiring in-depth knowledge of operating system interfaces, cabling requirements, software etc. System management tasks include:—

- Accounting Management — providing information about users' resource consumption (disk space, communication band width and processor time used), which is later used for billing.
- Back Up and File Restoration — regular, systematic file system archival is essential not only to guard against users accidentally deleting files but to protect the organisation from catastrophic disk failures such as head crashes. File archives need to be carefully labelled, catalogued and held in a secure, fireproof environment "just in case."
- Communications System Management — providing reliable secure communications between resources in the distributed system. This includes the actual wiring between components, physical interfaces to the network and controllers, as well as the logical system structure.
- Environment Management — ensuring that a noise/dust free, properly air conditioned environment is available for both employees and resources.
- Hardware Configuration and Maintenance — this includes upgrading resources, adding (rewiring) hardware devices and processor units.

Many organisations may sub-contract hardware maintenance to computer manufacturers and take out "on site" service agreements etc.

- Liaising with Other Management Departments — such as co-operative working and managing shared resources.
- Liaising with Users — such as answering users' queries and giving advice.
- Safety Management — ensuring that the distributed computer system does not endanger the organisation's employees. For example, that computers located in a "machine room" are properly configured on a 3 phase power supply.
- Security Management and User Authorisation — including "policing" the distributed system, granting users login and physical access to resources.
- Software Installation and Maintenance — ensuring that new releases are properly configured and smoothly installed on all appropriate machines. This can be particularly difficult as some software may require vendor/hardware specific configurations and different documentation.

1.4 DEPENDABILITY AND SYSTEM MANAGEMENT

In previous sections we have briefly considered the role of management (in general) and specifically managing large computer systems and the tasks performed by system administrators. We have noted that many organisations are heavily computer reliant and "cannot cope" without a high level of information technology support. Effective system management techniques are therefore required to compliment the organisation's goals, strategies and policies etc.

Organisations *must* be able to *rely* upon their distributed computer system and the ability of their system managers. Very large distributed computer systems raise particular problems for system administrators, due to their size, complexity and the sheer number of interconnected resources. These problems include:—

- Incompatabilities between software and hardware architectures. This leads to administrators maintaining several different versions of software distributions and replicated documentation.
- Inconsistent configuration information regarding network topologies, resources and distributed resources. This often occurs when configuration information is incorrectly propagated across the distributed system and leads to disjoint services etc.
- Side effects of management operations causing *unplanned* system configurations, i.e. system administrators corrupting resources while reconfiguring *other* system components.
- Debris from failed management operations scattered across the distributed computer system. This is particularly apparent when *major changes* to a large number of resources. In the event of *major* operations failing "en-route", system administrators must restore corrupted configuration files, network connections etc.

Some activities can be automated with the aid of management information systems. These allow system administrators to perform operations across a communications system and alter a physical (logical) resource's state and sometimes provide a query language to prepare management reports etc. For example, the Internet Working Group prepared two prototype systems, The High Level Entity Management System (HEMS)[Partridge88a] and Protocol (HEMP)[Partridge87a], and Simple Gateway Monitoring Protocol

(SGMP)[Case88a] which are superseded by the Simple Network Monitoring Protocol (SNMP)[Schoffstall89a].

Fault tolerance techniques must be incorporated when designing and implementing a management information system in order to ensure the dependability of management operations upon managed resources. In particular:—

- Support for failure atomicity of management operations, i.e. management operations upon physical (logical) resources either complete (i.e. commit) *or* "cleanly" abort. Any *non-recoverable* operations (such as printed output) must be compensated (such as discarding print jobs) in order to maintain the system's state.
- Management operations on resources should not interfere with other system activity (from system users or other managers). Management operations are therefore serialized, allowing *only* one system manager to alter a resource at any given time.
- When performing *major changes* to distributed resources or their topology, it *may* be desirable for system managers *gracefully* to shut down operations in the event of failure rather than (fully) aborting work already performed.

Ideally a system manager can log into the management information system and select a managed resource. She can then alter the physical (logical) resource's state / properties and perform management operations across the communications system. Should the operation fail, the resource's state etc. will be recovered and and appropriate action will be taken to clean up any debris.

Information should also be maintained concerning the distributed system's logical (physical) structure, such as network ports, sub-networks and ports etc. which can also be viewed (altered) using the management information system. This information should be used to maintain consistency between system components and used to generate (replace) system configuration files.

1.5 THESIS CONTRIBUTION, AIMS AND OBJECTIVES

This thesis concerns the design, implementation and application of a dependable management information system to aid system administrators control (monitor) a large, complex distributed computer system. We first consider the role of large, complex computer systems within an organisation (Chapter 2), standards and current management initiatives. Even though several research projects have considered system management and produced prototype management information systems, there are several shortcoming exist:—

- Some prototype systems use an unnatural model of management (if any!).
- Cleaning up failed operations, where an operation has crashed enroute leaving resources in an "intermittent" state.
- Management systems should incorporate (and abstract) operations upon logical (physical) resources in a *clean* and *portable* manner.
- Supporting *major* changes (i.e. both planned and unplanned) to the distributed computer system's logical (physical) structure and components.

Our management information system is based on a (deliberately) person centred model of management. "Agents" in management "roles" are allowed to inspect ("view") resources providing they hold a "contract". These contracts encode reporting responsibilities, job descriptions and conditions imposed on the management agent (e.g. major reconfigurations can only be performed at the weekend). Furthermore, delegation of duty is considered which is particularly useful for encoding trading arrangements between organisational departments ("domains").

A "system management information base" was constructed based on this model of management and implemented in C++ using the Arjuna distributed programming toolkit to access remote "probes" which enact management control (monitoring) actions upon physical resources. Design diversity techniques have been used to implement a variety of methods of integrating management operations, these include: the system programmer's interface, existing system software and even contacting a human operator to perform manual operations. These are incorporated using "recovery blocks" [Randell75a] implemented using atomic actions. Existing system software, configuration files and other "legacy applications" [Sventek94a] are incorporated using "integration objects" which provide an clean and portable interface to physical (logical) resources.

The management information system was used to manage one of the University's machine clusters made up of a large number of resources (Suns, HP Workstations, disk devices etc) and used to illustrate the prototype system in industrial use. The configuration was adapted to simulate machines crashing and nodes becoming unreachable. A scheme for enacting major configuration changes and disaster recovery based upon "dependable change schedules" was implemented and demonstrated for a large number of resources.

1.6 THESIS OUTLINE

The thesis is structured as follows:— Chapter 2 considers the task of managing large systems and contains a literature survey. Chapters 3, 4, 5 and 6 discuss the design, implementation and application of our management information system. Conclusions, evaluation and future research are contained in

Chapter 7.

Four appendices are provided and consider the physical integration of resources (appendix A), constructing unrecoverable (Appendix B) and recoverable controllers (Appendix C). Several example user interface "views" are provided in Appendix D. A glossary, list of acronyms, trademarks and references are provided at the end of the thesis.

Chapter 2

MANAGING LARGE, COMPLEX SYSTEMS

So many workstations, so little time ... [Harrison92a]

The widespread use of distributed computing has been an invaluable asset to many organisations. Old, large mainframe computers located in centralised machine rooms with many terminal connections within the organisation have been replaced by modern high performance mini and micro computers, organisation wide communications topologies and resource sharing between individual departments. This is particularly true in the academic environment where

> *"The revolution in campus computing caused by the development of micro computers has fostered a second revolution in computer networking ... the convergence of computing and communication represents the most profound technical development since the development of movable type and the modern printing press" [Arms88c]*

Distributed systems have now become an indispensable part of universities research and education programmes through a strong commitment to shared resources and information. This lead to the potential for:—[Arms88d]

- Increased research output by improving access to information, super computers and other specialised computational resources, experimental devices and databases.
- Advancing the quality of academic research and instruction by expanding opportunities for collaboration and scholarly work.
- Reducing the transmission time for conveying basic research results from the academic to private sector and thus enhancing the national research and production capacity.

- Allowing researchers to communicate with colleagues and other professional staff from diverse and geographically separate departments using electronic mail, information servers and online conferencing.

These goals are not limited to the academic (and campus) environment but extend to other institutions and research organisations connected to national (international) networks. Many commercial organisations are connected to the internet allowing their employees to share information via the World Wide Web and communicate with colleagues using electronic mail etc.

Even within individual organisations the widespread use of decentralised computing can provide a highly individual service, local resource autonomy and increase the potential for raised productivity[King83a]. However, these very large distributed computer systems have caused a nightmare for computer systems administrators and operators faced with the unenviable task of managing thousands of workstations distributed across a potentially wide area and complex communication networks.

2.1 STANDARDS AND SYSTEMS MANAGEMENT

There have been important initiatives by the International Standards Organisation (ISO) and other bodies to develop standards in networking and application level components. These include ISO, the International Standards Organisation, CCITT, International Telegraph and Telephone Consultative Committee, ECMA, European Computer Manufacturers Association; IEEE, Institute of Electrical and Electronic Engineers, and MAP/TOP — Manufacturing Automation Protocol/Technical Office Protocol (General Motors). These group's work have converged within the ISO/OSI Management Model which has influenced some of the prototype systems discussed later in this chapter.

The ISO Open Systems Interconnection reference model[ISO87a] provides a structure to enable uniform (vendor independent) communications between systems and in particular, a management environment which specifies the requirements to control and supervise "managed objects". This "information model" provides guidelines for defining managed objects and their respective interrelationships, classes and names. These managed objects are defined in an abstract transfer syntax (Abstract Syntax Notation 1)[Steedman90a] and are grouped into "object classes" within the OSI "directory"[ISO88b]. Managed objects are controlled (monitored) by an information exchange protocol (CMIP)[ISO88c] whose services are provided by an information exchange system (CMIS)[ISO88d].

Organisations are modelled using a domain based representation and management activities are described using a functional model. This taxonomy describes the following management activities:—

- *Accounting Management* — "the set of facilities which enables charges to be established for the use of managed objects and costs to be identified for the use of those managed objects". This includes setting resource quotas and billing.
- *Configuration and Name Management* — exercises control over and identifies data about managed objects, to assist in achieving continuous operation of interconnected services. This includes setting parameters, initialising managed objects and associating names.
- *Fault Management* — "the set of facilities which enables the detection, isolation and correction of abnormal operation of the OSI environment" this includes maintaining error logs, accepting error reports and diagnosing faults.

- *Performance Management* — is "needed to evaluate the behaviour of managed objects and the effectiveness of communication activities". This includes gathering statistics, planning and analysis.
- *Security Management* — protecting managed objects and includes authentication, access control, key management and security audits.

OSI's goal is the long term provision for vendor independent homogeneous distributed computing, although short term migration problems are likely[Burnett87a]. MAP/TOP also described actions and functions which should be used to prevent components (and the whole system) from catastrophic failure. These include provision for both graceful and forced shutdown, status verification, physical recovery and restarting nodes. Unfortunately this provision was dropped when MAP 2.1. converged to ISO/OSI.

2.2 MANAGING COMPUTATIONAL RESOURCES

In chapter 1 we considered the role of management in general, the task of managing very large (organisation wide) computer systems and day to day operational management performed by system administrators. We noted that effective system management is fundamental to the smooth running of computer intensive organisations and that the traditional "one manager per computer system" approach is not appropriate for very large distributed computer systems.

It is possible to automate *some* management tasks using a management information system. Such a management information system would include the following components:—

- The information system's model of the organisation and management tasks should reflect the organisation's management structure, reporting relationships and activities performed by their system managers[Ross77a].

- The management information system should be resilient to partial system failure and include error recovery techniques. In the event of system failure, automatic re-boot utilities should be provided to minimise *manual* intervention.
- The management information system should provide *both* a graphical and command line user interface catering for both *expert* and *novice* users. Many software utilities available in the UNIX operating system are used within "shell scripts" and executed using system daemons and therefore providing a command line interface would be very useful.
- The system should incorporate (and encode) physical and logical system resources and integrate existing system software, network services, system entry routines etc. All management operations performed by the information system *must* be reversible "just in case".
- Where possible, the system should provide *on-line* system manual pages and other relevant documentation.

In this section we will provide an overview of selected research into computer systems management and prototype management information systems. In particular, we will evaluate each project considering its:—

- Model of management (if any)
- Encoding of management resources — either via a database, abstract transfer syntax etc.
- System architecture, implementation and application.
- Reliability mechanisms and fault reporting.

Since some of the prototype systems described use conflicting terminology a common frame of reference will be described in our comparison. (A glossary, at the end of the thesis will define terminology used in our management information system.)

2.2.1 PROJECT ATHENA (M.I.T.)

Project Athena commenced in 1983 with the aim of creating a new educational computing environment using high performance graphical workstations, high speed networking and centrally managed network services by 1988. Their network services comprise:—

- Authentication (Kerberos)[Miller87a] which uses trusted third party private keys and distribution services based on Needman & Schroeder's protocol[Needham78a].
- Service Management (Moira)[Rosenstein88a] which provides a replicated central repository of configuration information which is accessed by a set of library functions (and authenticated by Kerberos) across TCP/IP network connections — implemented using RTI Ingres[INGRES85a]
- Name Service (Hesiod) —[Dyer88a] providing a high speed "front end" to the Service Management System (Moira), and uses the BIND to achieve a hierarchical name space. Hesiod does *not* perform any processing or data interpretation and *only* retrieves data and transforms "logical" resource names into "physical" names used by the Berkley Internet Name Daemon (BIND), which provides a hierarchical name space, subsidiary name services and local caching.
- Real Time Message Delivery (Zephyr) —[DellaFera88a] Zephyr is implemented using UDP/IP to deliver reasonably short (preferably under eight hundred characters) to users *actually logged into* Athena. Typical applications include: conferencing, broadcasting emergency messages, phone services etc; and is therefore not intended to support electronic mail.

- Print System (Palladium) —[Handspiker89a] specifically designed for a distributed environment and conforms to ECMA printing standards[ECMA88a] and allows centralized management of printers, users and spool queues etc.
- File System — Athena workstations are "dataless nodes" supported by an extensive file system implemented using the Network File System[Sun87a] (NFS), the Andrew File System (AFS)[Mauro89a] and Remote Virtual Disk (RVD)[Greenweld86a]. RVD holds system software libraries and application software replicated on each subnet. Whereas NFS is used for shared access to private files.
- Electronic Mail — structured in terms of a central "mail hub", "post offices" and individual "mail boxes".
- On-Line Consulting — where users request help from "experts" on "topics".
- Conferencing (Discuss) —[Raeburn89a] modelled on "electronic meetings" on specific subjects and has evolved from an earlier system called *Forum* supported on MULTICS.

Athena primarily addresses configuration and user management and little attention is given to other management areas. In 1991, Moira managed 15,000 user accounts approximately 1,300 workstations and 100 network services held in a 15 M.byte database. Moira provides centralised data administration and update comprising a single master copy with specialised replication of subset data. Should Moira fail, Athena can still operate but managed resources cannot be reconfigured. The database is regularly archived and a recovery plan is used in the event of a failure.

2.2.2 *EMA*

Digital's Enterprise Management Architecture (EMA) defines a framework for managing a heterogeneous multi-vendor distributed operating environments and communications sub-systems [DIGITAL89a] and is designed to enable systems administrators to both design and implement organisation specific management environments. EMA is based on OSI Management standards and uses "well defined" architectural models:—

- Director/Entity Framework — defining the structure of management interfaces and interactions between directors (management systems) and entities (managed objects).
- Entity Model — defining management information and operations on managed objects.
- Director model — defining an open modular platform for managing resources.

Their management information system allows "users" to control (monitor) managed objects based upon the organisation's "policies" and the user's "view" of the system. An open modular environment ("director model") provides a workbench for managers to convey operations on resources ("entity model"). Managers can also access tools used by "remote directors" via distributed director modules.

Managed objects are encoded using an object oriented model, whose objects define "directives" (i.e. class interface operations), "attributes", "attribute groups"(i.e. types) and "events" (e.g. exception messages). (Single type inheritance is achieved through "subordinate entity classes".) Object states are held in a "management information repository" (i.e. an object store) which can be interrogated by the "management module" components.

Management modules are sub-divided into "presentation modules" which support the director's user interface, "function modules" providing management operations (divided as per the ISO/OSI taxonomy) and "access modules" provides transparent communication to remote objects.

2.2.3 DSM

The Distributed Systems Management (DSM) project [Wang89a, Wang88a] at the University of Lancaster developed a distributed management information system for controlling (monitoring) an organisation's resources. Their model is based on a set of system managers, maintenance engineers and end-users accessing managed resources through a management centre. Information is collected from resources using "agent" probes (one per resource) which control and monitor the physical resource in real time and enact management tasks. Once collected, information is analysed and stored in a management database implemented using Ingres.

Individual domain's management information systems are connected via LAN "Manager Modules" which exchange information across domains, access management data and forward fault reports. These fault reports are directed to maintenance engineers as appropriate. (The names and addresses of maintenance engineers stored in the database.) Information elements in the management database and encoded using SySL[Sommerville89a] and classified into the ISO functional areas.

2.2.4 NETMAN

The Netman project [Dean92a] at the University of Lancaster aims to produce an "Integrated Enterprise Management Framework" (IEMF) which investigates managing "enterprise wide" networked computer systems.

System management is viewed as a group activity (i.e. systems managers interacting) and modelled via "power" and "motivation" relationships[Moffett92a].

Their model of management embodies an explicit representation of system management information. The "shared system model" (encoded in the Enterprise Management and Query Language — EMANUEL, based on SySL[Sommerville89a]) includes resources such as printers, workstations and disk devices; relationships between resources, such as dependencies and organisation issues such as "responsibility", "expertise" and "interest".

"Responsibility", "expertise" and "interest" are encoded using EMANUEL mapping relations in a similar way to ANSA's "structural roles". Users (managers etc.) are responsible for resources, have expertise in certain areas and are interested in various topics (used in mailing lists etc.)

The IEMF tool set architecture comprises:—

- Graphical Browser / Editor — which presents views of the model and performs control (monitoring) operations on resources.
- Internal Objects — i.e. the actual resource controller.
- Semantic Analyser — checks system consistency.
- Parser — converts EMANUAL descriptions to internal representations.
- Installer — maintains software distributions and configurations.

No attempt is made to provide fault tolerance or recovery in the event of system failure.

2.2.5 OSF

The "Open Systems Foundation" (OSF) Distributed Management Environment[OSF90a, OSF92a] integrates the management of systems, networks and user applications within a conceptual model comprising the following components:—

- Human Interface — providing both command line and graphical interface to managed resources.
- Management Applications — such as systems software and other utility programs such as print spooler management and file systems administration etc., which access managed objects across "clear and concise interfaces" to "common management services".
- Common Management Services — such as the communications subsystem, resource naming and placement.
- Management Information Storage Services — providing the programmers' interface to managed objects.
- Managed Objects — representing logical and physical managed resources, such as devices, mail and print systems, users and application software.

Unlike the other management information systems, OSF integrates resources using a distributed environment. The "Open Software Environment" allows the (vendor transparent) integration of networks, operating systems etc. via a Application and System Programmer's Interfaces (API and SPI) and therefore achieves software portability.

2.2.6 TOBIAS

The Esprit funded TOBIAS (Tools for Object Based Integrated Administration of Systems) project had two main aims:—[TOBIAS89a]

- The development of an object based model of a typical computing installation taking account of both system and human resources.

- Providing a coherent interface to system administration tools.

A prototype system was developed by the project's partners (Rigel S.A, University of Newcastle upon Tyne, Intrasoft S.A., GIE Emeraude and Planet) and comprised several layers which communicated via a consistent interface:—

- User Interface — providing an object based interface to the management information system based on the idea of "views" of resources. This interface was implemented using the InterViews graphical tool kit (based on X windows)[Schiefer86a].
- System Management Object Controller (SMOC) — which sends messages between layers in the TOBIAS model and implements management operations upon physical resources. Error codes and information messages are returned to the caller via the integration and secure communications layers.
- Integration Module — providing a similar function to stub objects used in remote procedure call software, converting status (monitoring) requests between managed resources and the SMOC.
- Secure Communications — providing reliable, sequenced communications between system components.

An extra module, an expert advisor, was proposed to provide a sophisticated support tool for system managers for both system configuration and fault management. System managers and other users were modelled as "agents" taking on "roles" to perform a particular task as specified by way of a "contract". Agents and resources are located in "domains" which functionally subdivide the organisation.

2.3 EVALUATION

Whilst the prototype management information systems we have considered may use conflicting terminology and apparently different management architectures, they have much similarity in terms of their models of management and system structuring:—

- Prototype systems were developed with the aim of assisting the human system administrator to implement policies.
- Several management roles and functionality are common — based on OSI etc.
- Some method of recording resource states and probing the physical resource is always present.

Let us first consider the terminology used by research groups, how their management information base is encoded and management functions supported. These are shown in the table below:—

Terminology Table

	Athena	EMA	Tobias	DSM
Human Manager	Operator	Director	Agent	Users etc
User Interface	Moira/Hesiod	Entity Model Interface	View	n.a.
Coms Interface	TCP/IP	n.a.	TCP/IP	UDP/IP
Resource Controller	n.a.	Agent	Surrogate	Agent
Resource	n.a.	Managed Object	Managed Resource	Resource
Encoding	Database	ASN.1	Objects	SySL
Storage	Ingres	ISO DB	Database	Database
Management Functions	Config only	Director Modules	Via Agents	5 areas (ISO)

Managing a vast number of managed resources has become a major problem in many organisations and hence it is very important for management information systems to scale. Although Athena was specifically designed for these resource intensive environments, the fact that Athena does *not* use any method of functionally subdividing their organisation (such as "domains") and not having a distributed management information system is a major disadvantage, leading to very large centralised databases. The distributed management information system developed at the University of Lancaster (DSM), with one LAN Manager module per organisational domain provides an elegant approach to this problem.

The relationship between organisational policy and job allocation has been addressed by TOBIAS and Netman — using "contracts" and "power-motivation policies" respectively. Contracts recording role adoption, responsibility for managing resources etc. provide a cleaner and superior approach to job allocation. Netman's "power-motivation policies" are based upon Moffett's work as part of DOMINO. Moffett's later work has much in common with "structural roles" developed by ANSA [ANSA90a] and ORDIT [ORDIT89a] which record the relationship between agents, roles and resources. ORDIT does not have explicit "contracts" like TOBIAS but includes a "contractual schema" in its "information projection". [Blyth95a] TOBIAS's contracts can be easily extended to information (as per ORDIT etc.) and include resource allocation and explicit service trading.

OSF, unlike the other prototype systems provides a "management environment" rather than a "management information system" and therefore relies on other prototype information systems to provide management functionality. Management information systems such as Moira (Athena etc.) can be incorporated in OSF's environment using application and systems interfaces etc. OSF therefore has made a considerable contribution to portability in distributed "open systems".

While Athena provides replication of system critical information, other fault tolerance measures have not been addressed by any of the prototype management information systems; even though fault reporting is implemented, explicit recovery from operation failure on external resources is not documented in any of the prototypes considered. A summary is shown in the following table:—

Comparison Table

Criteria	Athena	Dec	Tobias	DSM
Domains	n	Multi	single	y
Management Areas	Config	5 (ISO)	Config Backup, User etc.	5 (ISO)
View Based Interface	n	y	y	y
Secure Comms	y	n	y	n
Object Oriented Model	n	y	y	n
Data Replication	y	n	n	n
Data Partition By Domain	n	n	y	y
Job Allocation	n	n	Contract	n
Explicit Manual Ops.	n	n	n	n
Failed Op Clean up	y	n	n	n
Fault Reporting	y	(Fault Manager)	y	y

2.4 CONCLUSIONS

In previous sections we have considered the role of very large distributed computer systems in (University) organisations and prototype management information systems which have been developed. Advantages gained by organisations in using these highly complex distributed systems have been met by increasingly complex computer (and network) management strategies. The traditional UNIX system management approach for "one computer system, one system manager" is not acceptable on the scale of 1,000 workstations, 100 servers and 10,000 users [Champine91a] and therefore some degree of automated systems administration is required in order to help systems administrators keep track of their managed resources. Automated system management can be performed using:—

- embedded resource controllers,
- systems software, or better still
- using a Work bench (as developed by TOBIAS etc).

Within the remainder of this thesis, we will consider the design, implementation and application of a prototype management information system based on a "management workbench" (c.f. TOBIAS and EMA) and a set of "probes" which integrate managed resources (such as printers, computers, disk units etc.). We will "expand" TOBIAS's contract based model to include trading, resource as well as job allocation and include ideas drawn from ANSA's (and ORDIT's) "enterprise projection".

We will consider fault tolerance and dependability in the design and implementation of the management information system — an area overlooked by the prototype systems discussed earlier. Hence, we will consider "graceful shutdown", "forced shutdown", "status verification", "physical recovery" and "node restarting" and consider a mechanism for performing major reconfiguration / recovering from major failure. We will not attempt to design "standard" interfaces etc. for integrating real resources *but* where possible will adhere to existing standards such as POSIX[Lewine91a, POSIX90a] etc. Various methods of incorporating resources will be considered such as using programmer's interfaces, systems software, network services and even contacting human operators.

Chapter 3

DESIGN OF INFORMATION SYSTEM

Here is the answer which I will give President Roosevelt ... Give us the tools and we will finish the job *

This chapter considers the design of a management information system and in particular the interactions between members of the organisation and resources. We will examine the explicit person oriented management model used by the prototype system and discuss a way of representing the organisation's policy and trading arrangements using contracts.

Our model of management has been developed from ideas used by several research groups. These include: ANSA[ANSA90a], ORDIT[ORDIT89a] Enterprise Modelling; Role Based Management developed by HP Laboratories [Bedford-Robers91a] and Contracts used by the Esprit funded Tobias project[Marshall90a]. Much of the underlying theory behind enterprise modelling is drawn from Checkland's work [Checkland86a] on "Soft Systems Methodology". Interested readers are referred to his book for further details on "Conceptual Modelling" and "root definitions."

3.1 MANAGEMENT MODEL

The model is based on the idea of members of the organisation ("agents") adopting "roles" in order to manage resources Each "agent" can adopt zero (i.e. unemployed) or more roles within the organisation. These agent—role bindings are described using "Contracts" [Dowson87a, Stenning86a] which record reporting responsibilities to other agents, job descriptions, working

* Sir Winston Churchill (1874 - 1965); Radio broadcast in Feb. 1941

practices, conditions of work and resource access — these will be discussed in subsequent sections.

3.1.1 AGENTS AND ROLES

This model of management is deliberately person centred and is based on the interactions between agents, other agents and managed resources. Members of the organisation are modelled as a set of "agents" who adopt "roles" to perform tasks. Hence the management role is abstracted and isolated from the person performing the work[Schutz70a]. Agents and role adoption can therefore be likened to agents "wearing hats" [Marshall90a, Bedford-Robers91a] and each agent adopting a role is assumed to be an "expert" in that "field of interest"[Dean92a]. Within a computer organisation, agents can adopt many roles; these could include:—

- *Asset Management* — concerning the location of resources, serial numbers, resource allocation etc. This is particularly complex due to the increasingly diverse variety of resources, vendors and users' requirements.
- *Backup Management* — managing the state of archived objects, file libraries, object restoration.
- *Change Management* — managing system re-configuration.
- *Configuration Management* — concerning the logical and physical configuration of the distributed system, cabling, and the physical interfacing of equipment.
- *Environment Management* — concerning the non-computing aspects of the installation, such as air conditioning, fire alarms and powering.
- *Fault Management* — managing fault diagnosis and rectification.

- *Name Management* — as the name suggests, managing the names of resources, network addresses and their resolution.
- *Performance Management* — detailed system monitoring is carried out in order to obtain management information in order to provide data for planning and detect intermittent faults. Besides "watching the system", trend statistics can be obtained and used for simulation purposes.
- *Safety Management* — concerning the health and safety executive's role in the installation.
- *Security Management* — concerns *policing* the distributed system, performing audits and preventing unauthorised access.
- *User Management* — concerning the management of end users, allocating resources, quotas and capabilities.

3.1.2 RESPONSIBILITIES

Responsibilities define the expected behaviour, rights of and obligations upon the agent playing the role ("role holder") and the relationships with other roles and resources. For example, an agent in the role "System Manager" may have a team of operators (superior—subordinates) who are responsible for a printer cluster. The operators' jobs would be allocated by the System Manager who would observe their behaviour and assess their performance.

Responsibilities can be sub-divided into:—

- Peer relationships — where two or more agents share a common supervisor (i.e. colleagues).
- Power responsibilities — reporting responsibilities and formal chains of communication between agents i.e. Supervisor — Subordinate etc.,

where one agent makes and enforces commands on other agents.

- Resource relationships — where an agent owns or maintains logical or physical resources.
- Service relationships — where goods and services are traded between agents and organisations (producer — customer, supplier etc.).

We will further consider responsibilities towards the end of this chapter when modelling organisational relationships using contracts.

3.1.3 JOB DESCRIPTIONS

In addition to agents adopting roles and relating to other agents, agents are given job descriptions for describing the actions of "playing" a particular role. For example, our team of printer operators ensure that the printers have sufficient paper, toner etc; stack/deliver print-outs for users and keep the print room tidy.

Whilst agents may have similar job descriptions, their role in the organisation may be inherently different, for example, legal and lexicographical proof reading. Both sets of proof readers may have the same rights and access privileges on documents and closely scrutinise their contents *but* the effects of their actions have widely differing consequences.

3.1.4 VIEWS

Checkland's "Soft Systems Methodology" pays particular attention to describing an organisation in terms of a "rich picture" (i.e. conceptual model), where user "viewpoints" and "conflicts of interest" are explicitly recorded. Views are also used within data base circles to present an external "schema" to users. There are certain advantages to such an approach:—[Date86a]

- they provide some logical independence in the form of data base restructuring.
- views allow the same data to be seen and used by different users.
- they simplify users' perception of resources and they provide automatic security for hidden data.

Views can therefore be used as a mechanism for filtering management operations and resource data from agents playing different roles. For example, while configuration and safety managers may manage a particular resource, the configuration managers will require data in order to perform hardware and software installation, connections and resource connectivity *but* will not *need* to know in-depth information regarding safety policy. Similarly, the Safety Manager will delegate all configuration management operations to the configuration manager and will *not* be expected directly to perform configuration management.

Each resource will have an associated *role dependent* view, presenting the external schema to management agents. Thus, a printer resource will have a set of views for the performance, fault, configuration manager and other managers. The performance manager's view of the printer resource may be composed of graphical charts and graphs. Whereas, the fault manager will have access to the diagnostic tests and fault logs.

3.1.5 MESSAGES

Management control and monitoring actions are performed using messages which are conveyed between agents and resources. Agents can select and edit fields on a resource's view and appropriate message is sent to the managed object representing physical resources. Whilst it is possible to describe this communications protocol using "speech acts" [Austin62a, Searle69a] and flow charts [Blyth95a] and formally specify interactions, a much simpler approach has been adopted. Messages simply invoke interface functions exported by managed resources in the same way as remote procedure call (or message passing) primitives — operations are performed on resources, parameters set (retrieved) and results returned to the user interface (resource view etc.).

3.1.6 CONDITIONS OF WORK

Besides responsibilities, job descriptions and role dependent views of the organisation, agents may have limitations imposed on their behaviour. These include:—

- Accounting fees — where charges are made for consuming resources and credit arrangements imposed.
- Legislation such as health and safety requirements.
- Physical factors such as environmental conditions, for example, ensuring the air conditioning in a machine room is between certain temperatures.
- Resource utilisation such as storage "quotas" and c.p.u. usage.
- Temporal constraints such as out of bounds times, particular days of operation etc.

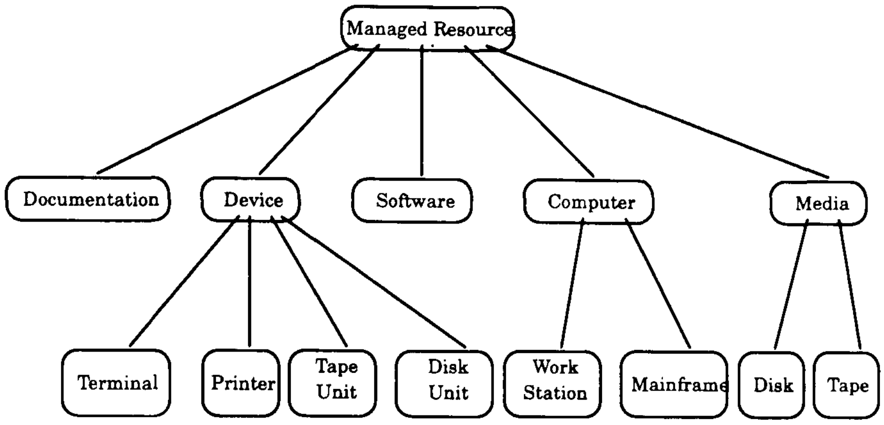
3.1.7 RESOURCES

An organisation's resources are represented by a set of managed objects within the management model whose state corresponds to other particular real word entities. For example, a printer entity corresponds to and is linked with one of the organisation's printer resources. Managed objects are encoded using an object oriented model and are mapped into a dependable object store. Managed resources include:—

- Cabling — used physically to connect devices with computational resources, and computers to the communications subsystem. These include fibre-optic, co-axial (co-ax) and twisted pair.
- (Communications) Devices — including fan-outs, bridges, routers and gateways.
- Communication Connectors — such as taps.
- Computing Resources — the organisation's processing capability. These comprise personal computers, lap-tops, workstations, and mainframes.
- (Peripheral) Devices — these include input devices such as terminals, card and paper tape readers; and output devices such as visual displays, printers and plotters. Some devices, such as tape and disk units, which from part of memory sub-system are both input *and* output devices.
- Documentation — comprising printed and on-line manuals.
- Media — comprising the set of consumables used by the organisation. These include magnetic tape, disks, CD-Rom, listing paper etc.
- Memory — comprising the actual persistent RAM, ROM etc. used by the computer as part of its memory subsystem and paging mechanism.
- Services — print and terminal servers.
- Software — including system software, operating systems and applications packages.

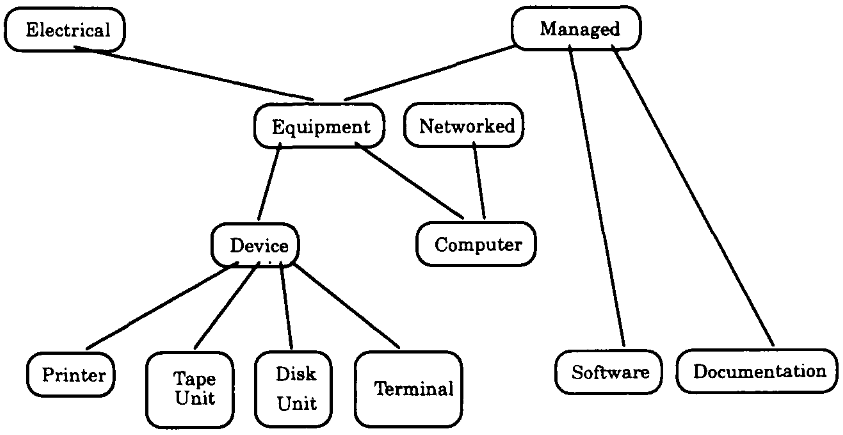
These components can be modelled using both single and multiple inheritance. In the single inheritance hierarchy, "devices," "computers," "media", "documentation" and "software" are derived from a base class "managed resource". The base class would include instance data such as asset identifiers, serial numbers and configuration information common to all managed resources. This is shown in the following diagram:—

Single Inheritance Hierarchy



It is possible to encode managed resources using multiple inheritance, using base classes such as "managed", "electrical", and "networked". For example, the managed base class would record asset names, location, maintenance details etc; electrical —power points and phases etc; networked — network address, internet ports etc. This is shown in the following diagram:—

Multiple Inheritance Hierarchy

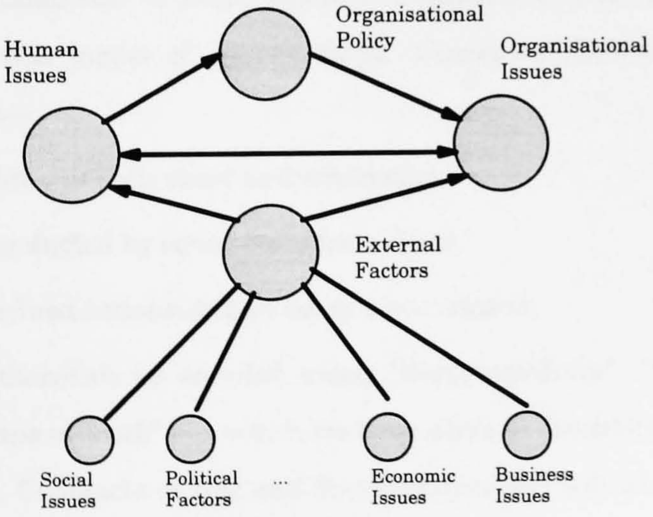


Whilst the multiple type hierarchy may have some advantages, the single inheritance model is far simpler and cleaner. We will see in Chapter 4 that due to restrictions in the Arjuna stub-generator software, the multiple hierarchy would be very difficult to implement.

3.2 REPRESENTING ORGANISATIONAL POLICY

An organisation gets its character from the policies which it adopts for directing its activities even though these policies may not be precisely defined and employees may not be aware of the existence of organisation policy at all levels of management. Management policy is influenced by a series of inter-related issues[ANSA90a] which include:—

- Human issues — including the safety of employees, trade union's relationship and productivity.
- Organisational aims and objectives — such as hopes, aspirations, goals and constraints.
- External factors — including social, political, economic and business issues, such as ethics, legal statutes and the availability of product supplies and transportation costs.



Some Factors which affect Organisational Policy

Objectives are typically described in the form of "general" policy statements which are implemented and interpreted by system managers. General policy statements can be redefined into detailed action plans using "policy hierarchies" in which sub-policies describes the implementation of higher level policies[Moffett92a]. Thus, the high level policy "no more than one day's work should be lost in the event of data loss" can be broken down into "the computer department's back up managers will archive all disks at midnight". Agent "Graham", in his role as "back up manager", can then be contracted to perform the disk archival etc. Graham can delegate backup on a particular machine to a (sub-ordinate) operator etc. which raises various issues regarding responsibility and accountability. Graham still remains responsible for the disk archival even though the task has been delegated. Roles are *responsible for* the tasks that they are contracted and *accountable to* their clients. These tasks can be *observed* by the clients; and the agents adopting role can be *liable* in the event of failure.

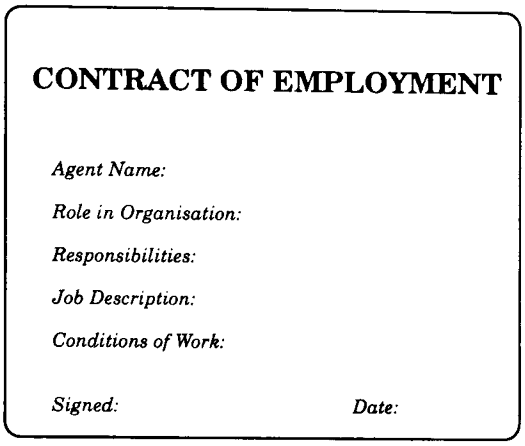
"Job allocation", "resource utilisation" and even "trading" are encoded using contracts (i.e. "Contracts of Employment", "Contracts of Use" and "Service Contracts") in this model of management. Contracts [Dowson87a, Stenning86a] define:—

- *Responsibilities* of both *client* and *contractor.*
- *Activities* conducted by *contractors* for a *client.*
- Precisely defined *actions* and an acceptance *criteria.*

Contracts can therefore be encoded using "Responsibilities", "Job Descriptions", "Conditions of Work" etc. which we have already considered. Contracts of Employment, Contracts of Use and Service Contracts will be discussed in the next three sub-sections.

3.2.1 JOB ALLOCATION

Job allocation is modelled using "contracts of employment" between the agent and the organisation. These contracts record the agent's reporting responsibilities, role in the organisation, conditions of employment etc. An example contract form is shown below:—



CONTRACT OF EMPLOYMENT

Agent Name:

Role in Organisation:

Responsibilities:

Job Description:

Conditions of Work:

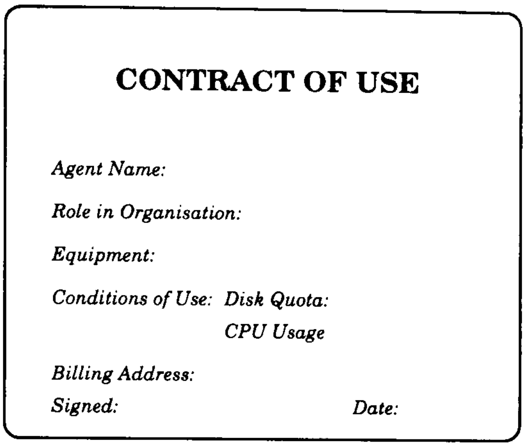
Signed: *Date:*

Agents can hold zero (i.e. unemployed) or more contracts of employment describing different roles and responsibilities in the organisation. For example, a fault manager can hold a contract for configuring resources (i.e. configuration manager) or even be a “safety officer”. With agents holding multiple contracts there is a danger of “conflicts of interest” between different roles. This is particularly apparent when agents reside in multiple domains, where the agent may have conflicts of loyalty or even be expected to perform tasks against the other domain’s policy.

