A REFERENCE MODEL FOR THE PROCESS CONTROL DOMAIN OF APPLICATION

by

NIRVANI DHEVCHARRAN

submitted in part fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROFESSOR A L STEENKAMP

JOINT SUPERVISOR: DR V RAM

NOVEMBER 1995

ACKNOWLEDGMENTS

The successful completion of this dissertation is attributed to a few key people and a heartfelt thanks is extended to them:

Vinesh: My Husband, for his persistent encouragement and support.
Reshina: My Sister, for unrelentlessly assisting me in a number of activities.
Doctor Ram: My Joint Supervisor for his advice and guidance.
Professor Steenkamp: My Supervisor, for her inspiration and advice.

ABSTRACT

The process control domain is intrinsically complex and dynamic. It has proved to be difficult to construct and maintain process control systems under the traditional software development methodologies.

Object Orientation is the latest paradigm in software development. The reason for its widespread acceptance is that it allows the application of the principles of hierarchical structuring and component abstraction which is essential in building large systems. It also promotes component reusability which makes systems easier to maintain and modify.

For the process control domain, these are important benefits. Furthermore, most process control systems have physical devices which can be modeled naturally as objects with the timing and performance issues of each object directly addressed.

A Target System Reference Model which addresses various aspects of the process control domain is proposed within this dissertation. The objective is to provide a frame of reference within which a process control system can function.

KEYWORDS

Distributed Databases, Methodology, Object Orientation, Paradigm, Process Control, Real-time Object Oriented Modeling, Reference Model, Software Engineering, Software Engineering Environments, Systems Engineering, Target System, Temporal Behaviour.

TABLE OF CONTENTS

PREFACE vii

LIST OF ACRONYMS ix

1. CHAPTER ONE: THE CONTEXT OF THE PROCESS CONTROL DOMAIN 2

1.1. INTRODUCTION 2
1.2. RELEVANCE OF PROCESS CONTROL ENVIRONMENT 4
1.3. CONTEXT OF RESEARCH 6
1.3.1. Scope of Research 6
1.3.2. The Environment Aspect 6
1.3.3. The Information Aspect 7
1.3.4. The Software Engineering Aspect 7
1.3.5. The Systems Engineering Aspect 7
1.4. ASSUMPTIONS 7
1.5. CONSTRAINTS 8
1.6. AXIOMS 8
1.7. METHOD OF INVESTIGATION 9
1.8. DISSERTATION FORMAT 10

2. CHAPTER TWO: PROPERTIES OF THE PROCESS CONTROL ENVIRONMENT 15

2.1. INTRODUCTION 15
2.2. HARDWARE CHARACTERISTICS OF THE PROCESS CONTROL ENVIRONMENT 18
2.2.1. Computing Power 18
2.2.2. Operating Environment 19
2.2.3. Operating Systems 19
2.2.4. Networking 20
2.2.4.1. LANs & PC Based data Acquisition 20
2.2.5. Redundant Distributed System 23
2.3. PROPERTIES OF PROCESS CONTROL SOFTWARE 23
2.3.1. Distributed Databases 23
2.3.2. Support for Distributed Object Applications 25
2.3.3. General Constraints 25
2.3.3.1. Synchronous and Asynchronous Monitoring 26
2.3.3.2. Avoiding the Introduction of Discontinuities 27
2.3.3.3. Using Graphical Representation 27
2.4. OO PRINCIPLES IN PROCESS CONTROL 28

2.4.1. Binding *28*
2.4.2. Polymorphism *28*
2.4.3. Multiple Inheritance *29*
2.4.4. Reusability *31*
2.4.5. Encapsulation *33*
2.4.6. Distribution Transparency *34*
2.4.7. Extensibility and Maintainability *36*
2.5. STANDARDS 37
2.5.1. Object Management Architecture *38*
2.5.2. Strengths of CORBA *38*
2.6. PARADIGMS FOR MODELING FUNCTIONALITY 39
2.7. PARADIGMS FOR MODELING TEMPORAL BEHAVIOUR 40
2.8. PARADIGMS FOR MODELING SYSTEM, CONTROL AND DATA STRUCTURE 44
2.8.1. Modeling data and control *44*
2.8.2. Rigour of Representation *44*
2.8.3. Modeling the System *45*
2.9. METHODOLOGY FOR THE DEVELOPMENT OF PROCESS CONTROL SYSTEMS 47
2.9.1. Software Process Models for Process Control *48*
2.9.1.1. Revised Spiral Life Cycle Model 48
2.9.1.2. Object Modeling Technique 49
2.9.1.3. Object Oriented Design 51
2.9.1.4. Jackson Structured Development 52
2.9.2. ROOM *53*
2.9.3. Key Elements of ROOM *54*
2.9.4. The Operational Approach *55*
2.9.5. A Phase Independent Set of Modeling Abstractions *56*
2.9.6. The Object Paradigm *59*
2.9.6.1. Objects as Instances: Abstract Data Types 59
2.9.6.2. Objects as Software Machines 60
2.9.6.3. Objects as Logical Machines 61
2.9.6.4. Messages 62
2.10. CONCLUSION 63

3. CHAPTER THREE: ASPECTS OF THE TARGET SYSTEM 70

3.1. INTRODUCTION 70
3.2. WORK ORGANISATIONAL ASPECT 71
3.2.1. Product-Oriented versus Project-Oriented Development *72*
3.2.2. Product Requirements *73*

3.2.3. Product Development Activities ... *74*
3.2.4. Deliverables ... *76*
3.2.5. Universal Model Relationships ... *76*
3.2.6. Project Team Organisation ... *77*
3.2.7. Project Management and Tracking ... *79*
3.3. META PRIMITIVES OF THE ENVIRONMENT ASPECT ... 80
3.3.1. Hardware Considerations ... *80*
3.3.2. Interface Considerations ... *82*
3.4. META PRIMITIVES OF THE INFORMATION ASPECT ... 82
3.4.1. Representation of Information ... *83*
3.4.1.1. Data Independence, Binding and Efficiency ... 83
3.4.2. Data Organisation ... *85*
3.4.2.1. Characteristics of Data in Real Time Database Systems ... 87
3.4.3. Manipulation of Information ... *88*
3.4.4. Interpretation of Information ... *90*
3.5. META PRIMITIVES OF THE SYSTEMS ENGINEERING ASPECT ... 91
3.5.1. Timeliness Issues ... *92*
3.5.2. Dynamic Internal Structure ... *93*
3.5.3. Reactiveness Issues ... *94*
3.5.4. Concurrency Issues ... *94*
3.5.5. Distribution Issues ... *95*
3.6. META PRIMITIVES OF THE SOFTWARE ENGINEERING ASPECT ... 96
3.6.1. High Level Structure Modeling ... *99*
3.6.1.1. Communications ... 102
3.6.1.2. Communication Relationships ... 104
3.6.1.3. Actor Structures ... 106
3.6.1.4. The Relationship between Structure and Behaviour ... 110
3.6.2. High Level Behaviour Modeling ... *113*
3.6.2.1. Events ... 114
3.6.2.2. State Machines ... 116
3.6.3. High Level Inheritance ... *122*
3.6.4. The Detail Level ... *123*
3.7. PROCESS CONTROL REFERENCE MODEL ... 125
3.7.1. Reference Model for the ROOM Virtual Machine ... *126*
3.7.1.1. The Services System ... 127
3.7.1.2. The Control System ... 129
3.7.1.3. The Timing Service ... 130
3.7.1.4. The Processing Service ... 131

7.1. CEBAF DATA ACQUISITION SYSTEM 182
7.1.1. Evaluation *183*
7.2. AN OO OPERATOR'S INTERFACE FOR REAL TIME PROCESS CONTROL EXPERT SYSTEMS 184
7.2.1. Overview of Functionality *185*
7.2.2. OI's OO Design and Development *185*
7.2.3. Summary *185*
7.3. ADROIT 186
7.4. SMALLTALK 187
7.5. PRINTFLOW 187
8. REFERENCES 188
8.1. BOOKS 188
8.2. ARTICLES 189

PREFACE

This dissertation is of limited scope and has been completed in partial fulfillment of the MSc degree in Computer Science at the University of South Africa. It forms part of a project entitled "Object Oriented Information Systems Engineering Environment" which is currently underway in the Department of Computer Science and Information Systems. The objective of the project is to provide an environment within which team sized research projects may be undertaken at postgraduate level Steenkamp[1995].

The scope of this dissertation was to propose a Reference Model for the Process Control Domain of Application. The fundamental paradigm for the reference model is object orientation. An object-oriented methodology, for real-time systems has been identified and evaluated for its suitability to the process control domain, whilst taking into consideration the paradigms for this domain.

In addition to this dissertation, five other examinable modules were completed:

1. Software Engineering: The various software life cycles were studied. Management, staffing and project leading a software project was also described. The "definition" of this module is that Software Engineering is "a discipline whose aim is the production of quality software, software that is delivered on time, within budget and that satisfies its requirements", according to Schach[1990].

2. Software Engineering Environments: A software engineering environment is one that provides computer support for all the phases in a methodology. It should at least include an integrated set of software tools and it should enforce the procedures and techniques on a methodology.

3. Object Orientation: This module focused on the study of object orientation and how these principles are adopted in software engineering. It also explained how object orientation integrates with the principles of structured software development.

4. Operating Systems: The aim of this module was to gain an insight into concurrency with special reference to the operating systems - the need for it, the problems involved and the techniques available to implement it.

5. Computer Architecture: Technology is rapidly changing and this module presented the student with an understanding of the architecture/hardware and design of computers. Technology independence was stressed, i.e. the move to "open systems".

LIST OF ACRONYMS

DCS	Distributed Control System
GUI	Graphical User Interface
IS	Information System
LAN	Local Area Network
NT	New Technology
OLE	Object Linking and Embedding
OO	Object Orientation
OOD	Object Oriented Design
OOP	Object Oriented Programming
PLC	Programmable Logic Controller
ROOM	Real-time Object Oriented Modeling
RTDBS	Real-Time Data Base System
RTU	Remote Terminal Unit
SAP	Service Access Point
SPP	Service Provision Point
SCADA	Supervisory Control And Data Acquisition
WAN	Wide Area Network

CHAPTER 1

THE CONTEXT OF THE PROCESS CONTROL DOMAIN

1.1. Introduction

1.2. Relevance of Process Control Environment

1.3. Context of Research

- 1.3.1. Scope of Research
- 1.3.2. The Environment Aspect
- 1.3.3 The Information Aspect
- 1.3.4. The Software Engineering Aspect
- 1.3.5. The Systems Engineering Aspect

1.4. Assumptions

1.5. Constraints

1.6. Axioms

1.7. Method of Investigation

1.8. Dissertation Format

1. CHAPTER ONE: THE CONTEXT OF THE PROCESS CONTROL DOMAIN

1.1. Introduction

An environment which provides computer support for a methodology is termed a Software Engineering Environment (SEE) as described by Bornman[1988]. The SEE should enforce the procedures and standards of a methodology and facilitate its application. It should provide automated support for the methods, techniques, tools and procedures of the methodology and make this as well as other software utilities of the host computer environment available to the user as part of a system development workbench. A central repository of information about the developing target system is fundamental to a SEE. Furthermore, a SEE should an least include an integrated set of software tools.

This research forms part of a larger project being undertaken by the Department of Computer Science and Information Systems at the University of South Africa (UNISA), that is addressing the development of an Information Systems Engineering Environment (ISEE), as explained by Steenkamp[1995]. The fundamental paradigm for the project is Object-Orientation (OO). The project has identified a general framework of four reference models. The models being proposed are:

1. Development Process Reference Model

2. Quality Reference Model

3. Technology Reference Model

4. Target System Reference Model.

Each of the above-mentioned reference models focuses on various aspects, as illustrated in Figure 1.

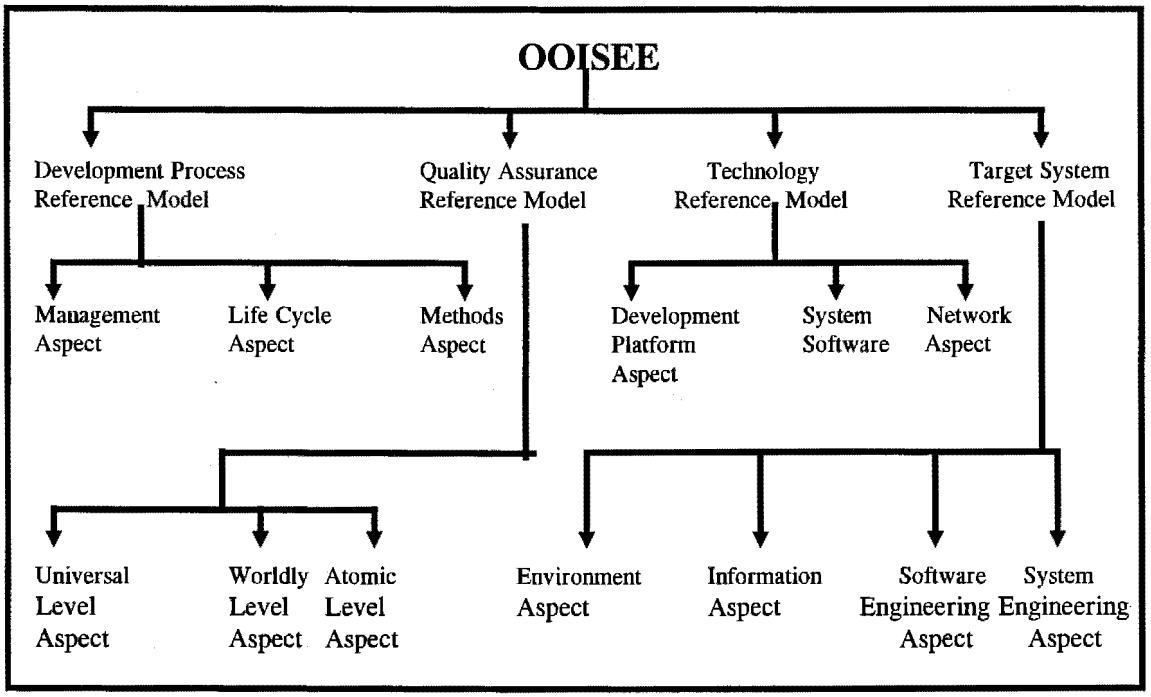


Figure 1. Reference Models of the OOISEE Project

Some of the objectives for developing the OOISEE are:

- To build a database of information regarding OO technology that is accessible to all members of the project.

- To formalise all aspects of information systems engineering; where formalism here is used to denote formal notations for specification, design and implementation, prescriptive methods, formalised evaluation, review and testing.

- To explore expert system support for the development process, and to build a knowledge base to support information systems development.

- To explore reusability in the broad and specific senses.

- To specify the requirements of an OO methodology companion.

- To define the required aspects of specialised target application domains, e.g. embedded real time, process control, multi-media, etc.

This dissertation focuses on the Target System Reference Model, in particular within the process control domain. Section 1.3 provides more detail on the reference model.

In the process control environment, the processes operate in real time, and there is a need for customised software for this environment. There are high expectations of the OO paradigm for the development of software in the application domain. OO mechanisms such as inheritance, polymorphism and dynamic binding offers the potential for the development of more flexible, more easily maintainable and less error-prone process control systems.

Although there has been a substantial amount of literature on the object paradigm and associated methodologies, (for example, Rumbaugh[1991], Winblad[1990]), very little of it has focused on the process control domain or even the real-time domain, Selic[1994]. History has shown that general-purpose methodologies must be substantially augmented (with additional modeling concepts, for example) before they can be truly useful for application to real time.

By making optimal use of the powerful OO features and further by establishing a Reference Model, for the process control environment, a structured framework for software development is created.

1.2. Relevance of Process Control Environment

The process control environment is normally very complex and detailed, with numerous manufacturing steps occurring simultaneously. Therefore, if a process control application has to be developed for such an environment, there has to be comprehensive understanding of the manufacturing business and the technology.

The developed application will comprise many different computer system processes that must be performed concurrently and within hard real-time constraints. There is also, a requirement for the safe operation of these processes so that system failures do not cause catastrophic process upsets or destruction. The communication between these concurrent processes should be seamless, which poses one of the greatest challenges. The requirement is not to merely provide a communication path for non co-operative programs to communicate in one well defined environment, but across an enterprise wide network of heterogeneous computing environments.

Utilising OO methods for implementing process control systems has a number of advantages.

- Firstly, the OO approach allows the independence of multiple distributed process control databases to be maintained but still provides for the integration of these databases.

- Secondly, and potentially the most important in the long term, is that an OO approach proffers integration with other OO approaches that create integrated manufacturing environments.

Workstations, represented as objects in these environments encapsulate processing knowledge as well as the protocols required to communicate with other objects (workstations). Given this representation, object-based quality control techniques may be incorporated directly into a distributed object-based process control system.

In summary, an OO architecture with industry standards, as opposed to vendor-specific standards, holds strong potential as a solution to the problems of integration, seamless communications between "objects" and reusability of software. OO designs allow software developers to create intricate system and control environments with interdependent functions. Object databases are used to specify non-standard relationships between objects that are stored within databases Gillen[1993].

1.3. Context of Research

The Target System Reference Model provides a frame of reference for viewing various aspects of an information system to be developed in support of the real world. Typical application domains that can be supported by the reference model are embedded real-time, process control, logistics management, etc.

1.3.1. Scope of Research

This dissertation will concentrate on developing a reference model for the process control domain of application within the context of The Target System Reference model. The following aspects have, thus far been identified within the reference model:

- Environment
- Information
- Software Engineering
- Systems Engineering.

1.3.2. The Environment Aspect

The environment aspect occupies the highest level within any methodology. This aspect considers the structure of the process control environment in terms of its interface to the real world as well as its interface to physical devices. More details regarding this aspect, are provided in Chapter 3.

1.3.3. The Information Aspect

The manner in which information is modeled within the process control domain is considered. Typically, information has to be represented, manipulated and interpreted. Data in this domain is normally represented in a real time database. This data has to be logically and temporally consistent as stated by Son[1995]. The latter arises from the need to preserve the temporal validity of data items that reflect the state of the environment that is being controlled by the system. This aspect is discussed further in Chapter 3.

1.3.4. The Software Engineering Aspect

This aspect expresses the target system in terms of the interacting object metaphor. The intra-object structure, inter-object relationships, the functionality of the objects and the dynamic behaviour of objects are addressed. Paradigms are required to model the structure of the system, of the data and of control, and are discussed in Chapter 2.

1.3.5. The Systems Engineering Aspect

The main focus of this aspect is on those engineering principles relevant at the operating and networking level. More specifically, the timeliness issue, the dynamic internal structure, the reactiveness issue, the concurrency issue and the distributed issue of the process control domain is described in Chapter 3.

1.4. Assumptions

The hardware platform to instantiate the proposed meta model exists. The configuration should be a Local Area Network (LAN) based development environment that is open systems compliant as proposed by Steenkamp[1995].

1.5. Constraints

- Willingness of developers of OO Process Control Systems to discuss the methodologies adopted and rationale for doing so.

- The investigation forms part of the OOISEE project of the Department of Computer Science and Information Systems at UNISA.

- The scope of the investigation is determined by the requirements of a dissertation of limited scope.

1.6. Axioms

- An attempt will be made to adopt the revised spiral life cycle model, as described by du Plessis and van der Walt [1992]. The initial spiral model was proposed by Boehm [Boehm1986] and has many features that are embodied in other life cycle models. The advantages of this model over other models are that the spiral model makes provision for evolutionary development with risk analysis and a control component. These are important properties for OO Development (OOD) as the life cycle phases in OOD are not distinct. The modifications to adapt it for OOD are by adding additional emphasis on analysis, inclusion of activities associated with OOD and adding more control checkpoints to the model as proposed by du Plessis[1992]. See Chapter 2.

1.7. Method of Investigation

The steps involved in this investigation were:

1. Surveying of OO literature:
    - in Real-time Systems generally and
    - in the Process Control Domain specifically.

2. Approaching suppliers of process control systems for comment on the usage or lack thereof of OO concepts. Most of the suppliers commented that OO techniques were being implemented but none were prepared to disclose the specifics of methodology.

3. Evaluating the following software development approaches:

    - Real -time Object Oriented Modeling (ROOM) by Selic [1994]
    - Object Modeling Technique by Rumbaugh[1991]
    - Object Oriented Design by Booch[1991]
    - Jackson Structure Diagram as discussed by Schach[1990]
    - Revised Spiral Life Cycle as proposed by du Plessis and van der Walt[1992].

    ROOM was considered the most suitable for the process control domain of application. One of the main reasons for its selection is that it eliminates development phase discontinuities which are inherent in some of the other methodologies. This selection is discussed in more detail in Chapter 2.

4. Defining a number of primitives for each aspect of the Reference Model as well as elaborating upon the attributes for the primitive.

5. Describing the ROOM Virtual Machine.

6. Proposing the Reference Model for the Process Control Domain of Application. All the aspects and primitives constitute the model. It is illustrated conceptually and then using ROOM's notation.

7. Demonstrating the concept of ROOM. Umgeni Water is a water purification organisation in Kwa Zulu/Natal and has recently upgraded its process control system. An extension to this new system was requested shortly after installation. ROOM was demonstrated against this extension and is discussed in greater detail in Chapter 4.

1.8. Dissertation Format

This dissertation focuses on all the aspects of a process control domain, i.e. it incorporates the hardware and software aspects, with more emphasis being placed on the software rather than hardware issues. Paradigms for the software aspect are discussed in detail. A reference model is proposed and a demonstration of the preferred methodology is presented. The format of the rest of the dissertation is as follows:

Chapter 2

An introduction to the process control domain is presented followed by consideration of the hardware characteristics of the process control domain, and the more common networks in this domain. The chapter moves on to the software aspects, i.e. properties of process control software and the OO principles for the software model. There are a number of OO concepts but only the ones that have relevance to the process control environment are described.

There has been a considerable lack of standards in the OO environment mainly because the OO concepts are still very new and developers are still experimenting with it. However, standards are now starting to emerge and are briefly discussed in this chapter.

Further, paradigms specific to the process control domain for modeling functionality, temporal behaviour and for modeling system, control and data structures are addressed.

A real time OO modeling technique (ROOM) is identified as a suitable methodology for the modeling of the process control domain and is also reviewed, Selic[1994].

Chapter 3

This chapter addresses the four aspects of the target system reference model and its meta-primitives. The software engineering aspect is discussed at length with particular emphasis placed on how the OO paradigm is adopted in the ROOM methodology for the process control domain. Reference models as proposed by ROOM and also as suggested for the process control application are discussed.

Chapter 4

The ROOM methodology is applied to a process control application, in particular an application within a Water Treatment Plant. An iterative approach is performed to model the construction and validation of the solution.

Chapter 5

A brief overview of the realisation of the objectives of this dissertation is presented, together with the approach to each objective. The effectiveness of the ROOM methodology as applied in Chapter 4 is analysed. The analysis is in terms of

ROOM's inherent limitations and constraints. Areas for future research are suggested.

CHAPTER 2
PROPERTIES OF THE PROCESS CONTROL ENVIRONMENT

2.1. Introduction

2.2. Hardware Characteristics of the Process Control Environment

- 2.2.1. Computing Power
- 2.2.2. Operating Environment
- 2.2.3. Operating Systems
- 2.2.4. Networking
  - 2.2.4.1. LANs & PC Based Data Acquisition
- 2.2.5. Redundant Distributed system

2.3. Properties of Process Control Software

- 2.3.1. Distributed Databases
- 2.3.2. Support for Distributed Object Applications
- 2.3.3. General Constraints
  - 2.3.3.1. Synchronous and Asynchronous Monitoring
  - 2.3.3.2. Avoiding the Introduction of Discontinuities
  - 2.3.3.3. Using Graphical Representation

2.4. OO Principles in Process Control

- 2.4.1. Binding
- 2.4.2. Polymorphism
- 2.4.3. Multiple Inheritance
- 2.4.4. Reusability
- 2.4.5. Encapsulation
- 2.4.6. Distribution Transparency
- 2.4.7. Extensibility & Maintainability

2.5. Standards

- 2.5.1. Object Management Architecture
- 2.5.2. Strengths of CORBA

2.6. Paradigms for Modeling Functionality
2.7. Paradigms for Modeling Temporal Behaviour
2.8. Paradigms for Modeling System, Control and Data Structure
2.8.1. Modeling Data and Control
2.8.2. Rigour of Representation
2.8.3. Modeling the System
2.9. Methodology for the Development of Process Control Systems
2.9.1. Software Process Model for Process Control
2.9.1.1. Revised Spiral Life Cycle
2.9.1.2. Object Modeling Technique
2.9.1.3. Object-Oriented Design
2.9.1.4. Jackson Structured Development
2.9.2. ROOM
2.9.3. Key Elements of ROOM
2.9.4. The Operational Approach
2.9.5. A Phase Independent Set of Modeling Abstractions
2.9.6. The Object Paradigm
2.9.6.1. Objects as Instances: Abstract Data Types
2.9.6.2. Objects as Software Machines
2.9.6.3. Objects as Logical Machines
2.9.6.4. Messages
2.10. Conclusion

2. CHAPTER TWO: PROPERTIES OF THE PROCESS CONTROL ENVIRONMENT

2.1. Introduction

Process Control deals with the technical, economic and safety dimensions of applications such as those found in chemical, oil and gas refineries; textile and paper mills; and municipal water and sewage treatment. Each of these industries has data to be acquired, data to be disseminated, decisions to be processed, communications to be performed and reports to be generated. The challenge is to develop a control system that addresses the tremendous but unique complexity of each application without reinventing a thousand new wheels each and every time as described by Beam[1993].

Traditionally, programming has been used to solve specific individual problems. While this approach may sometimes lead to quick solutions, often each new problem will have to be solved from scratch. The goal of OO development is just the opposite - to build solid, working models that can be used to solve any number of related problems. However, the solutions these models support will only be as good as the thinking that went into them.

Carter[1994] asserts that, twenty years ago, the design of a centralised control room in a plant was relatively straight forward: there was usually a display screen (mimic panel); procedures (sequencing) and switch setting (interlocks) were done using relays and timers; and the biggest decision focused on the choice of devices, e.g. controllers and recorders, to use.

As Programmable Logic Controllers (PLCs) began taking over sequencing, timing and interlocking, so Distributed Control Systems (DCSs) began to replace the traditional

controllers and recorders. A whole new concept, Supervisory Control And Data Acquisition (SCADA) emerged to complicate the issue. Added to this, low-cost computing power became easily available in the form of PCs which, with the advent of Graphical User Interfaces (GUI) like Windows, brought powerful graphics capabilities to the control room.

SCADA has been in use in various forms for over thirty years. Telemetry systems are a key element of a SCADA system providing the necessary transfer of analogue and digital data from the Remote Terminal Unit (RTU) to the master stations (see Figure 2). Telemetry is generally used for wide area monitoring and control while a PLC is normally used for local monitoring and control (on site/plant) although it is possible to have a combination of both.

The central site is structured as a distributed approach with the operator stations connected together on a Local Area Network (LAN) for added flexibility. The aim of the distributed architecture is to increase the overall reliability of the system and provide for a more flexible system.