Recording the agent’s responsibilities in the contract is particularly useful as it explicitly shows which agent is responsible for a particular resource, and therefore fault reports can be directed as appropriate. Similarly, in the case of sub-contracting, the explicit responsibility relationship provides a structure showing dissatisfied clients where to complain if things go wrong!

3.2.2 RESOURCE ALLOCATION

Resource allocation is modelled using “Contracts of Use” between the agent and the organisation. Like “Contracts of Employment”, “Contracts of Use” record the agent's name and role in the organisation as well as an inventory of resources, billing conditions etc. An example contract form is shown below:—



CONTRACT OF USE

Agent Name:

Role in Organisation:

Equipment:

Conditions of Use: *Disk Quota:*
CPU Usage

Billing Address:

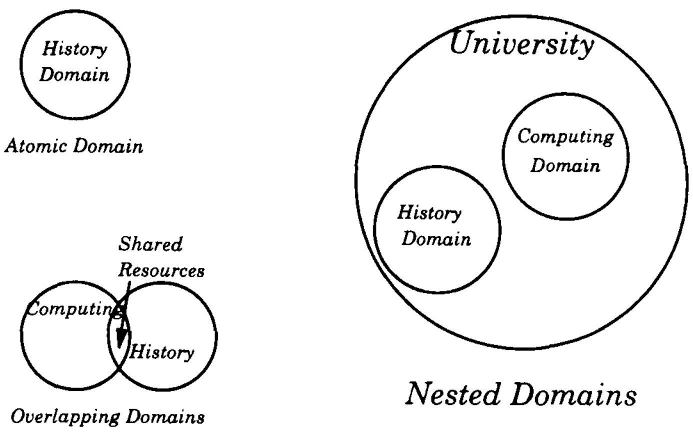
Signed: *Date:*

It is common for organisations to issue some form of contract when leasing equipment to users. These describe rules and regulations, legal requirements, such as the "Computer Misuse Act (1990)" [HMSO90a] and "Data Protection Act (1984)"[HMSO84a] etc. Information concerning lease of equipment and billing etc. could then be forwarded to the asset and accounting managers responsible for those resources.

3.2.3 DOMAINS

A domain is a multiple set of components which share a common attribute or are managed by the same management agents[Sloman87a]. Domains can be used to represent physical hardware components such as a workstation's CPU, disk units display etc.; communication topologies such as networks and resource names; organisational departments, research groups and projects; and network services, such as electronic mail. Domains therefore represent an excellent method of coping with the physical reality of resources, their interconnection and administration[Marshall93a]. Examples of domains are shown in the following figure:—

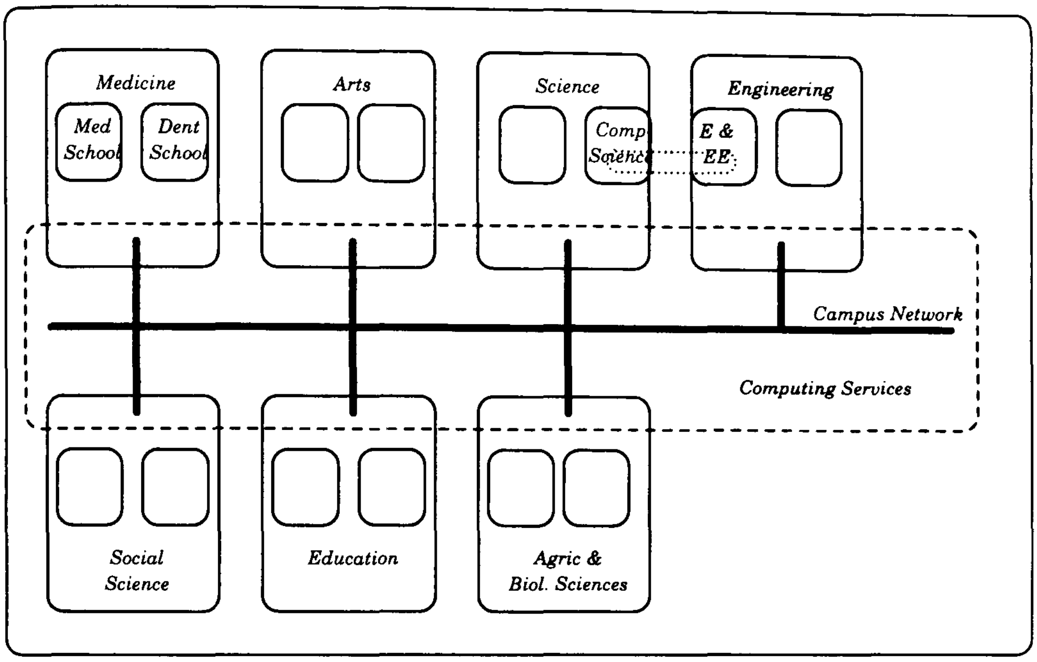
Domains (Based on University Departments)



Domains can be "atomic" (in the case of the single "history" domain); "overlapping", where domains share resources etc.; and "nested" where domains are grouped together to form a "superset".

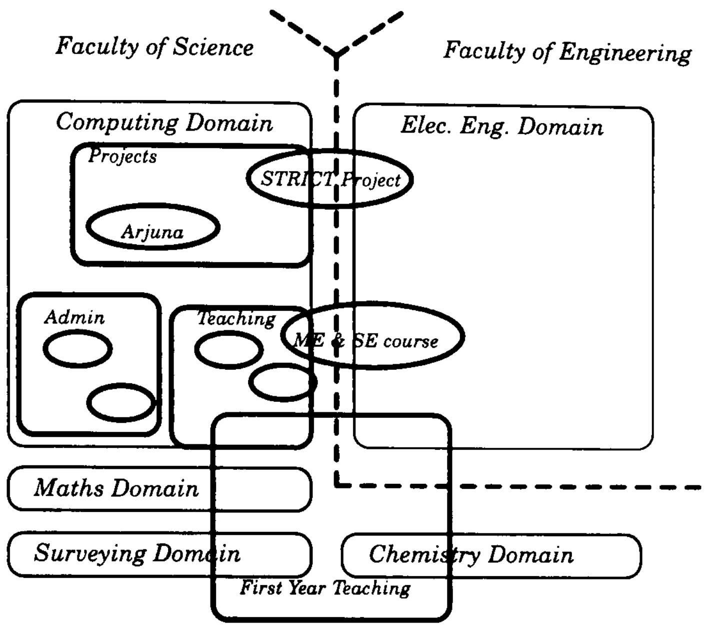
For example, the University of Newcastle on Tyne's academic structure is shown below. The University is structured in terms of Faculties — Medicine, Art, Science etc; which are composed of schools, departments and research projects. The campus (fibre optic) backbone is shared between all faculties and is managed by the Computing Services.

Newcastle University Domain Structure (Simplified)



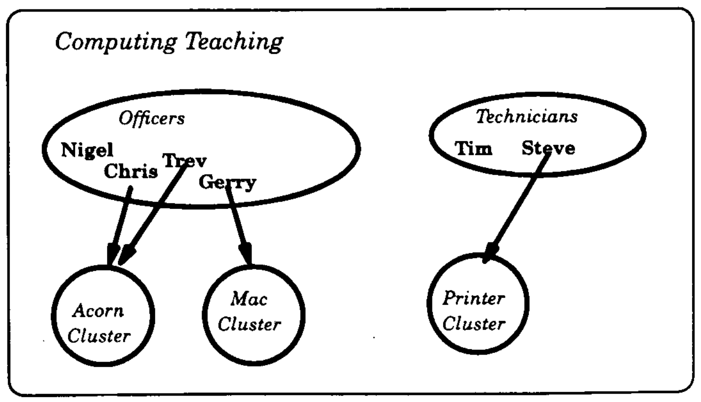
Examination of the Computing Science and Electrical Engineering domains shows that whilst the two departments are located in different faculties they share a joint research project (concerned with V.L.S.I. design) and teach a common undergraduate course (Micro Electronics and Software Engineering).

Computing and Electrical Engineering Domains



Computing Science's teaching resources are managed by a team of Computing Officers and Technicians. Chris and Trev manage an Acorn Workstation Cluster, Gerry — an Apple Macintosh Cluster etc. Furthermore, the Workstation Clusters share common file systems, name-spaces, and printers which are not shown for the sake of simplicity.

Computing Domain's Resources



3.2.4 INTERACTIONS BETWEEN DOMAINS

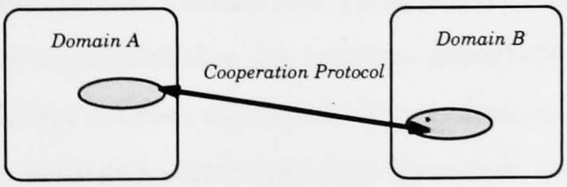
Interactions between independent domains are an essential requirement for managing distributed, complex systems and allow organisational departments to sub-contract (i.e. trade) services between organisational domains and allow individual managers to "co-operate" when performing complex tasks. For example, an organisation may choose to sub-contract hardware maintenance to an external firm. Network managers may need to co-ordinate management tasks to maintain connectivity across the organisation.

While domains aid visualization of, and reasoning about the relationships between members of the organisation, they do not directly provide a method for domains to trade services. There are implicit relationships between agent and managed resources in "nested domains". For example, if "agent a" manages "agent b" and "agent b" manages "agent c" — "agent a" by implication, manages "agent c" ("transitive relationship"). It is also possible for managers to have "reflexive relationships" with resources and managers to control (monitor) resources in non-local domains ("interacting domains").

There are two different ways to allow managers to co-operate:—[Sloman89a]

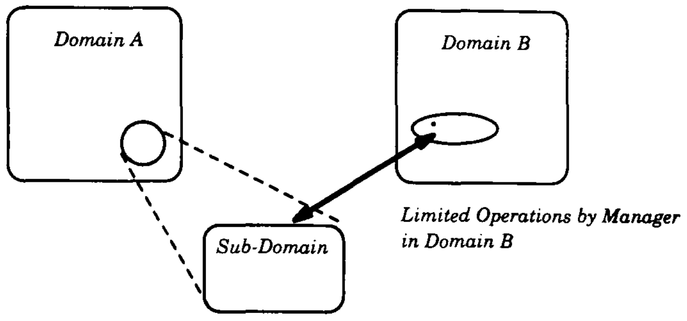
- "Indirect Management" — where management functions are "delegated" to another manager and some interaction protocol exists between both managers.

Indirect Management



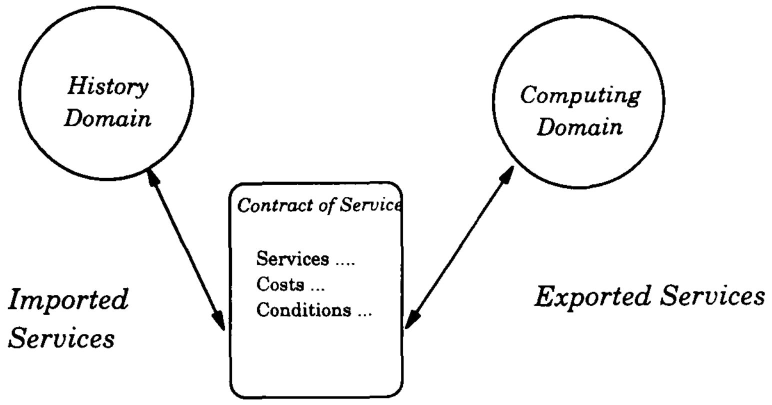
- "Direct Management" — where limited management functions are performed directly by a non-local manager on a subset of resources.

Direct Management



Co-operative work, delegation and "trading" are modelled using "Contracts of Service" which allow domains (or individual managers) to "import" ("export") services. Thus, "Service Providers" can "negotiate" with "Service Consumers" and agree a "contract". For example, the "history domain" could sub-contract disk maintenance to the "computing domain", which is shown in the following diagram—

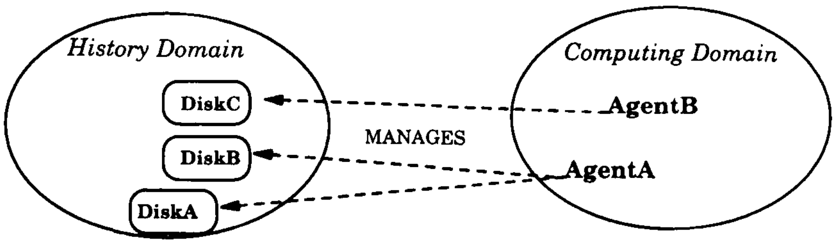
Direct Trading Between Domains

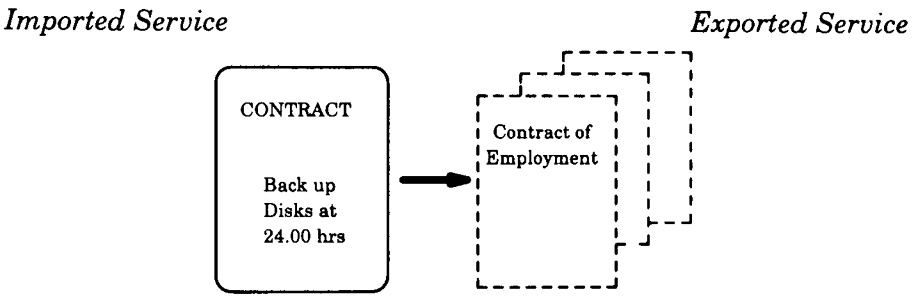


"Contracts of Service" like "Contracts of Employment" and "Contracts of Use" are composed of responsibilities, job (service) descriptions and "conditions". Service relationships between managers (or organisational domains) delegate tasks to other managers (domains) and therefore differentiate between "observer", "customer" and "executor" roles.

The importing domain allows managers in the exporting domain to directly perform tasks on their resources. The initial "Contract of Service" is broken down into jobs ("Contracts of Employment") which are allocated to managers in the exporting domain. Resources are still owned by the "importing domain", managers in the "exporting domain" are still accountable to their superiors etc. *but* there is also a direct trading relationship (contract) between both domains. This is illustrated in the following diagram:—

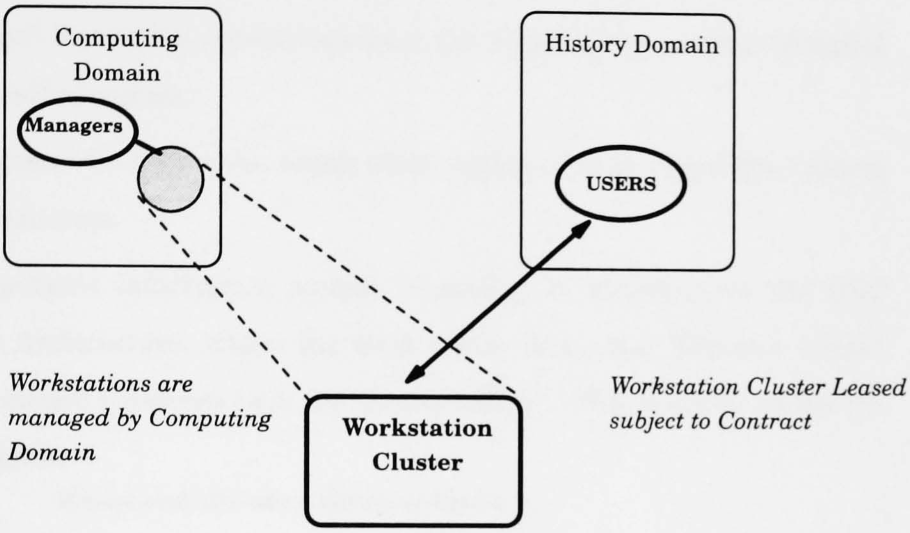
Direct Trading





The contract to archive the history domain disks is broken down into three contracts. "AgentA" and "AgentB" (from the computing domain) archive disk drives "DiskA", "DiskB" and "DiskC" every night (at 24:00) The computing agents are responsible to their manager (in the computing domain) but are observed by agents in the history domain. Similarly, computing agents are allocated capabilities in the history domain in order to archive the disk units. Should the history domain be dissatisfied with their "imported service" they have redress to the computing domain.

It is also possible to use "Contracts of Service" when recording equipment hire or "time sharing", where instead of allocating people (ie. agents) to perform a task, equipment (or processing time) is hired to (or used by) the importing domain. This is illustrated in the following diagram:—


Computing Domain
Managers
History Domain
USERS
Workstations are managed by Computing Domain
Workstation Cluster
Workstation Cluster Leased subject to Contract

In this example, the computing domain leases a workstation cluster to the history domain. The computing domain still manage the workstation cluster, arranging maintenance cover, disk back-up etc. and a group of users in the history domain are allocated workstations in their office subject to conditions of use. However, should the computing domain fail to deliver the agreed quality of service, the history domain can seek redress for breach of contract.

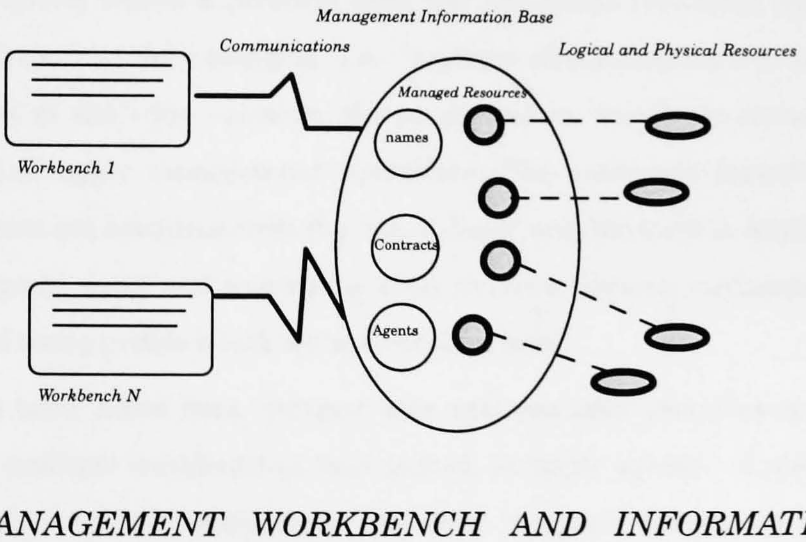
3.3 CONSTRUCTING A MANAGEMENT INFORMATION SYSTEM

Having discussed our model of management and encoding interactions within the organisation using contracts, let us turn our attention to constructing the management information system. Our model of management is based on the idea of agents adopting roles to view resources, and regulating resource access using contracts which is reflected in the design of the management information system. The management information system is composed of:—

- Management Information Base, which records organisational domains and interactions, contractual information, agents and resource details.
- Managed Resources, which represent the state and properties of logical (physical) resources.
- Management workbenchs, which allow agents in their respective roles to view resources.

The management information system is similar in structure to the DEC Enterprise Architecture, where the work bench forms the "Director Model" and the managed resources form the "Entity Model". This is shown in the following diagram:—

Management Information System Architecture



3.3.1 *MANAGEMENT WORKBENCH AND INFORMATION BASE*

Agents access the management information system via a management workbench. The workbench which forms the management information system's user interface is used to view "domains" (such as organisational departments, communication networks and composite resources) and managed resources. In order to manage a very large distributed computer system, the organisation's logical structure and resources are first modelled using domains and mapped into the management information system's "organisation base" The organisation's employees, allocated resources and interactions are then mapped into the "contract base" using "contracts of employment", "contracts of use" and "contracts of service". Agent and resource names are then mapped into the "agent base" and "asset base" (respectively), which are used to forward mail messages to agents, locate resources etc.

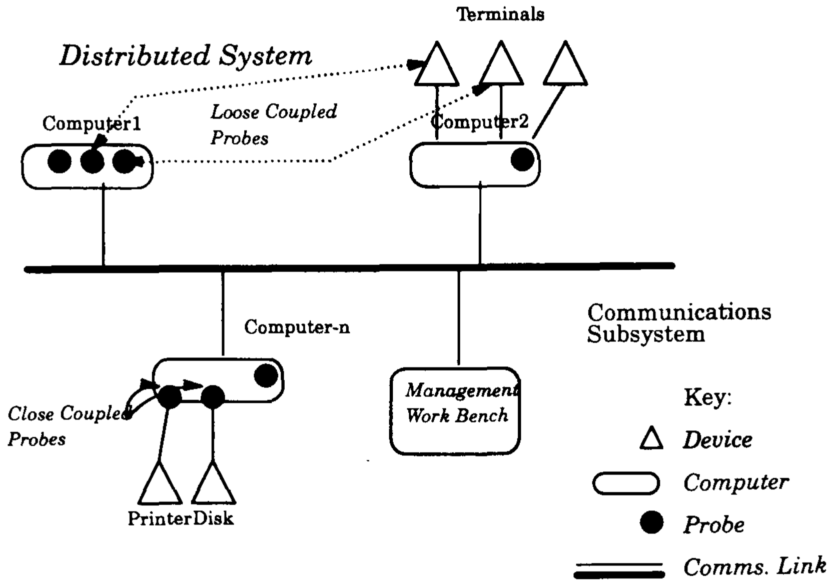
Once the agent logs into the management information system, her name and password are checked against an "agent base" which contains a list of agent names, contact addresses and login capabilities. Providing her capabilities are correct, she can view the organisation's logical (physical or communications) structure, create a personal desk top and select resources. Should she hold the contract for managing (i.e. "contract of employment") or using (i.e. "contract of use") the resource, she is allowed to "view" the resource's properties and apply management operations. The resource's logical and physical names are retrieved from the "name base" and the view is displayed using a property sheet and operations upon physical (logical) resources are implemented using probes which we will consider later.

The agent base, name base, contract base and managed resources can be accessed by multiple workbenches (and indeed, multiple agents). Appropriate concurrency control mechanisms and fault tolerance techniques must therefore be incorporated in the management information system which we will consider in later.

3.3.2 MANAGED RESOURCES AND PROBES

Managed resources such as computers, terminals and printers are controlled (monitored) using "probes" which implement management operations upon physical (logical) resources. Managed resources are encoded using an object oriented model (which we described earlier) and mapped into probes which act as servers to the workbench clients. Hence, when an agent sets (gets) properties or alters a resource's state, request messages are conveyed over the communications sub-system and enacted upon the resource.

Probes are "coupled" to their *external* resource. Resources can be "close coupled", where the probe is physically located on the same node as the external resource; "loosely coupled", where the external resource is remotely managed; or "uncoupled", such as when the external resource is disconnected. For example, computer workstations can be managed by close coupled probes, whereas terminals and printers could be loose coupled and remotely controlled (monitored) using network administration software. [Encore87a, Encore87b] This is shown in the following diagram:—

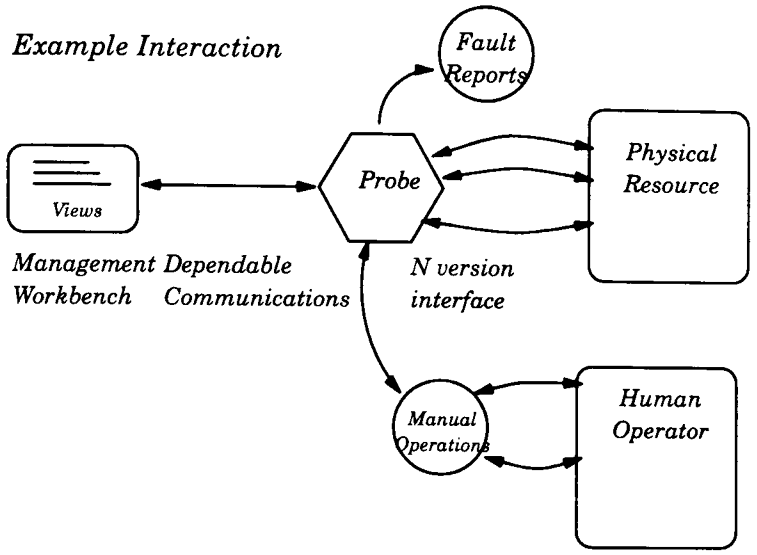


Resource coupling depends upon the method of integrating operations upon the external resource. For example, operations implemented using system entry routines and some system software *may* require close coupled probes, where as those operations implemented using network services are inherently loose coupled.

3.3.3 DEPENDABLE RESOURCE INTEGRATION

Management operations upon physical (logical) resources will be performed using a variety of techniques such as system entry routines, system software, network services, configuration data files and contacting (human) system managers / operators etc. Where possible we will adhere to existing standards such as POSIX in order to maximise portability and confine machine specific routines to lower layers in the management information system. This will be discussed further in Chapter 4.

Properties can be set (retrieved) via the view based user interface on the management workbench and conveyed to (from) the control probe via a dependable communication channel. The controller then attempts a management operation via system entry calls, system utilities etc. Should the operation fail, any debris will be cleaned up and a fault report sent to the fault manager responsible for the resource. Otherwise an acknowledgement is sent back to the workbench confirming that the operation has been performed correctly.



Probes (servers) will be implemented as server objects communicating with a workbench using remote procedure call primitives[Nelson81a] and holding their instance data on stable storage[Lampson79a]. Once a request is received by the object manager to access (create) a probe, a server is activated, obtains its state from stable storage and performs operations upon the physical (logical) resource. Operations on managed resources will be performed using write (read) locked atomic transactions and these will be serialised, enjoying failure atomicity and permanence of effect. Failed operations on external resources will be compensated and undone via operation based backward error recovery. In the event of managed resources failing, a fault report will be sent to the appropriate fault manager and the probe marked "out of order," preventing other agents from accessing a failed resource. The resource will remain out of order until replaced or repaired etc. Should the server crash enroute, recovery operations will be performed to clean up any partial computations and reconcile the probe's state to the external resource i.e. the probe's state will be re-initialised to that of the resource. Hence any performance data held by the probe will be kept "upto date".

3.4 CONCLUSIONS

Having considered the requirements for a Systems Management Information base (chapter 2) and the design of our system, let us compare our design with leading "competitors". Our Management Information base is designed in terms of a graphical work bench in the same way as TOBIAS and DEC, although the graphical user interface is not essential to the workbench's operation. A command line interface (for expert users) could easily be incorporated into the workbench via the view mechanism. Our view-based interface is very similar to DEC's "Director Modules" in that views provide tools and obtain properties etc. to assist agents in their particular roles to manage resources. Managed resources are encoded in an object oriented model (Appendix C) and have a more complex interface than TOBIAS. (Management functions as well as setting/getting properties). Unlike TOBIAS/DEC etc, explicit manual operations and fault reporting have been included in the resource interfaces.

The management workbench is not intended to replace operating system configuration files (like Moira) or provide rigorous interface definitions or even incorporate existing Management Information Systems (like OSF) although existing system software could easily be used to implement management operations (and even be modified to use the management workbench).

Unlike Netman, the model of management is based upon ANSA's "Enterprise Projection" and TOBIAS' contracts. Hence, rather than an agent having the power (authorisation) and the motivation to perform a task, an agent in a role manages a resource. Although both models are similar in their objectives, the role/contract model is more "natural" to the workings of an organisation. Contracts explicitly record responsibilities, job descriptions and conditions and hence are more complex than TOBIAS' implementation. Similarly there is a distinction between contracts of employment, contracts of use and trading arrangements. Contracts of use are not intended to replace existing password/log-on files but can be used to generate these configuration files.

There is no attempt to model policy hierarchies explicitly in our model management and therefore we assume that contracts etc. are consistent with the organisation's policies. Ensuring contract consistency and "resolving" conflicts of interest etc. are therefore outside the scope of our model. Interested readers are therefore referred to Moffett's work. A summary of the design features is shown below:—

Criteria	Feature
Domains	Multiple domains, used to encode organisational departments, resources etc.
View Based Interface	Multiple, role specific "property sheets" of each resource. Property sheets can be supplement using command line interfaces where appropriate.
Secure Comms	Prototype system will incorporate underlying communications channels.
Object Oriented Model	Single inheritance used to encode managed resources
Data Replication	Resilent management information servers used to maintain resource names, contracts etc.
Data Partition By Domain	Distributed databases could be incorporated in prototype.
Job Allocation	Encoded and regulated by contracts.
Explicit Manual Ops	yes
Failed Op Clean up	yes
Fault Reporting	fault reports are automatically sent to fault manager.

Chapter 4

PROTOTYPE IMPLEMENTATION

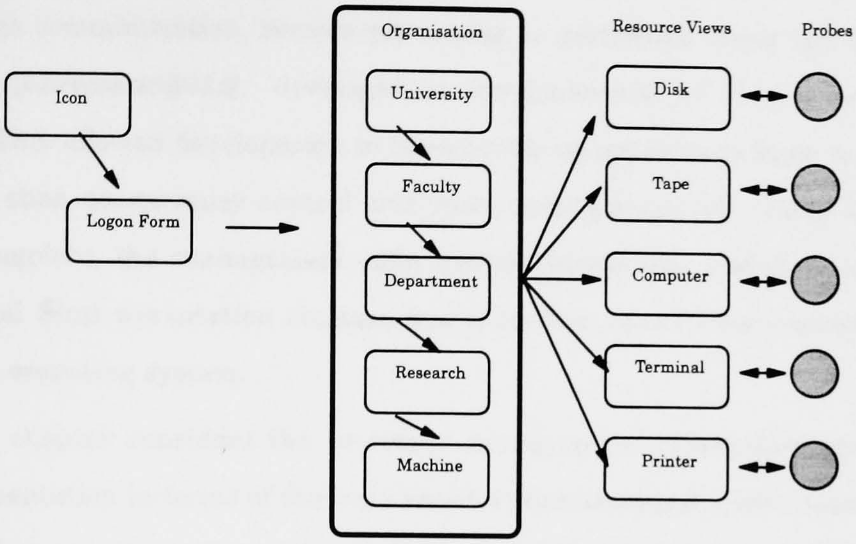
Anyone who isn't confused here doesn't really understand what's going on *

The prototype's application level architecture closely follows the model of management and system design which we have already considered in Chapter 3: agents adopt roles to manage resources *provided* they hold a contract. Agents navigate through a series of "organisational views" in order to select resources whose properties can be inspected (or altered) using property sheets etc.

The management information system is structured in terms of a "management workbench", which forms the information system's user interface; organisational information, such as agent and resource details; and a set of probes which access logical (physical) resources. Management information users (i.e. agents) click on an icon and "log in" via a property sheet. They then "view" their organisation via a set of "domain views" and (eventually) selects a resource. Resource views are then displayed showing the resource's properties and state etc. This is shown in the following diagram:—

* Belfast Citizen, 1970

Management Information System Structure



Interactions between the workbench and managed resources are outlined in the following program pseudo-code:—

```
FOR EVER DO
BEGIN
        Get Agent's Name Password;
        IF Agent CAN Login
        BEGIN
                Agent in Role Selects Resource;
                IF Agent IN Role MANAGES Resource
                BEGIN
                        Show Property Sheet;
                        Apply Operations to Resource;
                END
                ELSE
                        Report "No Contract to Access Resource";
        END
        ELSE
                Report "Agent Cannot Login";
END
```

The management information system was initially developed and tested on a Sun workstation running the Sun's 4.1.3 release of the UNIX operating system. The workstation was connected to the campus network via the laboratory Ethernet [Metcalf76a] and had a local hard disk physically attached to the console. The workstation's disk unit, console, keyboard etc. were allocated a "probe" and connected to a the workbench's graphical user interface.

Although it would have been possible to implement remote communication between the workbench (client) and probes (server) using basic process-message communication, remote processing is performed using the Arjuna tool kit[Shrivastava91a}. developed at the University of Newcastle upon Tyne. This allowed development to concentrate on *application layer* modules rather than concurrency control and state management etc. After testing was complete, the management information system was ported to several (HP and Sun) workstation clusters and a further workstation running the Solaris operating system.

This chapter considers the prototype management information system's implementation in terms of the work bench's organisational views, initial configuration, resource views and probes. Further information regarding specific modules and some sample screen dumps are provided in the appendix.

4.1 MANAGING CAMPUS RESOURCES

The University of Newcastle upon Tyne's academic structure was modelled using "domains" and mapped into the management information workbench. Obviously, any organisation structure could be mapped into the workbench and the University was chosen purely for convenience. A geographically dispersed organisation with branches across the country (or even the world) could be modelled using domains etc. in the same way as the University organisation although communicating with resources across a wide area would pose particular problems such as propagation delays, packet loss (and corruption) etc.

Newcastle University is structured in terms of several Faculties (Medicine, Arts, Science, Engineering, Social and Environmental Sciences, Law, Education, Agricultural and Biological Science), Schools (Medical and Dental School etc.), Departments (Computing Science, Electrical Engineering etc.) and Departmental groups such as research projects and administration etc. A complete list of Faculties etc. is given in the University Academic Staff Handbook.

Rather than implementing a (distributed) database containing the University's organisational structure, where each domain could manage their own structure; each faculty (school, department etc.) is encoded using a Tcl/Tk procedure. In future releases of the workbench, a combination of hard coding and a domain database could be used to combine the flexibility of a data base with the quick response times of hard coding. (Assuming that the Univerity's academic structure is fairly static!) Each faculty etc. is encoded as a labelled icon which when clicked reveals the contents of the individual domain.

No attempt has been made to restrict access to non-local agents accessing the contents of domains even though the individual resources can only be viewed if the agent holds a contract. Domain based "view restriction" could easily be added to "filter" view traversal. This view restriction would prevent agents from one domain viewing the contents of another domain and could be implemented using contracts etc. This could be easily implemented in a similar way to Robinson's domain based access control mechanism[Robinson88a]. Non-local agents who have been contracted to manage local resources would only be allowed to view sub-domains as illustrated in Chapter 3.

4.2 MANAGEMENT INFORMATION BASE

In order to support the management information system, certain organisational and resource data is maintained within the management information base. This information is currently held in text files, even though in future releases of the prototype system a distributed (resilent) data base would be more appropriate. The following information is held in the management information system:—

- Agent Base — recording a list of all agents in the organisation. Each agent's name, password, office location, telephone number and electronic mail address etc. is stored. This information is used to verify login capabilities and direct automatic reports via electronic mail etc.
- Asset Base — recording all assets held within the organisation, manufacture's and vendor references, maintenance details etc.
- Availability Table — which records any resources (probes) which are "out of order".
- Contract Base — recording contracts held by agents in the organisation. At present, only "contracts of employment" are recorded but this could be easily extended to record "contracts of service" and "contracts of use".
- Name Base — this stores resource locations and application level names for each managed resource. Ideally probes *should* be location transparent but these *physical* locations are maintained due a bug in the Arjuna name server which prevented (high level) Arjuna (location independent) naming and node binding between HP and Sun workstation clusters.

4.3 INITIAL CONFIGURATION

Configuration management concerns the distributed system's topology and interconnections. Not only does this concern the placement of management probes used by our prototype system, but also the configuration of management resources. Our definition of "Configuration Management" deliberately restricts the ISO/OSI "Configuration and name management" [ISO88a] to purely "Configuration Management" as name, asset and environment management etc. are considered as separate entities.

Configuration management is therefore distinct from "Configuration programming" (or "programming in the large")[DeRemer75a] which concerns the actual binding of processes to nodes and identifying communication paths between processes. Configuration management is subtly different from "change management" which concerns *reconfiguration*. Software upgrades for example, are included in "Change Management". It is possible for the same agent to perform both configuration and change management — providing the agent holds the appropriate contract for managing the particular resource.

4.3.1 WORKSTATIONS INTEGRATED

When configuring the management information system, the following resources were required:—

- Computers — one probe per host,
- Disk — one per file system (the disk controller was simplified and assumed that each disk unit had only one disk partition),
- Terminal — one per device,
- Printer — one per device (not one per spool queue).

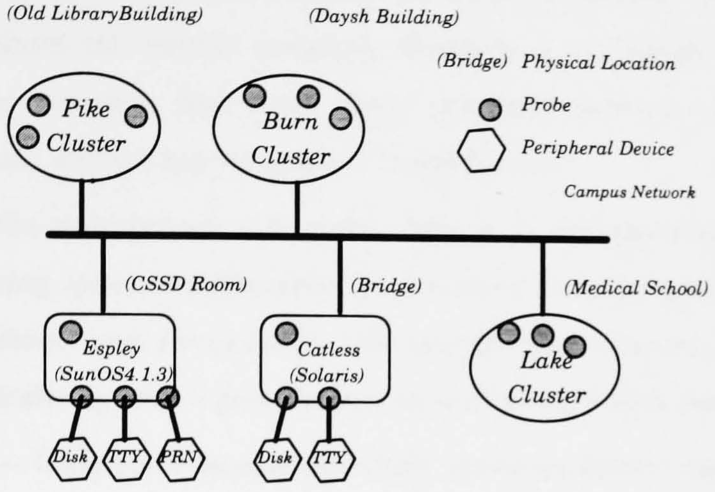
The following resources were managed by our prototype:—

HP Clusters	9 (5 workstations per cluster)
Sun Workstations	6
Disk Units	6 (3 or more partitions per disk device)
Printers	1
Tape Units	1

The workstation clusters shared a common network file system and printing services across the computer network. Rather than housing replicated disk and printer probes on every machine, disk probes were placed on nodes exporting file systems and nodes where printers were physically attached.

(Printers were deliberately simplified to probes on "exporting" nodes even though each "importing" node has its own spool queue etc.)

Application Level Architecture



The management workbench is located on a Sun5 workstation ("Catless"), located in the Department of Computer Science. When agents in their respective roles select the "Computing Service domain" from the University's faculty, structure managed resource clusters are displayed on the screen. This includes the "pike", "lake" and "burn" H.P workstation clusters, located in the Old Library Building, Medical School and Daysh Building. Selecting individual HP workstations causes "Computer View" property sheets to be displayed on the screen showing workstation configuration properties, load metrics, process queue statistics etc. Selecting the "Computing Teaching" domain (located in the "Faculty of Science" — "Department of Computing Science") and clicking on the "CSSD" (M.Sc. course) domain reveals teaching resources and student user groups. This includes the "Espley" Sun4 workstation located in the CSSD project room.

4.3.2 AUTOMATIC CONFIGURATION

It is possible to use the management information workbench to configure resources, in practice, many workstations are supplied already configured with disk interfaces etc. already installed. Computers are "simply" assembled by system managers, disk units "daisy chained", network cable and tapes attached etc. and the host is (almost) "ready for use".

Configuring the management information system is performed automatically by inspecting system configuration files such as `/etc/printcap` (line printer descriptions) and `/etc/fstab` (file system table). Hence, for each physical (or logical) resource, a probe is created and instance data obtained:—

- Computer — Computer Name, architecture, operating system memory.
- DiskUnits — Disk Controller, interfaces, physical (and logical) geometry etc.
- FloppyUnits — ditto.
- TapeUnits — Block size, tape controller, media etc.
- Printers — "Printcap" entry, spool queue state, synopses of jobs already printed.
- Terminals — baud rates, protocols, parity etc. are obtained from the "Terminal Controller".

4.4 VIEWS

Views present a "role based" user interface to managed (recoverable) resources and are therefore part of the management workbench. By selecting a resource icon (printer, terminal, computer etc.), the resource's name and location are passed to the "view selection" procedure which then displays the appropriate property sheet on the screen. (Although the Arjuna name-service provides location transparent access to remote probes, there are unfortunately incompatibilities between HP and SunOS releases. Hence the location tables are maintained in the management workbench.) The view selection procedure is outlined below:—

```
Display Computer

IF resource IS available THEN
      Get Name and location from table
      IF resource is NOT being used by another agent
            Show Property Sheet
      ELSE
            Report "Resource in Use"

ELSE
      Report "Resource Not Available"
```

Views are used to present a "role based" user interface to managed (recoverable) resources. Views therefore are part of the (application layer) management workbench and access resources across a dependable communications channel. Four views of each resource are provided:—

- Asset View, which accesses asset identifiers, physical locations etc.
- Configuration / Change View, which set (get) properties such as printer baud rates, configuration files etc.
- Environment View, setting (getting) power supply information etc.
- Performance View, which allow resources to be periodically probed and properties "watched".

These views are implemented in Tcl/Tk [Ousterhout94a] and access remote resources using small Arjuna "client" programs. Client programs (written in C++) simply access remote servers (via an "ArjunaName" or "UniqueIdentifier") and perform remote procedure calls to set (get) properties or apply state operations. Output to (input from) these clients is "piped" to the Tcl/Tk procedure which displays output to the screen.