SCADA is referenced considerably in this dissertation and it is encapsulated within the process control environment as shown below:

1. Management Information Systems (MIS)
2. SCADA
3. Front End Devices (PLCs and Telemetry)
4. Field Devices

The process control environment is composed of steps 2 to 4 above.

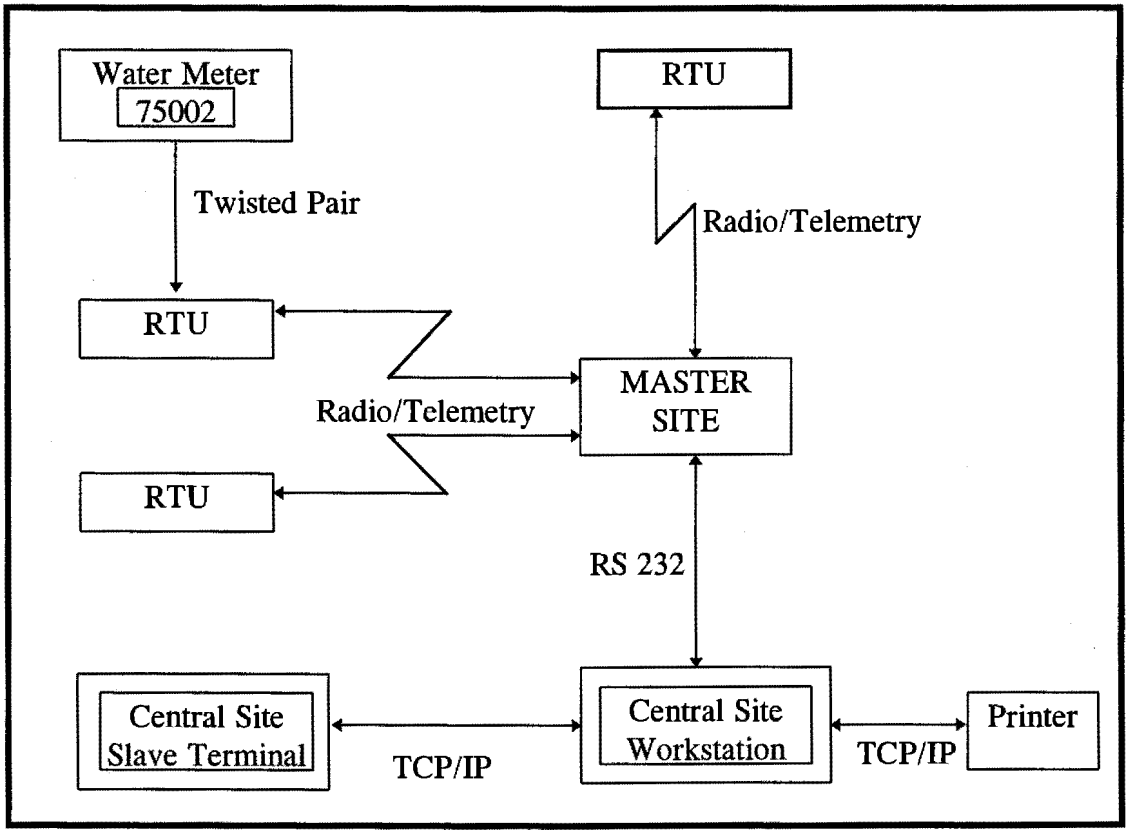


Figure 2. Example of a Process Control System

As can be seen from the discussion so far, the process control domain is specialised and technical. There are a number of unique features within this domain that have to be taken cognisance of, for example:

- Absolute timing of events and processes.
- Protocols used across the network for efficient messaging between the SCADA systems and the PLCs in the field.
- Real-time access to data at the plant as well as across distributed SCADA systems.

The rest of this chapter addresses these concepts and the paradigms specific to the process control domain in terms of hardware and software.

2.2. *Hardware Characteristics of the Process Control Environment*

At first, the focus of SCADA was on simply displaying plant information, but now users are becoming more demanding and are exploring how they can use the available information to improve process performance[Strydom1993]. The rapid change in technology means, however, that some systems can quickly become obsolete. Further, some systems may not offer adequate future expansion, thus necessitating a complete replacement.

The Process Control industry also follows the trend of moving to open systems, which is not only to alleviate technology obsolescence but to also help with porting information across different platforms.

2.2.1. Computing Power

The goals of OO development are ease of use, robustness under conditions of change and an increase in productivity relative to conventional approaches. Experience has shown that all of the above features can be achieved within the rigours of an on-line environment in an efficient manner with the central processing unit power now readily available[Quarrie1992]. Given appropriate modification to the algorithms, OO programs can in fact be more efficient than the equivalent conventional programs.

With all the benefits, however, many believe that OO is counter-productive since there may be extra heavy overheads in dynamic binding and dynamic memory management. However, the overheads in these mechanisms can be minimised with careful compiler and run time implementation. Moreover, OO process control systems may provide more flexibility which may result in even higher performance gains than those attained by non-OO systems. There is a strict requirement of

graceful system degradation in safety critical process control systems. The OO approach uses a flexible scheduling framework and by integrating the performance consideration into the structures of object types and the whole system, enhances the predictability of system performance.

2.2.2. Operating Environment

Distributed computer systems like those used for industrial process control are large and complex. Although connecting computers in a local area network is relatively easy, programming them requires skilled personnel, especially when the project becomes large and there are added requirements at run-time. The tools developed by mainframe manufacturers for single processor machines fall short in distributed systems. In particular, they do not meet the important requirements of industrial control systems: distribution, real time response and fault tolerance [Aschmann1991].

2.2.3. Operating Systems

Windows NT® has become one of the strongest players as a server and workstation operating system. It is uniquely suited for process control in two ways. First, it is a viable server for corporate data, one of the main markets targeted by Microsoft. The pre-emptive multi-tasking and memory management also make it a solid client platform for process applications. Windows 95® is likely to be a strong player for general client (desktop) applications. Only process control applications which have been developed from the ground up to run under NT® can take full advantage of these powerful new platforms.

The architecture of the future version of Windows NT® (code named Cairo) is based on object technology evolving from Object Linking and Embedding (OLE) 2

Windows NT ®& Windows 95® are registered Microsoft Trademarks

and promises easier development of more powerful, flexible and usable software. The design of process control systems that has a solid OO foundation will allow for the co-existence with the new technology and therefore ensure the long term production of users' investments[Carter1994].

2.2.4. Networking

Normally the front-end devices, PLCs, have their own protocol specification and the SCADA system needs to implement a protocol driver that will understand and correctly interpret the protocol used by the PLC as described by Strydom [Strydom1993]. The communication may be over a variety of physical media. The most widely used standard is the RS232 serial port. This is similar to the interface used to connect a PC to a mouse or, in some cases, to a printer.

With the increasing trend to larger distributed systems, network based communications have become a common requirement. This type of communication tends to be more complex and expensive, and there is a need for fast interface capabilities. Some network vendors provide special hardware cards that can be placed into computers to do a lot of the network interfacing at the hardware level. This is especially necessary where the process control network is a proprietary network.

2.2.4.1. LANs & PC Based data Acquisition

Ethernet vs. Token Networks

Amongst the many communication methods used in the past, Ethernet and Token Networks have emerged as the overwhelming favourites in PC LANs and in the process control environment in terms of open systems. Of the two, Ethernet is the most popular and lowest cost technique[Gunderson1995]. Token networks, on the other hand, offer reliability and repeatable timing characteristics that are considered valuable.

Ethernet is a bus scheme, where every node is attached to every other node as shown in Figure 3.

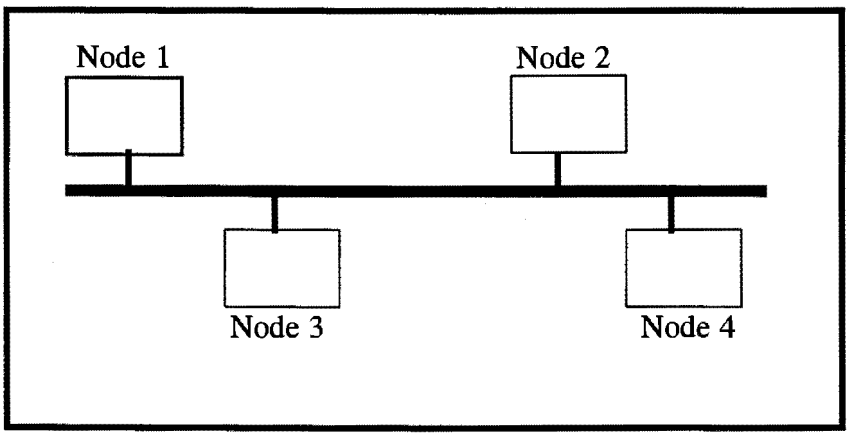


Figure 3. Ethernet Structure

In an Ethernet implementation, each node is allowed to transmit whenever it senses that the bus is idle and every node receives every transmitted packet. This leads to the possibility of two or more nodes starting a transmission at about the same time (called a collision). When an Ethernet transmitter detects a collision it abandons its transmission, waits for a brief interval and tries again. This scheme is called CSMA/CD (Carrier Sense Multiple Access with Collision Detection). Ethernet is standardised by IEEE Standard 802.3 (ISO 8802.3).

The main disadvantage of Ethernet is that the time for a data packet to get from the transmitter node to a receiver node cannot be determined absolutely. The time depends on network loading and the number of transmit collisions that are taking place. As a result, Ethernet should not be used for time critical messages such as safety alarms and real-time machine control information.

In a token ring (IEEE 802.5, ISO 8802.5), each node is attached to only two other nodes, as shown in Figure 4.

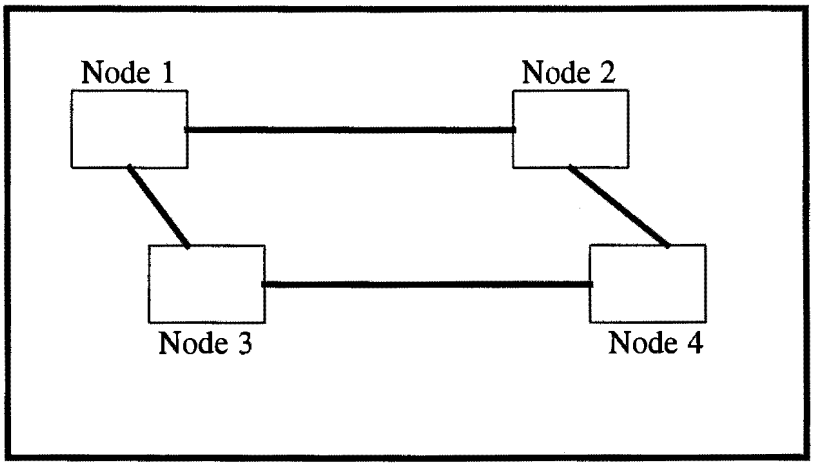


Figure 4. Token Ring Structure

Transmitted packets travel from node to node around the ring. Each node receives the packets transmitted by two neighbour nodes, recognises those that are addressed to it and repeats only those packets addressed to other nodes. Only the node holding the 'token' may originate a transmission. The token is a flag, that is passed around the network to allow each node an equal opportunity to source transmissions. Packets may travel in either direction around the ring, allowing operation of the net even when the ring is broken at some point.

A Token Bus (IEEE-802.4) has the same bus arrangement as an Ethernet, but uses the token passing scheme to control transmission permission.

Since each node gets an equal opportunity to transmit and no collisions are possible in a token network the amount of time needed to complete the transmission of a packet can be absolutely determined. This makes this network topology popular in situations where time critical messages must be sent over the net.

The main disadvantage of Token Networks is the higher cost of wiring and interfaces relative to Ethernet hardware. Token Network interface cards are more expensive than Ethernet cards because of the increased logic required to perform token passing and because they are made in lower quantities than Ethernet cards.

2.2.5. Redundant Distributed System

In process control at the supervisory level, the main requirement is high availability. Basic availability is best provided by a duplex structure. Functional network redundancy is provided by additional nodes in the network and duplication of communication links. When a node fails, its backup node takes over its function. The backup node is functionally redundant to the on-line node. In particular, the backup has the same access to I/O devices as the on-line node, either through dual-ported devices or redundant devices.

2.3. Properties of Process Control Software

The process control domain is typically distributed not only across a Wide Area Network (WAN) but also by having a number of remote devices communicating with it. It is imperative to have access to this data from such a distributed network as soon as possible because process control systems operate in real-time.

2.3.1. Distributed Databases

Object-oriented designs allow software developers to create intricate system and control environments with interdependent functions. Object databases are used to specify non-standard relationships between objects that are stored within databases. One strength of an object-oriented system is its ability to simulate a complex environment, such as the process control environment, where one process or mechanism impacts another through multiple relationships. An object database must support these relationships as well.

Furthermore the objects that constitute the object database must be:

- individually tested to ensure high quality;
- modifiable with ease in response to change;
- easily maintained, Roedner[1994].

An OO approach to the architectural challenges of these process control systems has much to recommend it. The two hallmark features of an OO architecture, encapsulation and message passing, provide significant leverage for addressing the problems of distributed, real-time applications. They provide the locality of reference, precise interfaces and explicit communication essential to effectively and efficiently design these applications.

A client object invokes a service of another object by sending a message to the supplier object.

All messages consist of two parts: their addressing and their content. These attributes of message objects can be examined and manipulated by the receiver of message. The addressing of a message consists (potentially) of three fields: the sending object, the receiving object and the correspondent object to which the response of the message is sent. The contents consists of the name of the service requested and the sequence of arguments provided.

With these features, messages and stream take on a tangible reality, without committing to the more radical option of introducing an explicit messaging system object. Since all object interactions occur via message passing, access to the message stream provides a single point of leverage for monitoring system activity.