4.4.1 COMPUTER CONFIGURATION

The "Computer Configuration View" (shown in Appendix D) comprises a title "Computer View", menu and property sheet; each are implemented as frames within the window. Property names are encoded as labels and value entries are "text variables" in a raised box. In order to provide a consistent interface, a configuration file provides default fonts, point sizes, window colours etc. Four menu options are provided on all resource views to present a consistent, ergonomic user interface:—

- State — Operations on the resource state,
- Properties — set, get and clear property values,
- Help — view help (manual) page,
- Quit — exit view.

In the case of the configuration view, an extra menu operation "RestoreProperties" is provided which allows configuration managers to attempt different resource configurations and "back track". Each time properties are "set" the resources state is saved to a stack and therefore incorrect configurations can be "rolled back".

When the user clicks on the "GetProperties" menu, the "doGetProperies" procedure is evaluated, which in turn executes an Arjuna client. During each session, remote procedure call timeout and re-try values are defined via "ClientActionRPC" structures. Should the Arjuna remote procedure call mechanism timeout and exceed the "retry limit", exceptions are raised. These user defined handers include:—

- Constructor Failure — where the Arjuna server cannot be created, for example, the probe has not been installed properly on the host machine.
- Destructor Failure — where the server has not terminated properly.

- (Other) Remote Procedure Call timeout — where either, the server has crashed or communications failure etc.

In the event of these exceptions being raised, the client will terminate with and send an error message to the view procedure. Otherwise, several remote procedure calls are made to the server obtaining properties (machine name, operating system, loading etc.) which are then read by the view. "Setting properties" is performed in the same way to "getting properties" except that data is sent from the property sheet to the client (and in turn to the probe etc.).

4.4.2 COMPUTER PERFORMANCE VIEW

Besides setting (getting) properties and altering a resource's state, it is possible to watch resources via the performance manager view. For each resource, a watcher client can be invoked over a specific time (for example, 8 a.m. through to 8 p.m.) which polls the resource at regular intervals and checks if certain properties are within specified bounds.

Clicking on "watchComputer" executes the `doWatchComputer` function, which in turn activates a computer watcher program "at" * the start time. Once activated, the "computerPerformance" program periodically obtains the computer's properties and is outlined the the following pseudo-code:—

* The "at" program is used to activate processes at a specific time etc.

```
EVERY time interval DO
BEGIN
      Get Performance Metrics from Resource
      IF Cannot obtain Metrics
      BEGIN
            Report Error to manager
            Log Error
      END
      ELSE
      BEGIN
            Log Performance Metrics to file
            IF (Metrics OUTSIDE Criteria) AND NOT Already Reported
            BEGIN
                  Send Performance warning to manager
                  Log "warning flags" etc.
            END
      END
END
```

Every pre-determined time interval, performance metrics are obtained from the resource. If the resource is “dead” or “unreachable”, this is reported to the fault manager and the fault logged. Similarly, if the current performance data is outside predetermined boundaries, error reports are raised. These reports are only raised once to prevent the manager becoming “swamped” with error reports.

The computer “watcher” program is implemented as an Arjuna client, in the same way as the configuration client discussed in the previous section. (Except that “watcher” obtains properties multiple times.) In order to reduce the overhead of probes, servers are *only* active during each time interval and their properties (loading, process queue etc.) are logged. These logs (written as ASCII files) can then be inspected by the performance manager — either using a graph utility or simply printed as text. In the event of communications or server failure (i.e. RPC timeouts on server construction etc.), error messages are logged and messages are sent to the fault manager *responsible for* the computer (i.e. holding the contract).

4.5 PROBES

Probes obtain (set) properties from and apply management operations to physical (logical) resources. These are implemented in C++ and executed as "servers" by the Arjuna state and lock manager daemons. Each resource managed by the prototype system is therefore controlled (monitored) by an instance of probe and "coupled" to the external resource. These at present include computers, disk and tape units, printers and software distributions, but extra probes could easily be incorporated to manage other hardware, software and network resources.

Resource probes are encoded using an object oriented model, closely resembling the single inheritance hierarchy (shown in Chapter 3) and further structured into three layers.:—

- Recoverable Layer — which provides "recoverable" operations upon external resource. Operations are applyed to the external resource and recovery (and compensation) techniques used in the event of either the resource or operations failing.
- Unrecoverable Layer — this provides a basic interface to external resources and uses "n-versions" of each operation. Operations can be applyed to external resources using system entry routines, system software, network services and even contacting people.
- Integration Layer — operations upon external resources are implemented using "integration objects" which "wrap" existing system software etc. No error recovery is provided by the integration layer as all *application* level recovery is performed by the recoverable layer.

Each layer will be discussed in subsequent sections.

4.5.1 INTEGRATION LAYER

The prototype's integration layer provides the first layer of abstraction above the system programmer's interface to managed resources and is designed to present a clean, *abstract* resource interface to the upper layers of the management information system. This interface is presented in terms of a set of C++ classes which interact with manufacturers' software and system library calls etc., and provides for both manual and automatic integration of management control (monitoring) requests.

In the following sections we will consider integrating managed resources using the system programmer's interface, system software, obtaining configuration information, integrating servers and performing manual operations. Each section is illustrated using code taken from the prototype Management Information System. Readers are referred to Appendix 1 for further details regarding individual integration modules.

4.5.1.1 SYSTEM PROGRAMMER'S INTERFACE

Physical hardware and software resources can be incorporated using the programmer's interface to the particular resource. All of these system library routines are implemented in the C programming language [Harbison91a] (or assembly language with a C function prototype) which can be easilily accessed from C++. [Lippman89a] Generally speaking, many of these integration routines are well described using outline manual pages and other manufacturer's documentation.

Terminal, tape and disk devices are often manipulated by variants of the "ioctl" (Input Output ConTrL) function which can perform tasks such as setting (getting) baud rates, control settings, disk interfaces and even ejecting/formatting floppy disks. Ioctl calls are of the form:—

```
int fd = open(deviceName, MODE);

result = ioctl(fd, REQUEST [, PARAMETER]);

    ...
close(fd);
```

Once the device has been opened and an ioctl call is performed with set (get) requests and additional arguments. An example from the disk controller module is shown below:—

```
class DiskController
{
public:

        DiskController(char *);          //device name
        ~DiskController();

        Error getController();
        Error getControllerAddress(int &);
        Error getDiskInterface(short &);
        Error getUnitAddress(short &);
        Error getUnitFlags(short &);

protected:

        char *theDeviceName;

        int   theControllerAddress;
        short theDiskInterface;
        short theUnitAddress;
        short theUnitFlags;
};
```

The disk controller, is based on the SunOS disk interface (“dkio”) and obtains configuration information (controller address, disk interface code, unit address etc. from the disk unit. Once the class has been constructed with the raw device name (e.g.`/dev/rsd0a`) the controller can be accessed (`getController`) which sets instance data in the class. The `getController` method is shown below:—

```
Error DiskController::getController()
{
      int   fp;
      struct  dk_info    theInfo;

      fp = open(theDeviceName, O_RDONLY);
      if (fp < 0)
      {
            return BadOpen;
      }

      if (ioctl(fp, DKIOCINFO, &(theInfo) ) != 0)
      {
            close(fp);
            return BadIOCTL;
      }

      close(fp);

      theControllerAddress       = theInfo.dki_unit;
      theDiskInterface           = theInfo.dki_ctype;
      theUnitAddress             = theInfo.dki_ctlr;
      theUnitFlags               = theInfo.dki_flags;

      return OkOperation;
}
```

In order to obtain the disk controller's configuration, the (raw) disk device is first opened, "ioctl'ed" and `theInfo` decomposed. Configuration information retrieved from the disk controller include: the controller's address, interface type (eg. SCSI) and unit address. This operation must be executed using "system administrator" (i.e. root) privileges otherwise error conditions (`BadOpen`, cannot open file; `BadIOCTL`, cannot perform IOCTL) are returned.

A large amount of performance data can be obtained from the UNIX operating system kernel. This includes: system loading metrics ("load averages"), virtual (physical) memory consumption and the process table. The system kernel is modelled as a C++ class, which is shown below:—

```
class Kernel
{
public:

        Kernel();
        ~Kernel();

        Error initKernel();
        Error closeKernel();
        Error getImage();
        Error checkImage();
        Error getLoadAverage(LoadAverage &);
        Error getMemory(int &, int &, int &, int &, int &);
        Error getVMstats(int &,int &,  int &, int &);
        ...
};
```

In order to obtain metrics, an "nlist" structure is created which is used to transport data to (from) the Kernel. This nlist is created by the Kernel's constructor and contains symbols describing load average, memory usage, disk transfers etc. The operating system kernel is then opened (`initKernel`) which obtains a `Kvm_token`. Kernel metrics are then transported from the kernel (`getImage`) and checked for completeness (`checkImage`) and can be decomposed before closing the kernel. For example, in order to obtain memory usage, the `X_TOTAL` component is then accessed from the `nlst` and `vmtotal` is obtained, as shown in the following program segment:—

```
Error Kernel::getMemory(int & realMem,
                        int & availRealMem,
                        int & virtMem,
                        int & availVirtMem,
                        int & freeMem)
{
        struct vmtotal      total;
        Error theResult;

        theResult  = getKernelValue(kd, nlst[X_TOTAL].n_value,
                                    (char *)(&total),
                                    sizeof(total));

        if (theResult == OkOperation)
        {
                realMem      = pagetok(total.t_rm);
                availRealMem = pagetok(total.t_arm);
                virtMem      = pagetok(total.t_vm);
                availVirtMem = pagetok(total.t_avm);
                freeMem      = pagetok(total.t_free);
        }

        return theResult;
}
```

The getKernelValue operation is implemented using the SunOS kernel library and simply accesses the X_TOTAL element from the nlst structure. (Accessing elements from the nlist is often performed using the C library routines lseek and read in non-SunOS implementations of the UNIX operating system. Alternatively, the process state vector can be examined and a synopsis obtained, as shown in the next code segment:—

```
Error Kernel::getProcessStats()
{
        structproc   *theProcess  = new struct proc;

        zeroProcs();
        if ( kvm_setproc(kd) != 0)
        {
                // ...
                return UnKnownError;
        }

        while ( (theProcess = kvm_nextproc(kd))  != NULL )
        {
                switch (theProcess->p_stat)
                {
                        case SSLEEP: ++procsSleeping; break;
                        case SWAIT:  ++procsWaiting;  break;
                        case SRUN:   ++procsRunning;  break;
                        case SIDL:   ++procsIntermed; break;
                        case SZOMB:  ++procsZombied;  break;
                        case SSTOP:  ++procsStopped;  break;
                }
        }

        // ...
        return OkOperation;
}
```

The running totals are cleared (zeroProcs and theProcess pointer is set to the head of the process table (kvm_setproc), the process table is then traversed and the process summaries are collected.

Unfortunately some of these system library functions do produce "corrupt" data values even though they apparently "succeeded" and care must be taken to determine if "dubious" data has been obtained. For example, the physical memory metrics produced by getMemory allege that the Sun workstation has approximately half a megabyte of physical memory instead of 16 megabytes!! This could be due to a combination of poor (kernel) documentation and non-portable sample programs.

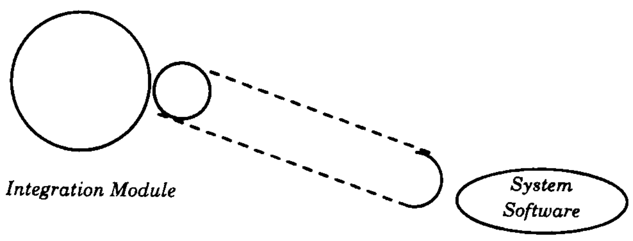
4.5.1.2 INTEGRATING SYSTEM SOFTWARE

Whilst many integration modules have been implemented using system entry routines, some interfaces are subtly different between operating system releases (and *not* backward compatible) and are therefore non-portable. For example, the SunOS 4 disk interface "DKIO" is very similar to the SunOS 5 (Solaris) implementation, some data structures have slightly different names and the floppy disk interface is located elsewhere. A more portable method of integrating physical resources concerns "wrapping" existing software using integration objects.

One method of integrating existing applications in the management information system is to modify the "legacy application's" [Sventek94a] source code and incorporate it into an integration object. (Which requires a good understanding of the original code!) The integration object's exported interface would abstract the legacy application's interface to "mimic" the user's commands etc. While this approach is at least possible, particularly with small (and very simple) applications (for example, the tape controller module); in some cases, the source code is *very* complex or not available.

Rather than modifying application source code, "wrappers" were formed to encapsulate the legacy application with an interface reflecting the original application's functionality. These wrappers execute system software utilities (such as the line printer controller) using UNIX inter-process communication primitives (pipes etc.)[Leffler86a, Sechrest86a], which is illustrated in the following diagram:—

Integrating System Software



System software has been integrated using popen calls which allow processes executing system software to be spawned and results collected etc. These popen calls have been abstracted within an integration module with four members:— Open (Close) pipe, Write to (Read from) pipe. This is shown in the following figure:—

```
class Pipe
{
public:

        Pipe(char *);            // the command
        ~Pipe();

        Error  openPipe(PipeMode);
        Error  closePipe();

        Error  writeTo(char *);
        Error  readFrom(char *);

        Error  readLine(char *);

protected:

        FILE *      thePipe;
        PipeMode    thePipeMode;
        char *      theCommand;
};
```

Hence, once an instance of `Pipe` has been constructed with the program's name and other command line arguments, a pipe is then opened and results read from (data written to) the process. Therefore, providing we know the expected inputs to particular system software (of the format of output data), we can integrate system software. Unfortunately, the popen routines can *only* be used with batch style utilities and “pseudo-terminal” techniques must be used with interactive software. The following system utilities have been integrated into the management information system:—

- DFcommand — obtains file system, size, usage, availability and mount point.
- LPCcommand — the line printer controller.
- PScommand — examines the process queue.
- VMSTATcommand — examines virtual memory statistics.
- MakeCommand — maintains software distributions.
- MailCommand — integrates electronic mail.
- LPRcommand — prints documents using line printer software.

LPCcommand as its name suggests integrates line printer controller software. In order to design this integration object, LPC's source code was examined and an syntax of all screen output was abstracted. Command line arguments were then incorporated into LPCcommand's exported interace, which is shown below:—

```
class LPCcommand
{
public:
        LPCcommand();
        ~LPCcommand();

        Error abortPrinter(char *);
        Error cleanPrinter(char *);

        Error disablePrinter(char *);
        Error enablePrinter(char *);

        Error upPrinter(char *);
        Error downPrinter(char *, char *);    // printername, message

        Error startPrinter(char *);
        Error stopPrinter(char *);
        Error reStartPrinter(char *);

        Error statusPrinter(char *, Boolean &, Boolean &,
                            Boolean &, unsigned& );

        Error topQ(char *);

};
```

Hence, by creating an instance of LPCcommand, a line printer can be started (enabling the printer), stopped (disabling the printer), queue enabled (disabled). The `startPrinter` and `statusPrinter` members are shown below:—

```
Error LPCcommand::startPrinter(char * thePrinter)
{
        char *lprIssue  = new char[100];
        Buffer theBuffer = new char[BufferSize];
        Error theResult;

        sprintf(lprIssue,"%s %s %s",
                    LPcontroller, StartCommand, thePrinter);

        theResult = doPrinterCommand(lprIssue, theBuffer);

        // ...

        if (cannotAccessPrinter(theBuffer))
        {
                theResult = BadPrinter;
        }
        else if (isPrintingOn(theBuffer) &&
            isDaemonStarted(theBuffer))
        {
                theResult = OkOperation;
        }
        else
        {
                theResult = UnKnownError;
        }

        // ...
        return theResult;
}
```

In order to start the printer, command line arguments are constructed and issued to the “lpc” process. Output from lpc is then captured in a buffer and parsed to determine if printing is enabled and a daemon started. In the event of a failure, `UnKnownError` is returned as insufficient diagnostics are provided by the underlying software.

Obtaining a printer's status (spool queue, queue size etc.) is similar to the previous example. A command line is formed, issued to lpc and output is parsed. This is shown in the code segment below:—

```
Error LPCcommand::statusPrinter(char *    thePrinter,
                                Boolean & isQenabled,
                                Boolean & isPrintEnabled,
                                Boolean & canExamineSpoolArea,
                                unsigned& jobsToPrint)
{
        char *lprIssue  = new char[100];
        BuffertheBuffer = new char[BufferSize];
        Error theResult = UnKnownError;

        sprintf(lprIssue,"%s %s %s",
                    LPcontroller, GetStatusCommand, thePrinter);

        theResult = doPrinterCommand(lprIssue, theBuffer);

        // ...

        if (! cannotAccessPrinter(theBuffer))
        {
        isPrintEnabled      = getPrintEnabled(theBuffer);
        isQenabled          = getQenabled(theBuffer);
        canExamineSpoolArea = getCanExamineSpool(theBuffer);

              if (canExamineSpoolArea)
                    jobsToPrint = getSpoolEntries(theBuffer);

        }

        // ...

        returnOkOperation;
}
```

Command line arguments are constructed (of the form: `lpc status PrinterName`) and executed in a pipe. Provided the printer can be accessed (i.e. the printer exists!), the output buffer is parsed using simple string comparisons. If `isPrinterEnabled`, `isQenabled` flags are set, and the number of entries in the spool queue are extracted.

4.5.1.3 OBTAINING CONFIGURATION SPECIFICS

Configuration specific data is often held in text files and parsed by the operating system when the computer is booted or when controller daemons are started, and by parsing these text files it is possible to obtain a large volume of configuration data. Each of the most "useful" configuration files has been abstracted using a C++ class for ease of parsing. These include:—

- `Password` — the password/account file,

- Printcap — printer description file,
- FSTAB — file system table.

The PrintCapEntry module, shown below, is based on configuration entries in the /etc/printcap:—

```
class PrintCapEntry
{
public:

        PrintCapEntry(char *);
        ~PrintCapEntry();

        Error  readEntry();
        Error  writeEntry(char *);            // printcap file
        Error  getBaudRate(unsigned &);
        Error  getPageDims(unsigned &, unsigned &); // length, width
        Error  getMaxCopies(unsigned &);
        Error  getMaxSize(unsigned &);
        Error  getSpoolDir(char *);
        Error  getLogFile(char *);

protected:

        // instance data ...

        void   setDefaults();
};
```

Rather than attempt to parse the file, the termcap library function tgetent is used to obtain the entry which is then decomposed by tgetstr. Two extra functions setNumeric are setString were written to check null (i.e. default) entries.

```
Error PrintCapEntry::readEntry()
{
        char *buffer= new char[1024];
        char *scratch = new char[1024];

        environ            = newenv;

        if (tgetent(buffer, printerName) <= 0)
        {
                // ...
                return UnKnownError;
        }

        setString(acFile,        tgetstr("af", & scratch));
        setString(cFfilter,      tgetstr("cf", & scratch));

        // etc ....

        setNumeric(maxCopies,     tgetstr("mc", & scratch));
        setNumeric(maxSize,       tgetstr("mx", & scratch));
        setNumeric(price,         tgetstr("pc", & scratch));
        setNumeric(pageLength,    tgetstr("pl", & scratch));
        setNumeric(pageWidth,     tgetstr("pw", & scratch));

        // etc ...

        return OkOperation;
}
```

Provided that the `printcap` entry exists, the printer's entry is obtained using `tgetent` * and held in a `buffer`. String and numeric entries, such as "log files", filters and page dimensions are then extracted and held as class instance data. Should the printcap entry be incomplete default entries are preserved. An example printcap entry (produced by the `writeEntry` method) is shown below:—

* By altering an environment variable, it is possible to use the `termcap` library routings to obtain "printcaps".

Print Cap Entry	*Variable Name*
`bramley:\`	*Printer Name*
`:lp=:\`	*Line Printer*
`:af=:\`	*Account File*
`:bu#0:\`	*Baud Rate*
`:cf=:\`	*Filter*
`:df=:\`	*Ditto*
`:du=0:\`	*User Id of Daemon*
`:lf=/var/spool/lpd/bramley/log:\`	*Log File*
`:lo=lock:\`	*Lock*
`:mc#0:\`	*Max Copies*
`:mx#1000:\`	*Max Size*
`:pc#200:\`	*Price per Copy*
`:pl#66:\`	*Page Length*
`:pw#132:\`	*Page Width*
`:rm=semillon:\`	*Remote Machine*
`:rp=bramley:\`	*Remote Printer*
`:sd=/var/spool/lpd/bramley:`	*Spool Directory*

4.5.1.4 SERVER INTEGRATION

It is further possible to integrate networked services i.e. accessing resources across the network via message passing or remote procedure call primitives, into the Management Information base. These include the line printer sub-system which we have considered earlier (as the user interface is actually a system utility programme) and Rstat, which obtains kernel metrics using Sun Remote procedure call techniques.

The `Rstat` has been integrated in the same way as other integration modules, in the form of a C++ class which is shown below:—

```
class Rstat
{
public:

        Rstat(char *);              // hostname
        ~Rstat();

        Error hasHardDisk(int &);

        Error getStatistics();

        Error getLoadAverage(LoadAverage &);
        Error getVMstats(int &, int &, int &, int &);
        Error getBootTime(struct timeval &);
        Error getCPUstates(int &, int &, int &, int &);

protected:

        // instance data ...
};
```

Instances of `Rstat` are constructed using the remote host's name, statistics obtained (`getStatistics`) which can be output. The getStatistics method is shown below:—

```
Error Rstat::getStatistics()
{
        struct statstime *statp  = new struct statstime;
        int    rstatResult;

        rstatResult = rstat(theHostName, statp);

        if (rstatResult < 0)
        {
                // ...
                return UnKnownError;
        }

        thePagesIn   = statp->v_pgpgin;
        thePagesOut  = statp->v_pgpgout;
        theSwapIn    = statp->v_pswpin;
        theSwapOut   = statp->v_pswpout;

        // etc ...

        delete statp;
        return OkOperation;
}
```

Hence, the rstat operation is performed on `theHostName` and a `statstime` structure returned. Providing that the remote procedure call has succeeded (i.e. the return value is non-negative), instance data is obtained from `statp`. Unfortunately, while testing this routine the `rstat` call "succeeded" even though the network service was *not* enabled on the Workstation!!!

4.5.1.5 MANUAL COMMANDS

While it is possible to integrate many systems management tasks using systems calls and software, there is a residue of operations which can only be performed manually. These include physically installing resources, loading tape decks etc. Manual operations upon managed resources are implemented *explicitly* in the management information base, and request the human operator *responsible for* the resource (i.e. holding the contract) to perform operations as requested. Communication to (from) the management information system from (to) the operator is implemented using a "mailbox", which is illustrated in the following algorithm:—

```
Management Information Base

START MANUAL OPERATION
      Find Operator
      Place Request in  Mailbox
      WAIT for reply WITHIN time out
      IF reply == ok      RETURN OkOperation
      ELSE                RETURN FailedOperation
END MANUAL OPERATION
```

Once the operator responsible for the resource has been identified (via a contract data-base), she is asked to perform an operation manually. The actual mechanism for contacting operators (and getting responses) is implemented using Arjuna clients (the `ManualOperation` integration module) and the server (`OperatorsMailBox`). Timeouts and exception handling are also provided by Arjuna tool kit — and are explained in more detail later in this chapter. In the event of manual operations failing (for whatever reason), their effects are cleaned up and undone by the information system's recoverable layer.

```
Operator

FOR EVER DO
BEGIN
      Get Request from Mailbox
      Do Request
      Place Reply in Mailbox
END
```

The `ManDiskUnit` (i.e. manual operations upon disk units) integration module is shown below:—

```
class ManDiskUnit
{
public:

      ManDiskUnit();
      ~ManDiskUnit();

      Error doPowerUp(char *);
      Error doPowerDown(char *);
      Error doLoadDisk(char *, char *); // Diskdrive, Diskmedia
      Error doRemoveDisk(char *);

      Error doScrapDisk(char *);
};
```

`ManDiskUnit` exports methods for powering up (down) the disk unit, loading (removing) floppy disks etc. and an extra operation for scrapping bad disks is included as a compensation action for failed (floppy) disk format operations.

Manual operations are also used by the unrecoverable layer to perform operations upon unreachable (and unconnected) resources. Manual operations are particularly important when considering powering up (down) computational resources — these resources can only be switched off if the probe has been moved from the host.

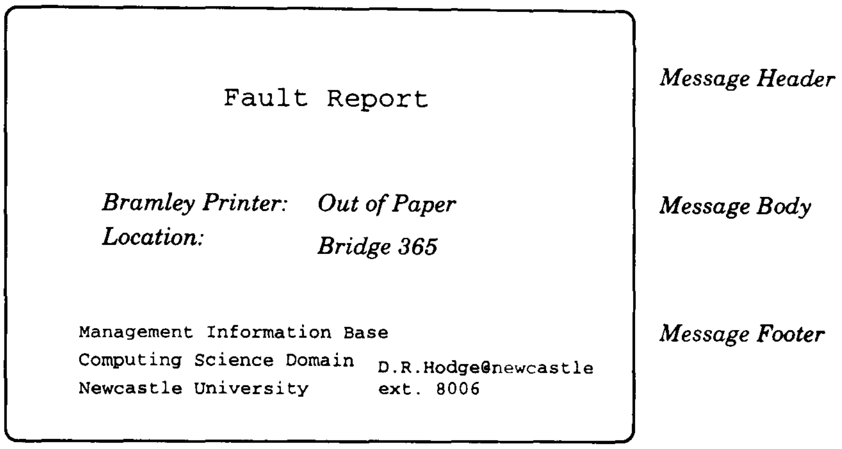
4.5.1.6 REPORTING

In order to provide fault reporting, broadcast messages and mail sub-system integration four modules are provided:—

- `FaultReport` (implemented using `MailCommand`).
- `MailCommand` — interfacing to the electronic mail sub-system.

- `Wall` — which sends broadcast messages to users.
- `News` — interfacing to the electronic news groups etc.

`FaultReport`, implemented using `MailCommand` adds predefined "headers and footers" to fault reports which are delivered using electronic mail. An example fault report is shown below:—



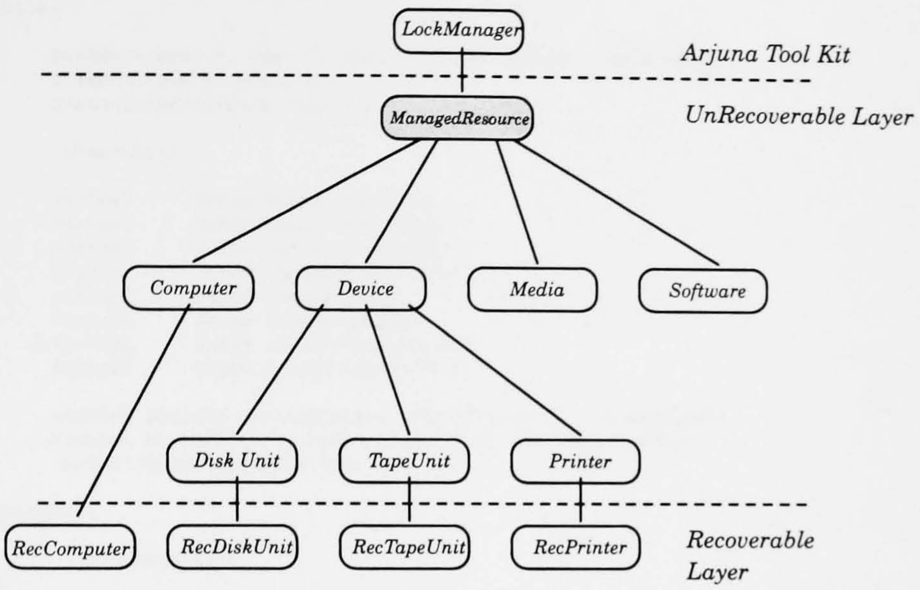
```
Fault Report

Bramley Printer:   Out of Paper
Location:          Bridge 365

Management Information Base
Computing Science Domain   D.R.Hodge@newcastle
Newcastle University       ext. 8006
```

Message Header
Message Body
Message Footer

4.5.2 *UNRECOVERABLE LAYER*

Having considered the physical interface and integrating managed resources, let us turn our attention to implementing resource controllers. These resource controllers are split across two layers; the unrecoverable layer and the recoverable layer which is discussed in the next section. This layer provides a set of unrecoverable operations which are exported from the resource controller classes. (No attempt is made to recover from or report failures as these tasks are performed by the recoverable layer.) These resource controllers are implemented in terms of a single class hierarchy using a model which is shown in the diagram below:—

Type Hierarchy (Principle Classes)



This single level inheritance was chosen due to restrictions imposed by the Arjuna's stub generator and the Cfront C++ compiler. * Extensive use has been made of virtual class members which allows the redefinition of specific member functions by lower levels in the tree. For example, the `DiskUnit` device class is shown below:—

* The stub generator only supports single inheritance and the compiler only permits a certain number of levels in the inheritance tree.

```
class DiskUnit : public Device
{
public:

        DiskUnit(char *, char *, char *, NodeCoupling,  ObjectKind);
        DiskUnit(Uid &, Error &);
        DiskUnit(ArjunaName, Error &);

        ~DiskUnit();

        virtual      Error manualPowerUp();
        virtual      Error manualPowerDown();
        virtual      Error autoFormatMedia();
        virtual      Error autoGetController();
        virtual      Error getStatus();
        virtual      Error autoGetUsage();
        virtual      Error autoGetPartitions();
        virtual      Error autoGetGeometry();

        virtual Boolean restore_state (ObjectState & , ObjectType);
        virtual Boolean save_state (ObjectState & , ObjectType);
        virtual const TypeName type () const ;

protected:

        // instance data ...
};
```

The disk unit, derived from device and managed resource has three constructors: creating a disk unit from scratch and accessing persistent `DiskUnits` using an Arjuna name and unique identifier. Several methods are exported which power the device, obtain configuration and disk usage. These methods are prefixed "manual" — referring to manual operations, "auto" — automatic operations which integrate system calls and software; "remote" (used in the computer class) which integrates network services.

Most of the disk unit's instance data is set at configuration time (disk controller geometry and partition size) and therefore once initialised, periodic calls to `getUsage` are required to keep the probe in step with the physical disk device. `autoGetController`, `autoGetPartitions` and `autoGetGeometry` use `DiskController`, `DiskPartitions` and `DiskGeometry` integration modules respectively in order to obtain configuration information. Whereas `autoGetUsage` uses `dfCommand` in order to obtain disk usage. (Although it is possible to use system library calls to obtain disk usage, it was far simpler to parse the output from "df"). autoGetController is shown below:—

```
Error DiskUnit::autoGetController()
{
      if (theCoupling != CloseCoupled)      return PreFailure;

      DiskController      theDiskController(theDeviceName);
      Error               theResult;
      short               tempInterface;

      theResult = theDiskController.getController();
      if (theResult != OkOperation)         return theResult;

      theDiskController.getControllerAddress(theControllerAddress);
      theDiskController.getUnitAddress(theUnitAddress);
      theDiskController.getDiskInterface(tempInterface);
      theDiskInterface = short2DiskInterface(tempInterface);

      return OkOperation;
}
```

Hence, provided that the processing node is close coupled, an instance of the DiskController integration class is constructed with the disk device's raw pathname. The configuration is then obtained and instance data set. A conversion function short2DiskInterface is used to convert the disk interface code used by dkio to an enumeration type.

4.5.3 RECOVERABLE LAYER

Recoverability can be incorporated in system components using type inheritance[Dixon88a], reflection [Stroud95a, Stroud94a] or delegation. [Rubira94a] In the case of type inheritance, a recoverable class is derived from its unrecoverable counterpart and a set of "super objects" concerning state (operation) based recovery, persistence and serialization. [McCue92a, Shrivastava91a] Whereas, reflection transparently provides recovery, serialization etc. using "meta-objects" which trap operations on application layer objects.

This contrasts with the "delegation" approach developed by Rubira which provides error recovery in a computer controlled train set. Rubira's equivalent of "recoverable components" are "friends" of the components "normal" and "abnormal" state. Her "Recoverable Components" "delegate" operations upon the component (i.e. trains, points, train controller etc.) depending upon the component's state (normal, abnormal etc.) — this can be compared with Kramer et al's work on Dynamic Reconfiguration. [Kramer88a]

There are two approaches to achieving recovery from failed operations. One approach takes a copy of the component's *state* before the object is modified and during recovery the *current* (i.e. modified) state is replaced by the old state of its "snap shot". This is contrasted with *operation based* recovery which records operations upon the system component (for example, in an operation log) and uses inverse (or "anti") operations to recover the object's state.

Recoverability has been incorporated in the management information system using type inheritance in the same way as the examples shown in Dixon's thesis. The unrecoverable layer is used as the baseline interface to manage resources and enacts control (and monitoring) operations through the integration layer to the physical resource. Similarly the non-recoverable layer exports multiple implementations (i.e."n versions") of control operations (manual operations, automatic integration using system software, physical interface via system calls etc.) which are used by the recoverable layer.

Each exported class member from the recoverable layer components is implemented using recovery blocks [Randell75a] incorporating Atomic Actions: using the n-version interface and design diversity from the unrecoverable layer; adding serialisation, persistence and failure atomicity of atomic actions. [Lomet77a, Lampson81a, Spector83a, Marshall80a] Our recovery block algorithm is outlined below:—

```
ensure PRINTER IS ENABLED by
      action automatically enable printer;
end "primary" else-by
      action manually enable printer
end "secondary"
else fail;
```

In the above example, the acceptance criteria (post condition) `PRINTER IS ENABLED` is attempted using a list of alternative atomic operations. First the primary action is attempted to enable the printer automatically, locks are obtained and control signals sent to the printer. Providing the acceptance criteria are met, the operation is committed and no further operations are attempted. However, should the control signals fail to enable the printer, the operation's effects are compensated and the action is aborted; ready to attempt the secondary implementation to enable the printer. In the event of external operations upon the printer succeeding *but* the atomic action not committing, the entire operation is undone by an "anti-operation". This process of attempting operations is performed until the list is exhausted and the recovery block signals failure.

Similarly, by incorporating appropriate pre-conditions within unrecoverable layer components, it is possible to perform management actions regardless of the connectivity or reachability of the external resource i.e.:—

- all calls using the physical interface must be on reachable, close coupled resources.
- loose coupled controllers require the resource to be connected only.
- manual operations do not depend upon the connectivity or reachability of the resource.

For example, the `RecComputer` (i.e. recoverable computer) class is shown below:—

```
class RecComputer : public Computer
{
public:

        RecComputer(char *, char *, char *, NodeCoupling, ObjectKind);
        RecComputer(Uid &, Error &);
        RecComputer(ArjunaName, Error &);

        ~RecComputer();

        virtual     Error onLine();
        virtual     Error offLine();

        virtual     Error powerUp();
        virtual     Error powerDown();
        virtual     Error boot();
        virtual     Error shutDown();

        virtual     Error getStatus();
        virtual     Error getConfig();
        virtual     Error getPerform();

        // ...

        virtual     Error compensate(Error);
        virtual     Error reportError(char *);
        virtual     Error reportError(Error);

        virtual Boolean restore_state (ObjectState & , ObjectType);
        virtual Boolean save_state (ObjectState & , ObjectType);
        virtual const TypeName type () const ;
};
```

The `RecComputer` like the `DiskUnit` in the previous section, has three constructors (creating a recoverable computer from scratch and accessing persistent object using a unique identifier or an `ArjunaName`), destructor, operations to save (restore) state and provide a `TypeName`. Recoverable layer classes unlike other layers, *do not* have their own instance data and therefore inherit the "unrecoverable layer's" data. *

Methods such as `powerUp`, `boot` are provided, but unlike unrecoverable layers, no variants are permitted. Each of these operations is encoded using recovery blocks and implemented in terms of (write locked) atomic actions. For example, the `getKernelMetrics` method is shown below:—

* "Public" type inheritance is used only due to restrictions in the Arjuna stub generator.

```
Error RecComputer::getKernelMetrics()
{
#define ANTI_OPERATION   noOperation();
#define PRE_CONDITION    TRUE
#define POST_CONDITION   TRUE

      BEGIN_RECBLOCK
            BY(autoGetKernelMetrics())
            ELSE_BY(remoteGetKernelMetrics())
            REPORT_FAIL("cannot get kernel metrics")
      END_RECBLOCK
}
```

Rather than encode the recovery block in terms of templates etc.[Rubira-Calsavara94a], macros are used to define an "anti-operation", pre and post conditions etc. Thus, provided the PRE_CONDITION holds, autoGetKernelMetrics is attempted within a (write locked) atomic action. The recovery block macros are shown below:—

- BEGIN_RECBLOCK, declares an atomic action.
- BY(OPERATION), executes operation within a write locked atomic action. If the operation fails its effects are compensated and the atomic action is aborted. If operation succeeds but the atomic action fails to commit, an anti-operation is performed. Otherwise, the operation returns "succeeded."
- ELSE_BY(OPERATION), c.f BY(OPERATION).
- REPORT_FAIL(MESSAGE), reportError(MESSAGE).
- END_RECBLOCK, Return operation failed.

Hence autoGetKernelMetrics is then attempted in the "unrecoverable layer" (i.e. Computer class). Should that operation fail, the atomic action is aborted and secondary operations (remoteGetKernelMetrics) "tried".

Other recoverable operations, such as powerUp have predefined anti-operations (i.e. powerDown) which are used to provide operation based recovery. Hence, should operations succeed on the "external resource" and the atomic actions fail to commit (for example, because the object store is full) the anti-operation is applied. Similarly, if operations fail enroute, such as while the `RecPrinter` is printing jobs, a compensation operation (for example, `ScrapPrintJob`) is performed.

The extent that partial system failure can be detected is largely determined by external resource's interfaces and exceptions raised by the Arjuna remote procedure call system. Some printers for example only indicate that the printer has misfed when the controller "refuses" to write to a file descriptor rather than provide any sophisticated error diagnostics. Let us consider the following scenario: The agent in her role as configuration manager selects a resource via a view and sets printer properties.

- Operations upon the probe are performed within a recovery block implemented using atomic transactions. Resource properties are locked and operations are applied to the external resource.
- If an individual operation fails, its effects are compensated and the transaction is aborted. Operations are applied within the recovery block until a variant succeeds. Should all variant operations fail, a fault report is issued and the resource marked `ResourceFailed`.
- If the server crashes during the operation, the resource and server's state become inconsistent. The server's state remains on stable storage and is reconciled with the resource as part of "crash recovery".
- If the client crashes during the operation, the server will (should!) detect the failure as part of an orphan detection mechanism and Arjuna's "state manager" will kill the server.
- If the communications system fails, the client's remote procedure calls will timeout and raise appropriate exceptions.

Recovery is performed at four levels:—

- Repairing (Replacing) External Resources, i.e. calling out an enginner, disconnecting the resource and taking appropriate action.
- Repairing the Communications Subsystem, i.e. replacing severed network cable; erecting a "fire wall", restarting routers/ bridges etc.
- Restarting Arjuna, cleaning up any shared memory, killing servers and applying co-operating termination protocols to any inconsistent atomic actions. Arjuna management daemons can then be restarted.
- Ensuring that the probe's state is consistent with the external resource. This is simply performed by getting the resource's (latest) properties.

Resource failure and recovery will be discussed further when considering fault injection (in Chapter 5).

4.6 DISCUSSION

This chapter has considered the application of our prototype management information system to the task of managing a large distributed computer system. The University of Newcastle upon Tyne's academic structure was mapped into the workbench's user interface which allows management agents in their respective roles to access HP and Sun workstations, disk units, printers etc. *anywhere* on campus.

Dependencies between resources were modelled using "resource domains" and mapped into the management workbench in the same way as organisational domains etc. Multiple views of resource (and organisational) domains are displayed by the workbench, and therefore resource domains can record both the logical *and* physical structure of managed resources. This proved much simpler to implement than using a configuration language such as PCL. [Sommerville95a] Although PCL provides a "convenient" syntax for recording a distributed system's structure, PCL does not provide physical operations on managed resource states or invariants between dependent resources. Our managed resources are encoded as C++ objects which control and monitor physical resources and composite objects implemented using domains. Therefore change managers can reconfigure the distributed system and maintain both structural (and functional) integrity between resources. Variations between hardware (software) platforms are also recorded using the domain structure. For example, a software package developed for a Sun 4 architecture is located within a Sun4 domain etc. — This avoids adding value expressions in resource relationships.

The probes were deliberately designed in the form of 4 layers, ranging from an operating system specific integration layer through to the view based user interface. Several probes were implemented: terminal, printer, disk device, floppy device, mag tape unit, computer, software. Although this list is not an exhaustive set of an organisation's resources. Extra controllers can be easily incorporated, such as Bridges, routers, terminal concentrators and network services, for example the Network File System and Network Information Service[Stern92a].