2.3.2. Support for Distributed Object Applications

"Supporting distributed object application" has two essential meanings. First, can a technology support interactions between objects across networks? Second, does a technology give software developers all the facilities they need to build distributed object applications?

2.3.3. General Constraints

Computations in a process control environment are usually triggered by both external events and internal timers. The program specification includes the times at, before, or after which events and responses may occur, as well as the minimum and maximum time intervals that may elapse between events. To ensure that a program meets its specifications, a process control programming language must allow

- programmers to express different types of timing constraints,
- compilers to check the feasibility of meeting the timing requirements,
- systems to enforce timing constraints either before or at run time.

Given a coded program and the timing constraints on it, the system must know how much time and resources the program need in order to check if its timing constraints can be satisfied. The timing problem is non trivial since many factors may affect the execution time of the program. For example, many processors have operation pipelines which have different execution times depending on the number of branch operations executed. Factors like data dependent execution paths make compile time analysis impossible. In fact, in the general case, the problem of determining the program execution time is equivalent to the halting problem, which means that one may not be able to predict if a program execution will terminate.

2.3.3.1. Synchronous and Asynchronous Monitoring

In developing a process control system a programmer, should also have available with a notation for specifying complex constraints that are to be monitored at run time. For example, a timing constraint may specify an end-to-end deadline on a set of actions that must be executed upon receiving a new sensor value, or a safety assertion may require that a specific procedure must hold among the actions while the system is in a certain mode. In deciding on a model to handle this, a system constraint can be viewed as an assertion on the relationship between the occurrences of the observable events. This model should distinguish between the two general ways in which event histories can be utilised in specifying and monitoring system constraints: *synchronous* vs. a*synchronous*.

Synchronous monitoring is when there is continuous monitoring of events while asynchronous monitoring is done at irregular intervals.

In synchronous monitoring the programmer can explicitly check for the satisfiability of a constraint at a particular point in the execution of the program and modify the computation accordingly. This is done by directly manipulating the event histories that are shared by co-operating tasks. Thus, testing and handling of any violation of the constraint is carried out synchronously on the threads of the executing tasks.

Alternatively, in asynchronous monitoring, the constraint is enforced during the entire execution of the program. Thus, testing and handling of exceptions are performed asynchronously. The events generated by the application tasks are sent to the system monitor (a separate event) which is responsible for maintaining the event histories. Whenever an event occurs that may violate the satisfiability of the constraint, the system monitor re-evaluates the expression and invokes the appropriate handler if the constraint is no longer satisfiable. The rationale for asynchronous monitoring is that, for certain assertions, it may be impossible to insert a test at a particular point in the program and synchronously check for its satisfiability.

2.3.3.2. Avoiding the Introduction of Discontinuities

One of the major trouble spots in traditional process control systems development is the presence of discontinuities that occur within the development process [Selic1994]. These discontinuities are caused by the lack of formal relationships between different notions. For example, high level designs are often formulated in terms of abstract and high level concepts such as layered diagrams and state machines. These representations are far removed from the concepts supported by implementations by means of standard programming language constructs. Typically, it is not obvious how such models are to be recast in terms of programming language constructs. Consequently, the process of generating an implementation from such models tend to be extremely unreliable.

These discontinuities also make it difficult to trace the linkages between the process control system requirements and the implementation that is supposed to satisfy them. Maintaining the linkage is important to ensure not only that all the requirements are met, but also that (as the system evolves) the effects of any change can be determined precisely in terms of its effect on the original requirements.

2.3.3.3. Using Graphical Representation

It is believed that graphics-based representations facilitate communication among all parties involved in system development. Therefore it was imposed that as a general constraint, an OO language must provide graphical representation for those aspects that, based on experience, are best communicated by graphical means. For example, state machines are traditionally rendered as graphs, since that representation seems to provide insight faster than equivalent textual and tabular forms. However, one of the traditional shortcomings of process control graphical representations is their impracticality for capturing detail.

2.4. OO Principles in Process Control

An object is a concept or abstraction with crisp boundaries and meaning for an application. OO principles such as polymorphism and binding are not new to the programming domain. The difference in OO is that these elements are brought together into a synergistic way.

2.4.1. Binding

Programs cater to different groups of people, such as operators, production supervisors, process chemists and plant management. Each group needs different types of data and levels of control over a process, and these requirements are better met in an OO environment than in a traditional development environment. The late binding feature of Object-Oriented Programming (OOP) makes a control system more flexible for initial configurations and subsequent modifications. A large distributed control environment without late binding can be very limiting, and difficult to modify Ghosh[1991]. However, care should be exercised in implementing late binding since it can make a system less secure. For one, addressing-errors may not be discovered until run time.

These security problems can be avoided by thoroughly checking out a system before a production run. Or, alternatively, users can employ diagnostic tools to check the validity of all the addresses.

2.4.2. Polymorphism

Hiding alternative procedures behind a common interface is called polymorphism, a Greek term meaning "many forms". Polymorphism is the capability of a single variable to refer to different objects that fulfill certain message protocol responsibilities. For example, an instance variable called *Water_Balance* can hold a

Pipeline Balance or a Reservoir Balance at different times. See Figure 5. No matter which type of object the variable holds at a given time, it can be sent a calculate balance message.

Polymorphism is important enough to be considered as one of the defining characteristics of OO technology. The key benefits of polymorphism are that it makes objects more independent of each other and allows new objects to be added with minimal changes to existing objects. These benefits, in turn, lead to much simpler systems that are far more capable of evolving over time to meet changing needs.

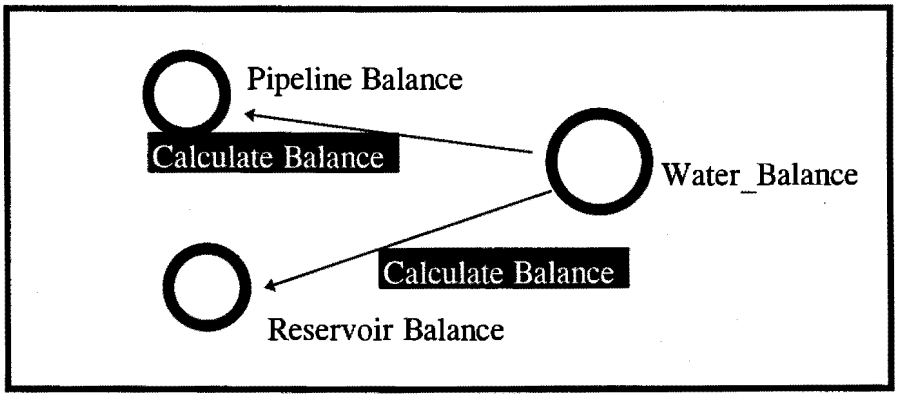


Figure 5. Water Balance Options

In essence, polymorphism provides the simple ability to use the same method name in more than one class.

2.4.3. Multiple Inheritance

Inheritance is the mechanism for automatically sharing methods and data among classes, subclasses and objects [Winblad1990]. A powerful mechanism not found in procedural systems, inheritance, allows programmers to create new classes by programming only the difference from the parent class.

Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents. This permits mixing of information from two or more sources [Rumbaugh1991].

Although multiple inheritance can simplify certain situations, it can also lead to complications. For example, suppose both Supervisor and ShiftOperator contained a method call ReportStatus, see Figure 6. Then a ShiftOperatorForeman would inherit two different versions of this method. Which one should it use? There are ways of dealing with this problem, but none of them are entirely satisfactory.

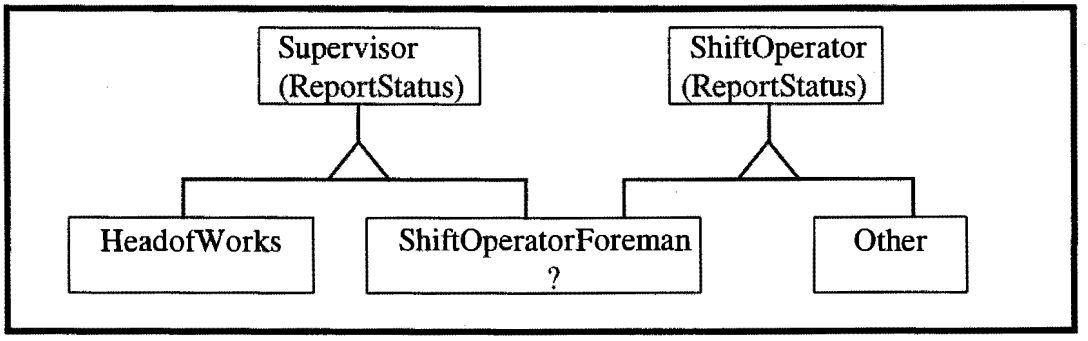


Figure 6. Multiple Inheritance Example

Another problem with multiple inheritance is that it is often misused. To apply multiple inheritance correctly, one must be certain that a class is truly an example of two or more classes. For example, the preceding illustration assumes that a ShiftOperatorForeman really is a ShiftOperator (one of her jobs is to do water quality testing) and spends time on the line performing the task. If this were true - if the foreman supervised ShiftOperators but did not actually do any water quality testing - then inheriting the qualities of ShiftOperator would not be appropriate.

Given these complications, many authorities question whether the benefits of multiple inheritance outweigh its potential costs. The trend in OO development appears to be toward supporting multiple inheritance, but it is definitely a feature that should be used judiciously.

2.4.4. Reusability

Reusability is a by-product of the development process. A by-product that the hardware industry is not only familiar with, but one that is a standard processing step inherent in any hardware design as described by Beam [Beam1993]. Engineers rarely reinvent the wheel, but rather use mature established "parts" to develop and create new solutions (products). In the case of software the idea is that a shell would wrap around the code segment (encapsulation) so that an end-user, application developer or vendor could access the code, available to everyone, and determine if it has the capability (assets) to assist a specific application. The end result would be a dramatic decrease in programming and development costs while information availability (one of the most important issues of the Process Control Industry today) would increase.

In process control, OO techniques offer an alternative to writing the same processes repeatedly. The OO programmer modifies a program's functionality by replacing old elements or objects with new objects or by simply "plugging" new objects into the application. General instructions (messages) require no modification because specific implementation details (methods and data) are encapsulated within the object. That is, each object knows how to carry out its own behaviour. This notion is in sharp contrast to procedural programming where operations and rules act on separate sets of data. In the procedural approach, programmers focus their attention on language issues, whereas in the OO environment, the important issue is cultivating a robust class library or sets of objects that can be used in a pre-existing repository of code that has been written, tested and debugged to provide high-quality application building blocks. Classes provide not only modularity and information but also reusability, enhanced by inheritance and polymorphism.

To create mimics of the production processes (typically within a SCADA application) it is now merely a matter of extracting objects (e.g., a valve) from an object library and contextualising it. An illustration is given below:

- Consider, for example the process tank in Figure 7. It has an inlet valve, an agitator, a heating element (heater), high and low level switches and a level gauge. Stored in the mimic library, the complete process tank can be called up as an object and then sized, positioned, annotated and coloured according to the needs of the mimic diagram [Hill1993].

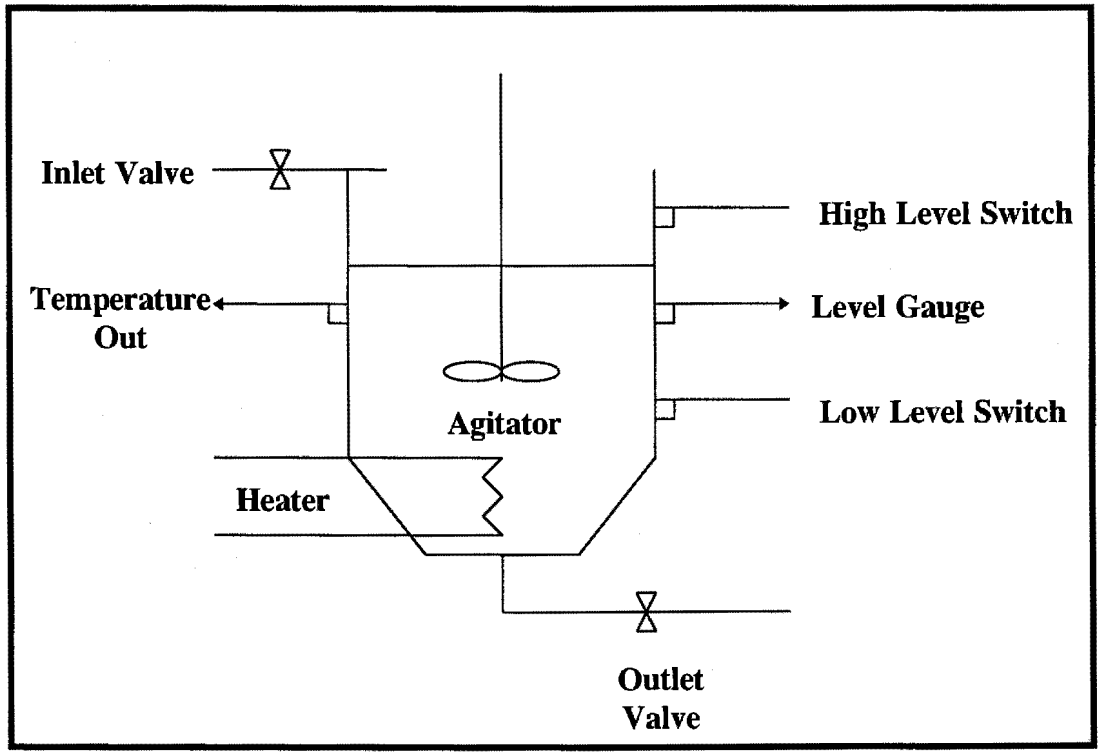


Figure 7. Process Tank

But suppose it is just a straightforward tank with no agitator and no heating element. Using OO technology it is not necessary to create a new object. In fact, the procedure would entail calling up the complete "process tank" as what is termed a template or class. From this template (e.g. Class A) it is now possible to create another class (e.g. Class B) that only incorporates the attributes and behaviour required, i.e., no agitator and no heater. This new class (Class B) thus inherits all the other features of the original template (i.e., the "process tank") and may now be put back in the library as a sub-class of Class A - a new template called the "tank" class. If, there were three classes, a "tank", an "agitator" and a "heater", it could be combined to form a "super-class", the "process tank". Both the "process tank" (Class A) and the "tank"

(Class B) are templates that may now be used in specific cases or instances. Thus, instance 1 of Class A might refer to "process tank 1"; instance 2 of Class A to "process tank 2"; and so on.