"Inherited recovery" cleanly (and simply) abstracted unrecoverable layer multiple implementations from higher layers in the prototype. Recoverable layer class components therefore consist largely of a set of recovery blocks which hide lower levels in the prototype. Similarly, using the Arjuna tool kit to construct server and clients proved valuable and considerably eased the burden of distributed programming. Arjuna has now been ported to ANSAWare, HP Workstations and even personal computers running the Linux operating system and therefore provides a portable distributed programming environment.

Although using Arjuna had many advantages, one of its greatest drawbacks concerns its ability to manage a very large number of managed objects. Each object's state is represented in "core dump" format which forms a very large, highly fragmented file. Incorporating a query language in the management information system would be useful for collating data and producing reports. For example, queries such as:—

```
SELECT
      ComputerName, Load
FROM
      AllComputers
WHERE
      Load Increased by 10%
```

Arjuna's object store has been mapped into a commercial data base and object multiplexing attempted. This would be particularly useful in user (and software) management where large numbers of objects are manipulated.

In the current release of Arjuna software (PR3.2), configuring session remote procedure call timeouts (and re-try values), binding servers to hosts is a fairly "low level" task — creating RPC structures, binding them to `ClientAction` and eventually to the server's stub interface. [Parrington95a] The actual server and RPC bindings could be performed using a configuration language similar to Conic[Magee89a] which would make the programmer's life much easier! For example:—

```
RecComputer  A;
ComputerClient      B;

CREATE A ON ncl.catless
CREATE B ON ncl.espley

LINK B TO A
      WITH 16 Retries, 10 min. TIMEOUT
```

Or even, in the case of replicated servers,

```
ContractBase A;
Client     . B;

CREATE A ON ncl.catless, ncl.espley, ...
LINK B TO A

// etc.
```

Design diversity techniques are particularly useful in "tolerating" object migration (i.e. moving probes between nodes in the distributed system). Thus probes which are not closely coupled can access their managed resource using network services or even a "remote integration layer". Manual operations do cause particular problems in the management information base, particularly concerning longer timeouts and scheduling (human) operators. However, performing operations "by hand" is implemented in the same way as contacting a network service. Integrating "automatic" operations using existing system software may be less efficient than using system entry routings *but* can produce more portable program code.

Although "system entry" standardization has been addressed by POSIX etc., much more work is required in producing consistent interfaces between operating system releases (and implementations). OSF for example have produced both application (and system level) interface standards as part of their distributed environment but unfortunately very little has been published in this area.

Tcl/Tk proved a great asset in constructing the prototype's user interface. Not only was Tk very easy to use but allowed development of a forms based interface. Although Tcl/Tk is an interpreted language, this has no significant effect on performance, particularly as clients/servers are executed as binaries.

A summary is shown below:—

Views	Multiple views of resources
Servers	Terminal, printer, mag tape and disk units Computer, software
Server Size	.5 meg *
Client Size	.5 meg *
Recovery	Backward error recovery and compensation actions
Fault reporting	Fault reports are send to the "Fault Manager" responsible for the resource.
Manual operations	Performed by resource operator
Unreachable/ unconnected	via manual operations

* Compiled on SunOS 4.1.3 using Cfront 3.0.1 C++ Compiler. (Stripped Executable Code)

Chapter 5

EVALUATION OF PROTOTYPE

Because this (CMU) is a research university, much of the equipment and software is experimental. ... The number of ways that a computer loaded with "questionable" software can fail is virtually unlimited ... this produces wonderful opportunities for groups to blame each other for undiagnosed problems[Arms88e]

In the previous chapter we detailed the implementation of our prototype management information system. Our prototype system allowed management agents to control (monitor) resources connected to the University of Newcastle campus network and demonstrated using type inheritance to incorporate error recovery. The prototype system was used to manage resources such as terminals, printers, disk and tape units, work stations and software distributions and we noted the importance of adhering to standards when porting probes to variants of the UNIX operating system.

Using a management information system to control (monitor) campus resources carries a price. Our prototype system used the Arjuna tool kit to provide concurrency control, state management, remote procedure call etc. and we added application layer recovery techniques (recovery blocks, compensation etc.) to resource probes. In this chapter we will consider the performance overhead of incorporating error recovery and demonstrate the prototype's behaviour in the presence of faults.

5.1 PERFORMANCE EVALUATION

In order to evaluate the prototype's performance, a series of tests were performed on normally * loaded workstations. When assessing the prototype's

* Each workstation was running standard daemons and two or three users each running several

performance, four factors must be considered:—

- Measuring time is of course hardware dependent, and whilst system calls such as `gettimeofday` and `clock` (ANSI C library function) claim to measure microseconds, this depends upon the machine's physical clock rate.
- "Arjuna overhead" — the *quoted* performance of the Arjuna tool kit compared to the times observed in experiments.
- "Integration Overhead" — i.e. the load imposed by performing operations on managed objects, such as building software or probing device drivers *independently* of Arjuna.
- "Combined evaluation" — times observed by executing recoverable layer components (i.e. Arjuna *and* the integration layer).

Performance data was obtained from the integration and recoverable layers of the management information system by adding standard system calls to test harnesses. Each operation was performed 1,000 times and the average performance measured, and some experiments were performed several times to ensure the data were consistent.

5.1.1 ARJUNA OVERHEAD

Analysing an Arjuna application's performance is particularly complex because of the impact of factors such as kernel file buffer and object store file descriptor caches. A recent paper [Parrington95b] cites a performance evaluation of the Arjuna tool kit — "standard operations" were applied to objects containing 1024 bytes of instance data. (Times were measured using standard system calls etc.)

window applications.

The following performance times are quoted:—

Operation	*Read Only*	*Write*
Top Level Action (commit)	9.5 ms	101.0 ms
Top Level Action (abort)	9.5 ms	10.5 ms
Nested Action (commit)*	5.0 ms	5.5 ms
Nested Action (abort)	5.0 ms	5.5 ms
Distributed Top Level (commit)	19.0 ms	130.0 ms

Write locked atomic actions are dominated by writing the object's state to disk and "intention list handling" which represent 70% of execution time.

Server Objects are initiated when a child process (the actual server) is detached from the Arjuna object manager. Remote procedure calls between the client and server are performed using UNIX interprocess communication primitives (sockets etc.) and servers are terminated by "killing them". The following statistics are quoted by the Arjuna group:—

Operation	*Time*
Server Initiation	233.0 ms
Server Termination	4.0 ms
Null RPC Round Trip	3.0 ms

A client program (test harness) was adapted to time calls to a `RecComputer` server. The server was constructed from scratch, properties obtained and terminated — we will consider the performance of other operations later.

File	*Sizes*
Client	1102112 bytes **
Server	4374032 bytes **
State SnapShot	404 bytes (fragmented)

The following results were observed:—

* Nested commits exploit caching

** Both file sizes represent stripped executables (dynamically linked), compiled and built using G++ version 2.6.2.

Operation	*Time*
Server Creation	100 ms
Server Termination	20 ms
Save State	1.64 ms
Restore State	0.3 ms
Read (Commit)	16 ms
Write (Commit)	118 ms
Write (Abort)	17 ms

When activating a `RecComputer`, the client constructs a (local) server which in turn activates a (remote) server via remote procedure calls and stub-code. `RecComputer`'s construction not only initialises fault injection routines (which are part discussed later in this chapter) but each derived class (i.e. computer, managed resource) and other Arjuna modules (such as lock and state manager etc.) The state is then written to the object store under the defined `typename`. Although our performance times appear considerably faster than the quoted times, this could be simply due to environmental conditions. "Null" operations within read (write) locked atomic actions were significantly higher than quoted figures (16 ms and 118 ms for read-commit and write-commit compared with 9.5 ms and 101 ms respectively).

5.1.2 INTEGRATION OVERHEADS

Managed resources are integrated into the management information base using a variety of system calls, utility software, network services and even contacting a human operator. In order to obtain performance data from the integration layer, system calls to time operations were added to several modules' test harnesses. These include:—

- "System Call Integration" — getting device properties using the UTMP, `PrintCapEntry` and `TerminalController` classes.

- "Utility Program Integration" — summising a computer's process queue (`PScommand`) and virtual memory (`VMStatCommand`).

These tests were performed on Sun 4c (running SunOS 4.1.1), Sun 4m Sparc Station (running Solaris) and a HP workstation running HP-UX.

The following results were obtained:—

Module	Command	SunOS4	Solaris	HP-UX
TerminalController	getController	16.5 ms	100 ms	n.a.
PrintCapEntry	readEntry	16.7 ms	10 ms	n.a.
Utmp	getUsers	1.6 ms	0.52 ms	n.a.
PScommand	getProcessQ	30 ms	177.92 ms	100 ms
VMStatCommand	getVMstat	16 ms	12.409 ms	60 ms
MakeCommand	makeAll	n.a.	16000 ms	n.a.

Obtaining properties from a device involves opening a device (`open`), performing several control functions (`ioctl`) and closing the device. In the case of the `TerminalController` a structure (`termios`) is decomposed to reveal many properties "or'ed" together whereas the printcap entry is obtained using the `curses` library.

The `PScommand` and `VMStatCcommand` classes were developed simply because insufficient documentation was available to incorporate Solaris and HP-UX kernels directly. Both classes execute systems software using popen and parse any output. `MakeCommand` compiled and linked a 15k C++ program written by a M.Sc. student. His program comprised 21 modules and used standard libraries. Most of the time shown was spent compiling the program with 160 ms taken to link.

As expected, integration via system entry routines was approximately ten times faster than via systems (and other utility) software. Obtaining the number of users logged into Sun workstations was particularly fast even though the 0.52 ms time represents reading a binary file. Producing process queue and virtual memory statistics from the three operating systems produced greatly different performance times: SunOS 4.1.1 values were approximately six times faster than Solaris. * Even though implementations of the `PsCommand` module produced a synopsis of the workstation's process queue by parsing a large buffer, Solaris and HP-UX's module implementations are based on UNIX System 5 version of the "ps" software. Solaris's `PsCommand` performance time is by far the slowest.

5.1.3 RECOVERABLE LAYER

Three recoverable layer modules were timed as part of this performance evaluation. `RecTerminal`, `RecComputer` and `RecSoftware`. Operations were executed on a Sun 4m (running Solaris) with both client and server located on the same workstation. These experiments were performed on a single workstation as Arjuna's performance data is only available for this machine architecture. A test harness was adapted to time calls to a "recoverable" server. The client created a server from scratch, obtained properties and terminated the server. Properties were obtained within "write locked" atomic actions and later transferred to the client within "read locked" actions.

5.1.3.1 COMPUTER

The `RecComputer`'s test harness, used earlier as a comparison against the Arjuna group's performance data, was adapted to provide times taken when getting properties from physical resources and returning values to the client. Two experiments were performed, the first, getting (showing) properties using the first operation in the recovery block, and secondly, performing an

* The Solaris experiments were later repeated with only one user "logged on" to ensure accuracy of performance times.

operation after the first "alternative" had failed. (i.e. aborted atomic action followed by a successful operation.) The following times were observed:—

Operation	*Lock*	*Time*
getProcStates	Write (Commit)	340 ms
getUsers	Write (Commit)	200 ms
getMemory	Write (Abort)	680 ms
showPS	Read (Commit)	24 ms
showUsers	Read (Commit)	26 ms

Getting the process queue, users and memory requires evaluating a recovery block (using the "UnRecoverable" layer) executing write-locked Atomic Actions, packing (unpacking) the object's state, writing to (read from) disk, and calling the integration layer. In the case of `getProcStates` the observed time is almost double the integration layer time and `getUsers` — almost all of the observed time is take up by Arjuna!

Obtaining the machine's physical memory represents a write-locked transaction *aborting* and sending a fault report using electronic mail. Composing the fault report requires concatenating signature files (mail header and footer) and a small fault report. This disk access and string handling probably accounts for the large performance time observed in this experiment. "Showing" the process state and number of users simply involves acquiring a read-lock and copying instance data across the communications subsystem. Both times observed in this experiment are double quoted figures (20 ms vs 10 ms quoted).

In a second experiment, the "recoverable computer" was altered to demonstrate the effect of operation failure. `getProcStates` first attempted to obtain values from the kernel (which returned `FailedOperation`) followed by using `PsCommand`. The following time was observed:—

Operation	*Time*
getProcStates	337.6 ms

The 337.6 ms represents: getting kernel metrics (within an aborted write locked action) followed by using the `PsCommand` (committed write locked action) and is slightly lower than the expected time of 357 ms.

5.1.3.2 TERMINAL

A single experiment was performed on a `RecTerminal` client which timed operations to get (show) properties. Constructing the terminal's server was slightly slower than the `RecComputer` in the previous set of experiments. Obtaining the terminal's controller required calling `autoGetController` in the unrecoverable layer within a write-locked atomic action. This was slightly slower than the expected time of 110 ms, whereas showing the terminal's speed was the same as Arjuna's benchmark. The following times were observed:—

Operation	*Lock*	*Time*
Construction	n.a.	193 ms
Termination	n.a.	18 ms
getController	Write (commit)	114 ms
showSpeeds	Read (commit)	10 ms

5.1.3.3 SOFTWARE

To provide a contrast with recoverable computers and terminals, a `RecSoftware` client was adapted and performance data obtained. This server was the slowest in the three sets of experiments this and could be due to local "environmental" conditions.

Operation	*Lock*	*Time*
Construction	n.a.	400 ms
Termination	n.a.	52 ms
makeAll	Write (commit)	18 s

5.1.4 EVALUATION

Producing "bench marks" for the prototype's performance is very difficult. The time taken to perform operations depends upon the computer's architecture and physical configuration, loading and even user's behaviour. On our Sun 4m running Solaris, the machine was used by two or three people, running window applications and a World Wide Web server. Performance times could have been affected by memory limitations and other external factors.

When evaluating the prototype's performance, we first compared the quoted Arjuna figures with one of our servers (which had been adapted to perform null operations) which provides the basis for subsequent bench marks. Our hypothesis was simply that recoverable layer operations should take "integration layer time *plus* Arjuna": these layers were timed separately, compared and contrasted.

Generally speaking, integration layer operations performed using system calls took approximately 10 ms; and using system software took 150 ms. However, once these same operations were performed within an Arjuna server, the observed times were dominated by write locked atomic actions committing. The Arjuna group's performance evaluation highlights this problem and concludes that a bottleneck exists in their object store.

Although our performance times are much slower than expected, operations are still performed in "real time" and should not adversely affect the management agent using her workbench. Obtaining a set of properties from a managed resource (even across a network) should take a matter of seconds. Providing operations on servers are performed at appropriate intervals (for example, when opening and closing views) she should not notice the effects of bottlenecks in Arjuna.

5.2 TESTING THE MANAGEMENT INFORMATION SYSTEM

The management information system was initially tested (and debugged) using a test harness which created probes (servers) on both Sun and HP workstations, modified and periodically obtained properties using performance views of managed resources. At each layer of the prototype system (integration, unrecoverable, recoverable etc.) components were tested and produced correct results. Several bugs were discovered which concerned Arjuna's configuration on the Sun workstation (resulting in the workstation running out of swap space) and inappropriate remote procedure call timeouts / retry values when connecting to HP workstations.

In order to test the prototype's fault tolerance provision, fault injection was incorporated into the management information system. Fault injection techniques have been recognised as a useful way of testing error detection schemes, studying the behaviour under faulty conditions and examining the adequacy of fault tolerance mechanisms. Faults can be injected into the communications sub-system, protocols, algorithms or even the actual "target system" and can be implemented using both hardware and software techniques. For example, using test pins on hardware chips, special proms or using specific software test harnesses. Rather than using special hardware (or even unplugging boards and devices!) software was developed based on Ingham's "delayline" wide area network simulation tool[Ingham92a].

Fault injection techniques used in the management information base are based on a fault injection library containing statistical formulae (used to generate probability distributions) and routines used to emulate faulty system behaviour such as resource (operation) failure and nodes crashing. Each resource is allocated a set of probabilities for each failure class and a mean operation time (and distribution) which are used within the fault injection algorithm, which is outlined below:—

```
For each fault class Load
      Fault Rate, Mean Operation Time and Distribution
Sample Injection Probability FROM Uniform Probability Distribution
IF Injection Probability > Fault Class's Fault Probability
      INJECT Fault Into Management Information Base
ELSE
      Perform Operation as Normal
```

Within the management information system, five classes of failure exist:—

- Long Operations, where operations upon the particular resource take much longer than usual.
- Operation Failure, where control (monitoring) operations are not successful.
- Physical Resource Failure, such as peripheral (disk, printer etc.) and computer failure.
- Node failure, where the processor node fails causing all of the servers located on that node to fail.
- Communications failures, such as network delays, message corruption and loss.

We will consider application and communication level fault injection in the following sections.

5.2.1 APPLICATION LEVEL FAULT INJECTION

Application level fault injection was incorporated in the management information base by compiling recoverable layer probes with a `FAULT_INJECT` flag which redefined recovery block macros and incorporated injection operations, probability calculations etc. This avoided manually altering a lot of program code and allowed selected servers to be "injected". For each server, the following information is required:—

- Fault Rates: long operations, operation failure, server crash.
- Delay Time and distribution. Delay distributions can be: fixed, uniform, exponential, poisson and gaussian (i.e. normal distribution).

These probability distributions, fault rates and delay times are held as global variables in the server. Default values can be redefined in the probe's constructor although ideally these could be loaded from a configuration file. When "faulty" operations are selected, the following actions are performed:—

Fault Class	*Action*
Normal Operation	Perform Operation as usual
Long Operation	Perform Operation as usual after delay time
Server Crash	Terminate Server
Resource Crash	Set error code
Operation Failure	Set error code

Random number calculations used throughout the fault injection routines and probabilities are based on POSIX (48 bit) random number functions and use the time as a "seed." Random numbers are initiated at the start of each recovery block as part of an `initFaultInjection` routine.

"Uniform" and Poisson distributions are already provided by the Delayline software. (Uniform distribution are simply constructed using a random number generator.) Extra distributions such as Gaussian (i.e. normal distributions) and exponential are incorporated into the application level fault injection suite using functions from "Numerical Recipes". [Press86a] Readers are referred to Numerical Recipes for further details regarding particular algorithms.

Delay Distribution	*Delay Time Calculation*
Poisson	Get delay using mean value
Exponential	Multiply delay by Exponential Ordinate
Gaussian	Gaussian Ordinate + mean *
Uniform	Multiply delay by mean
Fixed	Mean time value

5.2.2 *COMMUNICATION LEVEL FAULT INJECTION*

Communication between clients and servers in object based systems can be disrupted in several ways. These include physically severing the communications media (or isolating network segments), intercepting messages between processes [Tao95a] and redefining inter-process communications routines[Ingham92a].

Arjuna could be recompiled using Ingham's software which would transparently intercept (delay, "garble" and lose) messages between processing nodes. This is performed by redefining UNIX inter-process communication primitives (sockets etc.) and replacing a system header file. Delayline could also be used to simulate "network partitions" between host groups, ARP storms and other undesirable characteristics (such as "black holes", "poison packets etc.)[Bosak88a] Unfortunately, this software is no longer available and the Arjuna group are currently redeveloping their crash simulation software!

Rather than using Delayline, communication level faults are incorporated in the management information system by modifying remote procedure call timeouts and retry values. By allowing "long operations" and server crashes,

* Assuming that the distribution has a standard deviation of 1

(hopefully) the Arjuna remote procedure call mechanism would be fooled into "believing" that communications between clients and servers had been disrupted and take appropriate action. In the event of timeouts etc. exceptions are raised by remote procedure call mechanism and caught by the client. Orphan detection and "killing" is also performed automatically by the Arjuna system.

5.2.3 FAULTY TERMINAL EXPERIMENT

A `RecTerminal` server was recompiled with application layer fault injection macros in order to simulate a slow, faulty and generally unreliable device. (Communication level faults were not included in this experiment in order to concentrate on the application layer.) The usual recovery block, macros were replaced and fault injection/operation logging included. Whilst it is possible to include Arjuna debugging information (object locking, remote procedure calls, atomic actions etc.), this level of detail was *not* recorded to avoid "debug information" overload.

Two experiments were performed using the (faulty) `RecTerminal` server using the probability values and delay distributions shown in the table below:—

Exp.	Probability Values					
	Long Op	Resource Failure	Server Failure	Operation Failure	Mean Delay	Distribution
i)	0.75	0.75	0.00	0.50	50 s	Uniform
ii)	0.50	0.50	0.05	0.50	70 s	Gaussian

As expected, long operations taking *less* than the remote procedure call time out behaved as "normal". Operations taking longer than the remote procedure call timeout value caused remote procedure call retries and eventually "time out" exceptions in the client. When operations "failed", fault reports were issued and their effects "compensated" by the recovery block algorithm.

Failed operations were then aborted and the terminal server's state was restored from stable storage. When the (physical) terminal resource "failed" the probe was marked "external resource out of order", the server's state was then committed to the object store and further processing by the client abandoned.

5.2.4 FAULTY SOFTWARE EXPERIMENT

When testing the `RecSoftware` server, recompiling and installing software distributions located on (remote) HP workstations proved notoriously slow and often caused the client's remote procedure call timeouts (and retries). To further demonstrate the server's behaviour in the presence of faults, the server was recompiled with application layer fault injection macros.

Two experiments were performed using the (faulty) `RecSoftware` server using the probability values and delay distributions shown in the table below:—

Exp.	Probability Values					
	Long Op	Resource Failure	Server Failure	Operation Failure	Mean Delay	Distribution
i)	0.75	0.05	0.00	0.90	300 s	Uniform
ii)	0.90	0.05	0.05	0.50	500 s	Gaussian

In both experiments, performing `makeAll` operations on software distributions failed and were compensated by `makeClean`. Delayed operations raised remote procedure call time out exceptions even though the server was still active. The server was terminated normally by the client's destructor.

5.2.5 DISCUSSION

Although the application layer fault injection library demonstrated the management information system's behaviour in the presence of faults and highlighted remote procedure call time out problems in the `RecSoftware's` client programs, these experiments were limited by the lack of Arjuna crash simulation software. In order to fully demonstrate the prototype's behaviour in the presence of faults, communication and distributed processing environment fault injection is also required.

Due to the lack of appropriate software, we could not illustrate the effects of communication failure (in particular, network partition and congestion) or probes crashing during two phase commit protocols. Furthermore, Arjuna's fault management software only provides host level crash management. When servers crashed, their shared memory etc. was still retained by the Arjuna system and local state management daemons had to be restarted manually.

However the fault injection software illustrates the management information system's behaviour in the presence of *application layer* faults and demonstrated issuing fault reports and operation compensation (clean up). While performing fault injection experiments a bug was discovered in the recovery block macros concerning the reuse of atomic actions. Arjuna uses a "state machine" model of atomic actions (i.e. begin, commit, abort etc.) and we (incorrectly) assumed that previously committed (aborted) actions could be restarted. This "misunderstanding" was easily corrected by a minor alteration to the macros and may not have "surfaced" except through the fault injection experiments. Our fault injection system was later used to illustrate the crash (and restart) semantics of dependable change schedules and is further discussed within the section on fault management.

5.3 SUMMARY

The management information system's performance was evaluated on a Sun workstation running the Solaris operating system. Integration and recoverable layers of the information system's architecture were timed, and compared (contrasted) with *quoted* Arjuna performance data. Our tests illustrated bottlenecks in the Arjuna object store which severely affected the prototype's performance. Unfortunately recoverable objects in the Arjuna system are implemented as heavy weight server processes and are dedicated to managing *one* resource's state. Probes within the management information system are therefore only active for a minimum time interval (i.e. while properties are transferred between the workbench and servers) and this therefore avoids the host computer being "hogged" by the management information system.

Faults were injected into the mangement information system in order to illustrate the prototype's behaviour in the presence of errors. Faults were incorporated into the recoverable layer of the prototype's architecture to "simulate" delayed and faulty operations, servers crashing etc. However, in order fully to test the prototype, a more sophisticated system is required which injects communication *and* distributed environment faults. Unfortunately, the current Arjuna implementation does not fully support crash simulation (as their software is still being developed) and only host level crash recovery is provided. Hence, when servers crash as part of our fault injection, the Arjuna system must be manually cleaned up. This is of course far from adequate.

Chapter 6

MANAGING CHANGE

Now here's another fine mess you've gotten us into *

In the previous chapter we have considered the management information system's application level architecture, initial configuration and testing. Unfortunately, distributed computer systems are rarely static. Computer systems *evolve* over time as resources are superseded by new technology and the organisation's computing requirements expand. Furthermore, many organisations become so dependent upon their computing resources that changes to distributed components must be performed while their computer system is active. These "dynamic changes" can cause major disruption (and even financial loss) to the organisation and therefore must be carefully planned. Furthermore, major changes to very large, complex distributed computer systems can affect *many* resources and interconnections, and (potentially) take a long time to complete.

This chapter concerns the management of changes to a distributed computer system. These include both "planned changes" (i.e. *Change Management*) and "unplanned changes" (i.e. *Fault Management*). Changes to the distributed systems are implemented using "views" and enacted using "change schedules". Schedules are used to implement *planned* changes to managed resources and isolate (reconnect) failed components. We will consider implementing schedules using systems of long running actions and a method of gracefully shutting down (and restarting) schedules.

* Oliver Hardy (1892 - 1957)

6.1 RECONFIGURATION

Permitting dynamic changes to distributed applications is becoming increasingly desirable. This allows continual services for those system components not affected by change and provides the minimum disruption to system users. Distributed computer systems have become so essential to organisations that "down-time" due to reconfiguration could cause financial loss as well as annoying system users. Reconfiguring system resources should therefore be carefully planned (for example, only performing major changes at week-ends) and synchronised, preventing inconsistencies between components during the change process. This could involve migrating objects between nodes, replacing hardware to improve system response times or even "sub-netting" the communication's sub-system.

The Arjuna system, for example, allows clustering of composite objects and type specific locking to increase system concurrency. An example cited in their programmer's guide concerns a distributed newspaper. Articles are edited by journalists on their personal workstations before being sent (migrated) to the newspaper editor. Local editing by journalists and the newspaper editor avoids the overheads of paging across the network and increases response times etc.

6.1.1 CHANGE MANAGEMENT AND SCHEDULES

Within the management information system, (major) changes to dependent resources are performed using the change manager's view of "resource domains". This view comprises icons representing the composite resource, displayed in the same way as other domains within the management information system, with an extra menu bar which allows change managers to alter a resource's structure (state). For example, adding a printer to a workstation or upgrading a software package across a workstation cluster. This approach is similar to the "Architect's Assistant" developed at Imperial College[Kramer93a] as part of the Darwin project[Magee94a] and the PCL editor[Bowers94a].

One method of performing major system changes requires the use of "change schedules" where (dynamic) reconfiguration is encoded using a configuration language and implemented via a "Change Algorithm" [Kramer88a] and protocol [Young91a] as demonstrated in the Conic system[Magee89a]. Objects are created (deleted), activated (deactivated), linked to (unlinked from) other objects using change actions described in a schedule. These change actions are applied to the system's structure and assume a fault free environment. Change actions can, of course, be permitted in a concurrent system with appropriate locking but little attention is given to *application level* reconfiguration or reliability considerations. For example, while configuration files can encode a nurse/patient monitoring system, "safety critical" features are not described.

For example, in order to connect a printer device to a workstation, the following steps are required:— In order to connect a printer device to a computer, the following steps are required:—

Step	*Operation*
i	BOTH printer and resource must be shut down and unpowered
ii	Physically connect device to computer
iii	Power up computer and device
iv	Boot computer
v	Configure printer
vi	enable spool queue and printer

And, in order to prevent other system administrators from performing configuration changes while the device is being installed, the entire change operation must be locked. This is outlined in the change schedule below:—

```
BEGIN CHANGE SCHEDULE
      LOCK Printer, Computer
      SHUTDOWN Printer, Computer
      CONNECT Printer TO Computer
      BOOT Computer
      INSTALL Printer AND CONFIGURE
      UNLOCK Printer, Computer
END CHANGE SCHEDULE
```

Hence, the entire change schedule is serialised and each of the enclosed operations is performed using the managed resource's probes. The change schedule maintains invariants and prevents inconsistences between between dependent resources. Change schedules can be used to reconfigure application level resources, the management information system's structure, isolate failed components and recover from failure.

6.1.2 IMPLEMENTING DEPENDABLE SCHEDULES

Change schedules could be implemented using nested transactions, where each operation is encoded as a single atomic action within a top level action. Should individual operations fail (either the external resource or the atomic action is aborted etc.) appropriate error recovery is performed. However, should the top level action fail to commit, the entire change schedule must be aborted and constituent components undone. For example, the device connection change schedule is shown below:—

```
RecPrinter          thePrinter( ... );
RecComputer         theComputer( ... );

AtomicAction SA, GA1, GA2, GA3;

SA.Begin();

  GA1.Begin();

      // stage 1: power off / shut down

      GA1.lock(write);

      theComputer.shutDown();
      theComputer.powerOff();
      thePrinter.powerOff();
  GA1.End();
  GA2.Begin();

      GA2.lock(write);
      // stage 2: connect and device
      theComputer.connectDevice(theDevice);
      theComputer.installDevice(theDevice);
  GA2.End();
  GA3.Begin();

      GA2.lock(write);

      // stage 3: power up resources ....

      theComputer.powerOn();
      theComputer.boot();
      thePrinter.powerOn();
  GA3.End();
SA.End();
```

In this change schedule, encoded using "traditional" (nested) atomic actions, individual steps are implemented using actions nested within an top level action. The computer and printer's locks are maintained throughout the entire change schedule, and should the outermost transaction fail, the entire schedule is aborted. This is not a major problem in this small (and very simple) change schedule *but* could abort a lot of work in the case of large, complex schedules. For example, where the change schedule are executed over a long period of time and performs operations on *many* managed resources.

One method of rectifying these problems is by using "long lived transactions". These "enhanced" actions include:—[Wheater89a]

- Serialising Actions — which allow a system of actions to share objects while denying other actions access. These are particularly useful if the whole system of actions need not have failure atomicity (e.g. if it has been running for a long time and performed a lot of work).

- Top level Independent Actions — unrecoverable actions invoked outside other actions which are particularly useful for logging operations, garbage collection and caching.
- Common Actions — which allow systems of actions to share common atomic actions.
- Glued actions — which allow the transfer and release of locks between (and by) individual actions. Glued actions are particularly useful when a large number of objects are being manipulated by a system of actions: once computation has finished on an included object its lock can be released allowing other actions access.

(The interested reader is referred to Wheater's thesis for a fuller explanation of these actions and associated semantics.)

To encode the "device-connection" change schedule using long lived actions, the entire schedule is enclosed within a "Serialising Action" and constituent operations, within "Glued Actions". Change actions can therefore "partly commit" at each major stage in the schedule and share (release) locks on objects. Should these enhanced actions be included in later releases of the Arjuna software, the change schedule could be implemented as follows:—

```
RecPrinter          thePrinter( ... );
RecComputer         theComputer( ... );

SerialisingAction   SA;
GluedAction         GA1, GA2, GA3;

SA.Begin();

  GA1.Begin();

      // stage 1: power off / shut down

      GA1.lock(thePrinter, write);
      GA1.lock(theComputer, write);

      theComputer.shutDown();
      theComputer.powerOff();
      thePrinter.powerOff();

      // transfer locks to next glued action

      GA1.transferLock(theComputer, thePrinter);
  GA1.End();
  GA2.Begin();

      // stage 2: connect and device
      theComputer.connectDevice(theDevice);
      theComputer.installDevice(theDevice);
      GA3.transferLock(theComputer, thePrinter);

      // transfer locks ...
  GA2.End();
  GA3.Begin();

      // stage 3: power up resources ....
      theComputer.powerOn();
      theComputer.boot();
      thePrinter.powerOn();

      // release locks ...

      GA3.unlock(theComputer);
      GA3.unlock(thePrinter);

  GA3.End();
SA.End();
```

Device connection is performed using three stages: powering off resources, connection and finally powering the computer and printer. Each stage is performed inside a glued action (nested within a serialising action). Operation logging could be added to the schedule, using top level independent actions (which would not be undone in the event of failure) and complex schedules could share operations (using common actions). Glued actions are particularly suited to systems which manipulate a large number of managed resources, for example, when compiling and building software components.

Software distributions comprising many individual packages could be locked during compilation and building, and released when installed.

In the event of operations failing (causing glued actions to abort), intermediate computation is compensated to the end of the previously committed (glued) action. The change schedule can then be gracefully aborted by aborting the serialising action. As the serialing action carries *no* recovery information, operations already committed are preserved and all resources unlocked. Extra *application layer* recovery information can be maintained during the change schedule to record steps successfully performed and the state of locked objects. This information can be used to later restart or abort the schedule.

6.1.3 SOFTWARE UPDATE EXAMPLE

For the vast majority of software systems, software installation is an *ad hoc* process with limited automated support[Dean95a]. Software distributions may require a specific file system structure, certain system files and particular versions of other installed software. Although configuration (and installation) constraints may be expressed informally in software documentation, the first sign of installation problems are often when the newly installed software fails to execute.

The proposed standard for software installation has identified four stages in software installation:—[POSIX93a]

- Selection — file identification,
- Analysis — Verifying pre-requisites,
- Load — Copying files onto the host system,
- Configuration — Changing the software's environment.

It is possible to describe software environments using a configuration language (such as PCL) which records: hardware platforms, software structures, installation constraints and instructions. Configuration descriptions can create “makefiles” and generate “distfiles” which are executed by UNIX “make” and “rdist” utilities.

Within our prototype management information system, software distributions, like other (logical and physical) external resources are encoded as instances of (recoverable) managed resources. The `RecSoftware` class records information concerning the distribution’s host directory, file size and hardware (software) platforms etc. Software configurations and dependencies are encoded using “domains” and are graphically displayed as part of the management workbench. Software updates etc. are then performed using dependable change schedules.

We have not attempted formally to verify software compatibility with the underlying hardware architecture, co-processors, file systems or environments (which could be performed using an “expert advisor” etc.) but have concentrated on the reliability considerations concerning installing software distributions on a large number of resources. We assume that each software distribution is held in a single directory, compilation and installation rules encoded in a makefile (or imakefile) with configuration flags catering for specific hardware / software environments.

One “traditional” way of performing software up-date is using the “rdist” utility program which copies files between hosts, builds and installs programs etc. Software distributions and configurations are described using “distfiles” which encode a set of variables (such as host and file names), production rules and commands (such as installation and notification). Just as the “make” program can be distributed and constructed from long running actions[Wheater89a] it is possible to implement rdist using dependable change schedules. When installing a new software release, the following steps are required:—

```
(i)   Load distribution media into disk/tape/CD rom device
(ii)  Transfer distribution from media to memory
(iii) downgrade existing software
(iv)  unpack/uncompress distribution: apply ''diffs'' etc.
(v)   build distribution with appropriate configuration
(vi)  install software
```

When encoding "rdist" using dependable change schedules, the entire software update must be serialised and individual operations — loading software, building, installing etc. must be self contained. Should an individual operation within the schedule fail, it should be undone to the end of the previous step. Errors are then reported to the fault manager who can then decide either to undo the entire change schedule, or fix the bug and restart.

Our software installation change schedule is implemented using instances of `RecTapeUnit` and `RecSoftware` probes. `RecTapeUnit` is used to load the software distribution on to the host machine, and `RecSoftware` is then used to build (and install) software components across the distributed computer system. The change schedule is outlined below:—

```
BEGIN_SERIALISING_ACTION
      BEGIN_ACTION
            read from media
            unpack
      END_ACTION

      ON EACH MACHINE DO
      BEGIN
            BEGIN_GLUED_ACTION
                  Obtain Locks
                  Obtain Distribution
                  Transfer Locks to next Glued Action
            END_GLUED_ACTION
            BEGIN_GLUED_ACTION
                  Downgrade Old Software Release
                  Transfer Locks to next glued action
            END_GLUED_ACTION
            BEGIN_GLUED_ACTION
                  Compile and Link New release
                  Transfer Locks to Next Glued Action
            END_GLUED_ACTION
            BEGIN_GLUED_ACTION
                  Install new release
            END_GLUED_ACTION
      END
END_SERIALISING_ACTION
```

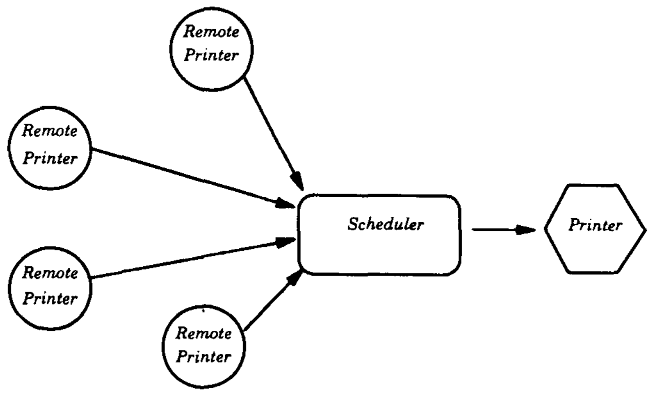
Software installation is potentially a time consuming process and therefore remote procedure calls to software probes (located on HP workstations) were set to maximum timeout and retry values. When installing new versions of software (upgrading as well as downgrading releases), all affected software components (on *all* hosts) are locked preventing inconsistencies *during* the update. On each machine, old software is downgraded, new software is compiled, built and then installed. Once software objects are installed on the host machine, the locks are released.

6.1.4 HARDWARE EXAMPLE

Composite hardware components, like software distributions are encoded using resource domains and are displayed using views on the management workbench. Examples include: workstations, comprising processors, disk and tape units, printers etc.; network concentrators, comprising "fan-outs" and multiple (terminal) connections; and line printing systems, comprising multiple printers attached to a server. Altering dependencies on the (hardware) resource domain's view causes reconfiguration of physical resources implemented using dependable change schedules.

In this example, we will consider a line-printing subsystem comprising the following components:—

Line Printer Subsystem



Distributed line printing subsystems can be compared to (passive) replicated objects, where print jobs are sent by (remote) printer spoolers to a (primary) spool queue dedicated to a physical printing device. With our print system, multiple (remote) printers "feed" the central spool queue from local schedulers. We can note the following dependencies:—

- Remote printers — each remote printer has one spool queue forwarding jobs to a scheduler.
- Scheduler — one local resilient scheduler with a spool queue sending jobs to a central printer.
- Local (Physical) Printer — has one spool queue.

In order to reconfigure the print system, the entire line printing subsystem must be quiescent. If the printer scheduler and local printer were migrated to another machine, *all* remote printers must obviously stop sending jobs to the initial machine and the local printer has to physically be moved to a second machine. Should the local printer fail, remote printing should be stopped to prevent a large back log of print jobs in the scheduler.

For example, replacing the local computer's printer requires shutting down the line printing subsystem while physical reconfiguration is performed. Hence, all jobs in the print queue must be completed (on the old printer), local and remote (printing) queuing is stopped and the host computer is powered off. The old printer's configuration also replaced on the local machine before restarting the print system. This is shown in the change schedule below:—

```
BEGIN_CHANGE_SCHEDULE
      STOP queuing ON remote machines
      STOP queuing ON local machine
      STOP printing ON local machine

      REMOVE Old Printer's Configuration FROM local machine

      SHUTDOWN computer
      POWER DOWN computer
      POWER DOWN Printer

      DISCONNECT Printer FROM HostMachine
      POWER UP computer
      BOOT computer
      CONNECT NewPrinter TO HostMachine
      CREATE Printcap FOR NewMachine

      POWER UP NewPrinter
      START queuing ON local machine
      START printing ON local machine
      START printing ON remote machines
END_CHANGE_SCHEDULE
```

This change schedule could be implemented using either long lived *or* traditional atomic transactions. The schedule comprises four stages:—

Stage	*Operation*
i)	Locking resources and (draining) stopping the queue
ii)	Removing the old device
iii)	Adding (and configuring) the new device
iv)	Resuming service

Each of these stages is implemented using a glued transaction within a serialising action. This schedule could also be implemented using traditional atomic actions, where the glued and serialising actions are replaced by nested transactions. However, should the nested action fail, the *entire* change schedule must be undone to ensure failure atomicity. Obviously when performing operations on the computer (printers) while the machines are powered off, *all* probes are migrated from the host and management operations are performed manually. The remote printers are encoded as a (passively) replicated group to simplify the code segment shown below:—

```
RecComputer        theComputer( ... );
RecPrinter         theRemotePrinters( ... ); // replica group
RecPrinter         theLocalPrinter( ... );
RecPrinter         theNewPrinter( ... );

SerialisingAction  SA;
GluedAction        GA1, GA2, GA3, GA4;

SA.BeginAction();
    GA1.BeginAction();
      // stage 1

      // lock remote and remote printers

      GA1.lock(theRemotePrinters, write);
      GA1.lock(theLocalPrinter, write);

      // stop queuing on all printers

      theRemotePrinters.stopQueuing();
      theLocalPrinter.stopQueuing();
      theLocalPrinter.stopPrinting();

      GA1.transferLock(theRemotePrinters, theLocalPrinter);
    GA1.EndAction();
    GA2.BeginAction();

      // stage 2: remove old device

      GA2.lock(theComputer, write);

      theComputer.removePrintCapEntry(theLocalPrinter);
      theComputer.shutDown();
      theComputer.powerDown();
      theLocalPrinter.powerDown();
      theComputer.removeDevice(theLocalPrinter);

      GA2.unLock(theLocalPrinter);
      GA2.transferLock(theRemotePrinters);
    GA2.EndAction();
    GA3.BeginAction();

      // stage 3: add new device
      GA3.lock(theNewPrinter, write);

      theComputer.addDevice(theNewPrinter);
      theComputer.powerUp();
      theComputer.boot();
      theComputer.addPrintCapEntry(theNewPrinter);
      GA3.unLock(theComputer);
      GA3.transferLock(theNewPrinter, theRemotePrinters);
    GA3.EndAction();
    GA4.BeginAction();
      // stage 4: start printing

      theNewPrinter.powerUp();
      theNewPrinter.startPrinting();
      theNewPrinter.startQueuing();
      theRemotePrinters.startQueuing();

      GA4.unlock(theNewPrinter, theRemotePrinters);
    GA4.EndAction();
SA.EndAction();
```

6.1.5 DISCUSSION

Over the last fifteen years we have seen a major evolution in the size and complexity of organisation's computing requirements with the increased availability of cheap, sophisticated resources. Distributed computer systems have *evolved* with new information technology which has led to large, complex computer systems with multiple dependencies between system components. Organisations are now "computer dependent" and therefore require continual computing service even when their distributed systems are being reconfigured.