The code for future systems can be preserved and the code models (stored in classes) can be evolved over time, to gain more information relevant to differing situations and to discover hierarchies of classes which become stable over time.

2.4.5. Encapsulation

Encapsulation (also information hiding) consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects [Rumbaugh1991].

In OO systems, one cannot directly access an object's state data, but must ask for the object for services, which may include returning a piece of the data. There are a variety of reasons for keeping objects "black boxes" (hiding the internal workings), but the primary one is to minimise ripples caused by maintenance changes. After all, client code should not care how a service is provided, only that the desired result is obtained.

Thus, the encapsulation shell of an object serves multiple purposes:

- Firstly, it is an abstraction mechanism that enables the potentially complex functionality of an object to be comprehended as a single conceptual unit.

- Secondly, it is a decoupling mechanism that shield the environment from implementation changes within the object and vice versa.

- Finally, it can serve as a basis for security, since it prevents any external access to objects.

2.4.6. Distribution Transparency

Layered, process control systems achieve a new dimension with OO. OO structure is re-expressed as a collection of objects communicating with each other through messages; applications can be dynamically assembled from these objects when messages are sent, even across architectures.

OO design has been strongly influenced by the concepts of modelling and simulation. Instead of thinking of the process control system as a set of transformations to be performed on data, the designer thinks of the application as a collection of concurrently active communicating components.

To support object distribution and concurrency effectively, the software must break down the barriers of process and address space to allow transparent message passing between objects without concern for location or synchronisation as Gilbert[1993] has described. This transparency is accomplished by using the operating system's location services to route messages between objects in potentially different address spaces. The software communication ports of the operating system interlink object communities. All objects desiring to receive communications from objects in other communities register themselves as global objects. Global means the object is accessible from any address space managed by the operating system, including separate memories within a bus structure and remote computers connected by communication channels. Each globally accessible object is known by a name. A name is a character string and the establishment of the name is accomplished through the registering process, which utilises the global data store of the operating system to publish the global object's name for use by other objects in the system. The result of the registration process is the assignment of a global identity to the object that becomes its handle.

Communication with a global object is possible by any object that "links to" a global object by name, using the returned identity handle as the "address" of the communication. Through this run-time linking process, objects can obtain identity

handles and send messages to global objects as if they were local to the senders. The message dispatcher and operating system hide the actual message routing from the sending object's view. Often, however, the sending object is conceptually aware that the receiving object is potentially global. Message routing options are provided to make effective use of this knowledge. Objects sending global messages have the choice of performing the communications synchronously (wait for reply) or asynchronously (return without waiting for reply). Thus, message senders can avoid possible long communication waits when return values are not needed and objects are in remote locations.

In general, objects model the processes they represent. There is, generally, a strong correlation between the real-world objects and the objects within the software system. These objects communicate as one would intuitively expect: Proportional, Integral and Derivative (PID) loop objects get set-point messages and send process variable messages. Motor control objects receive start and stop messages and send over-current or over-temperature messages, etc. This model serves as its own documentation and changes in the model are expected to produce corresponding changes in the control system [Meyer1993].

The most general version of the object paradigm defines an object as a logical machine (of whatever level of granularity) that may be interconnected with other logical machines to realise a system.

Further, object communication by messages reinforces the loose coupling introduced by encapsulation, by restricting the extent of detail that objects must "know" about one another in order to communicate.

2.4.7. Extensibility and Maintainability

Within a SCADA environment of a rapidly growing organisation, there is ongoing extensions to the system configuration, as new out-stations are added. Therefore it is essential to implement a system that can accommodate these changes easily.

It is easier to modify and extend an OO process control application than a procedural one as stated by [Winblad1990]. Inheritance allows new objects to be built out of old objects. Methods are easy to change because they reside in a single location, rather than being scattered and potentially repeated throughout the program. Thus, with the OO approach, it is no longer necessary to search and replace functions and variables throughout a large body of procedures.

The features of OO are extremely useful during maintenance. Modularity makes it easier to contain the effects of changes to a program. Polymorphism reduces the number of procedures and thus the size of the program that the maintainer must understand. Class inheritance permits a new version of a program without affecting the old. The mechanism of inheritance facilitates the documentation of program changes as subclasses, representing the history of changes made to the superclass.

Creating a subclass, the OO technique of defining a new class by describing only the differences between it and its parent class, makes it easier to determine how a program differs from its prior version. A set of subclasses represents the history of changes made to a superclass. Inheritance mechanisms reduce the likelihood of human error because changes in one class are automatically propagated to all subordinate subclasses.

Note, however that the subclass hierarchy could also exist as part of a design with intention of change.

2.5. *Standards*

In the process control domain where there is frequently a requirement to combine and reuse code from two different vendors, interoperability becomes an issue. That is, the software from both sources must be able to co-exist without damaging the environment. An additional requirement may be that the two pieces of software must communicate with each other.

Solving the interoperability barrier is largely a technical issue. Two examples of such issues are control of system resources, and name spaces. There cannot be two pieces of software demanding control of system resources and causing contention. Neither can each insist on using a particular object name for two different purposes.

Standards are being developed for distributed objects, for call-in and call-out between various programming dialects, and for common class libraries that will behave the same in different programming dialects. Solutions to the barriers of standards and interoperability will greatly enhance in the ability to reuse third-party OO software.

Only recently has there been de facto standards for OO languages established such as rules for encapsulation and inheritance [Winblad1990]. When developers of OO languages cannot decide what an object is, it is not surprising that OO database developers, who must integrate their databases with these languages, cannot decide on how to query the object. The problem is aggravated by the natural resistance of developers to adhere to standards. Developers want flexibility so that they can meet the needs of changing markets, and they want to differentiate their products to maintain an edge over their competitors. For wide spread acceptance, a common data model is needed so that applications written in different languages can easily share database objects. This is especially necessary for process control systems as different production steps could be utilising different packages and there is a need for an integration of these packages, more so from a planning point of view.

2.5.1. Object Management Architecture

The Object Management Group (OMG) is an interest group for object technology that was formed in 1989 with several hundred member companies including HP, Sun, IBM and DEC. In January 1992, the OMG completed its first standard. A year after issuing its first request for proposal, OMG published the Object Request Broker, ORB, component of the Object Management Architecture (OMA), which is the communications heart of the standard. ORB is the OMG's standard terminology for communications interfacing between objects, known as CORBA, the architecture standard for object-oriented messaging. CORBA is now the specification that virtually the entire industry uses.

CORBA provides for the infrastructure enabling objects to communicate, independent of the specific platforms, and techniques to implement the addressed objects. ORBs are designed to provide interoperability of objects over a network of heterogeneous systems.

In short, CORBA-compliance simply means that objects on different platforms can communicate transparently. An application that is compliant with the OMA consists of a set of interlocking classes and instances that interact via the ORB.

2.5.2. Strengths of CORBA

There are six essential benefits of the CORBA standard:

1. flexibility,
2. programmer productivity,
3. application simplicity,
4. decreased development time,
5. application ease of use and,
6. reduced development costs.

These benefits when coupled with improved reliability clearly demonstrate OMG's efforts to be the champion of standardisation driven solely by the end user's view point.

CORBA addresses both the definition of interfaces through which one piece of software can interact and collaborate with another. This is one part of a comprehensive development environment. In a complete environment, developers will also need languages and related tools to construct objects and the mechanisms through which they interact.

2.6. Paradigms for Modeling Functionality

The environment of a process control system often contains devices that act as the senses of the system. Broadly stated any system that accepts input may be said to be sensing what is occurring in the environment [Ward1985]. A process control system is typically attached to sensors such as thermocouples, optical scanners, etc., and can thus collect a continuous stream of relatively unstructured data.

The environment of a process control system often also contains devices that can affect physical changes as sensory inputs occur. Any system that produces outputs, of course, makes changes in its environment. However, the outputs of a process control system often are continuous in character and overlap the continuous inputs from the sensors. Such a system also changes the physical world in quite a literal way - by changing temperatures, valve positions and so on - rather than in the more abstract way of merely producing information to be acted on.

Process control systems often require concurrent processing of multiple inputs. True requirements for concurrency usually involve correlated processing of two or more inputs over the same time interval.

The time scales of many process control systems are fast by human standards. In terms of exchange of information between human beings one second is not a long delay. The devices that process control systems monitor and control, on the other hand, often operate on time scales in which one second is an extremely long time [Ward1985].

The end result of a distributed Process Control system is that any user in the system should and would be able to have timely access to information. This sounds like OO Technology's desired end result. The challenge for the process control designer is to develop and create an architecture that enables systems to become "intelligently adaptable to change."

2.7. Paradigms for Modeling Temporal Behaviour

Traditional databases deal with persistent data. Transactions access this data while maintaining its consistency. Serialisability is the usual correctness criterion associated with transactions. The goal of transaction and query processing approaches adopted in databases is to achieve a good throughput or response time.

In contrast, process control systems for the most part deal with temporal data, i.e. data that becomes outdated after a certain time. Due to the temporal nature of the data and the response time requirements imposed by the environment, tasks in process control environment possess time constraints, e.g., periods or deadlines. The resulting important difference is that the goal of real-time systems is to meet the time constraints of the activities.

One of the key points to remember here is that real time does not just imply fast. Also, real-time does not imply time constraints that are in nanoseconds or microseconds. Real-time implies the need to handle explicit time constraints in a

predictable fashion, that is, to use time cognisant protocols to deal with deadlines or periodicity constraints associated with activities.

Process control transactions are real time transactions that can be grouped into three categories:

- hard deadline
- firm deadline
- soft deadline.

The classification is based on how the application is affected by a violation of timing constraints. For a hard deadline application, missing a deadline is equivalent to a catastrophe. For firm or soft deadline applications however, missing deadlines leads to a performance penalty but does not entail catastrophic results.

A hard real-time transaction can be defined as a transaction that has hard response time requirements and temporal data consistency constraints. The Real Time Data Base System must guarantee that both timing and consistency requirements will always be met before it starts since a failure to meet these hard timing requirements will lead to a system failure.

A real-time transaction in a process control environment often has hard timing constraints, accesses highly perishable and pre-defined sets of data objects, requires only simple database functions, and arrives with a fixed periodicity.

Further, the distinction between continuous and discontinuous behaviour is of immense significance in the process control environment [Ward1985].

Consider the status of inputs and outputs as related to the "real-time" of a system. In other words, if one watched the inputs and outputs of a system as it operated, what would one see? Take the instance of an analogue circuit that monitors a signal from a thermocouple and produces a variable voltage output signal that controls the

power supplied to a heating coil. The inputs and outputs in this case are *time-continuous.* They have significant values at every point over the time intervals when the control loop is active. On the other hand, a system that monitors the impact of particles on a detector in a high energy physics experiment can be said to have input data that is *time discrete,* which exists only at isolated points in time.

In most process control systems there is a complex interplay between time-continuous and time-discrete behaviour. The temperature control circuit just mentioned, if activated and deactivated by an on-off switch, exhibits two different kinds of time-continuous behaviour. During periods after an "off" and before an "on", its output is null for any value of the input. During periods after an "on" and before an "off" its output is related to its input by the algorithm embedded in its control circuitry. The time-discrete events of the "on" and "off" signals causes transitions between one kind of behaviour and the other.

The distinction between time-continuous and time-discrete behaviour is closely related to the distinction between data and control. In the temperature-control circuit, the temperature is the data and the on-off switch is the control. However, the data/control perspective is not the same as the continuous/discrete perspective. The heater control output would often be referred to as a "control signal" yet it is time continuous in nature. On the other hand, the pulse indicating that a particle has hit the detector in the physics experiment could well be considered "data".

In order to model the time-related complexities of a process control environment/system, an adequate modeling language must be able to distinguish time-continuous and time-discrete behaviours and must be able to model the interactions between the two.

One approach to enhance the flexibility of process control systems so that timing constraints are always satisfied is to change the software structure so that the amount of work performed is based on the amount of time and resources available. In other words, instead of defining a fixed amount of work to be performed, a set of

workloads can be defined which may or may not be completely executed. During run-time or system reconfiguration, a subset of workloads is executed using only the amount of time available. The system design and scheduling issue is then to select the optimal subset of the workloads which gives the best reward under the available time and resources. An example of such a set workloads is the printing of routine reports.

In process control systems, the approach can be implemented in three different ways. First, a computation may actively evaluate its timing constraints to select the execution path with the most desirable response time. Second, the process control system, given global scheduling knowledge, may bind a real-time request dynamically to a server with appropriate time and resources available. Finally, a computation may resort to producing imprecise results if its timing constraints are so dynamic that they are beyond the control of the previous two mechanisms.

In OO systems, some methods in an object may be provided by other objects, these methods are bound based on the class hierarchy and by the parameters of the invocation. This concept has been further generalised in a system called *Flex*, to include the execution performance as one of the binding parameters, according to [Son1995]. This binding, is based on architectural or performance criteria and is a form of polymorphism. Instead of having multiple procedures that perform the same action on objects of different types, multiple procedures that perform similar functions based on different environmental constraints, can be defined in *Flex*.

This model of *performance polymorphism* raises some scheduling issues relating to the binding of the polymorphic operations. When jobs are performance polymorphic computations, jobs do not have a fixed amount of execution time, but can be executed for a variable amount of time. Each version defines a reward function which specifies how much reward can be received for a given execution time.

2.8. *Paradigms for Modeling System, Control and Data Structure*

2.8.1. Modeling data and control

From a processing perspective, there are two distinct issues relating to a process: the availability of the data necessary for the performance of the process and the occurrence of the environmental conditions sufficient for the process [Ward 1985]. There is a potential for confusion between these issues because there exists a large class of systems in which the availability of the data is the sufficient condition. From a requirements definition view-point, many typical business systems are transaction driven. For example, an inventory management system records an increase in the on-hand balance whenever an inventory item identification and quantity added is captured.

A modeling language that is to be adequate for process control systems must be capable of separating the data needed by a process from the control that actually makes the process operate. Consider, e.g., the position of a variable-flow valve that is input data for a process-control system. The data is always available. However, a process that uses the valve position may operate only during time intervals when the reservoir is being filled, intervals determined by environmental conditions. The process will not be triggered simply by the data's availability. An adequate systems modeling language must therefore allow modeling of control as well as data.

2.8.2. Rigour of Representation

In designing a model for a system, there are two ways in which the model must match the proposed system. First, the behaviour of an actual system built from the model must be demonstrated as being either consistent or inconsistent with the model. A model that is ambiguous or that describes a system in generalities will

therefore fail in terms of rigour of representation. More to the point, a modeling language must have the means to construct a representation that can be used to evaluate objectively the actual behaviour of a system. Secondly, the modeling language must be semantically rich. Thirdly, it must be capable of being used to investigate the behaviour of a system not yet built. This use of a system model is closer to the typical engineering use of a scale model (e.g., a scale model of an airplane subjected to wind-tunnel testing) than to the scientific case. However there is still a parallel. A scientific model, known to be consistent with certain experimental data is often used to predict not-yet observed behaviour. Of course, in the scientific case the next step is to measure a real system to verify the actual behaviour and to change the model if the behaviour does not match. On the other hand, in the engineering use of a system model, the next step is to simulate an imaginary system to evaluate the predicted behaviour and to change the model if the behaviour is judged unsatisfactory.

In addition to capability for predicting, system models should have internal characteristics that permit a choice to be made among several equivalent models.

When, creating a systems modeling language, it is vital to provide a notation that facilitates a conception of the problem with minimal artificial restrictions. The language must be capable of formulating problems and expressing solutions in terms of parallelism and data structures as well as in terms of sequencing and operations.

2.8.3. Modeling the System

Some of the attributes of process control systems are:

- Timing Constraints - e.g. deadlines.

- Criticalness - Measures how critical it is that a transaction meets its deadline. Different transactions may have different levels of criticalness. Note that

criticalness is a different concept from deadline. A transaction may have a very tight deadline but missing it may not cause great harm to the system.

- Value Function - Related to a transaction's criticalness is its value function. A value function of a transaction measures how valuable it is to complete the transaction at some point in time after the transaction arrives. Some typical examples are shown in Figure 8.

- Resource Requirements - This includes the number of Input/Output operations to be executed, expected CPU usage, etc.

- Expected Execution Time - This is usually hard to predict.

- Data Requirements - Read sets and writes sets of transactions.

- Periodicity - If a transaction is periodic what is its time interval?

- Time of Occurrence of Events - At what point in time will a transaction issue a read/write request?

- Other Semantics - Is the transaction read only? Does it conflict with any other transaction? Is so, will they ever be executed at the same time. How up-to-date is the data required to be by the transaction?

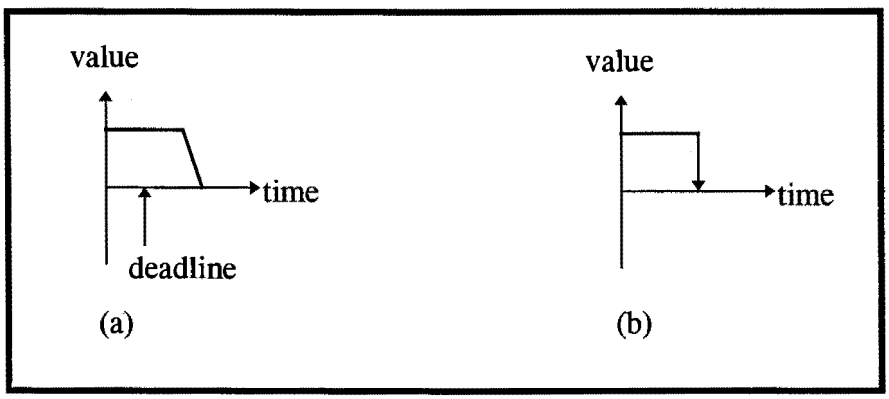


Figure 8. Value Function

2.9. *Methodology for the Development of Process Control Systems*

In its broadest sense, a systems development methodology is a comprehensive and consistent collection of the following three components:

- A modeling language.
- Modeling heuristics.
- A framework for organising and performing development work.

The dependency relationships between the three basic components of a methodology are shown in Figure 9 [Selic1994].

A modeling language provides the basic vocabulary necessary to capture high level system properties in addition to the low level abstractions captured by conventional programming languages.

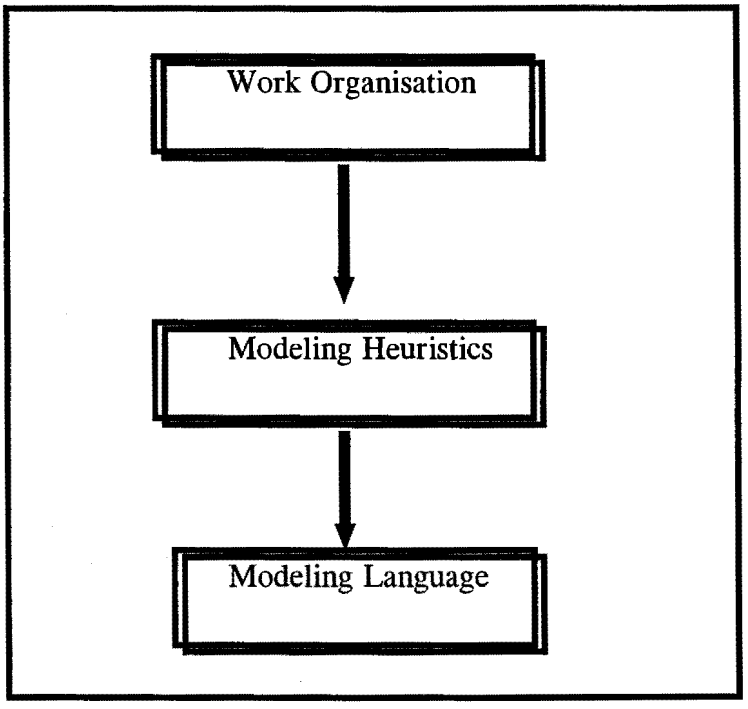


Figure 9. The Basic Structure of a Systems Development Methodology

A modeling heuristic is informal, i.e. there is no formal definition, it is based on experience and style and is therefore inconsistent across organisations.

The work organisation is directly influenced by the modeling language and the modeling heuristic. It provides a framework for an efficient organisation of time and work so that the modeling language and modeling heuristic can be used efficiently. The software process model is the generic model that underlies the work organisation concept.

2.9.1. Software Process Models for Process Control

The following approaches were evaluated for their suitability to the process control environment:

- Revised Spiral Life Cycle[du Plessis and van der Walt1992].
- Object Modeling Technique[Rumbaugh1991].
- Object Oriented Design[Booch1991].
- Jackson Structured Modeling[Schach1990].

A brief overview of each model followed by its evaluation is presented next.

2.9.1.1. Revised Spiral Life Cycle Model

The revised spiral life cycle as described by du Plessis and van der Walt[1992)] consists of five cycles:

1. *Feasibility Cycle* - system analysis and evaluation of the problem. Alternative systems configurations are evaluated.

2. *Architecture Cycle* - top level system hardware and software architecture are determined.

3. *Analysis Cycle* - risk analysis is done for the defined strategies. Prototyping of the top level class structure may be done to ensure the feasibility of the strategy.

4. *Design Cycle* - the system is designed and the sub-systems are developed.

5. *Implementation Cycle*- different ways of installing the hardware of the final system are determined. . The software is loaded, tested and system acceptance is done.

This model caters for OO development. Strong emphasis is placed on risk analysis before actual development work is done. This approach reduces the chance of project failure. Various reviews are held throughout the development life cycle as checkpoints to ensure control on the development process. By introducing the architecture cycle, where the top level system software architecture is defined, greater emphasis is placed on analysis.

Evaluation

There is no continuity in terms of models i.e., the models produced in one cycle cannot be used in the next. This approach places more emphasis on analysis and implementation as compared to the other cycles in the approach. There is a great focus on the deliverables and notation, with less focus on the functionality required.

2.9.1.2. Object Modeling Technique

Rumbaugh[1991], describes a methodology called Object Modeling Technique (OMT). OMT supports three orthogonal views of a system:

- *The Object model* - describes the static structure of the objects in a system and their relationships.

- *The Dynamic model* - describes aspects of a system that are affected by time.

- *The Functional model* - describes computations within a system.

The object model is considered by Rumbaugh [1991] to be the most important of the three models. Emphasis is placed on building a system (creating an OO model) around objects rather than around functionality. An OO model corresponds closely to the real world and is consequently more resilient with respect to change, hence the emphasis on objects.

Evaluation

The data flow diagrams and state machines used by OMT rely on natural language . This means that it can be ambiguous.

The purpose of the functional model is to show how output values are derived from input values without any regard for the order in which it occurred. However, by using data flow diagrams, invariably the order of computation is considered. The functional model is required to relate to the object model by making the bottom-level processes of the data flow diagram correspond to operations on objects. Clearly, any data flow diagram that shows object behaviour is prescribing, at least in part, a particular computation rather than specifying system behaviour.

The net effect is that the analysis models do not form a coherent set of descriptions, Hayes[1991]. They can be ambiguous and it is not possible to properly address consistency between models.

The OMT notation is rich but it can lead to object models that are complex and difficult to understand.
As with many other methods, topics such as management, sizing and testing are not covered in much detail.

2.9.1.3. Object Oriented Design

Booch[1991] discusses the following models within his methodology, Object oriented Design (OOD):

- *Physical View* - uses module and process diagrams to describe the concrete software and hardware components of an implementation.

- *Logical View* - uses class and object diagrams to describe the existence and meaning of the key abstractions that form the design.

- *Semantics (Static and Dynamic)* - combines the diagrams from the above two views together with state transitions and timing diagrams. Thus, a timing diagram can be used in conjunction with each objects' diagram to show the time ordering of messages as they are sent and evaluated.

Evaluation

The models of OOD are similar to OMT. However, OOD places more emphasis on design and less on analysis than OMT. The assumption is made that the user will use other traditional or object approaches to uncover requirements.

The inconsistency problems experienced by OMT are also prevalent within OOD.

The notation is complex and difficult to follow.

The approach suggested by Booch[1991] is based on the programming language C++. C++ is conducive to the concept of "openness" which is being promoted within industry. However the code developed is difficult to maintain unless it is written carefully. Further, it is very easy to write "dangerous" code in C++.

2.9.1.4. Jackson Structured Development

Jackson Structured Development (JSD) was developed for systems where timing is important, i.e. it is based on real-world modelling. Jackson's method consists of six steps, Schach [1993]:

1. *Entity action step*. The real-world area of interest is delimited. The entities and actions of the proposed product are listed.

2. *Entity structure step*. The actions performed by or on the entities are displayed using Jackson's notation. The time ordering of the actions is taken into account.

3. *Initial model step*. The entities and actions are represented by a process model.

4. *Function step*. Functions are specified that will result in the required outputs from the product.

5. *Product timing step*. Process scheduling aspects are examined.

6. *Implementation step*. Decisions are taken as to how the process will be mapped to the available processors (hardware).

There are essentially two stages to JSD, steps 1 to 5 which can be combined into specifications and step 6 is the implementation.

JSD is a bottom-up, compositional method and is suitable for environments where the designer is familiar with the application area.

Evaluation

Jackson approach is complex to understand mainly because of its heavy reliance on pseudo code. It has been designed to handle particularly difficult real-time problems and as such has quite a superior design. However associated with the superior design is the increased complexity.

The modules developed using JSD are bound by coincidental cohesion and are counterproductive from the viewpoint of reuse [Schach1993].

In considering the evaluations on the above approaches, it is clear that there are inherent problems of each approach. These problems make them unsuitable for the process control domain, even JSD. Although JSD was developed specifically for real-time systems its complexity makes it an unsuitable choice.

2.9.2. ROOM

An important point to consider is that there is no single modeling technique that is going to suit every application that will ever be built. The key to selecting the appropriate modeling technique is to understand the strengths and weaknesses of the approach, and knowing how it will into the needs of the application at hand. Based on the above research and its evaluations it was found that the methodology that suited the process control domain well is the Real-time OO Modeling model, as proposed by Selic[1994]. The main reason for its selection is its clear mapping of the physical domain to a model. This dissertation discusses the ROOM methodology incorporating all three elements (a modeling language, modeling heuristics and a framework for organising and performing development work), albeit at different levels of detail. The reasons for the selection of ROOM is also encapsulated in this discussion.