In our management information system, we have applied ideas from configuration languages (PCL and Darwin), change schedules (CONIC) and long running actions (Wheater) to perform dynamic reconfiguration of managed resources. We have concentrated upon application layer components (computers, peripherals and software) and considered the reliability aspects of change management. Although the configuration language PCL (Univerisity of Lancaster et al) encodes dependencies between "families of resources" and provides "relationship definitions", their model does not provide operations on physical hardware resources. The language purely records system configurations and is particularly suited to software configuration management.

Changes to multiple (dependent) resources are provided using the change management view of the management information system in a similar way to the Architect's Assistant (Darwin) and PCL editor, and reconfiguration is performed using "dependable change schedules". Unlike change schedules used within CONIC, our schedules consider reconfiguring application layer entities rather than the distributed system probes. Maintaining structural integrity during reconfiguration is very important and ensures, for example, that only quiescent resources are manipulated. When considering application level resources, maintaining invariants between resources is also required. For example, powering off a computer *before* connecting peripheral devices. Dependencies between managed resources are encoded using "Change Management Views" of resource domains. These views are at present hard coded but ideally should be expressed in a configuration language. Changes to resource dependences would still be performed using a graphical interface but using a configuration language would provide greater flexibility. For example, a computer comprising a processor, disk units, screen and keyboard could be expressed using the following notation:—

```
TYPE  Computer     = { Processor x Disks x TTY x Keyboard x Mouse };
      Disks        = LIST OF DiskUnit;
      TTY          = { Terminal x ScreenCard x Connections };
      Keyboard     = { KeyPad x Cable x Connections };
      Mouse        = { MousePad x Cable x Connections };
```

Composite components are encoded using cartesian products ({}) and lists, and represented using "domains" in the user interface. It is also possible to describe software architectures using this notation. For example, a computer comprising processor, local and remote software could be encoded thus:—

```
TYPE  Platform     = { Computer x LocalSoftware x RemoteSoftware };
      LocalSoftware      = LIST OF Software

      // ...
      GNUsoftware  = SUBTYPE OF Software
                         InstallDirectory = "/usr/local/GNU/bin";
                         ManDirectory  = "/usr/local/GNU/man";
                         // etc.
                     END;

      G++          = SUBTYPE OF GNUsoftware
                         UNION { G++unix, G++pc, GNUmac };
                     END;
```

Reconfiguring multiple components could cause inconsistencies between dependent components. For example, "re-daisychaining" a disk unit's "SCSI" cable while a workstation is still powered could cause short circuiting etc. Operation invariants are therefore required between multiple resources.

```
INVARIANT
        Computer.powerDown()        => isComputerShutDown();
        Terminal.powerDown()        => TRUE;

        DiskUnit.powerDown()        => isComputerPoweredDown();

        // etc.

EMERGENCY-ACTION

        DiskUnit.powerDown()        => TRUE
```

Thus, computers must be “shutDown” before “powerDown” and disk units can only be powered off when the computer is powered off (except in emergencies, for example, if the disk unit is on fire).

In the current Arjuna implementation, each managed resource is probed by a (single) server process. This is not a major problem when reconfiguring a small number of resources (for example, installing a device on a computer). When reconfiguring a *large* number of resources, Arjuna activates one (heavy weight) server per resource and care must be take to avoid “hogging” the system! Should Arjuna provide a way for servers to “multiplex” their object state (i.e. one server could probe multiple resources) this problem would be solved.

Dependable change schedules are primarily designed for performing large, complex reconfiguration of managed resources. These change schedules provide “graceful shutdown” and failed operation clean up which is particularly suited to software reconfiguration. Although these schedules are at present encoded as C++ programs and implemented using nested atomic actions, these should ideally be performed using long lived actions. Similarly, it should be possible to encode schedules using a (high level) configuration language which would be exported from the change mangement views. We will consider the application of change schedules to fault isolation and recovery from failure in the next section which concerns Fault Management.

6.2 FAULT MANAGEMENT

With the many benefits associated with modern distributed computer systems, organisations have replaced old mainframe computers supporting many terminal connections with local (and wide area) networks, high performance workstations and shared equipment. These large distributed computer systems provide automation of once labour intensive information processing, communications and data banks, causing modern organisations to become highly dependent on their information technology. Equipment and software failure is not only annoying to system users but could cause financial loss, loss of credibility *and* catastrophic failures could grind an organisation to a halt.

Faults can affect both software, computer equipment and communication sub-systems. They can be intermittent or permanent and may require sophisticated diagnosis equipment, consultants and hardware engineers. Very large distributed computer systems can suffer from communications storms, viruses and electrical failure which are compounded by the complex relationships (and dependencies) between resources and administrative domains. According to the ISO/OSI[ISO88a],

```
Fault management is the set of facilities to:
a) maintain and examine error logs
b) accept and act upon error detection notifications
c) trace faults;
d) carry out sequences of diagnostic tests
e) correct faults.
```

Although the ISO/OSI provides a working definition of "fault management" in their taxonomy, this does *not* address the actual task of managing failed resources. Fault management comprises the following phases[Lee90a] :—

- Error detection — where the fault is detected and reported by the underlying management information system.

- Damage confinement and assessment — where the fault is isolated preventing a knock-on effect of error propagation,
- Error recovery — reconfiguration after the initial failure.
- Fault treatment and continued service.

Within the management information system, faults can occur at three layers:—

- Application Layer — this includes external equipment and management operations failing.
- Distribution Layer — including probes and even the Arjuna tool kit failing.
- Communications Layer — i.e. servers become unreachable due to network congestion, partition etc.

We will consider each of these application layer failures and apply change schedules to fault isolation (recovery) in subsequent sections.

6.2.1 APPLICATION LAYER FAULTS

Agents adopting the role "fault manager" in the management information system can examine error logs and perform fault diagnosis etc. using the fault manager's view of managed resources and resource dependencies. Users can also adopt the role "fault reporter" and send reports via the workbench to fault managers responsible for particular resources. The "fault report view" is based upon Athena's "hotline" report and comprises check (and radio) buttons to recode specific faults and text for users to write notes. Intermittent faults *may* be detected by performance managers "watching" managed resources. For example, if the distributed computer system was being attacked by a virus, some of the symptoms might include very high system loading, high levels of network connections and a general degradation in response times. Once unusual behaviour has been detected, the performance manager can forward details to the fault manager responsible for affected resources.

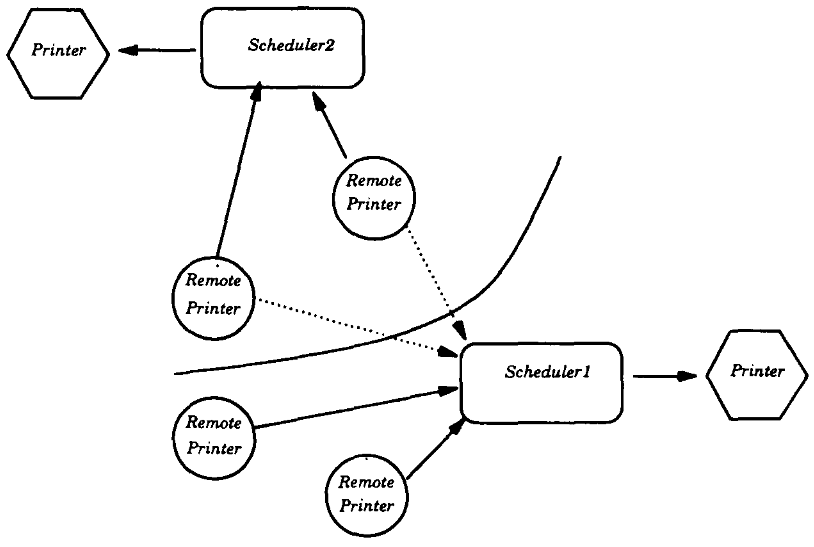
When physical equipment has been reported as faultly, the resource is marked out of order and users (and other agents) are denied access until the fault manager has investigated the problem. This is implemented simply by entering the resource in an "expected exception" table in the management workbench. (Resources listed in this table can only be accessed by change and fault managers.) Arrangements can then be made to call out an engineer, order replacement parts etc. Single resources such as "flickering" terminals and "smoking" laser printers can then be repaired and normal service resumed. Failures which involve multiple connected resource dependencies or where major reconfiguration is required are confined (isolated and reconnected) using dependable change schedules (which we have discussed earlier in this chapter.) For example, if the communications sub-system had been severed, change schedules could be used to isolate resources on each partitioned segment by reconfiguring network file systems and distributed printing servers.

6.2.2 FAULT MANAGEMENT USING CHANGE SCHEDULES

Dependable change schedules can be used to provide application layer fault isolation and repair in the same way as they are used to implement change management actions. Change schedules can create (delete) servers on processing nodes, link resources to communication ports in addition to applying control operations on application layer resources. For example, isolating remote printers from a faulty print spooler; erecting a network fire wall and "de-virusing" host computers and cleanly separating upstream and downstream partitioned network segments. Let us suppose that a network partition occurred between the remote printers and the print service which we considered earlier as a hardware change management example.

Remote printers would be unable to forward jobs to the (unreachable) spooler and any attempt (remotely) to access the spooler would result in communication timeouts etc. Furthermore, any (orphaned) remote computation performed during the partition would be destroyed by the remote procedure call mechanism. Therefore "partitioned" remote printers are isolated from the spooler and print jobs diverted to an alternative spooler. This is shown in the diagram below:—

(Partitioned) Line Printer Subsystem *Partitioned Resources*



When cleanly isolating the downstream printers, any jobs active on the remote printers are suspended and queuing disabled. All affected remote printers are then reconfigured to send jobs to an alternative service. Unaffected printers upstream of the partition do not require further action. This is shown in the change schedule below:—

```
BEGIN_CHANGE_SCHEDULE

      FOR ALL Remote Printers DO
      BEGIN
            Lock Computer, remote printer

            STOP queuing ON remote printer
            ABORT printing ON remote printer
            STOP  printing ON remote printer

            REPLACE printcap ON computer

            START queuing ON remote printer
            START printing ON remote printer
            Unlock Computer, remote printer
      END

END_CHANGE_SCHEDULE
```

6.2.3 CHANGE SCHEDULE RECOVERY

We have already illustrated using change schedules to implement major reconfiguration within the management information system. We noted that when performing major changes (for example, software reconfiguration), "graceful shutdown" is preferable to the "full abort" semantics of traditional atomic transactions and therefore avoids loosing a large amount of computation. When change schedules gracefully abort, partially completed stages in the change schedule are compensated up to the last committed (glued) action, locks are released and the schedule exits. By recording the last completed stage, it is possible to correct the failed section and restart the schedule. — Locks etc. are re-established and execution commences from the last completed stage. (Providing invariants for the particular stage are still maintained!)

In order to illustrate the failure and restart semantics of dependable change schedules, the software installation schedule (discussed earlier, within change management) was rebuilt with ("faulty") recoverable software servers. The software distribution used to performance test the `makeCommand` and `RecSoftware` classes was installed on ten HP workstations using the dependable change schedule (implemented using traditional atomic actions) is *outlined* below:—

```
// BeginChangeSchedule

        RecSoftware  theOldSoftware( ... );
        RecSoftware  theNewSoftware( ... );
        RecTapeUnit  theTapeUnit( ... );

        AtomicAction A0, A1, A2, A3, A4;

        // point 1

        BEGIN_ACTION(A0)
              WRITE_LOCK(A0)
              theTapeUnit.loadTape();
              theTapeUnit.restoreFromTape();
              theTapeUnit.ejectTape();
        END_ACTION(A0)

        // point 2

        for i in HostNames
        {
        BEGIN_ACTION(A1)
              // point 3
              BEGIN_ACTION(A2)
                    WRITE_LOCK(A2)
                    theOldSoftware.downGrade();
              COMMIT_ACTION(A2)
              // point 4
              BEGIN_ACTION(A3)
                    WRITE_LOCK(A3)
                    theNewSoftware.makeAll();
                    theNewSoftware.makeInstall();
                    theNewSoftware.upGrade();
              COMMIT_ACTION(A3)
        COMMIT_ACTION(A1)
        }

        // point 5
// EndChangeSchedule
```

The macros BEGIN_ACTION, COMMIT_ACTION and ABORT_ACTION have been used to hide Arjuna specifics and provide graceful shutdown points.

Let us consider the following scenarios:—

- If the software distribution cannot be loaded from the tape, the change schedule *has* (for obvious reasons) to abort.
- If the old software distribution cannot be downgraded on a particular host, actions A2 and A1 abort. Software installation therefore cannot proceed on the host and "Software Installation failure", host's identifier

etc. are recorded on the operation log. Software installation on the next host is attempted.

- If the new software distribution cannot be built (installed or upgraded), action A3 is aborted (and compensated) and A2 is undone using an Anti-operation (i.e. the old software is upgraded). Action A1 can be committed to preserve the old software's state.

Should the schedule be gracefully shutdown and any faults corrected (for example, replacing the tape), execution can be resumed. Either the schedule can be restarted (in the case of bad tapes) or installation re-attempted on previously failed hosts.

If the change schedule terminated abruptly (i.e. forced shutdown rather than graceful shutdown) *application* and *environmental* crash recovery would be performed on the change schedule. This requires restoring the object's state from stable storage, resolving blocked actions *as well* as ensuring the schedule's integrity. For example, compensating failed stages in the change schedule, performing anti-operations etc. Once crash management / recovery has been performed, the change schedule can be restarted. (Arjuna crash management is discussed in the next section.)

6.2.4 ARJUNA FAULT MANAGEMENT AND CRASH RECOVERY

Arjuna models distributed applications as a set of clients and servers connected by a communications sub-system. Servers and clients are located on ("fail silent") nodes in the distributed system. I.e. nodes fail silently and die *without* producing arbitrary results (Byzantine failures)[Lamport82a]. The current Arjuna implementation only considers processing node failure and does *not* provide recovery tools for failed server processes. Communications connectivity is monitored using a "ping daemon" and orphan detection (and killing) is provided in the remote procedure call protocol. (Passive) Replica object groups are maintained using a "GroupView" database[Little91a] which records process-node bindings, object usage etc.

Arjuna's crash recovery manager (`CrashMan`) is initiated at boot time and periodically records recovery information regarding the state of servers located on a particular node. Two crash recovery phases are provided:—

- *Phase 1 Recovery* — while the host is quiescent (i.e. clients are prevented from accessing servers), CrashMan's "work list" is collated. Any partially committed atomic actions and uncommitted server states are resolved and the replication database told about the host's failure.
- *Phase 2 Recovery* — concerns detecting the local effects of remote host crashes such as resolving orphaned atomic actions.

Once Arjuna's crash management system has declared the host safe, each management information system probe (i.e. server) is then reconfigured and obtains its state from its physical (external) resource. For example, the `RecCoverable` computer's probe obtains the computer's virtual memory usage and process load statistics. This is simply implemented using a shell script which executes `faultManagement` client processes.

6.2.5 DISCUSSION

Fault management is closely related to configuration and change management and addresses unscheduled reconfiguration of resources, the distributed processing environment and communication subsystems. Fault management is particularly complex due to dependencies between resources and diagnosis (repairing) physical equipment. Although our management information system may be able to detect some intermittent faults and isolate (reconnect) resources, many hardware faults can only be repaired *manually* by an experienced maintenance engineer.

Fault management "clean up" actions have been formally specified (in Z)[Spivey89a] by Young and used to reconfigure the CONIC system. Young's requirements for a fault management system state that:—[Young91a]

- Clean up should not require external intervention.
- Repair strategies should be declarative and flexible.
- Application contribution should be independent of the failure (and repair).
- Disruption to the rest of the distributed system should be minimised, and the whole system should *eventually* recover!
- The system should cope with single (multiple) failures.
- There should be minimal overhead imposed by the fault management system.

Application layer recovery (i.e. managed resources and change schedules) in our management information system is abstracted away from the distributed processing environment. Resolving partially committed atomic actions (using a co-operative termination protocol etc.) and removing shadow state data from object stores is performed by the Arjuna toolkit. Unfortunately Arjuna only provides host-level crash management and therefore, in the event of servers crashing etc. shared memory, semaphores etc. must be cleaned up *manually*. Young's "abort-in" and "abort-out" sets provide a formal basis for resolving blocked atomic actions, which is performed during the first phase of the Arjuna crash manager.

While it is possible formally to specify the structure of distributed components and describe repair strategies, providing "functional integrity" of (application level) fault management is very complex. Physical equipment faults can be manifested by many symptoms and may require manual intervention. An expert system ("expert advisor") could be used to identify multiple faults in the distributed computer system and integrated in the workbench.

Isolating (and reconnecting) failed components can be implemented using dependable change schedules. These could be generated from the fault manager's view of resource domains. For example, the fault manager could identify resources by clicking on particular icons, a change schedule created and enacted on physical resources. Multiple dependencies are recorded using resource domains and invariants maintained.

6.3 SUMMARY

Dependable change schedules were applied to both change and fault management and integrated into the workstation's view based interface. Change schedules which are implemented using long running actions are particularly suited to *major* reconfiguration where "graceful" shutdown (rather than complete abort) is required in the event of failure. Glued actions are particularly useful in providing recovery points in the change schedule and allow the transfer (and release of) object's locks between subsequent actions. Software updates and device installation were used to illustrate change schedules. The Software update schedule was implemented using traditional atomic actions using fault injection and demonstrate the schedule's behaviour in the presence of faults.

Chapter 7

CONCLUSIONS

I believe everything, I believe nothing
I suspect everyone, I suspect no one
I gather the facts, examine the clues
and before I know it, the case is solved *

This thesis concerns the role of dependability and enterprise modelling in managing a very large distributed computer system. We developed a person centred model of management which was used in the design of a prototype management information system and used fault tolerance techniques to provide error recovery and compensation when performing operations upon managed resources. Operations upon physical (logical) were performed using existing "legacy systems" such as system entry routines, system software, network services etc. which were incorporated using "integration objects". Once our prototype system was performance evaluated and tested, we presented a method of performing *major* reconfiguration using "dependable change schedules" which we applied to both change and fault management.

In Chapter 1, we considered management (in general), with the aid of "system's theory" and Kotter's framework, managing a computer intensive organisation and the practical aspects of "day to day" system administration. We noted that fault tolerance techniques *must* be incorporated in the design and implementation of management information systems to ensure the dependability of control (monitoring) actions upon logical (physical) resources. This includes failure atomicity and serialisation of management operations, error recovery etc.

* Jacques Clouseau on the Science of Criminology. (From the Peter Seller's film "A Shot in the Dark".)

In Chapter 2, we considered managing large, complex computer systems. This included the role of international standards within system management, in particular, their management taxonomy and OSI "directory". Although the ISO/OSI model describes and information exchange protocol and object store, their taxonomy fails to address the *actual* task of managing a large computer system. We then considered several research initiatives and prototype management information systems such as Project Athena, Digital's EMA, OSF, TOBIAS and The University of Lancaster's "Netman" and "DSM" projects. Even though these research projects have contributed to (automated) computer system management, we noted that several key areas were overlooked:—

- Providing a "natural" model of management,
- Providing fault tolerant management operations upon resources and automatic "clean up" in the event of failure,
- Providing a scheme for performing major reconfiguration and correcting major faults.

Chapter 3 concentrated on the design of our management information system. Our model of management was based on the idea of "agents" adopting "roles" in order to "view" managed resources and regulating interacts using "contracts". This model was based upon ANSA's Enterprise Projection and included ideas from the ESPRIT funded ORDIT and TOBIAS projects. Contracts "bind" agents to roles, record responsibilities, job descriptions, conditions of work etc. and are used to record job allocation, resource allocation and trading arrangements between organisational "domains". We agree with Moffett that high level organisational policies can be broken down into "policy hierarchies" and (eventually) to allocating jobs etc. (or indeed, deciding to sub-contract). Rather than allocating jobs, resources or sub-contracting arrangements using a notation similar to ANSA's "structural roles", we have used contracts.

Our prototype management information system was initially implemented (Chapter 4) on a single Sun work station and comprised a system management workbench (written in Tcl/Tk) and managed resources (represented using C++ objects). Management operations are performed using "probes" which were decomposed into "recoverable", "unrecoverable" and "integration" layers. Recoverability was incorporated using inheritance, which cleanly (and simply) abstracted unrecoverable multiple operation implementations from higher layers of the prototype and "automatic" error recovery, using recovery blocks (encoded using atomic-actions). Incorporating "legacy systems" using "integration objects" proved very simple to implement and encapsulated operating system (and software) specific features. Our prototype system could be easily extended to include extra managed resources and even other operating system platforms.

The prototype's performance was evaluated and testing in Chapter 5. We noted that although using Arjuna considerably slowed the prototype system down (due to bottlenecks in their object store), management operations are still performed in "real time" and indeed, the user should not notice any severe delays. Faults were injected into the management information system in order to demonstrate the prototype's behaviour in the presence of "errors". Unfortunately, the current release of Arjuna does not *fully* support crash simulation and only host level crash recovery is supported.

Large (distributed) computer systems are rarely static and due to the organisation's dependence upon information technology, changes to their system's behaviour (structure) *must* be performed while the remainder of their resources are *active*. We therefore considered (in chapter 6) a mechanism for managing change. Changes to the distributed system were implemented using “views” and encoded using “schedules”. Change schedules were based upon ideas from Imperial College and comprise “systems of atomic actions”. We noted that although it is possible to encode change schedules using traditional (nested) atomic actions, long running actions provide were particularly suited to *major* reconfiguration where *graceful* shutdown (rather than complete abort) is required in the event of failure. We illustrated the behaviour of change schedules in the “presence” of faults using fault injection and described a method of restarting failed schedules.

7.1 THESIS CONTRIBUTION

Very large distributed computer systems have revolutionised organisation’s information systems. These systems are here to stay and have only been made possible by the advent of cheap, very powerful equipment and reliable (high-bandwidth) data communications. It is therefore very important that such systems are *effectively* managed to avoid organisation’s grinding to a halt. Organisation wide computer systems must be considered in the context of the organisation’s policies, aims and objectives *as well* as the organisation’s social system, formal procedures, environment etc.

The major contribution of this thesis concerns the application of dependability and enterprise modelling techniques to managing very large distributed computer systems. The main points can be summarised below:—

- *Person-Centred Model of Management* — based upon “agents”, “roles”, “views”, “resources” and “contracts”. This model uses “natural” concepts and is based upon (and extends) previous work. Our model forms the basis for our prototype management information system, where “views” form the user interface, contracts regulate agent’s activity, and the organisation is represented in terms of "domains".

- *Fault-Tolerant Management Operations upon External Resources* — Management operations are encoded using recovery blocks and implemented using atomic actions. In the event of operations failing, any effects upon the physical resource are compensated and operation based recovery performed. Using "inherited recovery" techniques proved clean, simple and extendible.
- *Encoding Major Changes using "Schedules"* — atomic actions were incorporated into Imperial College's "change schedules" and linked to the prototype's "view" based interface. We demonstrated a method of "gracefully shutting down" and restarting change schedules used with application level crash recovery techniques.
- *Incorporating "legacy systems"* — Wrapping system software, network services, configuration files etc. using "integration objects" provides a clean, simple and easily maintainable method of enacting operations upon physical (logical) resources.

The prototype has demonstrated that it is possible to combine enterprise modelling and fault tolerance techniques when constructing a management information system. We further extrapolate that it possible to use systems of long running actions within configuration programming techniques, thus extending CONIC's work within change schedules.

7.2 EVALUATION OF RESEARCH

7.2.1 DESIGN

When designing and implementing a management information system, it is very important to consider the role of people within the organisation and their interactions with managed resources. Even though systems such as HEMS and SNMP have made a significant contribution to network management, their model or prototype does not consider the actual system users or network managers. Their prototype system does not regulate resource access or provide fault tolerant operations in real time. Within system management, Athena only considers centralised configuration management with "medium diversity" of computing resources. [Champine91a] Some fault tolerance and replication is provided but other management areas are not considered in their model.

Our management model uses natural concepts such as agents, roles, contracts and resources. We have not attempted formally to define or analyse conflicts of policy (or interest) but encode policy etc. using contracts. Contracts used in Chapter 3 are based on ANSA/ORDIT's structural and functional roles and are therefore more complex than TOBIAS's implementation. Using contracts to regulate interactions between agents and resources invites authentication: Contracts are issued by the contract manager on behalf of the organisation and accepted by the human agent. Incorporating an authentication system such as Kerberos etc. would therefore be a great benefit.

Domains [Sloman87a] have been incorporated into the model, but as we discovered when considering a large complex organisation (such as a University) domain relationships can get very complicated. Domains are, of course, scalable, but complex domain relationships (like multiply overlapping sets) become difficult to display. We have not attempted formally to define if contracts have been fulfilled by contractors (or accepted by clients) or if a set of sub-contracts meets (fulfills) a trading arrangement. (This is proposed as an area of future research.)

We designed a management information system using our management model. The management information system comprises: workbenches, which forms a view based user interface; managed resources which represent logical (physical) entities; and organisational information such as contracts etc. Although the prototype's structure is very similar to TOBIAS and Dec's DME, explicit application layer fault tolerance and recovery were incorporated in the prototype's design.

7.2.2 PROTOTYPE SYSTEM

Whilst the management information system has a simple, layered structure, one of its greatest drawbacks concerns using Arjuna's object store to hold persistent object states. Not only does this bottleneck severely affect the prototype's performance, but object multiplexing is not supported. Thus a 1 M.byte server is dedicated to a 400 byte object state!

Objects in Arjuna are instances of servers and are "heavy weight" processes. This is not a major problem when only considering host computers with a few peripherals but incorporating user management with a large number of user contracts requires a database. These problems can be solved, as demonstrated by Buzato's database system which was implemented using Arjuna (and uses object multiplexing)[Buzato92a].

Management operations are applied to physical resources using the integration layer and implemented in a similar way to Sventek's legacy objects. This provides a clean, simple and extensible mechanism for incorporating system utilities, network services, system entry functions etc. This integration layer in our prototype is therefore different to TOBIAS' integration layer. TOBIAS's integration layer is similar to "stub objects" in remote procedure calls) and operations on physical resources are performed by object controllers (the SMOC) TOBIAS also uses wrappers around utility programs and shell scripts but does not represent them explicitly as individual objects. This can create portability problems.

Armed with a large collection of integration objects, both the unrecoverable and recoverable layers proved very simple to implement. The recoverable layer, in particular, comprised largely recovery blocks and atomic transactions. This therefore separates the reliability mechanisms from the task of performing management operations. Although Arjuna requires extra methods in recoverable objects to save (restore) object states and provide a type name, this is not a major overhead. Atomic action code was masked using macros and recovery blocks which simplified implementation.

7.2.3 APPLICATION

Once initially tested on a single workstation, our prototype was used to manage a very large organisation wide computer system (Newcastle University). A view based interface was constructed using Tcl/Tk which communicates with probes on managed resources. The workbench could be extended to provide more functionality — for example, moving resources between domains, adding extra icons etc. but nevertheless the workbench demonstrated the suitability of this approach.

Porting the management information system between operating system releases proved relatively painless due to the integration layer's structure and adhering to POSIX standards. When applying the workstation to physical resources, portability is very important (and avoids major changes to integration models). Should POSIX and OSF etc. produce system entry routines for disk and other peripherals, these can easily be incorporated in the integration layer.

While our fault injection scheme demonstrated random faults in the management information systems, we could only show *application* level faults. In order to test the management information system properly, both communications and Arjuna layer faults should be injected. With the appropriate crash simulation software, we could demonstrate network partition, black holes, object store problems etc.

We have suggested that *major* reconfiguration can be implemented using systems comprising long running actions. This is particularly suited to software building and installation on a large number of machines etc. which not only could take a long time to implement *but* require graceful shutdown and restart semantics should the operations fail. Change schedules can be implemented using traditional atomic actions but carry the overheads of aborting entire schedules in the event of failure and retaining locks on large numbers of objects. Should long running actions be implemented in future Arjuna releases, it will be possible to fully demonstrate our techniques.

Performance testing "long running dependable change schedules" is dependent upon the actual underlying distributed processing environment. When implementing change schedules it is therefore very important that managed objects be active for a minimum time to avoid "swamping" the host computer with large number of servers. This could be solved by allowing multiplexing managed objects and should be incorporated in the Arjuna tool kit.

7.3 FUTURE WORK

Several areas of future research have been proposed:—

- *Model of Management* — Contracts can be applied to user management, allowing users to access resources; accounting and security management. Similarly, policy statements can be implemented using contract objects and therefore it should be possible to incorporate Moffett's work on Conflicts of Policy (Interest) in the model of management.

- *Prototype System* — Extra managed resources could be incorporated in the prototype — such as bridges, routers and cd-rom devices; and integrated in the same way as other hardware components. Similarly integrating workstations running operating systems other than UNIX could be investigated. Replication of managed resource probes could be investigated particularly considering the many to one relationship between probes and resources — active [Bernstein87a] and passive [Alsberg76a] replication techniques could be compared/contrasted. Similarly, the use of forward error recovery could be examined as a contrast to backward error recovery used in the prototype. Heuristic algorithms [Kar88a] could be used to display complex network structures and organisational domain relationships.
- *Configuration Languages* — Directly implementing change schedules using C++ programs is too low level for system managers. A better approach concerns using high level configuration languages (and even expert systems etc.) to describe reconfiguration actions which could be translated into our schedules. A similar approach has been adopted in the "System Architect's Assistant" where "Darwin" descriptions can be output as C++ programs; and within PCL, configuration details can be exported as makefiles.
- *Expert Advisors* — Using dependable change schedules could be complimented by "expert advisers" as originally proposed in the TOBIAS project. [TOBIAS89a] These "expert advisers" could assist in change schedule construction and configuration management[McDermott84a]. In particular checking consistency between components and aiding fault

diagnosis. Similarly, research on disaster prevention [Danish94a] could be incorporated within “Safety” and “Environment” management roles.

APPENDICES

Appendix A INTEGRATION LAYER

This appendix presents a *selection* of the management information system's (40 or so) integration modules in the form of UNIX style manual pages. While this is not by any means a complete list of integration modules, sufficient information is provided to appreciate higher levels of the prototype system. We will first consider devices, computers and manual operations.

A.1 DEVICE INTEGRATION

A.1.1 DiskController

DESCRIPTION

`DiskController` obtains configuration data from the disk unit's controller. These include: the disk's interface, unit and controller address and configuration flags. The disk controller is initialised with the particular raw device name (e.g. /dev/rst0a), configuration obtained (using ioctl's) which is accessed by other methods.

CLASS DEFINITION

```
class DiskController
{
public:

    DiskController(char *);       //device name
    ~DiskController();

    Error   getController();
    Error   getControllerAddress(int &);
    Error   getDiskInterface(short &);
    Error   getUnitAddress(short &);
    Error   getUnitFlags(short &);

protected:

    // instance data ...
};
```

INTERFACE

DiskController(char * itsDeviceName) Constructs the Disk Controller using the (RAW) devices path name

~DiskController() destructs the class

getController() opens the device, retrieves `dk_info` using an ioctl and sets appropriate instance data which is later retrieved by other methods

getControllerAddress(int & itsAddress) getDiskInterface(short & itsInterface) getUnitAddress(short & itsAddress) getUnitFlags(short & itsFlags) obtains instance data.

BUGS

None known

NOTES

i) Opening disk devices requires "super-user" access.

ii) Two implementations of `getController` exist catering for differences between SunOS and Solaris "dkio".

SEE ALSO

DiskPartititions, DiskGeometry, SCSIfloppy, TapeController

A.1.2 DiskParititions

DESCRIPTION

`DiskPartitions` obtains the disk partition map for a particular (raw) device identifier. These include the partition's start address and the number of disk clocks in the partition.

CLASS DEFINITION

```
class DiskPartitions
{
public:

    DiskPartitions(char *);        // device name
    ~DiskPartitions();

    Error   getDiskPartitions();
    Error   getStartingCylinder(long &);
    Error   getNumberOfBlocks(long &);

protected:

    // instance data ...

};
```

INTERFACE

DiskPartitions(char * itsdeviceName) Constructs the disk partitions class with the raw devices name.

~DiskPartitions() Destructs the Disk Partition Class

getDiskPartitions() opens the device, retrieves the `dk_map` using an ioctl call and sets class instance data

getStartingCylinder(long & itsStart) sets itsStart to the disk's start address

getNumberOfBlocks(long & itsNumberOfBlocks) sets itsNumberOfBlocks to the total number of disk blocks in the partition.

BUGS

none known

NOTES

i) Opening disk devices requires "super-user" access.

ii) Two implementations of `getDiskPartitions` exist catering for differences between SunOS and Solaris `dkio`.

SEE ALSO

DiskController, DiskGeometry, SCSIfloppy, TapeController

A.1.3 DiskGeometry

DESCRIPTION

`DiskGeometry` obtains the disk device's physical geometry. These include: the number of cylinders, sectors per track and interleave factor.

CLASS DEFINITION

```
class DiskGeometry
{
public:

    DiskGeometry(char *);        // device name
    ~DiskGeometry();

    Error   getDiskGeometry();
    Error   getCylinders(unsigned short &);
    Error   getAltCylinders(unsigned short &);
    Error   getHeads(unsigned short &);
    Error   getHeadOffset(unsigned short &);
    Error   getSectorsPerTrack(unsigned short &);
    Error   getInterleave(unsigned short &);

protected:

    // instance data ...

};
```

INTERFACE

DiskGeometry(char * itsDeviceName) constructs the class with the raw device identifier

~DiskGeometry() destructs the class

getDiskGeometry() opens the device, retrieves `dk_geom` using an ioctl and sets class instance data.

getCylinders(unsigned short & itsCylinders) getAltCylinders(unsigned short & itsAltCylinders) getHeads(unsigned short & itsHeads) getHeadOffset(unsigned short & itsHeadOffset) getInterleave(unsigned short & itsInterleave) obtain appropriate instance data.

BUGS

none known

NOTES

i) Opening disk devices requires "super-user" access.

ii) Two implementations of `getDiskGeometry` exist catering for differences between SunOS and Solaris `dkio`.

SEE ALSO

DiskController, TapeController, DiskPartitions, SCSIfloppy

A.1.4 SCSIfloppy

DESCRIPTION

`SCSIfloppy` obtains the floppy disk unit's configuration corresponding to the device name.

CLASS DEFINITION

```
class SCSIfloppy
{
public:

    SCSIfloppy(char *);  //device name
    ~SCSIfloppy();

    Error   getSCSIfloppy();
    Error   getTransferRate(int &);
    Error   getNumberOfCylinders(int &);
    Error   getNumberOfHeads(int &);
    Error   getSectorSize(int &);
    Error   getSectorsPerTrack(int &);
    Error   getStepsPerTrack(int &);

    Error   ejectDisk();

protected:

    // instance data ...
};
```

INTERFACE

SCSIfloppy(char * itsDeviceName) constructs the class with the floppy unit's device name

~SCSIfloppy() class destructor

getSCSIfloppy() opens the device name, performs an ioctl to obtain `fdsk_char` and sets instance data.

getTransferRate(int & itsTR) getNumberOfCylinders(int & itsNC) getNumberOf-Heads(int & itsNH) getSectorSize(int & itsSS) getSectorsPerTrack(int & itsSPT) getStepsPerTrack(int & itsSPT)

ejectDisk() ejects the floppy disk using `FDKEJECT`

BUGS

none known

NOTES

i) Opening disk devices requires "super-user" access.

ii) Two implementations of `getSCSIfloppy` exist catering for differences between SunOS and Solaris `dkio`.

SEE ALSO

DiskController, DiskPartitions, DiskGeometry

A.1.5 TapeController

DESCRIPTION

`TapeController` obtains configuration information and performs control operations on a tape device. This class is based on (and reimplements) the `mtio` system utility program available on the UNIX operating system.

CLASS DEFINITION

```
class TapeController
{
public:

    TapeController(char *);        // device name
    ~TapeController();

    Error   getTapeController();

    Error   eraseTape();
    Error   rewindTape();
    Error   ejectTape();
    Error   retensionTape();

    Error   showUnitKind(int &);      // theUnitKind
    Error   showFlags(u_short &);         // theFlags
    Error   showOptBlockFactor(int &);    // theOptBlockFactor


protected:

    // instance data ...
};
```

INTERFACE

TapeController(char * itsDeviceName) constructs the tape controller's class and sets theDeviceName.

~TapeController() destructs the class

getTapeController() opens the tape device, retrieves `mt_get` using an ioctl call and sets class instance data.

eraseTape() erases the tape using `MTERASE`

rewindTape() rewinds, using `MTREWIND`

ejectTape() ejects, using `MTEJECT`

retensionTape() retensions, using `MRETEN`

showUnitKind(int & itsUnit) showFlags(u_short & itsFlags) showOptBlockFactor(int & itsOptBf) retrieves instance data

BUGS

None known

NOTES

i) Operations on the magnetic tape require super-user access to the raw device name.

ii) All operations on the magnetic tape require a tape loaded into the tape-unit (i.e. pre-condition).

SEE ALSO

SCSIfloppy, DiskController etc.

A.1.6 TerminalController

DESCRIPTION

The `TerminalController` class is based upon and incorporates the POSIX `termios` system entry functions. Termios is very different from traditional terminal handling functions and in order to set (get) terminal characteristics a large data structure is manipulated.

CLASS DEFINITION

```
class TerminalController
{
public:

    TerminalController(char *);
    ~TerminalController();

    virtual Error    getTerminalController();
    virtual Error    getWindowSize();
    virtual Error    getLineDiscipline();

    virtual Error    showWindowSize(int &, int &);
    virtual Error    showSpeeds(int &, int &);
    virtual Error    showLineDiscipline(int &);
    virtual Error    showParity(int &);

protected:

    // instance data ...
};
```

INTERFACE

TerminalController(char * itsDeviceName) constructs the class with the terminal device's pathname.

~TerminalController() destructs the class **getTerminalController()** obtains the terminal's `termios` data using an ioctl. The structure is then decomposed the reveal line speed, partity setting, protocols etc.

getLineDiscipline() obtains the terminal's line discipline using `TIOCGETD`

getWindowSize() obtains the terminal's window size using `TIOCGWINSZ`

showWindowSize(int & itsRows, int & itsCols) showSpeeds(int & itsIn, int & itsOut) showLineDiscipline(int & itsLine) showParity(int & itsParity) retrieves instance data

BUGS

This class did not work on *some* releases of SunOS (possibly due to an operating system bug). TerminalController worked correctly on both SunOS 4.1.1 and 4.1.3.

NOTES

i) Opening terminal devices requires super-user access.

SEE ALSO

DiskController, SCSIfloppy etc.

A.1.7 PrinterStats

DESCRIPTION

`PrinterStats` reimplements a simplified version of the UNIX printer accounting program and obtains the total volume and number of jobs printed.

CLASS DEFINITION

```
class PrinterStats
{
public:

    PrinterStats(char *);          // printer name
    ~PrinterStats();

    Error   getStatistics();

    Error   showNumberOfJobs(int &);
    Error   showVolumePrinted(float &);   // feet or pages
    Error   showCost(float &);

protected:

    // instance data ...
};
```

INTERFACE

PrinterStats(char * itsPrinter) constructs the class with the printer's name

~PrinterStats() destructs the class

getStatistics() reads each entry in the printer's accounting file and totals the number and volume of print jobs.

showNumberOfJobs(int & itsJobs) showVolumePrinted(float & itsVol) show-Cost(float & itsCost) retrieves instance data

BUGS

none known

NOTES

In order to read entries from the accounting file, the printer must have accounting enabled!

SEE ALSO

PrintCapEntry

A.1.8 PrintCapEntry

DESCRIPTION

`PrintCapEntry` obtains configuration information concerning a particular printer device form the `printcap` file. These include the printer's baud rate, filters, page setting, account file and spool directory. These values are acquired using the `termcap` library functions `tgetent`, which gets a termcap entry; and `tgetstr` which gets string parameters.

CLASS DEFINITION

```
class PrintCapEntry
{
public:

    PrintCapEntry(char *);
    ~PrintCapEntry();

    Error    readEntry();
    Error    writeEntry(char *);       // printcap file
    Error    getBaudRate(unsigned &);
    Error    getPageDims(unsigned &, unsigned &); // length, width
    Error    getMaxCopies(unsigned &);
    Error    getMaxSize(unsigned &);
    Error    getSpoolDir(char *);
    Error    getLogFile(char *);

    Error    setBaudRate(unsigned)
    Error    setPageDims(unsigned, unsigned);
    Error    setMaxCopies(unsigned);
    Error    setMaxSize(unsigned);
    Error    setSpoolDir(char *);
    Error    setLogFile(char *);

protected:

    // instance data ...
};
```

INTERFACE

PrintCapEntry(char * itsPrinterName) constructs the class and sets default instance data. These defaults are listed in the UNIX manual page for /etc/printcap

~PrintCapEntry() destructs the class

readEntry() reads and decomposes the printer's printcap entry preserving default values for incomplete printcap fields.

writeEntry(char * itsFileName) writes an entry to the file name specified

getBaudRate(unsigned & itsBaud) getPageDims(unsigned & itsLen, unsigned & itsWid) getMaxCopies(unsigned & itsMax) getMaxSize(unsigned & itsMax) getSpoolDir(char * itsSD) getLogFile(char * itsLF) retrieves instance data

setBaudRate(unsigned itsBaud) setPageDims(unsigned itsLength, unsigned itsWidth) setMaxCopies(unsigned itsMaxCopies) setMaxSize(unsigned itsMaxSize) setSpoolDir(char * itsSpoolDir) setLogFile(char * itsLogFile) sets instance data

BUGS

i) writeEntry causes a core-dump when compiled using G++2.6.3 (due to a compiler bug!)

NOTES

In order ease printer reconfiguration, individual printcap entries are stored in separate files which are then "cat'ed" together forming the `printcap` file.

SEE ALSO

LPCcommand, PrinterStats

A.1.9 LPCcommand

DESCRIPTION

`LPCcommand` integrates the line printer controller program which controls (monitors) line printer devices. These include: starting (stopping) printing; enabling (disabling) the spool queue and obtaining the printer's state.

CLASS DEFINITION

```
class LPCcommand
{
public:
    LPCcommand();
    ~LPCcommand();

    Error    abortPrinter(char *);
    Error    cleanPrinter(char *);

    Error    disablePrinter(char *);
    Error    enablePrinter(char *);

    Error    upPrinter(char *);
    Error    downPrinter(char *, char *); // printername, message

    Error    startPrinter(char *);
    Error    stopPrinter(char *);
    Error    reStartPrinter(char *);

    Error    statusPrinter(char *,    Boolean &, Boolean &,
                   Boolean &, unsigned& );

    Error    topQ(char *);
};
```

INTERFACE

LPCcommand() ~LPCcommand() constructs (destructs) the lpc controller

abortPrinter(char * thePrinter) cleanPrinter(char * thePrinter) disablePrinter(char * thePrinter) downPrinter(char * thePrinter, char * theMessage) enablePrinter(char * thePrinter) startPrinter(char * thePrinter) reStartPrinter(char * thePrinter) creates a pipe to the line printer controller with appropriate command line argument. The output is then parsed to determine if the request is successful.

statusPrinter(char * thePrinter, Boolean & isQenabled, Boolean & isPrintEnabled, Boolean & canExamineSpoolArea unsigned&, jobsToPrint) parses the output from lpc to obtain the printer's status.

BUGS

none known

NOTES

Printer operations require super-user access.

SEE ALSO

PrintCapEntry

A.2 COMPUTER INTEGRATION

A.2.1 Kernel

DESCRIPTION

`Kernel` obtains metrics from the operating system's kernel. These include system loading (load average), physical and virtual memory usage, disk transfers and the process queue. The kernel is initialed (`initKernel`), image obtained (`getImage`) which is then decomposed by interface methods before closing the kernel.

CLASS DEFINITION

```
class Kernel
{
public:

    Kernel();
    ~Kernel();

    Error   initKernel();
    Error   closeKernel();
    Error   getImage();
    Error   checkImage();

    Error   getLoadAverage(LoadAverage &);
    Error   getMemory(int &, int &, int &, int &, int &);
    Error   getVMstats(int &, int &, int &, int &);
    Error   getDiskTransfers(DiskTransfers &);
    Error   getCPUstates(int &, int &, int &, int &);
    Error   getCPU(long & ccpu);
    Error   getClock(long &);
    Error   getMPID();
    Error   getNPROC(int &);
    Error   getBootTime(struct timeval &);
    Error   getNumberOfDisks(int &);
    Error   getProcessStats();

    Error   showProcessStats(int &, int &,  int &,
                             int &, int &,  int &);

protected:

    // instance data ...
};
```

INTERFACE

Kernel() constructs the kernel class by creating a `nlist` structure which is used to transport the image to (from) the kernel. The process queue metrics are also zeroised.

~Kernel() destructs the class

initKernel() opens the operating system kernel (using kwm_open)

getImage() retrieves the nlist image from the kernel.

checkImage() checks for missing nlist parameters.

closeKernel() closes the kernel

getLoadAverage(LoadAverage & theLoad) obtains the _avenrun metrics from the nlist.

getVMstats(int & thePagesIn, int & thePagesOut, int & theSwapsIn, int & theSwapsOut) gets the virtual memory metrics (vmmeter) from the nlist and returns the appropriate fields.

getMemory(int & realMem, int & availRealMem, int & virtRealMem, int & freeMem) gets physical memory metrics (vmtotal)

getDiskTransfers(DiskTransfers & theTransfers) gets disks transfers for each disk device

getCPUstates(int & theUser, int & theNice, int & theSys, int & theIdle) check!!!

getNumberOfDisks(int & theDisks) gets the number of disk devices in use.

getProcessStates() counts the number of processes in each process state in the queue.

showProcessStats(int & itsSleeps, int & itsWaits, int & itsRunns, int & itsIntermed, int & itsZoms, int & itsStopps) obtains the number of processes in each state.

BUGS

Due to a lack of complete documentation regarding the kernel's datum, only a subset of nlist values are retrieved by the kernel class. Physical memory metrics are highly dubious even though they are consistent with values yielded by other system utility programs.

NOTES

i) The kernel module is only implemented on SunOS 4.1.3

ii) In order to access the operating system kernel super-user permission is required.

SEE ALSO

rstat

A.2.2 Rstat

DESCRIPTION

`Rstat` uses Sun remote procedure call functions to obtain metrics from remote system kernels.

CLASS DEFINITION

```
class Rstat
{
public:

    Rstat(char *);          // hostname
    ~Rstat();

    Error   hasHardDisk(int &);
    Error   getStatistics();

    Error   getLoadAverage(LoadAverage &);
    Error   getVMstats(int &, int &, int &, int &);
    Error   getBootTime(struct timeval &);
    Error   getCPUstates(int &, int &, int &, int &);


protected:

    // instance data ...
};
```

INTERFACE

Rstat(char * itsHostName) constructs the class and set the host name.

~Rstat() destructs the class

hasHardDisk(int & itsDisks) performs `havedisk` remote procedure call and sets `itsDisks` to 1 if the remote system has a hard disk.

getStatistics() performs `statstime` remote procedure call and sets instance data.

getLoadAverage(LoadAverage & itsLoad) getVMstats(int & itsPagesIn, int & itsPagesOut, int & itsSwapIn, int & itsSwapOut) getBootTime(struct timeval & itsBoot) getCPUstates(int & itsCPUuser, int & itsCPUnice, int & itsCPUsys, int & itsCPUidle) retrieves instance data

BUGS

While testing this module, the remote procedure call succeeded even though rstat was *not* enabled on the workstation!

NOTES

SEE ALSO

kernel

A.2.3 UtsName

DESCRIPTION

UtsName retrieves data concerning the system architecture, operating system and machine name using POSIX uname functions.

CLASS DEFINITION

```
class UtsName
{
public:

    UtsName();
    ~UtsName();

    Error    getUname();
    Error    getSystemName(char *);
    Error    getNodeName(char *);
    Error    getNodeExt(char *);
    Error    getOpSystemRelease(char *);
    Error    getOpSystemVersion(char *);
    Error    getMachineName(char *);

protected:

    // instance data ...
};
```

INTERFACE

Uname() constructs the class

~Uname() class destructor

getUname() retrieves and decomposes a uname structure

getSystemName(char * itsSystemName) getNodeName(char * itsNodeName) getNodeExt(char * itsNodeExt) getOpSystemRelease(char * itsOpSystemRelease) getOpSystemVersion(char * itsOpSystemVersion) getMachineName(char * itsMachineName) retrieves instance data

BUGS

(see notes)

NOTES

Some variations exist in the content of uname between operating systems. Node extensions are provided for backward compatibility and are not implemented on SunOS.

SEE ALSO

UnameCommand

A.2.4 DFcommand

DESCRIPTION

`DFcommand` captures the output generated by executing a `df` process, which is later used by the DiskUnit

CLASS DEFINITION

```
class DFcommand
{
public:

    DFcommand();
    ~DFcommand();

    Error    issueCommand(char *);     // file system

    Error    getMetrics(Kbytes &, Kbytes &, Kbytes &, Percent &);
    Error    getMount(char * &);

protected:

    // instance data
};
```

INTERFACE

DFcommand() class constructor

~DFcommand() class destructor

issueCommand(char * itsFileSystem) creates a pipe to a process running `df` and obtains the file system's metrics: size, usage, availability and mount path.

getMetrics getMount retrieves instance data

BUGS

none known

NOTES

In order to access file system statistics, there is a precondition that the file system must already exist!

SEE ALSO

A.3 Manual Operations

A.3.1 ManDiskUnit

DESCRIPTION

`ManDiskUnit` integrates manual operations on a disk device: powering up (down), loading (removing, and scrapping) disks etc. The scrap disk method provides a compensation action for failed formats etc.

CLASS DEFINITION

```
class ManDiskUnit
{
public:

    ManDiskUnit();
    ~ManDiskUnit();

    Error    doPowerUp(char *);
    Error    doPowerDown(char *);
    Error    doLoadDisk(char *, char *); // Diskdrive, Diskmedia
    Error    doRemoveDisk(char *);

    Error    doScrapDisk(char *);
};
```

INTERFACE

ManDiskUnit() ~ManDiskUnit() class constructor (destructor)

doPowerUp(char * theDiskUnit) doPowerDown(char * theDiskUnit) doLoadDisk(char * theDiskUnit, char * theDisk) doRemoveDisk(char * theDiskUnit) doScrapDisk(char * theDiskUnit) contacts the operator to perform manual tasks.

BUGS

none known

NOTES

SEE ALSO

ManPrinter

A.3.2 ManPrinter

DESCRIPTION

`ManPrinter` integrates manual operations on a printing device: powering up (down), booting (shutting down), enabling (disabling) the printer, loading (removing) paper and ink supplies, collecting (scrapping) print jobs.

CLASS DEFINITION

```
class ManPrinter
{
public:

    ManPrinter();
    ~ManPrinter();

    Error   doPowerUp(char *);
    Error   doPowerDown(char *);
    Error   doBootPrinter(char *);
    Error   doShutDownPrinter(char *);
    Error   doOnLine(char *);
    Error   doOffLine(char *);
    Error   doInstall(char *);
    Error   doRemove(char *);
    Error   doLoadPaper(char *);
    Error   doRemovePaper(char *);
    Error   doLoadInk(char *);
    Error   doRemoveInk(char *);
    Error   doCollectJob(char *, char *);     // printer, job
    Error   doScrapJob(char *, char *);   // printer, job
};
```

INTERFACE

ManPrinter() ~ManPrinter() class constructor (destructor)

doPowerUp(char * thePrinter) doPowerDown(char * thePrinter) doBootPrinter(char * thePrinter) doShutDownPrinter(char * thePrinter) doOnLine(char * thePrinter) doOffLine(char * thePrinter) doInstall(char * thePrinter) doRemove(char * thePrinter) doLoadPaper(char * thePrinter) doRemovePaper(char * thePrinter) doLoadInk(char * thePrinter) doRemoveInk(char * thePrinter) doCollectJob(char * thePrinter, char * theJob) doScrapJob(char * thePrinter, char * theJob) performs manual operations on the printer.

BUGS

none known

NOTES

SEE ALSO

ManComputer etc.

A.3.3 ManComputer

DESCRIPTION

`ManComputer` integrates manual operations on a computer. These include powering up (down), booting (shutting down) and installing an operating system.

CLASS DEFINITION

```
class ManComputer
{
public:

    ManComputer();
    ~ManComputer();

    Error    doPowerUp(char *);    // computer name
    Error    doPowerDown(char *); // computer name
    Error    doBoot(char *);       // computer name
    Error    doShutDown(char *);  // computer name
    Error    doConfigure(char *); // computer name
    Error    doInstallOpSystem(char *); // computer name

};
```

INTERFACE

ManComputer() ~ManComputer() class constructor (destructor)

doPowerUp(char * itsCompName) doPowerDown(char * itsCompName) doBoot(char * itsCompName) doShutDown(char * itsCompName) doConfigure(char * itsCompName) doInstallOpSystem(char * itsCompName)

BUGS

none known

NOTES

SEE ALSO

ManPrinter

Appendix B
UnRecoverable Layer

B.1 ManagedResource

DESCRIPTION

`Managed Resource` forms the Management Information system's unrecoverable base class and is derived from the Arjuna tookkit's `LockManager` class i.e. Managed Resources are both serialised and persistent entities. The Managed resource base class has three constructors: creating a Managed Resource from scratch and accessing persistent resources using an Arjuna name and unique identifier. Although Arjuna supports higher level names which are mapped to unique identifiers, Arjuna names are largely unused in the Prototype Management Information System (simply because the Arjuna Name Server is not fully distributed at this point in time).

CLASS DEFINITION

```
class ManagedResource : public LockManager
{
public:

    ManagedResource(char *,  char *, NodeCoupling, ObjectKind);
    ManagedResource(Uid &, Error &);
    ManagedResource(ArjunaName, Error &);

    ~ManagedResource();

    virtual Error    onLine();
    virtual Error    offLine();

    virtual Error    getStatus();
    virtual Error    autoGetConfig();

    virtual Boolean  restore_state (ObjectState & , ObjectType);
    virtual Boolean  save_state (ObjectState & , ObjectType);
    virtual const    TypeName type () const ;


protected:

    // instance data ...
};
```

INTERFACE

ManagedResource(char* itsName, char* itsResourceLocation, NodeCoupling itsCoupling, ObjectKind itsKind) Constructs a new Managed Resource

ManagedResource (Uid& Uold, Error &result) Accesses a persistent Managed Resource with unique identifier Uold and returns result.

ManagedResource (ArjunaName Name, Error &result)

~ManagedResource() destructs the ManagedResource Class

onLine() activates a Resource Controller

offLine() de-activates an already active controller.

restore_state (ObjectState & Os, ObjectType ot) restores the object state from the persistent object store

save_state (ObjectState & Os, ObjectType ot) saves the object state.

INSTANCE DATA

Type	*Variable Name*	*Description*
char *	theObjectName	Arjuna Object Name
char *	theResourceLocation	Physical Location
ObjectKind	theObjectKind	nature of object
NodeCoupling	theCoupling	close, loose or uncouple object
NodeStatus	theStatus	object status
int	thePowerPointRef	power point
PowerPhase	thePowerPhase	red, blue or green power phase
int	theAssetRef	asset serial number
Boolean	isExternalOk	state of physical resource

BUGS

none known

NOTES

Arjuna names are not registered in the Arjuna name server.

SEE ALSO

Arjuna documentation

B.2 Device

DESCRIPTION

`Device` is derived from Managed Resource and provides virtual methods to power up (down), reset and connect (remove) the device. These methods are prefixed `auto` referring to automatic operations; `manual` — Manual operations and `remote` — accessing services via remote procedure call (used in the Computer class).

Generally speaking, all automatic operations are performed on close coupled nodes, remote operations — provided that the managed resource is reachable. This forms part of the operations precondition.

CLASS DEFINITION

```
class Device : public ManagedResource
{
public:

    Device(char *, char *, char *, NodeCoupling, ObjectKind);
    Device(Uid &, Error &);
    Device(ArjunaName, Error &);

    ~Device();

    virtual Error   manualPowerUp();
    virtual Error   autoPowerUp();
    virtual Error   manualPowerDown();
    virtual Error   autoPowerDown();
    virtual Error   autoReset();
    virtual Error   manualReset();
    virtual Error   getStatus();
    virtual Error   manualConnectTo();
    virtual Error   manualRemoveFrom();

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

protected:

    // instance data ...
};
```

INTERFACE

Device(char * AN, char * itsName, char * itsResourceLocn, NodeCoupling itsCoupling, ObjectKind itsKind) creates an unrecoverable device

Device(Uid & Uold, Error & res) Device(ArjunaName RN, Error & res) accesses an persistent device using a unique identifier and Arjuna name.

manualPowerUp() autoPowerUp() powers the device (manually and automatically)

manualPowerDown() autoPowerDown() unpowers the device

autoReset() manualReset() resets the device

manualConnectTo() manualRemoveFrom() connects (removes) the device to (from) a host computer

INSTANCE DATA

Type	*Variable Name*	*Description*
char *	theDeviceName	the device's name
DeviceState	theDeviceState	the device's state: powered (unpowered), on (off) line etc.
char *	theConnection	host name
char *	theCable	cable identifier
char *	thePort	port reference

BUGS

none known

NOTES

Most of the device's methods are redefined by derived device classes.

SEE ALSO

Terminal, Printer, TapeUnit, DiskUnit

B.3 Printer

DESCRIPTION

`Printer` is derived from and redefines virtual functions exported from `Device` class. Generally speaking, many of these functions are performed using the line printer controller (LPC command), Man Printer and Printcap. Further methods concerning creating and deleting configuration files and directories are implemented using the file (directory) integration class. Providing accounting is enabled, printer statistics can be obtained and hence it is possible to determine if the printer is out of paper.

CLASS DEFINITION

```
class Printer : public Device
{
public:

    Printer(char *,  char *, char *, char *, NodeCoupling, ObjectKind);
    Printer(Uid &, Error &);
    Printer(ArjunaName, Error &);

    ~Printer();

    virtual Error   manualPowerUp();
    virtual Error   manualPowerDown();
    virtual Error   manualStartPrinting();
    virtual Error   autoStartPrinting();
    virtual Error   manualStopPrinting();
    virtual Error   autoStopPrinting();
    virtual Error   manualStartQueuing();
    virtual Error   autoStartQueuing();
    virtual Error   manualStopQueuing();
    virtual Error   autoStopQueuing();
    virtual Error   autoAbort();
    virtual Error   autoRestart();

    virtual Error   manualLoadPaper();
    virtual Error   manualRemovePaper();
    virtual Error   autoPrintTest();
    virtual Error   autoPrintJob(char *);
    virtual Error   autoRemoveJob(char *);
    virtual Error   manualCollectJob(char *);
    virtual Error   manualScrapJob(char *);

    virtual Error   autoMakeSpoolDir();
    virtual Error   autoRemoveSpoolDir();
    virtual Error   autoMakeLogFile();
    virtual Error   autoRemoveLogFile();
    virtual Error   autoMakeAccountFile();
    virtual Error   autoRemoveAccountFile();
    virtual Error   autoMakePrintCapEntry();
    virtual Error   autoRemovePrintCapEntry();

    virtual Error   autoGetPrintStats();
    virtual Error   autoGetPrinterStatus();
    virtual Error   autoGetPrintCap();
    virtual Boolean isPaperExhausted();

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const   TypeName type () const ;

protected:

    // instance data ...
};
```

INTERFACE

Printer(char * AN, char * itsName, char * itsPRName, char * itsResourceLocn, NodeCoupling itsCoupling, ObjectKind itsObjectKind) constructs the unrecoverable printer and sets default instance data

Printer(Uid & Uold, Error & res) Printer(ArjunaName RN, Error & res) accesses a persistent printers using unique identifier and Arjuna Name

~Printer() destructs the class

autoStartPrinting(), autoStopPrinting() autoStartQueuing(), autoStopQueuing() enables (disables) printing (queuing) the printer using an LPC command

autoMakeSpoolDir(), autoRemoveSpoolDir() creates (deletes) the printer's Spool Directory

autoMakeLogFile(), autoRemoveLogFile() creates (deletes) the printer's Log File

autoMakeAccountFile(), autoRemoveAccountFile() creates (deletes) the printer's Account File

manualLoadPaper(), manualRemovePaper() requests the printer operator to load (remove) a pack of paper

autoPrintJob(char* itsFileName) sends the file to the printer

autoRemoveJob(char* itsUserName) removes the User's Print Job from the spool. If the job has already been printed, it can be scrapped using the Manual scrap job method.

autoGetPrinterStats() obtains the volume and number of jobs printed using the printer stats in integration class.

autoGetPrinterStatus() obtains the printer's status (number of jobs in queue, if the printer and queue is enabled etc,.) using LPC

autoGetPrinterCap() reads the printer's printcap entry.

INSTANCE DATA

Type	*Variable Name*	*Description*
char *	thePrinterName	printer's name
PrinterState	thePrinterState	printing, stopped, misfed etc.
PrintQuality	thePrintQuality	draft, nlq etc.
PageKind	thePageKind	a4, a5 etc.
SpoolerStatus	theSpoolerStatus	local, remote etc.
Boolean	isQenabled	queue enabled (disabled)
unsigned	theNoOfPrintJobs	number of jobs in queue
Boolean	canExamineSpooler	self explanatory
char *	theLogFile	log file name
char *	theSpoolDir	spool directory name
char *	theAccountFile	account file name
char *	theLocalPrinter	local printer name
char *	theRemotePrinter	remote printer name
char *	theRemoteMachine	remote host name

unsigned	theMaxFileSize	self explanatory
unsigned	theMaxCopies	ditto
unsigned	thePageLength	ditto
unsigned	thePageWidth	ditto
int	theJobsPrinted	ditto
float	theVolumePrinted	ditto
float	theVolumeLoaded	paper reservoir
int	theBaudRate	self explanatory

BUGS

none known

NOTES

SEE ALSO

PrintCapEntry, PrinterStats, LPCcommand, Device

B.4 Terminal

DESCRIPTION

The `Terminal` class is derived from `Device` and `ManagedResource` models an unrecoverable terminal. The class is based upon an Encore Annex networked terminal and records the terminal's (input and output) speed, parity, control and data bits, flow control protocols etc. Many of these parameters are obtained from the `TerminalController` integration class, which in turn is based upon the POSIX "termios" ioctls. Instead of recording configuration values in a large (packed) structure, data items are encoded in abstract data types. Several methods are provided to power up (down) the terminal, device connection (removal) is inherited from the device class.

CLASS DEFINITION

```
class Terminal : public Device
{
public:

    Terminal(char *, char *, char *, NodeCoupling, ObjectKind);
    Terminal(Uid &, Error &);
    Terminal(ArjunaName, Error &);

    ~Terminal();

    virtual Error    autoPowerUp();
    virtual Error    autoPowerDown();

    virtual Error    autoGetController();

    virtual Boolean  restore_state(ObjectState &, ObjectType);
    virtual Boolean  save_state(ObjectState &, ObjectType);
    virtual const    TypeName type() const ;

protected:

    // instance data
};
```

INTERFACE

Terminal(char * AN, char * itsName, char * itsResourceLocn, NodeCoupling itsCoupling, ObjectKind itsKind) creates a terminal from scratch

Terminal(Uid & Uold, Error & res) Terminal(ArjunaName RN, Error & res) accesses a terminal using UID and ArjunaName.

~Terminal() destructs the Terminal.

autoPowerUp() autoPowerDown() powers up (down) the terminal

autoGetController() obtains configuration data from the controller.

INSTANCE DATA

Type	*Variable Name*	*Description*
int	theInputSpeed	(baud rate)
int	theOutputSpeed	(ditto)
DataBits	theDataBits	number of data bits (5 - 8)
StopBits	theStopBits	number of stop bits (1, 1.5, 2)

PartityKind	thePartityKind	odd, even or no parity
ControlLines	theControlLines	modem or flow control protocol
DeviceKind	theDeviceKind	x25, dial in, hard-wired etc.
DeviceMode	theDeviceMode	slave etc.
int	theTimeOut	terminal's inactivity time out.
FlowControl	theInputFlowControl	flow control protocol
FlowControl	theOutputFlowControl	(ditto)
int	theInputBufferSize	blocks of 512 characters
int	theScreenWidth	characters
int	theScreenDepth	lines
LineDiscipline	theLineDiscipline	

BUGS

none known

NOTES

A large ammount of configuration data is retrieved using `autoGetController`.

SEE ALSO

TerminalController

B.5 TapeUnit

DESCRIPTION

The (unrecoverable) `TapeUnit` class like other peripherals is derived from the `Device` class and is based on the "mtio" (Magnetic tape input—output) systems utility program. Methods are provided to load, remove, eject and retention tapes, dumping (restoring) files and obtaining configuration data.

CLASS DEFINITION

```
class TapeUnit : public Device
{
public:

    TapeUnit(char *, char *, char *,  NodeCoupling, ObjectKind);
    TapeUnit(Uid &, Error &);
    TapeUnit(ArjunaName, Error &);

    ~TapeUnit();

    virtual Error    setBlockSize(int);
    virtual Error    manualPowerUp();
    virtual Error    manualPowerDown();

    virtual Error    manualLoadTape();
    virtual Error    manualRemoveTape();

    virtual Error    autoRewindTape();
    virtual Error    autoEraseTape();
    virtual Error    autoEjectTape();
    virtual Error    autoRetensionTape();

    virtual Error    autoDumpToTape();
    virtual Error    autoRestoreFromTape();

    virtual Error    autoGetController();

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

protected:

    // instance data ...
};
```

INTERFACE

TapeUnit(char * AN, char * itsName, char * itsResourceLocn, NodeCoupling itsCoupling, ObjectKind itsKind) constructs the tape unit

TapeUnit(Uid & Uold, Error & res) TapeUnit(ArjunaName AN, Error & res) access persistent tape units

~TapeUnit() destructs the class

manualPowerUp(), manualPowerDown() contacts the tape operator to manually power up (down) the tape unit

manualLoadTape(), manualRemoveTape() contacts the taper operator to manually load (remove) a tape.

autoRewindTape(), autoEraseTape(), autoRetensionTape() rewinds, erases and adjusts the tape's tension using the tape controller class.

autoGetController() obtains configuration information (tape kind, unit kind and unit description) from the tape controller. Several functions are used to determine the kind of tape (thickness and type of media. i.e. reel or cartridge)

INSTANCE DATA

Type	*Variable Name*	*Description*
int	theBlockSize	block size
TapeUnitState	theTapeUnitState	loaded, empty etc.
UnitKind	theTapeUnitKind	Vax, SCSI, SUN etc.
TapeKind	theTapeMedia	reel, cartridge etc.
char *	theUnitDescription	description of tape unit

BUGS

None known

NOTES

SEE ALSO

TapeController

B.6 DiskUnit

DESCRIPTION

DiskUnit is derived from the device class and presents a simplified disk device comprising disk controller, hard disk, read (write) heads etc. Several methods are exported, these include: getting the controller, partitions and disk geometry which are performed using integration layer classes; and obtaining the disk's usage.

CLASS DEFINITION

```
class DiskUnit : public Device
{
public:

    DiskUnit(char *, char *, char *, NodeCoupling, ObjectKind);
    DiskUnit(Uid &, Error &);
    DiskUnit(ArjunaName, Error &);

    ~DiskUnit();

    virtual Error    manualPowerUp();
    virtual Error    manualPowerDown();
    virtual Error    autoFormatMedia();
    virtual Error    autoGetController();
    virtual Error    getStatus();
    virtual Error    autoGetUsage();

    virtual Error    autoGetPartitions();
    virtual Error    autoGetGeometry();

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

protected:

    // instance data ...
};
```

INTERFACE

DiskUnit(char * AN, char * itsName, char * itsResourceLocn, NodeCoupling itsCoupling, ObjectKind itsKind) creates an unrecoverable disk unit

DiskUnit(Uid & Uold, Error & res) DiskUnit(ArjunaName AN, Error & res) accesses persistent disk units.

~DiskUnit() destructs the class

autoGetController() retrieves controller configuration data using the DiskController in integration class. The Disk interface is determined via a function which maps a "short" integer to the disk interface enumeration type.

autoGetPartitions() retrieves the partition map (starting cylinder and number of blocks using the DiskPartition integration class.

autoGetGeometry() retrieves the disk's geometry (number of cylinders, number of heads, sectors per track etc.) using the disk geometry class

autoGetUsage() obtains disk usage (size, usage, availability) via the dfcommand.

INSTANCE DATA

Type	*Variable Name*	*Description*
DiskInterface	theDiskInterface	WCC2880, XY450 etc.
short	theUnitAddress	self explanatory
int	theControllerAddress	ditto
long	theStartCylinder	ditto
long	theNumberBlocks	disk capacity
unsigned short	theNumberCylinders	self explanatory
unsigned short	theNumberAltCylinders	ditto
unsigned short	theNumberHeads	ditto
unsigned short	theHeadOffset	ditto
unsigned short	theSectorsPerTrack	ditto
unsigned short	theInterleave	interleave factor
Kbytes	theSize	self explanatory
Kbytes	theUsage	ditto
Kbytes	theAvail	ditto
Percent	theCapacity	ditto

BUGS

none known

NOTES

Assumes that the disk device houses a single partition.

SEE ALSO

DiskController, DiskGeometry, DiskPartitions

B.7 FloppyUnit

DESCRIPTION

FloppyUnit is a simplified disk unit which represents an external floppy disk device. Several methods are exported which concern loading, removing, ejecting and formatting the disk. An extra method to scrap the disk is provided as a compensation operation for badly formatted disks.

CLASS DEFINITION

```
class FloppyUnit : public Device
{
public:

    FloppyUnit(char *, char *, char *, NodeCoupling, ObjectKind);
    FloppyUnit(Uid &, Error &);
        FloppyUnit(ArjunaName, Error &);

    ~FloppyUnit();

    virtual Error   getStatus();
    virtual Error   manualLoadMedia();
    virtual Error   manualRemoveMedia();
    virtual Error   autoEjectMedia();
    virtual Error   autoFormatMedia();
    virtual Error   manScrapMedia();
    virtual Error   autoGetController();

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

protected:

     // instance data ...
};
```

INTERFACE

FloppyUnit(char * AN, char * itsName, char * itsResourceLocn, NodeCoupling itsCoupling, ObjectKind itsKind) creates an unrecoverable floppy unit from scratch

FloppyUnit(Uid & Uold, Error & res) FloppyUnit(ArjunaName AN, Error & res) accesses a persistent floppy unit.

~FloppyUnit() destructs the floppy unit.

manualLoadMedia (), manualRemoveMedia() contacts the disk operator to load (remove) a disk to(from) the disk unit.

autoEjectMedia(), autoFormatMedia() ejects and formats the floppy disk via the floppy unit controller.

autoGetController() obtains configuration characteristics from the floppy unit's controller (data transfer rate, number of cylinders, heads, sector size etc.)

INSTANCE DATA

Type	*Variable Name*	*Description*

FloppyUnitState	theFloppyUnitState	loaded etc.
int	theTransferRate	self explanatory
int	theNoOfCyl	number of cylinders
int	theNoOfHeads	self explanatory
int	theSectorSize	ditto
int	theSectorsPerTrack	ditto
int	theSteps	ditto

BUGS

none known

NOTES

SEE ALSO

SCSIfloppy, ManDiskUnit

B.8 Computer

DESCRIPTION

`Computer`, derived from `ManagedResource` represents a simplified workstation resource. Several methods are provided which power up (down), boot (shutdown) the physical resource and obtain status information such as loading, memory consumption etc.

CLASS DEFINITION

```
class Computer : public ManagedResource
{
public:

    Computer(char *, char *, char *,  NodeCoupling, ObjectKind);
    Computer(Uid &, Error &);
    Computer(ArjunaName, Error &);

    ~Computer();

    virtual Error   autoPowerUp();
    virtual Error   manualPowerUp();
    virtual Error   autoPowerDown();
    virtual Error   manualPowerDown();
    virtual Error   autoBoot();
    virtual Error   manualBoot();
    virtual Error   autoShutDown();
    virtual Error   manualShutDown();

    virtual Error   manualConnectToNetwork();
    virtual Error   manualRemoveFromNetwork();

    virtual Error   manualAddDevice();
    virtual Error   manualRemoveDevice();

    virtual Error   autoGetKernalMetrics();
    virtual Error   remoteGetKernalMetrics();
    virtual Error   autoGetMachine();
    virtual Error   autoGetMemory();
    virtual Error   autoGetProcStates();
    virtual Error   autoGetNumberOfUsers();

    virtual Boolean  restore_state (ObjectState & , ObjectType);
    virtual Boolean  save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

protected:

    // instance data ...
};
```

INTERFACE

Computer(char * AN, char * itsName, char * itsResourceLocn, NodeCoupling itsCoupling, ObjectKind itsKind) creates an unrecoverable computer

Computer(Uid & Uold, Error & res) Computer(ArjunaName RN, Error & res) accesses persistent computers

~Computer() destructs the computer

autoPowerUp(), autoPowerDown(), autoBoot(), autoShutDown() automatically powers up (down), and boots (shuts down) the computer

manualPowerUp(), manualPowerDown(), manualBoot(), manualShutDown() the manual counterparts of autoPowerUp() etc.

manualConnectToNetwork(), manualRemoveFromNetwork(), manualAddDevice(), manualRemoveDevice() manual operations to add (remove) devices and network connections

autoGetKernalMetrics(), remoteGetKernalMetrics() Obtains system loading and CPU state from the operating system kernel.

autoGetMachine() Obtains the operating system name, release and version; machine name and architecture.

autoGetMemory() Obtains virtual memory statistics from the kernel

autoGetProcStates() Obtains a summary of the process state vector from the kernel

autoGetNumberOfUsers() Obtains the number of workstation users

INSTANCE DATA

Type	*Variable Name*	*Description*
char *	theComputerName	the computer's host name
LoadAverage	theAvenrun	load metrics
int	theRealMem	physical memory size
int	theAvailRealMem	physical memory availability
int	theVirtMem	virtual memory size
int	theAvailVirtMem	virtual memory availability
int	theFreeMem	free memory
int	theProcsSleeping	self explanatory
int	theProcsWaiting	ditto
int	theProcsRunning	ditto
int	theProcsIntermed	ditto
int	theProcsZombied	ditto
int	theProcsStopped	ditto
int	thePagesIn	ditto
int	thePagesOut	ditto
int	theSwapsIn	ditto
int	theSwapsOut	ditto
int	theUserCPU	ditto
int	theNiceCPU	ditto
int	theSysCPU	ditto
int	theIdleCPU	ditto
int	theNumberOfUsers	ditto
char *	theSystemName	ditto
char *	theNodeName	ditto
char *	theNodeExt	ditto
char *	theOpSystemRelease	ditto
char *	theOpSystemVersion	ditto
char *	theMachineName	ditto
ComputerState	theComputerState	powered, booted etc.
char *	theEtherCard	ethernet card reference
char *	theNetAddress	network address
char *	theTapRef	tape reference
char *	theDropCable	drop cable reference

BUGS

NOTES

SEE ALSO

UTMP, Kernel, Rstat, Uname, ManComputer

B.9 Software

DESCRIPTION

`Software` represent a software distribution held in its own directory. Software components can be packed (unpacked), encoded (decoded) etc. Each distribution's configuration is encoded using a "makefile" which provides rules concerning compilation, installation and "clean"ing up directories.

CLASS DEFINITION

```
class Software : public ManagedResource
{
public:

    Software(char *,
         char *,
         char *,
         NodeCoupling,
         ObjectKind
        );

    Software(Uid &, Error &);
        Software(ArjunaName, Error &);
    ~Software();

    virtual  Error    autoUpGrade();
    virtual  Error    manualUpGrade();

    virtual  Error    autoDownGrade();
    virtual  Error    manualDownGrade();

    virtual  Error    autoEncode();
    virtual  Error    manualEncode();

    virtual  Error    autoDecode();
    virtual  Error    manualDecode();

    virtual  Error    autoTarPack();
    virtual  Error    autoZooPack();
    virtual  Error    manualPack();

    virtual  Error    autoTarUnpack();
    virtual  Error    autoZooUnpack();
    virtual  Error    manualUnpack();

    virtual  Error    autoMakeAll();
    virtual  Error    autoMakeClean();

    virtual  Error    autoMakeInstall();

    virtual  Error    autoGetConfig();

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

protected:

    // instance data
};
```

INTERFACE

Software(char *, char *, char *, NodeCoupling, ObjectKind) Creates an unrecoverable software server.

Software(Uid &, Error &) Software(ArjunaName, Error &) accesses persistent servers using unique identifier and Arjuna names.

~Software() destructs the server.

autoUpGrade(), autoDownGrade() manualUpGrade(), manualDownGrade() upgrades (downgrades) the distribution automatically (manually)

autoEncode(), autoDecode(), manualEncode(), manualDecode() encodes (decodes) the distribution using uuencode (uudecode)

autoTarPack(), autoTarUnpack(), autoZooPack(), autoTarUnpack(), autoZooUnpack() packs (unpacks) the distribution using "tar" and "zoo"

manualPack() manualUnpack() packs (unpacks) the software manually

autoMakeAll(), autoMakeClean(), autoMakeInstall() builds, installs and cleans up distributions using the UNIX make program

autoGetConfig() obtains file status information

INSTANCE DATA

Type	*Variable Name*	*Description*
char *	thePathName	host directory
char *	theSoftwareName	software name
SoftwareState	theSoftwareState	installed, built etc.
char *	theOwner	self explanatory
char *	theGroup	ditto
mode_t	theMode	ditto
off_t	theSize	ditto
time_t	theModifyTime	ditto
Compiler	theCompiler	ditto
Platform	thePlatform	ditto

BUGS

none known

NOTES

SEE ALSO

MakeCommand, ZooCommand, TarCommand, UUcommand, ManSoftware

Appendix C
RECOVERABLE LAYER

Having discussed the constituents of both the integration and recoverable layers in the Management Information System, let us turn our attention to the recoverable layer, Unlike the unrecoverable layer, this layer provides both backward error recovery, serialisability and fault reporting: each method exported from recoverable servers is implemented in terms of recovery blocks constructed from atomic actions.

Many operations in the recoverable layer are self explanatory and therefore only minimal descriptions are provided.

C.1 RecComputer

DESCRIPTION

RecComputer is derived from the (unrecoverable) computer class and provides single operations implementations which set (show) state data etc. Each control operation is encoded within a recovery block with preconditions and an acceptance test. In the event of operations failing within the recovery block its effects are compensated and errors are reported as appropriate.