2.9.3. Key Elements of ROOM

ROOM is organised around the following three key elements:

- The operational approach.
- A phase independent set of modeling abstractions.
- The object paradigm.

Traditionally, there have been problems in terms of having discontinuities between the various phases of the life cycle. The output from one phase did not necessarily feed as input into the next. ROOM however has tried to eliminate these discontinuities by the use of its modeling language.

In order to facilitate discussion of discontinuities, a simple classification scheme for commonly encountered discontinuities has been introduced by Selic [1994]. The differentiation is as follows:

- *Scope* discontinuities are caused by a lack of formal coupling between the representations of different *levels* of detail.
- *Semantic* discontinuities are caused by a lack of formal coupling between the representations of different kinds of related detail.
- *Development Phase* discontinuities are caused by a lack of formal coupling between requirements, design, and implementation representation.

2.9.4. The Operational Approach

ROOM has adopted graphical modeling techniques. The reason being that a large amount of information can be represented in a compact manner. Further, a single, integrated, formal set of modeling abstractions are used to eliminate discontinuities.

ROOM makes an attempt at this early stage to make the model executable. The details of an executable modeling language for process control systems and of the environment provided for its use on the development platform are dictated by the uses to which the executable models will be put. Making the models executable causes a longer modeling time but it resolves ambiguities. ROOM modeling is normally an iterative process. The models produced by ROOM must be highly observable. That is, it must be possible to clearly convey the semantics of a model as well as its execution. This is an argument for the use of graphical modeling concepts and for the ability to visualise model execution using the same graphics used for model construction, (see Figure 10).

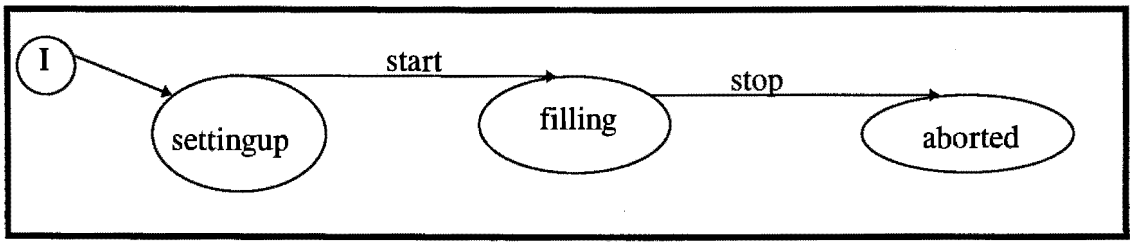


Figure 10. An Executable State Machine

The models

- Will be run on the development workstation, as well as on the implementation hardware.

- Will need to operate even when not connected to real sensors, actuators or other peripheral devices, and thus must be able to simulate these components.

- Will have to provide for stopping and restarting of execution and thus must be able to run with a simulated real-time clock, as well as with a real one.

- Will be able to collect information as well as to do real work.

- Will be used to verify high level, architectural properties of the proposed system, as well as low level detailed properties.

The fact that high level, architectural properties must be verified is a constraint on the modeling language itself, the language must be capable of explicitly expressing such high level properties. However, the other uses of the model can be accommodated by the model-building environment on the development platform and thus the modeling language can be defined without any specific features tailored to these uses. This usage independence of the language component is extremely important for the phase independence of ROOM, which is discussed next.

2.9.5. A Phase Independent Set of Modeling Abstractions

As mentioned earlier, ROOM has made a concerted effort to eliminate discontinuities, by using a single set of modeling concepts across the different life cycle phases. It is not easy to manage this, as each phase (e.g. requirements, specifications, design, etc.) are of varying complexity and the deliverables from each stage are different.

ROOM manages this situation by using OO, and its scope, as a modeling language, was limited in a variety of ways.

- The language was specialised for real-time systems. No attempt was made to create a general-purpose development language.

- The scope was restricted to abstract requirements modeling, abstract high level design, detailed software design and software implementation. No attempt was made to cover detailed hardware design.

- Various features to support requirements and design modeling were incorporated into the model building environment rather than into the modeling language, allowing the language to carry less overhead and thus to execute more efficiently.

The ROOM modeling language as used on a development platform (workstation) is depicted in schematic form in Figure 11 [Selic1994]. Two interfaces are presented to the modeler: the design (model building) interface and the run-time (model execution) interface.

The design interface allows the modeler to use the ROOM modeling language to create and modify models [Selic1994]. A particular model, after compilation, can be executed using the runtime interface. The run-time interface provides for controlling execution, gathering and displaying of information about an execution, tracing of errors and so on. Another software layer, the ROOM virtual machine, provides basic services (such as communications, timing and exception handling) to the executing model.

Figure 12 shows a ROOM model as installed on a target platform (that is, on the hardware on which the system will be implemented) [Selic1994]. Note, that both the detailed design model and the ROOM virtual machine have been ported from the development platform. This allows the code for the implemented system to use the same services (by way of the same interfaces) as on the development platform. It is assumed that a run-time environment, providing basic services such as an operating system kernel, is available on the target platform. It is also assumed that some components of the application (for example, user interface components) were not developed from ROOM models. This latter assumption requires that the interfaces between the ROOM model and the non-ROOM components were developed on the development platform by installing or by simulating these components [Selic1994].

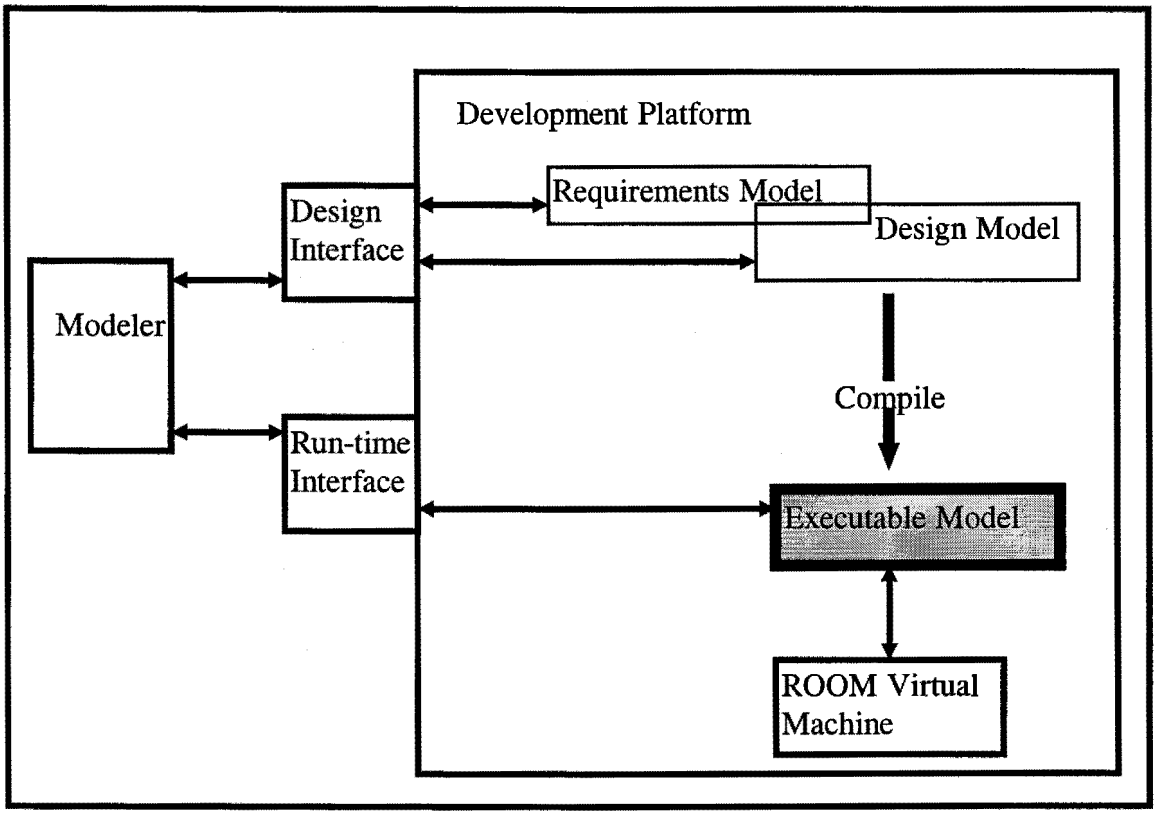


Figure 11. A ROOM Model on a Development Platform

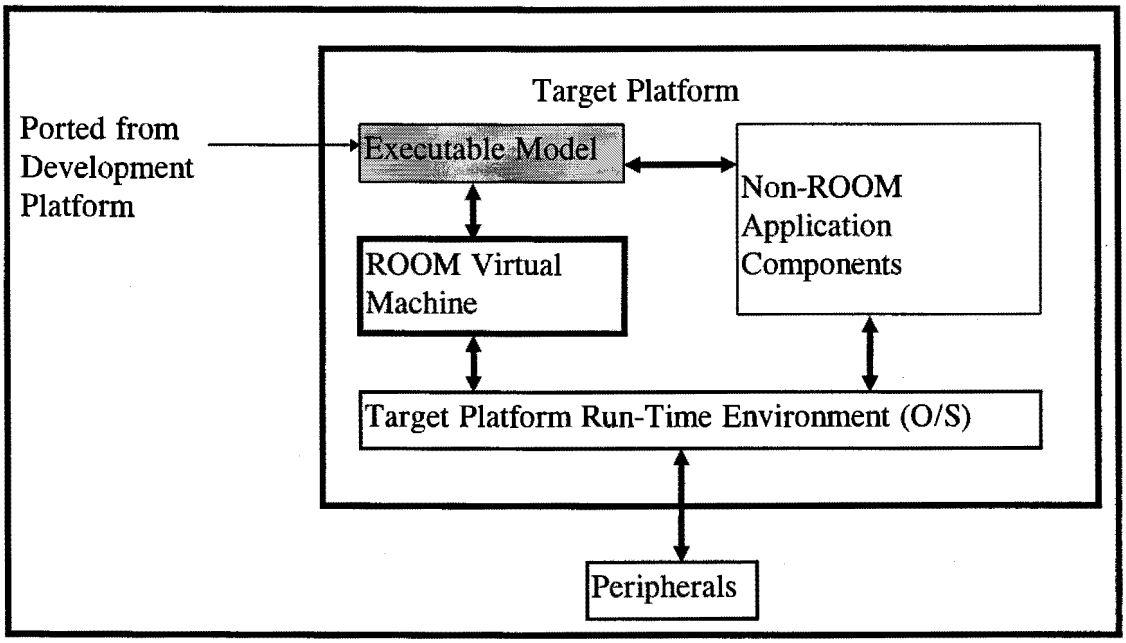


Figure 12. A ROOM Model on a Target Platform

ROOM allows the final form of the design model to be ported to the target hardware and to become the implemented system.

2.9.6. The Object Paradigm

Using the object paradigm (which is the OO paradigm), a system is realised by combining a set of smaller specialised component "machines". These components, called objects, are potentially reusable in a variety of different contexts. For example, one of the objects within a water purification control system is likely to be an operator interface. The same type of operator interface may be used for other applications.

2.9.6.1. Objects as Instances: Abstract Data Types

Consider an instance of an abstract data type as it might be used in the design of a backwashing a filter in a Water Treatment Plant. Assume that a central control processor is currently monitoring and controlling the progress of several filters being backwashed. The monitored data is provided (by way of a data communications line) by remote data collection processors attached to each filter. The data consists of packets, each containing a filter identifier and the time that has elapsed, in minutes, since the start of backwash. A typical packet might contain the following:

filter id: 21
elapsed time: 10

Since a single processor is controlling multiple filter backwashes, it is likely that a new packet of data will arrive while the processor is still working on previous packets. To handle the situation, the design can make use of a very common abstract data type, the First In First Out (FIFO) queue. An instance of FIFO is illustrated in Figure 13. The data is the collection of enqueued packets and the procedures are Put

and Get. The basic idea is that Put dumps a packet on top of the pile and that a Get retrieves a packet from the bottom of the pile.

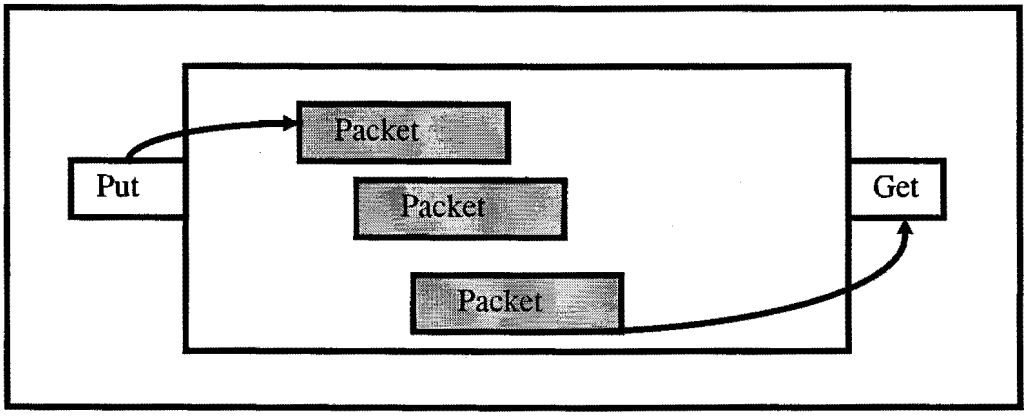


Figure 13. A FIFO Queue

2.9.6.2. Objects as Software Machines

It is feasible to define an object as a software machine because, in essence any computer is a "machine building" machine in the sense that it can be made to behave like some desired specialised machine simply by writing and executing an appropriate program. The same computer can be used to implement a range of different machines, from weather predictors to chess-playing automata.