CLASS DESCRIPTION

```
class RecComputer : public Computer
{
public:

    RecComputer(char *, char *, char *, NodeCoupling, ObjectKind);
    RecComputer(Uid &, Error &);
    RecComputer(ArjunaName, Error &);

    ~RecComputer();

    virtual Error   onLine();
    virtual Error   offLine();

    virtual Error   powerUp();
    virtual Error   powerDown();
    virtual Error   boot();
    virtual Error   shutDown();

    virtual Error   getStatus();
    virtual Error   addDevice();
    virtual Error   removeDevice();

    virtual Error   connectToNetwork();
    virtual Error   removeFromNetwork();

    virtual Error   getKernalMetrics();
    virtual Error   getMachine();
    virtual Error   getMemory();
    virtual Error   getProcStates();
    virtual Error   getNumberOfUsers();

    virtual Error   showLoadAverage(LoadAverage &);
    virtual Error   showMemory(int &, int &, int &, int &, int &);
    virtual Error   showProcesses(int &, int &, int &,
                      int &, int &, int &);
    virtual Error   showVMstats(int &, int &, int &, int &);
    virtual Error   showCPUstate(int &, int &, int &, int &);
    virtual Error   showNumUsers(int &);

    virtual Error   showComputerName(char * &, char * &, char * & );
    virtual Error   showOpSystem(char * &, char * &   );
    virtual Error   showMachineArch(char * &);

    virtual Error   showComputerState(ComputerState &);

    virtual Error   showLocation(char * &);
    virtual Error   showManufacturer(char * &);
    virtual Error   showPowerPoint(char * &);
    virtual Error   showPowerPhase(PowerPhase &);
    virtual Error   showContactName(char * &);
    virtual Error   showMaintenenceRef(char * &);
    virtual Error   showAssetRef(char * &);

    virtual Error   showEtherCard(char * &);
    virtual Error   showNetAddress(char * &);
    virtual Error   showTapRef(char * &);
    virtual Error   showDropCable(char * &);

    virtual Error   setLocation(char *);
    virtual Error   setManufacturer(char *);
    virtual Error   setPowerPoint(char *);
    virtual Error   setPowerPhase(PowerPhase);
    virtual Error   setContactName(char * );
    virtual Error   setMaintenenceRef(char *);
    virtual Error   setAssetRef(char *);

    virtual Error   setEtherCard(char *);
    virtual Error   setNetAddress(char *);
    virtual Error   setTapRef(char *);
    virtual Error   setDropCable(char *);

    virtual Error   compensate(Error);
    virtual Error   reportError(char *);
    virtual Error   reportError(Error);

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

};
```

CLASS INTERFACE

RecComputer(char *, char *, char *, NodeCoupling, ObjectKind), RecComputer(Uid &, Error &), RecComputer(ArjunaName, Error &) Constructs the class from scratch / accesses persistent object states using Uid and an ArjunaName

~RecComputer() terminates the RecComputer server.

onLine(), offLine() turns the probe on (off) line

powerUp(), boot(), shutDown(), powerDown() Powers (unpowers), boots (shuts down) the computer manually. Each operation has an inverse anti-operation and must be performed in sequence. I.e. the computer must be shut down before powering off.

addDevice(), removeDevice() manually adds (removes) devices.

connectToNetwork(), removeFromNetwork() manually connects (disconnects) the computer to (from) the network

getKernalMetrics() obtains kernel data (virtual memory, loading etc.) from the kernel using `autoGetKernelMetrics` and `remoteGetKernelMetrics`. An extra operations is available on workstations running Solaris and HP-UX which obtains this data using software.

getMachine() gets the machine architecture and operating system name using `autoGetMachine`

getMemory() obtains physical memory allocation using `autoGetMemory`

getProcStates() obtains the number of running (stopped, waiting etc.) processes using `autoGetProcStates`. On workstations running Solaris or HP-UX, an extra operation `softwareGetProcStates` is provided.

getNumberOfUsers() uses `autoGetNumberOfUsers` to obtain the number of users logged into the workstation.

showLoadAverage(LoadAverage & itsLoad) sets itsLoad to the workstations last known load average

showMemory(int & itsRealMem, int & itsAvailRealMem, int & itsVirtMem, int & itsAvailVirtMem, int & itsFreeMem) returns physical memory instance data.

showProcesses(int & itsProcsSleeping, int & itsProcsWaiting, int & itsProcsRunning, int & itsProcsIntermed, int & itsProcsZombied, int & itsProcsStopped) return the number of processes in each state.

showVMstats(int & itsPagesIn, int & itsPagesOut, int & itsSwapsIn, int & itsSwapsOut) obtains virtual memory statistics

showCPUstate(int & itsUserCPU, int & itsNiceCPU, int & itsSysCPU, int & itsIdleCPU) returns CPU state values

showNumUsers(int & itsNumberOfUsers) returns the number of user logged into the workstation.

showComputerName(char * & itsSystemName, char * & itsNodeName, char * & itsNodeExt) obtains the computer's name, node name and extension.

showOpSystem(char * & itsOpSystemRelease, char * & itsOpSystemVersion) obtains operating system name and version.

showMachineArch(char * & itsMachineName) obtains the machine's architecture.

showComputerState(ComputerState & itsComputerState) obtains the computer's state.

showLocation(char * & itsLocation), showManufacturer(char * & itsManufacturer), showPowerPoint(char * & itsPowerPoint), showPowerPhase(PowerPhase & itsPowerPhase), showContactName(char * & itsContactName), showMaintenenceRef(char * & itsMaintenanceRef), showAssetRef(char * & itsAssetRef) obtains asset data.

showEtherCard(char * & itsEtherCard), showNetAddress(char * & itsNetAddress) showTapRef(char * & itsTapeRef) showDropCable(char * & itsDropCable) obtains network connection data.

setLocation(char *) setManufacturer(char *) setPowerPoint(char *) setPowerPhase(PowerPhase) setContactName(char *) setMaintenenceRef(char *) setAssetRef(char *) setEtherCard(char *) setNetAddress(char *) setTapRef(char *) setDropCable(char *) sets asset, powering and network connection data.

compensate(Error) reportError(char *) reportError(Error) compensates and reports errors.

BUGS

Due to a bug in the Arjuna stub generator, showLoadAverage returns loading individually rather than in an array

NOTES

SELF TEST

Self testing the RecComputer requires initially powering the resource and getting kernel metrics, process states, and the numbers of users.

SEE ALSO

Computer

C.2 RecTerminal

DESCRIPTION

The (recoverable) terminal class `RecTerminal` is derived from (unrecoverable) `Terminal` and provides fault reporting, compensation and management control (monitoring) functions. Each operation is implemented using (read / write locked) atomic transactions with control functions implemented using recovery blocks.

CLASS DEFINITION

```
class RecTerminal : public Terminal
{
public:

    RecTerminal(char *, char *, char *, NodeCoupling, ObjectKind);
    RecTerminal(Uid &, Error &);
    RecTerminal(ArjunaName, Error &);

    ~RecTerminal();

    virtual Error   onLine();
    virtual Error   offLine();

    virtual Error   powerUp();
    virtual Error   powerDown();
    virtual Error   getStatus();
    virtual Error   getController();
    virtual Error   setController();

    virtual Error   showSpeeds(int &, int &);
    virtual Error   showDataBits(DataBits &);
    virtual Error   showStopBits(StopBits &);
    virtual Error   showParity(PartityKind &);

    virtual Error   showControlLines(ControlLines &);
    virtual Error   showDevice(DeviceKind &, DeviceMode &);
    virtual Error   showFlowControl(FlowControl &, FlowControl &);

    virtual Error   showScreenDims(int &, int &);
    virtual Error   showLineDiscipline(LineDiscipline &);

    virtual Error   showLocation(char * &);
    virtual Error   showManufacturer(char * &);
    virtual Error   showPowerPoint(char * &);
    virtual Error   showPowerPhase(PowerPhase &);
    virtual Error   showContactName(char * &);
    virtual Error   showMaintenenceRef(char * &);
    virtual Error   showAssetRef(char * &);

    virtual Error   setLocation(char *);
    virtual Error   setManufacturer(char *);
    virtual Error   setPowerPoint(char *);
    virtual Error   setPowerPhase(PowerPhase);
    virtual Error   setContactName(char * );
    virtual Error   setMaintenenceRef(char *);
    virtual Error   setAssetRef(char *);

    virtual Error   setSpeeds(int, int);
    virtual Error   setDataBits(DataBits);
    virtual Error    setStopBits(StopBits);
    virtual Error    setParity(PartityKind);
    virtual Error    setControlLines(ControlLines);
    virtual Error    setDevice(DeviceKind, DeviceMode);
    virtual Error    setFlowControl(FlowControl, FlowControl);
    virtual Error    setLineDiscipline(LineDiscipline);

    virtual Error    compensate(Error);
    virtual Error    reportError(char *);
    virtual Error    reportError(Error);

    virtual Boolean  restore_state(ObjectState &, ObjectType);
    virtual Boolean  save_state(ObjectState &, ObjectType);
    virtual const    TypeName type() const ;

};
```

CLASS INTERFACE

RecTerminal(char *, char *, char *, NodeCoupling, ObjectKind) RecTerminal(Uid &, Error &) RecTerminal(ArjunaName, Error &) constructs recoverable terminals from scratch or accesses terminal using Uid / ArjunaName

~RecTerminal() terminates the server

onLine(), offLine() switches the probe on (off) line

powerUp() powerDown() powers the terminal up (down)

getController() gets the terminal's controller using `autoGetController`

setController() sets the terminal controller data.

showSpeeds(int & itsInputSpeed, int & itsOutputSpeed) obtains the terminal's baud rate.

showDataBits(DataBits & itsDataBits) showStopBits(StopBits & itsStopBits) showParity(PartityKind & itsPartity) showControlLines(ControlLines & itsControlLines) showDevice(DeviceKind & itsDeviceKind, DeviceMode & itsDeviceMode) showFlowControl(FlowControl & itsInputFlowControl, FlowControl & itsOutputFlowControl) obtains configuration settings from the terminal's controller

showScreenDims(int & itsWidth, int & itsDepth) obtains screen width (characters) and depth (lines)

showLineDiscipline(LineDiscipline & itsLineDiscipline) obtains LineDiscipline (provided for backward compatibility)

setSpeeds(int itsInputSpeed, int itsOutputSpeed) setDataBits(DataBits itsDataBits) setStopBits(StopBits itsStopBits) setParity(PartityKind itsPartiy) setControlLines(ControlLines itsControl) setDevice(DeviceKind itsDeviceKind, DeviceMode itsDeviceMode) setFlowControl(FlowControl itsInputFlow, FlowControl itsOutputFlow) setLineDiscipline(LineDiscipline itsLineDiscipline) sets instance data

compensate(Error) reportError(char *) reportError(Error) compensates and reports errors

BUGS

none known

NOTES

SELF TEST

Once powered, the terminal's controller is obtained which sets instance data as appropriate

SEE ALSO

Terminal

C.3 RecPrinter

DESCRIPTION

`RecPrinter` defines a recoverable printer device which is derived from (unrecoverable) printer. Unlike its unrecoverable counterpart, only one of each method is exported from the class — most methods having an inverse operation which is used to provide backward error recovery, Monitoring operations are prefixed “show” and are used to return instance data; whereas those prefixed “get” are used to obtain configuration and other instance data.

CLASS DEFINITION

```
class RecPrinter : public Printer
{
public:

    RecPrinter(char *, char *, char *, char *, NodeCoupling, ObjectKind);
    RecPrinter(Uid &, Error &);
    RecPrinter(ArjunaName, Error &);

    ~RecPrinter();

    virtual Error   onLine();
    virtual Error   offLine();

    virtual Error   powerUp();
    virtual Error   powerDown();

    virtual Error   startPrinting();
    virtual Error   stopPrinting();

    virtual Error   startQueuing();
    virtual Error   stopQueuing();

    virtual Error   restart();

    virtual Error   printTest();
    virtual Error   printJob(char *);     // file name
    virtual Error   removeJob(char *);    // user name

    virtual Error   makeSpoolDir();
    virtual Error   makeLogFile();
    virtual Error   makeAccountFile();
    virtual Error   removeSpoolDir();
    virtual Error   removeLogFile();
    virtual Error   removeAccountFile();

    virtual Error   getPrinterStatus();
    virtual Error   getPrintStats();
    virtual Error   getPrintCap();

    virtual Error   makePrintCapEntry();
    virtual Error   removePrintCapEntry();

    virtual Error   showPrinterName(PrinterName &);
    virtual Error   showPageDims(unsigned &, unsigned &);

    virtual Error   showSpooler(SpoolerStatus &);
    virtual Error   showPrinterState(PrinterState &);
    virtual Error   showQueue(Boolean  &, unsigned &);
    virtual Error    showLogFile(char * &);
    virtual Error    showSpoolDir(char * &);
    virtual Error    showLimits(unsigned &, unsigned &);
    virtual Error    showJobsPrinted(int &, float & );
    virtual Error    showLocation(char * &);
    virtual Error    showManufacturer(char * &);
    virtual Error    showPowerPoint(char * &);
    virtual Error    showPowerPhase(PowerPhase &);
    virtual Error    showContactName(char * &);
    virtual Error    showMaintenenceRef(char * &);
    virtual Error    showAssetRef(char * &);

    virtual Error    setLocation(char *);
    virtual Error    setManufacturer(char *);
    virtual Error    setPowerPoint(char *);
    virtual Error    setPowerPhase(PowerPhase);
    virtual Error    setContactName(char * );
    virtual Error    setMaintenenceRef(char *);
    virtual Error    setAssetRef(char *);

    virtual Error    compensate(Error);
    virtual Error    reportError(char *);
    virtual Error    reportError(Error);

    virtual Boolean  restore_state (ObjectState & , ObjectType);
    virtual Boolean  save_state (ObjectState & , ObjectType);
    virtual const    TypeName type () const ;

};
```

CLASS INTERFACE

RecPrinter(char * AN, char * itsName, char * itsPrinterName, char * itsResource-Locn, NodeCoupling itsCoupling, ObjectKind itsKind) creates and activates a recoverable printer

RecPrinter(Uid & Uold, Error & res) RecPrinter(ArjunaName RN, Error & res) creates and activates a recoverable printer

~RecPrinter() destructs the class and de-activates the RecPrinter server **onLine(), offLine()** activates (deactivates) the probe

powerUp(), powerdown () powers (turns off) the printer by first attempting manual then automatic operation

startPrinting(), stopPrinting() starts (stops) the printer by attempting automatic then manual operations.

startQueuing (), stopQueuing () starts (stops) queuing by attempting automatic then manual operations

makeSpoolDir(), removeSpoolDir() makeLogfile(), removeLogfile () makeAccount-file(),removeAccountfile() creates (deletes) files and directories used by the line printer subsystem.

getPrinterStatus() gets printer status (spooler, queue and number of print jobs) automatically

getPrinterStats() gets the volume and number of jobs printed automatically

showPrinterName(PrinterName & itsPrinterName) showPageDims(unsigned & itsLength, unsigned & itsWidth) showSpooler(SpoolerStatus & itsSpooler) showPrinterState(PrinterState & itsPrinterState) showQueue(Boolean & itsEnabled, unsigned & itsJobs) showLogFile(char * & itsLog)

showSpoolDir(char * & itsSpoolDir) showLimits(unsigned & itsMaxSize, unsigned & itsMaxCopies) showJobsPrinted(int & itsJobs, float & itsVolume) retrieves instance data in a read locked atomic action.

compensate(Error theError) reportError(Error theError) reportError(char * theMessage) sends error reports the the fault manager responsible for the printer.

NOTES

(i) Using the line printer controller in the unrecoverable printer enables remote printer management

(ii) Getting printer statistics must be on a local node

BUGS

None Known

SELF TEST

Provided the printer is powered, queue and printing enabled, the status and statistics can be obtained. Similarly configuration parameters such as the spool directory and account files can be checked.

SEE ALSO

Printer

C.4 RecTapeUnit

DESCRIPTION

The (recoverable) tape unit provides a fault tolerant (distributed) implementation of `mtio` and exports methods to set (get) configuration properties in addition to control (monitoring) operations. Three constructors initialise the server, creating tape units from scratch and accessing persistent object state using an arjuna name (unique identifier). One destructor is provided which terminates the server.

CLASS DEFINITION

```
class RecTapeUnit : public TapeUnit
{
public:

    RecTapeUnit(char *, char *, char *, NodeCoupling, ObjectKind);
    RecTapeUnit(Uid &, Error &);
    RecTapeUnit(ArjunaName, Error &);

    ~RecTapeUnit();

    virtual Error   onLine();
    virtual Error   offLine();

    virtual Error   powerUp();
    virtual Error   powerDown();

    virtual Error   getStatus();
    virtual Error   getController();

    virtual Error   setBlockSize(int);

    virtual Error   rewindTape();
    virtual Error   eraseTape();
    virtual Error   ejectTape();
    virtual Error   retensionTape();
    virtual Error   loadTape();
    virtual Error   removeTape();

    virtual Error   dumpToTape();
    virtual Error   restoreFromTape();

    virtual Error   showBlockSize(int &);
    virtual Error   showTapeUnitState(TapeUnitState &);
    virtual Error   showTapeUnitKind(UnitKind &);
    virtual Error   showTapeMedia(TapeKind &);
    virtual Error   showUnitDescription(char * &);

    virtual Error   showLocation(char * &);
    virtual Error   showManufacturer(char * &);
    virtual Error   showPowerPoint(char * &);
    virtual Error   showPowerPhase(PowerPhase &);
    virtual Error   showContactName(char * &);
    virtual Error   showMaintenenceRef(char * &);
    virtual Error   showAssetRef(char * &);

    virtual Error   setLocation(char *);
    virtual Error   setManufacturer(char *);
    virtual Error   setPowerPoint(char *);
    virtual  Error    setPowerPhase(PowerPhase);
    virtual  Error    setContactName(char * );
    virtual  Error    setMaintenenceRef(char *);
    virtual  Error    setAssetRef(char *);

    virtual  Error    compensate(Error);
    virtual  Error    reportError(char *);
    virtual  Error    reportError(Error);

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

};
```

CLASS INTERFACE

RecTapeUnit(char * AN, char * ItsDeviceName, char * ResourceLocn, NodeCoupling ItsCoupling, ObjectKind ItsKind) creates and activates a recoverable disk server

RecTapeUnit(Uid & Uold, Error & res) RecTapeUnit(ArjunaName RN, Error & res) accesses (and activates) an existing tape unit server

~RecTapeUnit() terminates the tape unit

onLine(), offLine() activates (de-activates) the probe

powerUp (), powerDown() powers (unpowers) the tape unit by first attempting manual/automatic operations

loadTape(), removeTape() requests the human operator to load/remove the tape

rewindTape(), eraseTape(), retensionTape(), ejectTape() provided a tape is already loaded, rewind/erase/retension the tape

getController() obtains (automatically) the tape unit's configuration

dumpToTape() restoreFromTape() provided a tape is loaded, dump (restore) data

setBlockSize(int)

showBlockSize(int & itsBlocks) showTapeUnitState(TapeUnitState & itsState) showTapeUnitKind(UnitKind & itsKind) showTapeMedia(TapeKind & itsKind) showUnitDescription(char * & itsDecription) retrieves instance data within a read locked atomic action

compensate(Error theError)

reportError(char * theError) reportError(Error ErrorCode)

BUGS

none known

NOTES

SELF TEST

Provided the tape unit is already powered, any tape which is loaded is then ejected and the tape unit's configuration is obtained.

SEE ALSO

TapeUnit

C.5 RecDiskUnit

DESCRIPTION

RecDiskUnit defines a recoverable (external) disk unit and is derived from (unrecoverable) disk unit. Several methods are exported which power up (down) the device and obtain configuration information concerning the controller, physical geometry and partition map.

CLASS DEFINITION

```
class RecDiskUnit : public DiskUnit
{
public:

    RecDiskUnit(char *, char *, char *, NodeCoupling, ObjectKind);
    RecDiskUnit(Uid &, Error &);
    RecDiskUnit(ArjunaName, Error &);

    ~RecDiskUnit();

    virtual Error   onLine();
    virtual Error   offLine();

    virtual Error   powerUp();
    virtual Error   powerDown();

    virtual Error   getStatus();

    virtual Error   getController();
    virtual Error   getUsage();
    virtual Error   getPartitions();
    virtual Error   getGeometry();

    virtual Error   showController(DiskInterface &, short &, int &);
    virtual Error   showCylinders(unsigned short &, unsigned short &);
    virtual Error   showHeads(unsigned short &, unsigned short & );
    virtual Error   showSectors(unsigned short &, unsigned short &);
    virtual Error   showDimensions(long &, long &);

    virtual Error   showCapacity(Kbytes &, Kbytes &, Kbytes &);
    virtual Error   showLocation(char * &);
    virtual Error   showManufacturer(char * &);
    virtual Error   showPowerPoint(char * &);
    virtual Error   showPowerPhase(PowerPhase &);
    virtual Error   showContactName(char * &);
    virtual Error   showMaintenenceRef(char * &);
    virtual Error   showAssetRef(char * &);

    virtual Error   setLocation(char *);
    virtual Error   setManufacturer(char *);
    virtual Error   setPowerPoint(char *);
    virtual Error   setPowerPhase(PowerPhase);
    virtual Error   setContactName(char * );
    virtual Error   setMaintenenceRef(char *);
    virtual Error   setAssetRef(char *);

    virtual Error   compensate(Error);
    virtual Error   reportError(char *);
    virtual Error   reportError(Error);
    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;

};
```

INTERFACE

RecDiskUnit(char * AN, char * itsDeviceName, char * itsResourceLocn, NodeCoupling itsCoupling, ObjectKind itsKind) creates and activates a recoverable disk server

RecDiskUnit(Uid & Uold, Error & res) RecDiskUnit(ArjunaName RN, Error & res)

~RecDiskUnit() destructs and terminates the recoverable disk server

powerUp(), powerDown() powers up (down) the device by attempting manual then automatic variance

getControllers(), getPartitions(), getGeometry() obtains (automatically) configuration parameters

getUsage() obtains (automatically) disk usage, availability and total size

showController(DiskInterface & itsInterface, short & itsUnit, int & itsController) showDimensions(long & itsStart, long & itsBlocks) showCylinders(unsigned short & itsCylinders, unsigned short & itsAltCylinders) showHeads(unsigned short & itsHeads, unsigned short & itsOffset) showSectors(unsigned short & itsSectPerTrack, unsigned short & itsInterleave) showCapacity(Kbytes & itsSize, Kbytes & itsUsage, Kbytes & itsAvail) retrieves instance data within a read locked atomic action

reportError(char * theError) reportError(Error ErrorCode)

NOTES

BUGS

none known

SELF TEST

Once the disk device is powered, controller, usage etc. can be obtained automatically.

SEE ALSO

DiskUnit

C.6 RecSoftware

DESCRIPTION

Recoverable software classes are derived from the unrecoverable software class and provide fault tolerant control (monitoring) operations on the managed software distribution. Unlike the other recoverable layer classes, operation on software distributions (in particular building and installing new releases) is particularly slow and care must be taken in setting (client) remote procedure call timeout (and retry) values. A compromise must be achieved, having enough time to compile the distribution without waiting indefinitely for crashed servers.

CLASS DEFINITION

```
class RecSoftware : public Software
{
public:

    RecSoftware(char *, char *, char *, NodeCoupling, ObjectKind);
    RecSoftware(Uid &, Error &);
    RecSoftware(ArjunaName, Error &);

    ~RecSoftware();

    virtual Error   onLine();
    virtual Error   offLine();

    virtual Error   upGrade();
    virtual Error   downGrade();

    virtual Error   unpack();
    virtual Error   pack();

    virtual Error   makeAll();
    virtual Error   makeClean();

    virtual Error   makeInstall();

    virtual Error   getConfig();


    virtual Error   showOwner(char * &, char * &);

    virtual Error   showMode(mode_t &);
    virtual Error   showSize(off_t &);
    virtual Error   showModifyTime(time_t &);

    virtual Error   showSoftwareName(char *  &);
    virtual Error   showPathName(char *  &);
    virtual Error   showSoftwareState(SoftwareState &);

    virtual Error   compensate(Error);
    virtual Error   reportError(char *);
    virtual Error   reportError(Error);

    virtual Boolean restore_state (ObjectState & , ObjectType);
    virtual Boolean save_state (ObjectState & , ObjectType);
    virtual const TypeName type () const ;
};
```

INTERFACE

RecSoftware(char *, char *, char *, NodeCoupling, ObjectKind) creates a software server from scratch

RecSoftware(Uid &, Error &) RecSoftware(ArjunaName, Error &) accesses a persistent software class using Uid and ArjunaNames

~RecSoftware() terminates the software server.

onLine() offLine() turns the probe on (off) line

upGrade() downGrade() upgrades (downgrades) software automatically

unpack() pack() unpacks (packs) the software distribution by attempting Zoo and Tar utilities.

makeAll() makeClean() makeInstall() builds, cleans and installs using "make" program.

showOwner(char * & itsOwner, char * & itsGroup) obtains the software's owner and group.

showMode(mode_t & itsMode) obtains file permissions.

showSize(off_t & itsSize) showModifyTime(time_t & itsTime) obtains the software object's size and last modify time.

NOTES

SELF TEST

Once RecSoftware is created configuration data is obtained.

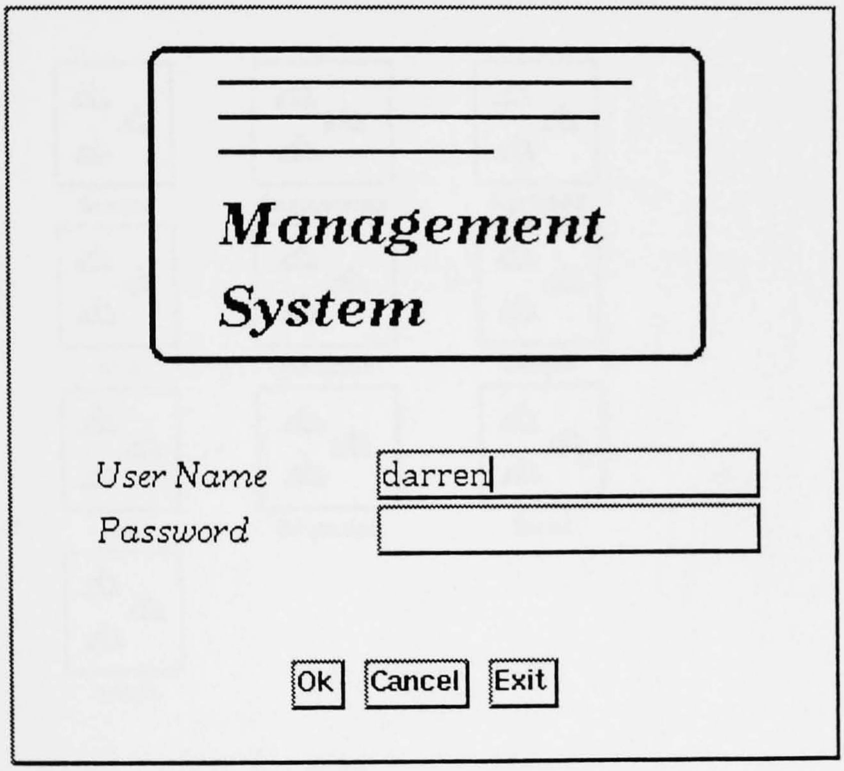
SEE ALSO

Software

Appendix D
APPLICATION LAYER

The following appendix contains several screen dumps of the management information workbench. These include the logon sheet, views of the organisation (faculty, departments and research groups), resources and property sheets.

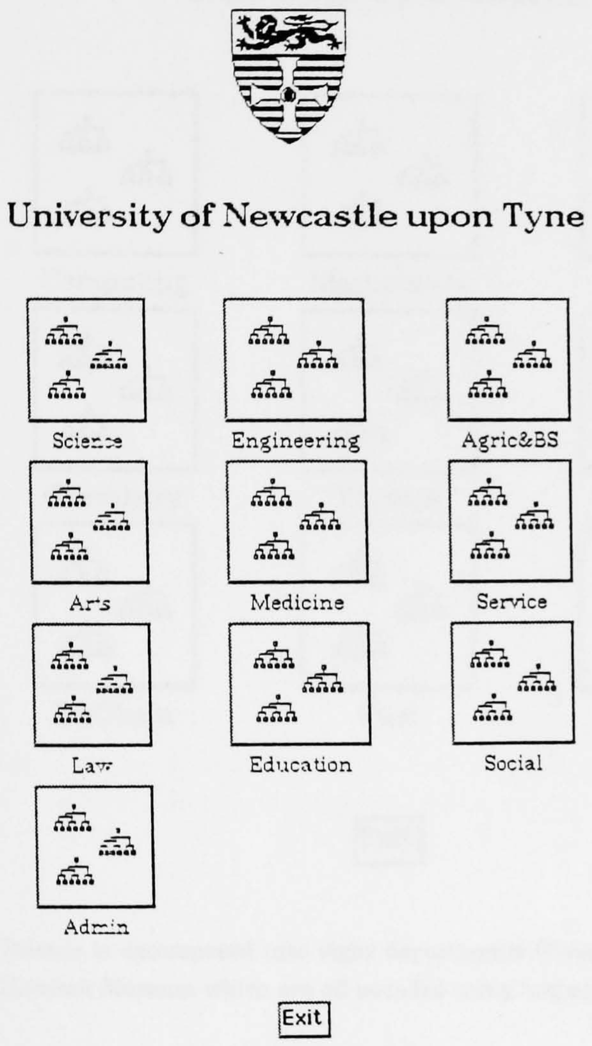
D.1 LOGON SHEET



User Name: darren
Password:

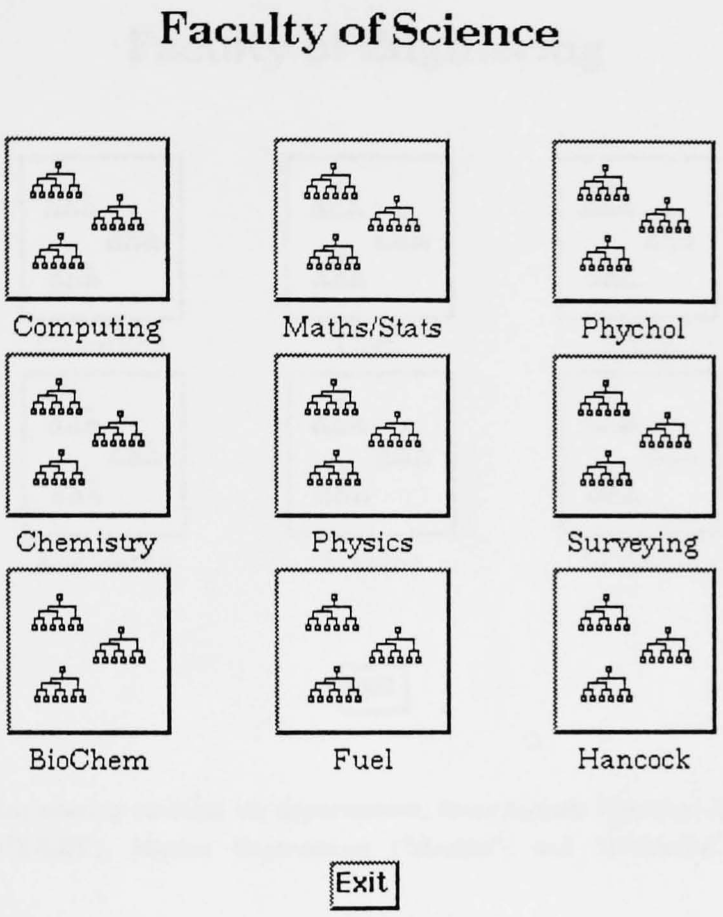
The workbench logon sheet simply contains a bitmap "logo" and user name / password fields. Three buttons are provided to enable the user (management agent) to proceed ("ok"), cancel or exit.

D.2 UNIVERSITY STRUCTURE



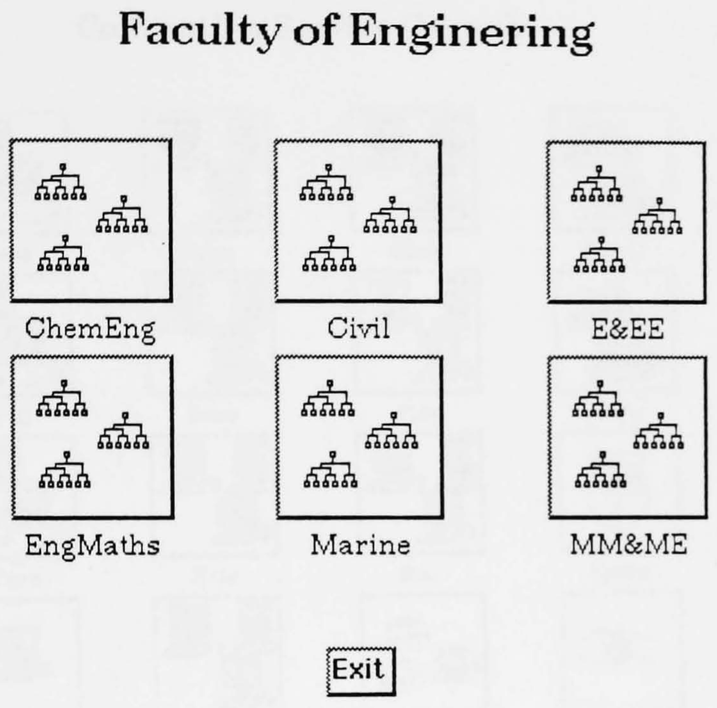
Newcastle University's "view" encodes the academic faculty structure into a series of "organisational domain" icons. These include the Faculty of Science, Engineering and Computing Service, which are shown below.

D.3 FACULTY OF SCIENCE



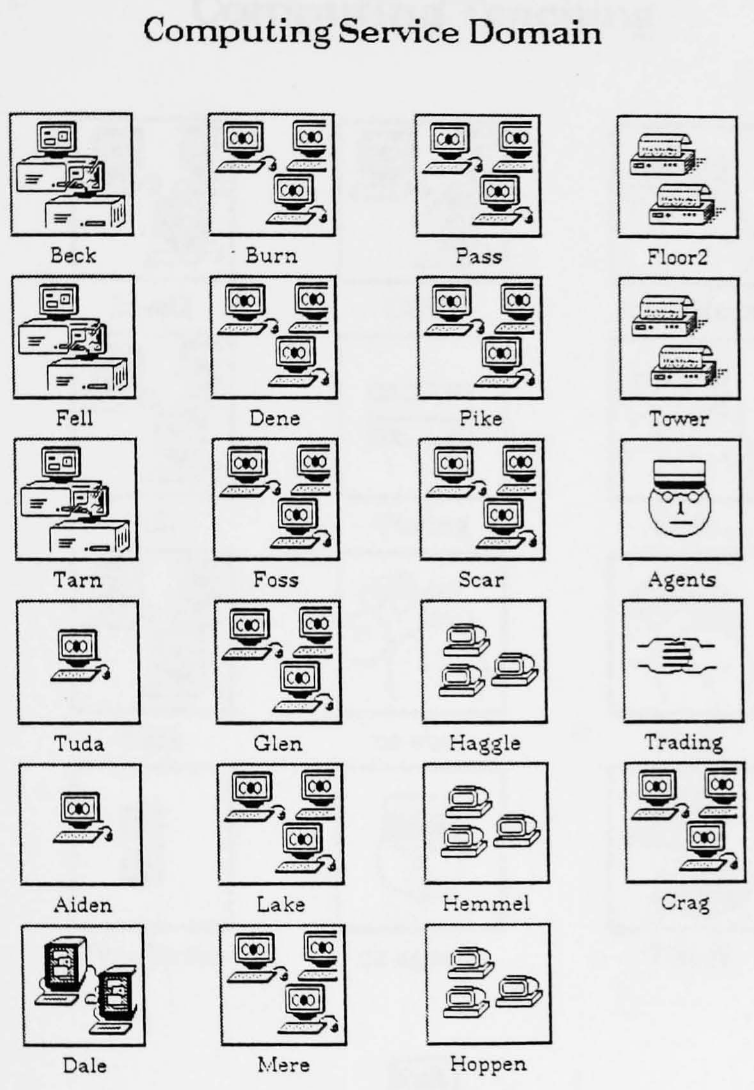
Faculty of Science is decomposed into eight departments (Computing Science, Chemistry etc.) and the Hancock Museum which are all encoded using "organisational domain" icons.

D.3.1 FACULTY OF ENGINEERING



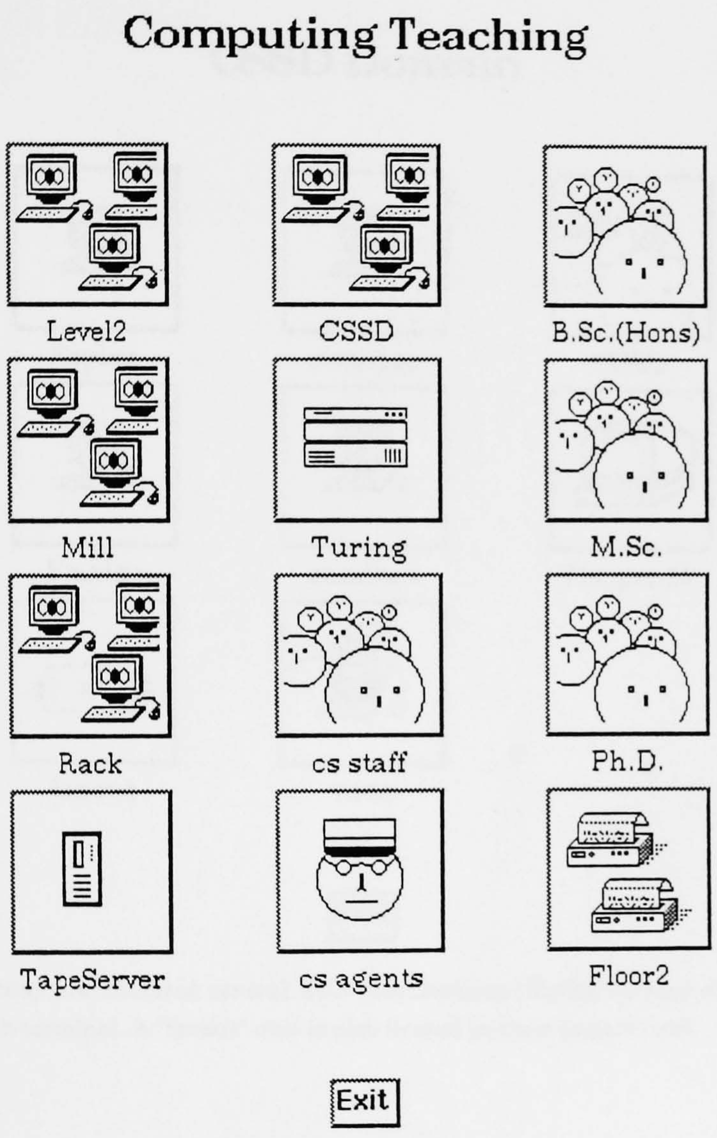
Faculty of Engineering contains six departments, these include Electrical and Electronic Engineering ("E&EE"), Marine Engineering ("Marine") and Mechanical Engineering ("MM&ME").

D.3.2 COMPUTING SERVICE



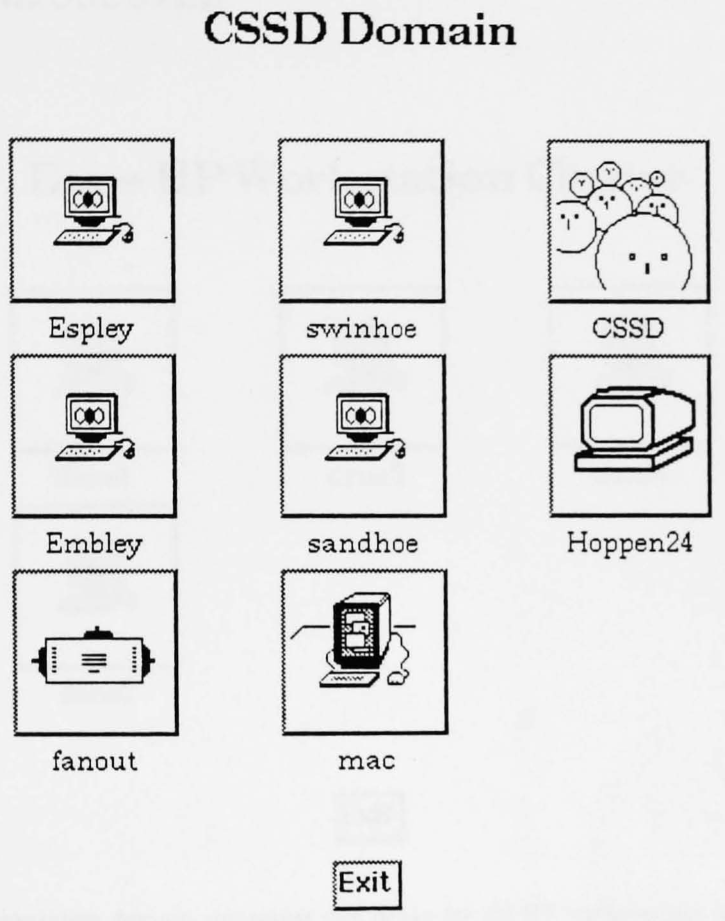
Computing Services manage over one hundred resources located across the university campus. These include P.C. clusters ("Beck", "Fell" and "Tarn"), a MAC cluster ("Dale"), H.P. workstations ("Burn", "Dene" etc.) and sixteen SUN workstations ("cragg"). Three Encore "annex" terminal clusters ("Haggle", "Hemmel" and "Hoppen") and two printer clusters ("floor2" and "Tower") are also managed. Clicking on each of these "resource domain" icons reveals the contents of the particular domain — which are shown later in this appendix.

D.3.3 COMPUTING (TEACHING)



Clicking on the Computing (teaching) icon (in the "Department of Computing Science" domain) reveals the "Computing Teaching" domain. This includes teaching equipment (workstations clusters located in the "Mill" and "Rack" laboratories, a printer cluster in the second floor etc.), user groups (staff, students etc.) and the M.Sc. "Computing Software and Systems Design" group. The CSSD domain is shown below.

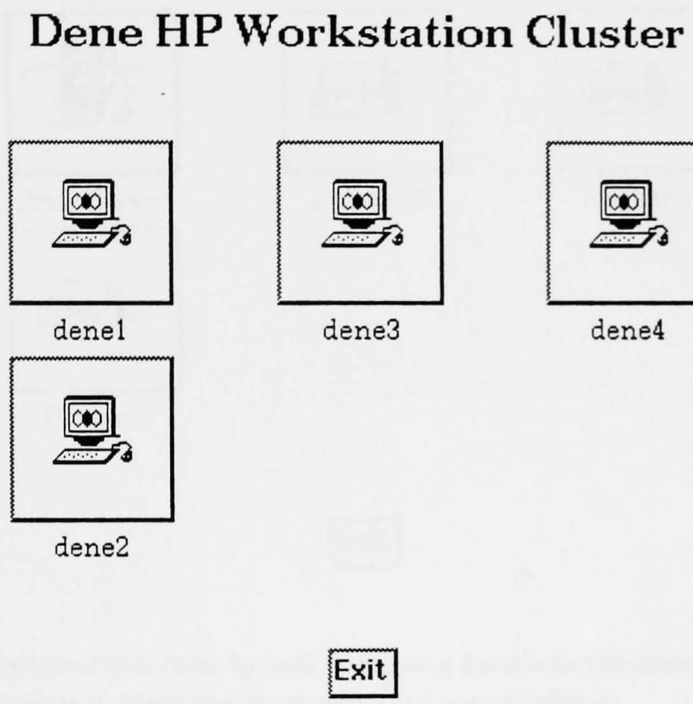
D.3.4 CSSD GROUP



The CSSD group are allocated several SUN workstations (Espley, Embley etc.), an Apple Mac and a dumb terminal. A "fanout" unit is also located in their project room.

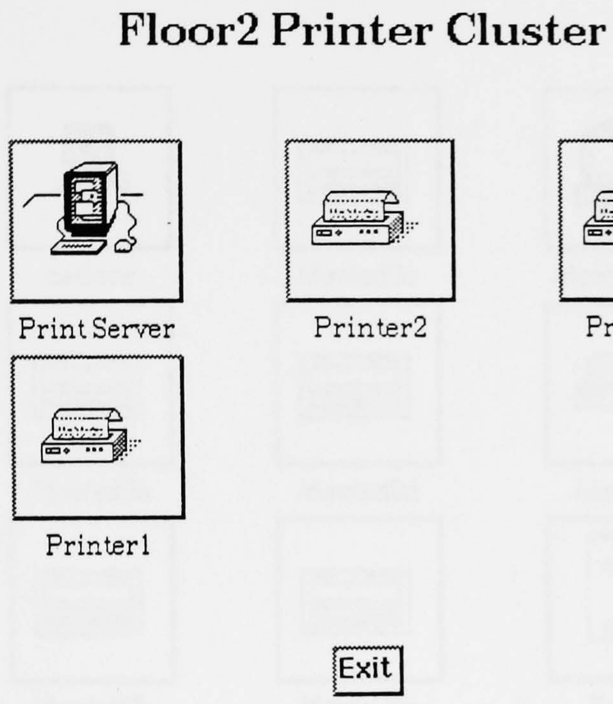
D.4 RESOURCE DOMAINS

D.4.1 DENE CLUSTER



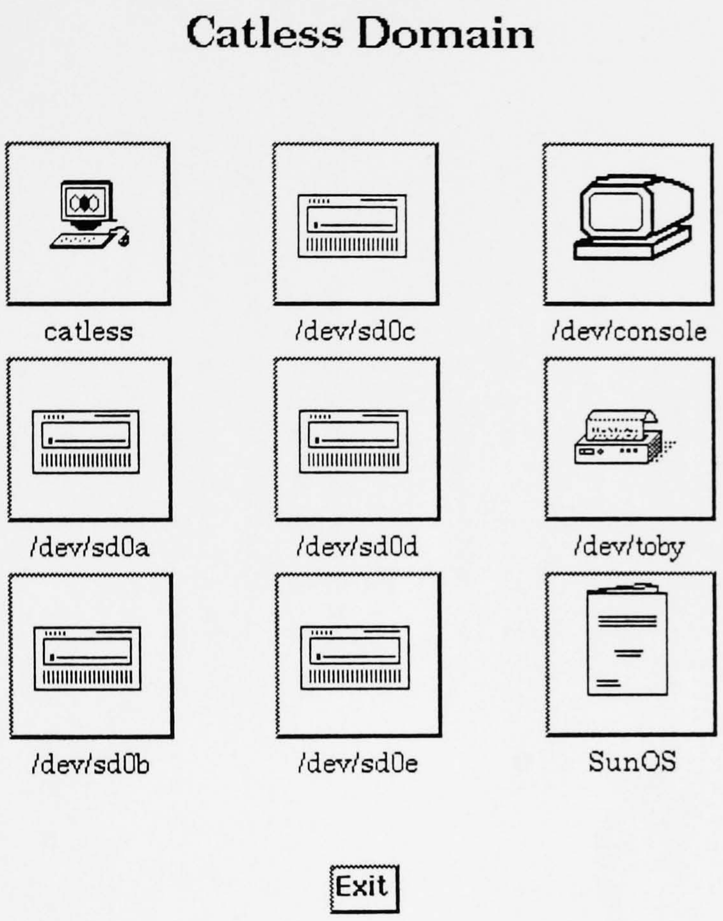
The Dene workstation domain comprises two dozen (or so) HP workstations and a printer. For the sake of simplicity, only four workstations are shown in the screen dump.

D.4.2 PRINTER CLUSTER



The “floor2” printer cluster (used by both Computing Service and Department of Computing Science) comprises an Apple Mac. print server and several printers.

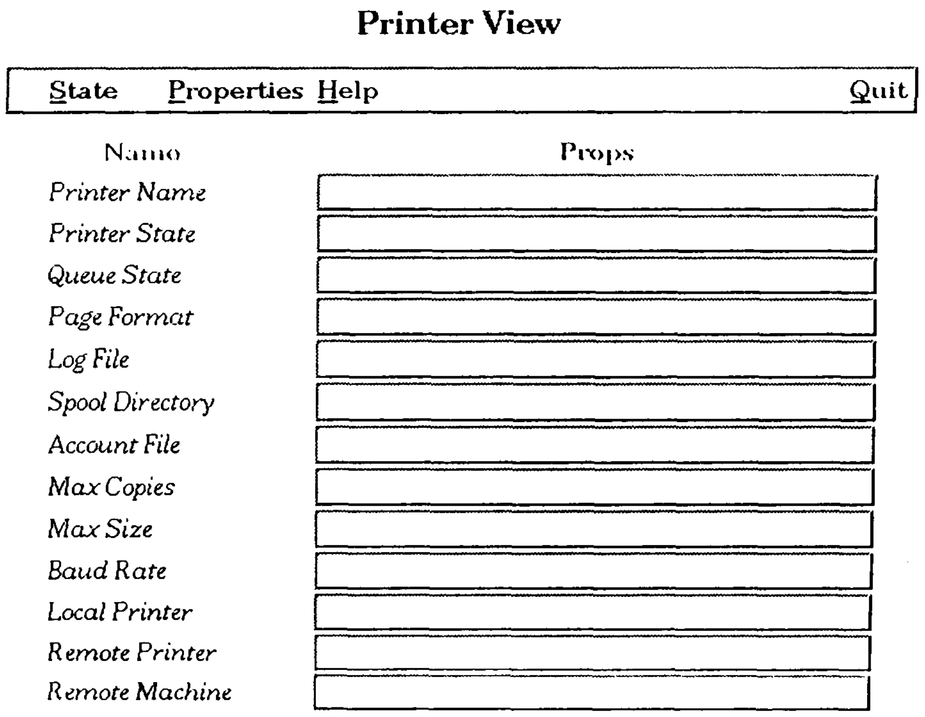
D.4.3 SUN4 WORKSTATION



The above screen dump shows Catless's configuration (before being upgraded to Solaris). This includes several disk partitions, console, printer and documentation.

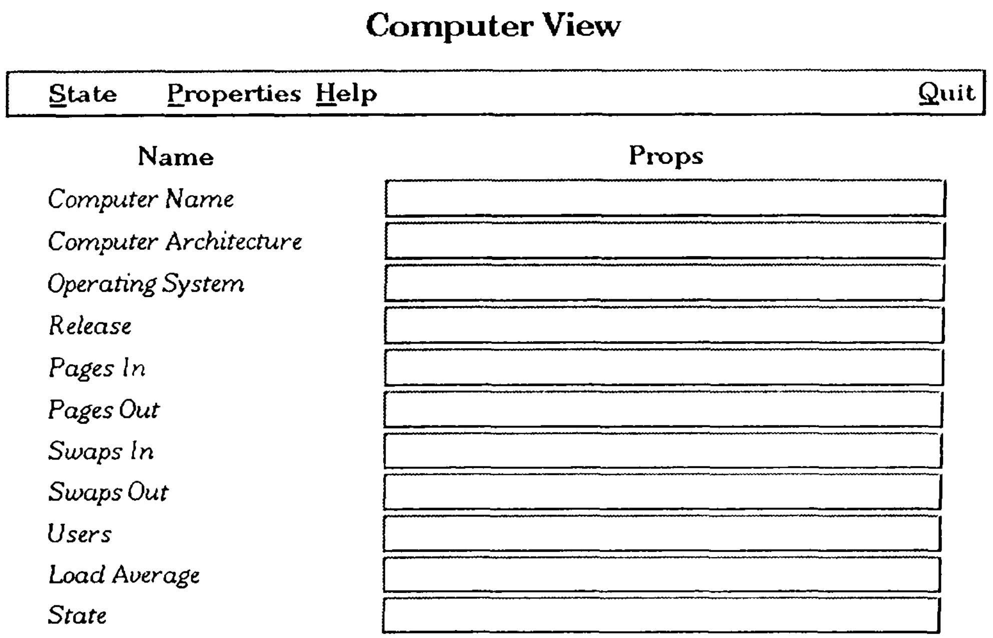
D.5 PROPERTY SHEETS

D.5.1 PRINTER CONFIGURATION VIEW



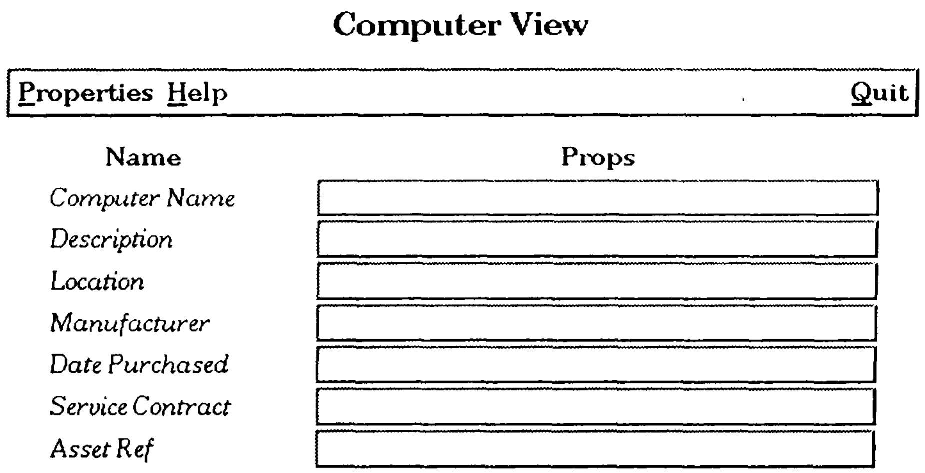
Name	Props
Printer Name	
Printer State	
Queue State	
Page Format	
Log File	
Spool Directory	
Account File	
Max Copies	
Max Size	
Baud Rate	
Local Printer	
Remote Printer	
Remote Machine	

D.5.2 COMPUTER CONFIGURATION VIEW



Name	Props
Computer Name	
Computer Architecture	
Operating System	
Release	
Pages In	
Pages Out	
Swaps In	
Swaps Out	
Users	
Load Average	
State	

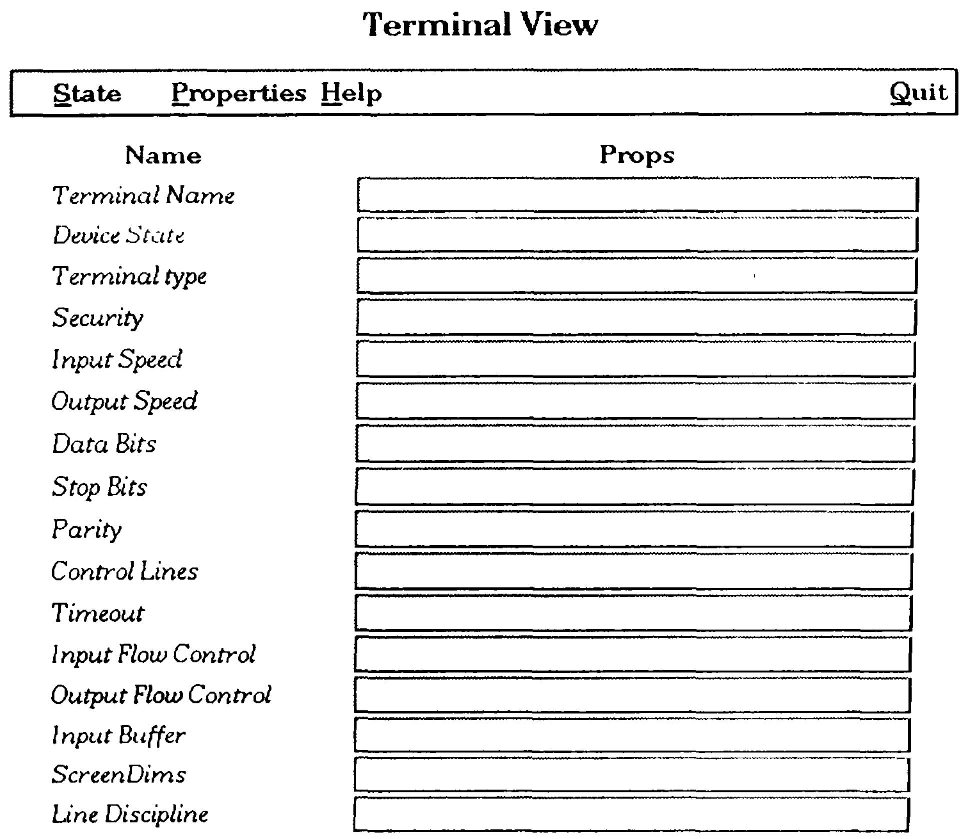
D.5.3 COMPUTER ASSET VIEW



Computer View

Name	Props
Computer Name	
Description	
Location	
Manufacturer	
Date Purchased	
Service Contract	
Asset Ref	

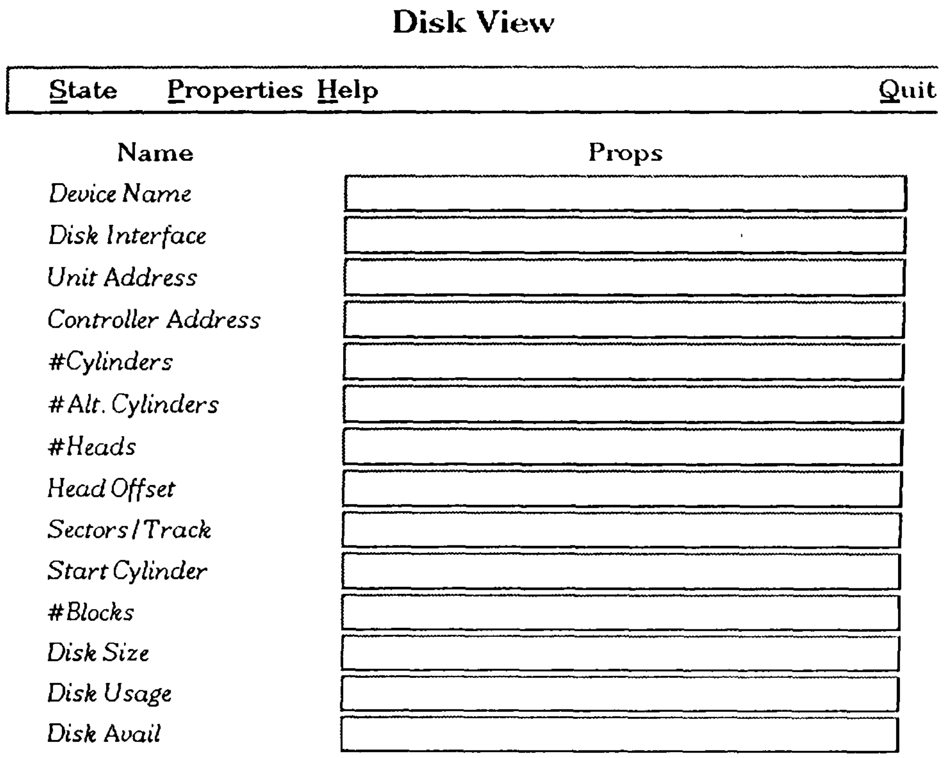
D.5.4 TERMINAL CONFIGURATION VIEW



Terminal View

Name	Props
Terminal Name	
Device State	
Terminal type	
Security	
Input Speed	
Output Speed	
Data Bits	
Stop Bits	
Parity	
Control Lines	
Timeout	
Input Flow Control	
Output Flow Control	
Input Buffer	
ScreenDims	
Line Discipline	

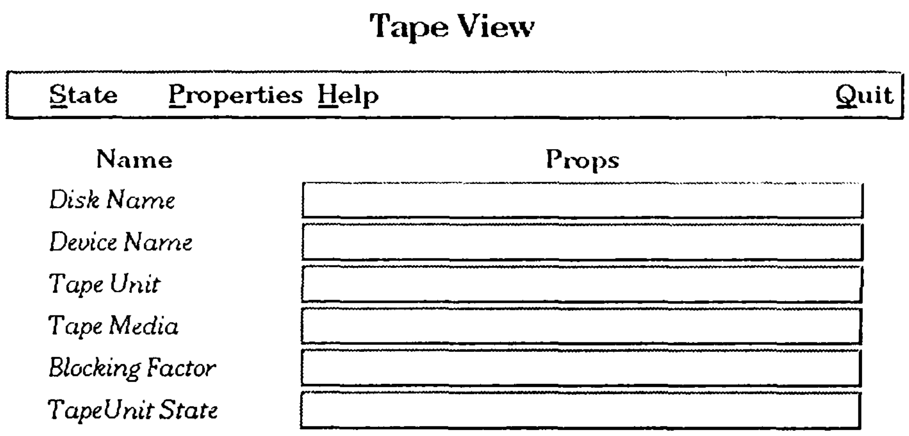
D.5.5 DISK CONFIGURATION VIEW


Disk View

State Properties Help Quit

Name	Props
Device Name	
Disk Interface	
Unit Address	
Controller Address	
#Cylinders	
#Alt. Cylinders	
#Heads	
Head Offset	
Sectors/Track	
Start Cylinder	
#Blocks	
Disk Size	
Disk Usage	
Disk Avail	

D.5.6 TAPE CONFIGURATION VIEW



Tape View

State Properties Help Quit

Name	Props
Disk Name	
Device Name	
Tape Unit	
Tape Media	
Blocking Factor	
TapeUnit State	

ACRONYMS

ANSA — Advanced Networked Systems Architecture

ASN.1 — Abstract Syntax Notation 1

CMIP — Common Management Information Protocol

CMIS — Common Management Information Service

FDDI — Fibre Distribution Data Interface

HEMS — High level Entity Management System

ISO — International Standards Organisation

SGMP — Simple Gateway Management Protocol

SIGDSM — Special Interest Group on Distributed Systems Management

SNMP — Simple Network Management Protocol

SSMP — Simple Screen Management Protocol

TOBIAS — Tools for Object Based Integration and Administration of Systems

GLOSSARY

Agent — An abstract entity used in the structural view of the organisation to perform management tasks.

Atomic Action — A recovery region possessing mechanisms for reliable management of shared data: failure atomicity, serialization and permanence of effect.

Change Management — The controlled and regulated manner of altering the configuration of a distributed system in order to minimise user disruption.

Configuration Management — The management of the system topology, its interconnections and resources.

Contract — A formal agreement between several parties, agents and the organisation.

Dependability — The trustworthiness of a component such that reliance can be justably be placed on the *service* it delivers.

Domain — A multiple set of components which share a common attribute or are managed by the same management agents. Much controversy exists surrounding the definition of "domain" and whether domains can overlap. In the model of management, domains can overlap (both partially or completely) and interact (trade) with other domains.

Delegation — The means by which one management agent allows or instructs another agent to perform a particular activity. Delegation is particularly used in the modelling of subcontracts.

Error — That part of a system state which is liable to lead to failure.

External Resource — A physical entity used by the organisation as part of its day to day operation. External resources, such as computing components are modelled and controlled by managed resource controllers.

Failure — Any service provided by the system which no longer complies with its specification.

Fault — The cause of an error

Fault Management — The management of isolating, diagnosing and rectifying erroneous system states.

Functional Role — The jobs performed by agents in a particular role.

Hat — A metaphor describing the adoption of roles by agents.

Managed Object — The management model entity representing a real world item.

Managed Resource — A resource which is controlled and monitored by agents in the installation.

Management — To organise, regulate and be in charge of an enterprise.

Message — Communications medium between system components.

Policy — The business objectives of the organisation and the effects of the system on the performance of the organisation.

Reliability — A measure of continuous correct delivery.

Service — The system behaviour as perceived by the user.

Structural Role — The responsibilities and obligations of agents.

System — A collection of components connected by a communications medium which can co-operate to perform some computation.

Unconnected Resource — A managed resource which is not physically attached to a communications medium.

Unreachable Resource — As contrasted with unconnected resources, an unconnected resource is unreachable resource is connected to the communications topology but cannot be accessed due to communications failure.

TRADEMARKS

References

Alsberg76a.

P.A. Alsberg and John D. Day, "A Principle for Resilient Sharing of Distributed Resources," *Proc 2nd Symp on Software Engineering*, 1976.

ANSA90a.

ANSA, *ANSA Reference Manual*, A, ANSA, 1990.

Arms88a.

Caroline Arms, *Campus Networking Strategies*, p. 4, Dec Press, USA, 1988.

Arms88b.

Caroline Arms, *Campus Networking Strategies*, pp. 67 - 89, Dec Press, USA, 1988.

Arms88c.

Caroline Arms, *Campus Networking Strategies*, p. 1, Dec Press, USA, 1988.

Arms88d.

Caroline Arms, *Campus Networking Strategies*, p. 2, Dec Press, USA, 1988.

Arms88e.

Caroline Arms, *Campus Networking Strategies*, p. 87, Dec Press, USA, 1988.

Austin62a.

J.L. Austin, *How to do things with words*, Clarendon Press, 1962.

Bedford-Robers91a.

James Bedford-Robers, *Concepts from Pythagoras*, HP Laboratories, Bristol, February 1991. Presented at Policy Workshop, Imperial College (1991)

Bernstein87a.

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

Blyth95a.

Andy Blyth, *Enterprise Modelling and its Application to Organisational Requirements, Capture and Definition*, University of Newcastle upon Tyne, 1995. (Ph.D. Thesis)

Bosak88a.

L. Bosak and C. Hedrick, "Problems in Large LANs," *IEEE Network*, vol. 2, no. 1, pp. 49 - 56, IEEE, January 1988.

Bowers94a.

John Bowers, James Pycock, Tom Rodden, and Graham Dean, "Running the Network: Supporting Co-operative Systems," *Information Technology and People*, vol. 7, no. 2, pp. 7 - 28, 1994.

Burnett87a.

Christopher Burnett, "Strategic aspects of migration to OSI," *International Open Systems 1987*, vol. 1, pp. 1 - 16, Online Publications, London, 1987. ISBN

0-86353-083-4

Buzato92a.
L.E. Buzato and A. Calsavara, "Stabilis: A Case Study in Writing Fault-Tolerant Distributed Applications using Persistent Objects," in *Proc. 5th International Workshop in Persistent Objects*, 1992. Held at San Miniato, Italy. September 1-4 1992

Case88a.
Jeffrey D. Case, James R. Davin, Mark S. Fedor, and Martin L. Schoffstall, "Introduction to the Simple Gateway Monitoring Protocol," *IEEE Network*, pp. 43 - 49, IEEE, March 1988.

Champine91a.
George A. Champine, *M.I.T Project Athena: A Model for Distributed Campus Computing*, Digital Press, USA, 1991.

Checkland86a.
Peter Checkland, *Systems Thinking, Systems Practice*, Wiley, 1986.

Danish94a.
Tawfig Danish, *A Knowledge-Based System for Disaster Prevention in Computer Systems*, University of Newcastle upon Tyne, 1994. (Ph.D thesis)

Date86a.
C.J. Date, *An Introduction to Database Systems*, 1, Addison Wesley, 1986.

Dean92a.
Graham Dean, David Hutchinson, and Ian Sommerville, "Distributed System Management as a Group Activity," Netman Report, University of Lancaster, December 1992.

Dean95a.
Graham Dean, "Configuration Language Support for Software Installation," PCL Report, University of Lancaster, 1995.

DellaFera88a.
DellaFera, "The Zephyr Notification System," in *Usenix Conference Proceedings*, USENIX, Winter 1988.

DeRemer75a.
F. DeRemer and H. Kron, "Programming in the Large versus Programming in the Small," in *Proceeding Conference on Reliable Systems*, pp. 124 - 121, 1975.

DIGITAL89a.
DIGITAL, *Enterprise Management Architecture (General Description)*, Digital, 1989.

Dixon88a.
G.N. Dixon, *Object Management for Persistence and Recoverability*, University of Newcastle upon Tyne, Newcastle, UK, December 1988. (Ph.D. Thesis)

Dowson87a.

M. Dowson, "An Integrated Project Support Environment," *SIGPLAN Notices*, vol. 22, no. 1, pp. 27 - 33, January 1987. Palo Alto, CA, 9 - 11 December 1986

Duncan81a.

V.J. Duncan, *Organisational Behaviour,* Houghton, 1981.

Dyer88a.

S.P. Dyer, "Hesiod," in *Usenix Conference Proceedings*, Winter 1988.

ECMA88a.

ECMA, "Standard ECMA - Document Print Service Description and Print Access Protocol Specification," Group TC35-TG5, European Computer Manufactures Assocation, September 1988.

Encore87a.

Encore, "Annex ii Hardware Installation Guide," Encore Manual, Encore, 1987.

Encore87b.

Encore, "Annex Network Administrators Guide," Encore Manual, Encore, 1987.

Greenweld86a.

M. Greenweld and J. Sciver, *Remote Virtual Disk Protocol Specification,* MIT, 1986.

Handspiker89a.

B. Handspiker, R. Hart, and M. Roman, *The Athena Palladium Print System,* MIT, Febuary 1989.

Harbison91a.

Samuel P. Harbison and Guy L. Steel Jr, *A C Reference Manual,* Prentice Hall Software Series, 1991.

Harrison92a.

Helen E. Harrison, "So Many Workstations, So Little Time," in *Proc. LISA VI*, pp. 79 - 86, USENIX, October 1992.

HMSO84a.

HMSO, *Data Protection Act 1984,* HMSO, 1984.

HMSO90a.

HMSO, *Computer Misuse Act 1990,* HMSO, 1990.

Ingham92a.

Dave Ingham, *Delayline: A Wide Area Network Simulation Tool,* University of Newcastle upon Tyne, 1992. (M.Sc. Dissertation)

INGRES85a.

INGRES, *The INGRES Papers: The Anatomy of a Relational Database Management System,* Addison Wesley, 1985.

ISO87a.

ISO, "Systems Management: Overview," *ISO/IEC JTC1/SC21*, ISO, 1987.

(working draft DP9595/2)

ISO88a.

ISO, "OSI-Basic Reference Model: Part 4 Management Framework," *ISO/IEC/DIS 7498-4*, March 1988.

ISO88b.

ISO, "Draft British Standard for OSI: The Directory, Part 1: Overview of Components, models and service," *ISO/IEC/DIS 9594-1*, July 1988.

ISO88c.

ISO, "Revised Text of 2nd DP 9596-2, Information Processing Systems - OSI - Management Information Processing Specification: Part 2 CMIP," *ISO/IEC JTC 1/SC 21 N 3070*, October 1988.

ISO88d.

ISO, "Revised Text of 2nd DP 9596-2, Information Processing Systems - OSI - Management Information Processing Specification: Part 1 CMIS," *ISO/IEC JTC 1/SC 21 N 3069*, October 1988.

Kar88a.

Gautam Kar, Brendan Madden, and Robert S. Gilbert, "Heuristic layout Algorithms for Network Management Presentation Services," *IEEE Network*, pp. 29 - 36, IEEE, November 1988.

King83a.

John Leslie King, "Centralized versus Decentralized Computing: Organisational Considerations and Management Options," *Computing Surveys*, vol. 15, no. 4, pp. 321 - 349, ACM, December 1983.

Kotter78a.

J.P. Kotter, *Organisational Dynamics: Diagnosis and Intervention,* Addision Wesley, 1978.

Kramer88a.

Jeff Kramer and Jeff Magee, "A Model for Change Management," *IEEE Dist Comp Sys in the 1990's*, IEEE, Hong Kong, September 1988.

Kramer93a.

Jeff Kramer, Jeff Magee, Keng Ng, and Morris Sloman, "The System Architect's Assistant for Design and Construction of Distributed Systems," in *Proceeding 4th IEEE Workshop on Future Trends of Distributed Computer Systems*, pp. 282 - 290, IEEE, September 1993.

Lamport82a.

L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM TOPLAS*, vol. 4, no. 3, ACM, July 1982.

Lampson79a.
B.W. Lampson and H.E. Sturgis, *Crash Recovery in a Distributed Data Storage System,* Xerox Parc, April 1979.

Lampson81a.
B.W. Lampson, "Atomic Transactions," in *Distributed Systems - Architecture and Implementation*, ed. B.W. Lampson, pp. 246-264, Springer-Verlag, 1981.

Lee90a.
P.A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice,* Springer-Verlag, New York, 1990.

Leffler86a.
Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller, and Chris Torek, "An Advanced 4.3 BSD IPC Tutorial," *PS 1: 8*, UNIX, February 1986.

Lewine91a.
Donald Lewine, *POSIX Programmer's Guide: Writing Portable UNIX Programs,* O'Reilly and Associates, Inc., 1991.

Lippman89a.
Stanley B. Lippman, *C++ Primer,* Addison Wesley, 1989.

Little91a.
Mark Little, *Object Replication in Distributed Systems,* University of Newcastle upon Tyne, September 1991. (Ph.D thesis)

Lomet77a.
D.B. Lomet, "Process Structuring, Synchronisation and Recovery using Atomic Actions," *ACM SIGPLAN Notices*, vol. 13, no. 3, March 1977.

Magee89a.
Jeff Magee, Jeff Kramer, Morris Sloman, and Naranker Dualy, "Constructing Distributed Systems in CONIC," *IEEE Transactions on Software Engineering*, vol. SE-15, no. 6, pp. 663-675, IEEE, June 1989.

Magee94a.
Jeff Magee, Naranker Dulay, and Jeff Kramer, "A Constructive Development Enviroment for Parallel and Distributed Programs," in *Proceedings of the IEEE 2nd International Workshop on Configurable Distributed Systems*, IEEE, March 1994.

Marshall80a.
Lindsay Marshall, *An Error Scheme for Concurrent Processes,* University of Newcastle upon Tyne, August 1980. (Ph.D. Thesis)

Marshall90a.
Lindsay Marshall, "Managing Management: The TOBIAS Approach," in *Esprit 1990 Conference*, 1990.

Marshall93a.

Lindsay Marshall, "Representing Management Policy using Contract Objects," in *Proc. IEEE 1st Int. Workshop on Systems Mananagement*, IEEE, Los Angeles, USA, 1993.

Mauro89a.

T. Mauro, *An Overview of the Andrew File System,* Transarc Corportation, 1989.

McCue92a.

Dan. L. McCue, *Selective Transparency in Distributed Transacrtion Processing,* University of Newcastle upon Tyne, April 1992. (Ph.D. Thesis)

McDermott84a.

J. McDermott, "R1 Revisited: Four Years in the Trenches," *AI Magazine*, pp. 21 - 32, Fall 1984.

Metcalf76a.

Robert A. Metcalf and David R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM*, vol. 19, no. 7, pp. 395 - 404, ACM, July 1976.

Miller87a.

S.P. Miller, B.C. Neuman, J.I. Schiller, and J.H. Saltzer, "Kerberos Authentication and Authorisation System," *Project Athena Technical Plan*, M.I.T., Massachusetts, December 1987.

Moffett92a.

Jonathan D. Moffett and Morris S. Sloman, "Policy Hierarchies for Distributed Systems Management," *Domino Report*, no. Arch/IC/6, Imperial College, London, 24 June 1992.

Needham78a.

R.M. Needham and M.D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *CACM*, vol. 21, pp. 993 - 999, December 1978.

Nelson81a.

B.J. Nelson, "Remote Procedure Call," *Xerox CLS-81-9*, Xerox, California, 1981.

ORDIT89a.

ORDIT, *Draft Technical annex,* ESPRIT, 1989.

OSF90a.

OSF, *Distributed Management Environment RFT,* Open Software Foundation, 1990.

OSF92a.

OSF, *EIDIS Workshop on Distributed Systems Management,* EIDIS, July 1992.

Ousterhout94a.

John K. Ousterhout, *Tcl and the TK Toolkit,* Addison Wesley, 1994.

Parrington95a.

G.D. Parrington, "A Stub Generation System for C++," *Computing Systems* -

journal for the USENIX Association, no. 8, pp. 135 - 170, University of Newcastle upon Tyne, May 1995.

Parrington95b.
Graham Parrington, Santosh Shrivastava, Stuart Wheater, and Mark Little, "The Design and Implementation of Arjuna," *USENIX Computer Systems*, 1995.

Partridge87a.
C. Partridge and G. Trewitt, "The High-Level Entity Management Protocol (HEMP)," *RFC*, no. 1022, October 1987.

Partridge88a.
Craig Partridge and Glenn Trewitt, "The High Level Entity Management System (HEMS)," *IEEE Network*, vol. 2, no. 2, pp. 37 - 42, IEEE, March 1988.

POSIX90a.
POSIX, "Draft Guide to POSIX Open Systems Environments," P1003.0/D9, IEEE Computer Soc., September 1990.

POSIX93a.
POSIX, "System Administration Interface," *POSIX 1003.7.2/D8.1, 1993*, IEEE, February 1993. (IEEE Draft Standard)

Press86a.
Willian H. Press, Brian P. Flannery, Saul A. Teukolsky, and Willian T. Vetterling, *Numerical Recipes - The Art of Scientific Computing*, Cambridge University Press, 1986.

Raeburn89a.
K. Raeburn, "DISCUSS: An Electronic Conferencing System for a Distributed Computing Environment," in *USENIX Conference Proceedings*, Winter 1989.

Randell75a.
Brian Randell, "System Structure for Software Fault Tolerence," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220 - 232, IEEE, June 1975.

Robinson88a.
D.C. Robinson and M.S. Sloman, "Domain Based Access Control for Distributed Computer Systemsc," *Software Engineering Journal*, vol. September 1988, pp. 161 - 170, IEE, 1988.

Rosenstein88a.
Mark A. Rosenstein, Daniel E. Geer jr, and Peter J. Levine, "The Athena Management System," in *USENIX Conference Proceedings*, pp. 203 - 212, USENIX Association, Dallas, Texas, February 9 - 12 1988.

Ross77a.
D. T. Ross, "Guest Editorial - Reflections on Requirement," *Transactions on Software Engineering*, vol. SE 3, no. 1, pp. 2 - 5, IEEE, 1977.

Rubira94a.

Cecilia M.F. Rubira, *Classification and Structuring of States and Behaviour in Object-Oriented Systems,* University of Newcastle upon Tyne, September 1994. (Ph.D. Thesis)

Rubira-Calsavara94a.

C.M.F. Rubira-Calsavara and R.J. Stroud., "Forward and Backward Error Recovery in C++.," *Object Oriented Systems*, no. 1, pp. 61-85, Chapman Hall, 1994.

Schein80a.

E.H. Schein, *Organisational Psychology,* Prentice Hall, 1980.

Schiefer86a.

R.M. Schiefer and J. Gettys, "The X Windows System," *ACM Trans on Graphics*, vol. 5, no. 2, pp. 79 - 109, ACM, April 1986.

Schoffstall89a.

M. Schoffstall, C. Davin, M. Fedor, and J. Case, "SNMP Over Ethernet," *RFC*, no. 1089, February 1989.

Schutz70a.

A. Schutz, "The Problem of Rationality in the Social World," in *Socological Theory and Philosphical Analysis*, ed. D. Emmet and A. MacIntyre, Macmillian, 1970.

Searle69a.

J.R. Searle, *Speech Acts - An Essay in the Philosophy of Language,* Cambridge University Press, 1969.

Sechrest86a.

Stuart Sechrest, "An Introductory 4.3 BSD IPC Turorial," *PS 1: 7*, UNIX, October 1986.

Shrivastava91a.

S.K. Shrivastava, G.N. Dixon, and G.D. Parrington, "An Overview of Arjuna - A Programming System for Reliable Distributed Computing," *IEEE Software*, IEEE, 1991.

Sloman87a.

Morris Sloman, "Distributed Systems Management," *Issues in LAN Management*, pp. 15 - 46, North Holland, July 1987.

Sloman89a.

Morris Sloman and Jonathan Moffett, "Managing Distributed Systems," Domino AI/IC/I, Imperial College, London, 25 September 1989.

Sommerville89a.

Ian Sommerville and R. Thomson, "An Approach to the Support of Software Revolution," *Computer Journal*, vol. 32, no. 5, BCS, 1989.

Sommerville95a.

Ian Sommerville and Graham Dean, "PCL: A Configuration Language for Modelling Evolving System Architectures," PCL Report, University of Lancaster, 1995.

Spector83a.

A.Z. Spector and P.M. Schwartz, "Transactions: A Construct for Reliable Distributed Computing," *ACM Operating Systems Review*, vol. 17, no. 2, April 1983.

Spivey89a.

J.M. Spivey, *The Z notation: A Reference Manual,* Prentice Hall, 1989.

Steedman90a.

Douglas Steedman, in *Abstract Syntax Notation 1,* Technology Appraisal Ltd, October 1990.

Stenning86a.

V. Stenning, "An Introduction to ISTAR," in *Software Engineering Environments*, ed. I. Sommerville, pp. 1 - 22, 1986.

Stern92a.

H. Stern, *Managing NFS and NIS,* O'Rieley, 1992.

Stroud94a.

R.J. Stroud and Z. Wu, "Using Meta-Objects to Adapt to Persistent Object System to Meet Application Needs," in *Proc. SIGOPS European Workshop*, 1994.

Stroud95a.

R.J. Stroud and Z. Wu, "Using Meta-Objects Protocols to Implement Atomic Data Types," in *Proc. ECOOP 95*, vol. 952, pp. 168 - 189, Springer-Verlag, 1995. 3-540-60160-0

Sun87a.

Sun, *NFS Protocol Specification and Service Manual Revision A,* Sun, 1987.

Sventek94a.

Joseph S. Sventek, "Distributed Objects as a Legacy Integration Mechanism," in *Proc. Systems Integration and Structuring*, ed. Brian Randell, University of Newcastle upon Tyne, September 1994.

Tao95a.

S. Tao, P.D. Ezhilchelvan, and S.K. Shrivastava, "Focused Fault Injection Testing of Software Implemented Fault Tolerance Mechanisms of Voltan TMR Nodes," *Journal of Distributed System Engineering*, vol. 1, no. 2, BCS and IEE, 1995.

TOBIAS89a.

TOBIAS, *Technical Annex,* TOBIAS, June 1989.

Wang88a.

B. Wang, D. Coffield, and D. Hutchinson, *Towards an Implementation of Domain Based Distributed System Management,* Department of Computing, University of

Lancaster, 1988.

Wang89a.
Baoyu Wang, David Coffield, and David Hutchinson, "Database / Domain Approach to Distributed System Management," *Computer Communications*, Butterworth, 1989.

Wheater89a.
Stuart M. Wheater, *Constructing Reliable Distributed Applications using Actions and Objects,* Newcastle University, September 1989. (Ph.D. Thesis)

Young91a.
Andrew Young, *A Structural Approach to Fault and Change Management in a Distributed System,* Imperial College, May 1991. (Ph.D. Thesis)