This can be illustrated with a schematic description of a filter backwashing control system illustrated by Figure 14. The Backwashing Unit Interface provides monitored variable values (elapsed time) to the Backwashing Controller and to the Operator Interface and accepts device commands from the Backwashing Controller. The Backwashing Unit Interface takes care of all the issues involving communications with the Data Collector objects associated with the filters and may provide some level of processing (such as interpolation of missing data values). The Operator Interface isolates the Backwashing Unit Interface and the Backwashing Controller from concerns about displaying information to and accepting commands from, the operator. The Backwashing Controller interacts with the Backwashing Unit Interface and with the Backwashing Specifications objects to establish the

parameters for a particular backwash cycle, and then interacts with the Backwashing Unit Interface to carry out the required control actions.

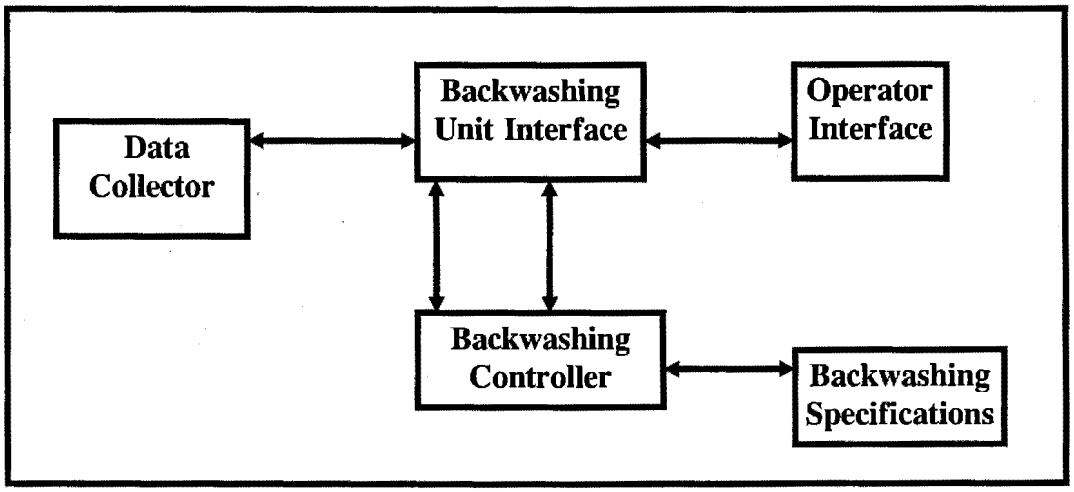


Figure 14. Schematic View of A Backwashing Control System

Software objects, unlike abstract data types, also have the property that they can be (at least conceptually) concurrently active. One can picture, for instance, the Operator Interface accepting an operator command while the Data Collector is delivering a monitored value to the Backwashing Unit Interface and the Backwashing Controller is doing a setpoint comparison.

2.9.6.3. Objects as Logical Machines

The idea of objects as software machines is broader and more generally useful than the idea of objects as abstract data types. However the concept of an object can be made more general.

For example, consider that many routine, commonly performed tasks originally encoded in software have ultimately been embodied in digital hardware circuits. Functions such as signal processing and pattern recognition are commonly performed by specialised chips. Now consider a system component such as the Operator Interface from Figure 14. Portions of this component that require high

performance and that are unlikely to vary may well be implemented in digital hardware. From the point of view of the other systems components whether the Operator Interface is implemented as software or as hardware or as some combination of the two, is irrelevant. These details can be hidden in exactly the same way as the physical data organisation of an abstract data type can be hidden.

Within the context of developing a real-time system, it is useful to be able to

- Represent a system component which may or may not be implemented in software.

- Represent a component that will not be implemented within the computer system at all, but will interact with the components of the computer system.

2.9.6.4. Messages

One of the benefits of the object paradigm is that objects tend to be relatively self-contained and autonomous.

Communications between objects are based on a *message-passing* model and the coupling between objects is made as weak as possible. The essential feature of this model is that information between objects is exchanged by means of an intermediate artifact - a *message*. The purpose of this message is to reduce the coupling between the senders and the receivers. The only thing that a sender and a receiver must share is the format and general semantics of the message. They do not have to know anything about each others' implementation. The sender packages the information that it wants to send into a message and then dispatches it to the destination. When the destination receives the message, it responds by performing the activity appropriate to that message. For example, if the control system in Figure 14 is required to display the elapsed time to the operator; the Backwashing Unit Interface will send a "current elapsed time" to the Operator Interface object and include in it

the real number representing the current elapsed time. The Operator Interface object responds to this message by displaying to the operator the value in an appropriate format.

There is also loose coupling between an object and its environment. This is highly desirable since among other things, it enables the object to be reused in contexts other than the one for which it was originally devised. For instance, the Operator Interface object in Figure 14, probably can be used in other applications that require interaction with an operator. Loose coupling also reduces the likelihood that a change in one component will affect others. Hence the costs of making changes are reduced.

2.10. Conclusion

Building a model of a complex system, such as a process control system, is too complex and error prone a task to be approached without careful consideration of the modeling language to be used. The choice of specific notations, modeling disciplines and technical methodologies can either greatly help or greatly hinder the systems development process. ROOM was selected because it is believed that it can support the complex process control development processes. Further, it is a methodology which closely resembles the physical processes, i.e. the real life situations.

In Chapter 1, it was stated that an attempt would be made to adopt the revised spiral life cycle model. It was compared against ROOM and it was found that although there are similarities between the two, ROOM was preferred. ROOM is more suited to a process control application and it does incorporate some of the steps of the revised spiral life cycle model. A comparison between the two is done via the five cycles in the revised spiral model:

Feasibility. This cycle specifies that a detailed problem statement with the constraints, description of the environment be provided. Contrary to this, with ROOM the problem is started with as much detail as is provided and the "specifier" is approached for more details as the need arises.

Architecture. The top level system software architecture and hardware architecture are determined in this cycle. Strategies are proposed and risk analysis on these strategies are performed. In ROOM, this cycle would occur when the boundary between the system and environment is identified, but a point of note is that just the hardware and software architecture is decided, the risks associated with the various strategies are not clearly defined.

Analysis, Design and Implementation. These three cycles of the revised spiral model are incorporated into iterative cycles within ROOM. In addition early validation of the model is performed against the requirements.

In addition, ROOM provides a number of features that extend the object paradigm to yield increased expressive power for modeling. These features are the ability to create actors that can be used as layers, the ability to allow an actor to appear at multiple places within a model by way of the multiple containment construct and the ability to declare a reference as substitutable with either an actor of a given class or an actor of an interface compatible class.

There are key features of ROOM that facilitate effective modeling of real time systems. These are, as described by Selic[1994]:

- *Timeliness*. ROOMs operational character permits simulation of time-related properties of a proposed system, so that these properties can be estimated from a model on a development workstation. Furthermore, a ROOM model can be ported to the implementation hardware so that critical time-related properties can be measured directly. Finally, the ROOM modeling abstractions are inherently

efficient, so that code generated from a ROOM model will typically be able to meet timing constraints.

- *Dynamic internal structure.* ROOM's support of the object paradigm includes the ability to model explicitly the creation and destruction of system components at run time and to exercise these properties by way of model execution.

- *Reactiveness.* The combination of discrete message passing and the use of event driven state machines to model internal object behaviour permits effective modeling of reactiveness.

- *Concurrency and Distribution.* ROOM supports an extended version of the object paradigm, which represents objects as independent logical machines with separate threads of control that communicate by message passing. Thus, a ROOM model is inherently concurrent and distributable.

The key features of ROOM also support common model-building strategies based on abstraction, as follows:

- *Recursion.* A ROOM object is an arbitrarily complex logical machine; objects may be modeled as being composed of other objects, to any desired depth. ROOM also permits the behaviour of an object to be modeled by a hierarchical state machine in which states may be decomposed into substates to any desired depth.

- *Incremental Modeling.* A partially complete ROOM model may be executed. This permits the expression and verification of model properties in an incremental fashion. Furthermore, the fact that a single notation is used throughout the modeling process, and that objects may be components of more than one model, permits repeated cycling among multiple models and relatively low overhead.

- *Reuse.* ROOMs support of the object paradigm includes inheritance which is a powerful technique for capturing abstractions in general and for supporting reuse in particular.

CHAPTER 3

ASPECTS OF THE TARGET SYSTEM

3.1. Introduction

3.2. Work Organisational Aspect

3.2.1. Product-Oriented versus Project-Oriented Development

3.2.2. Product Requirements

3.2.3. Product Development Activities

3.2.4. Deliverables

3.2.5. Universal Model Relationships

3.2.6. Project Team Organisation

3.2.7. Project Management and Tracking

3.3. Meta Primitives of the Environment Aspect

3.3.1. Hardware Considerations

3.3.2. Interface Considerations

3.4. Meta Primitives of the Information Aspect

3.4.1. Representation of Information

3.4.1.1. Data Independence, Binding and Efficiency

3.4.2. Data Organisation

3.4.2.1. Characteristics of Data in Real-time Database Systems

3.4.3. Manipulation of Information

3.4.4. Interpretation of Information

3.5. Meta Primitives of the Systems Engineering Aspect

3.5.1. Timeliness Issues

3.5.2. Dynamic Internal Structure

3.5.3. Reactiveness Issues

3.5.4. Concurrency Issues

3.5.5. Distribution Issues

3.6. Meta Primitives of the Software Engineering Aspect

3.6.1. High Level Structure Modeling

3.6.1.1. Communications

3.6.1.2. Communications Relationships

3.6.1.3. Actor Structures

3.6.1.4. The Relationship between Structure and Behaviour

3.6.2. High Level Behavior Modeling

3.6.2.1. Events

3.6.2.2. State Machines

3.6.3. High Level Inheritance

3.6.4. The Detail Level

3.7. Process Control Reference Model

3.7.1. Reference Model for the ROOM Virtual Machine

3.7.1.1. The Services System

3.7.1.2. The Control System

3.7.1.3. The Timing Service

3.7.1.4. The Processing Service

3.7.1.5. The Frame Service

3.7.1.6. The Communications Service

3.7.2. Mapping ROOM Specifications into the Traditional Real-time Environment

3.7.3. An OO Conceptual Model for Process Control Software

3.7.3.1. General Aspects

3.7.3.2. Special Requirements for Process Control Software

3.7.3.3. Process Control Framework

3.7.3.4. Communication Library

3.7.3.5. Data Management Library

3.7.3.6. Process Management Library

3.7.3.7. Process Control Library

3.7.3.8. Control Panel Library

3.8. Reference Model for Process Control
3.8.1. Conceptual Reference Model for the Process Control Domain
3.8.2. Conceptual Model in terms of ROOM
3.8.2.1. Process Control Application
3.8.2.1. Co-ordinating System
3.9. Conclusion

3. CHAPTER THREE: ASPECTS OF THE TARGET SYSTEM

3.1. Introduction

The Target System Reference model has to date been defined in terms of the following aspects:

- Environment.
- Information.
- Systems Engineering.
- Software Engineering.

The relevance of each these aspects to the process control domain is described. The implementation of ROOM within the systems engineering and software engineering aspects is also discussed. Reference models as proposed by ROOM as well as a model suitable for the process control domain are identified and elaborated on.

Before the aspects of the Target System Reference model are discussed, aspects that are part of the Development Process Reference Model of the OOISEE are discussed. These aspects are relevant to the process control environment and hence its inclusion in this section. It is referred to as the work organisational aspect.

3.2. *Work Organisational Aspect*

As described in Chapter 2, the work organisational aspect deals with operational and managerial issues. These include:

- What technical processes will be used?
- Who will manage the software documentation?
- Are there sufficient resources (hardware & people)?
- How should the development team be organised?
- Which intermediate development products need to be produced?
- In what order should the development activities be performed?
- How the progress of the work is tracked and controlled?

The organisation, i.e. the co-ordinating and facilitating is an important aspect in the development of software. To ensure the smooth operation of all the phases within the SEE there is a need for the proper resources. Resources can be categorised into two broad aspects, the human aspect and the appropriate tools.

In order to properly manage the organisation aspect, the person in charge has to understand the flow of work within the SEE and the deliverables expected for each phase. Further, she has to be firm in ensuring deadlines are met and that problems are resolved as soon as possible. This is by no matter of means, an easy task. To assist in this, there are CASE and SEE tools on the market.

Some primitives for consideration within the organisation aspect:

1. The distinction between a project-oriented and product-oriented approach to the development process.

2. The identification, organisation and prioritisation of product requirements.

3. The activities in product development.

4. The deliverables involved in product development.

5. The creation and maintenance of sequences of models.

6. The organisation of project teams.

7. Project management and tracking.

The first five form part of the Life Cycle Aspect while the last two are part of the Management Aspect. These primitives are discussed next.

3.2.1. Product-Oriented versus Project-Oriented Development

There are three key requirements in any SE project:

1. A software product to be built.
2. The resources to do the work.
3. A budget to do the work.

As products are being developed, the users are involved in the approval of the deliverables of the various stages of the life cycle. During these stages, they are made aware of the capabilities of a software system and start asking for more. This

is termed "scope creep". Project Leaders often find it very difficult to distinguish between what should actually be part of the current project and what should be considered an enhancement to the product.

If it is part of the current project, the scope will have to be extended and in all likelihood the budget as well. The aim of most projects is to deliver the product within time and budget, so this scope creep must be accordingly accommodated.

If it is an enhancement, it will not form part of the current project. The point to this discussion is that a project has a limited life-span while products are constantly being modified to meet the changing requirements of the environment. Thus, a product can be part of a number of projects.

3.2.2. Product Requirements

An original statement of requirements is very rarely complete. The missing requirements fall into the following categories, see Figure 15, Selic [1994]:

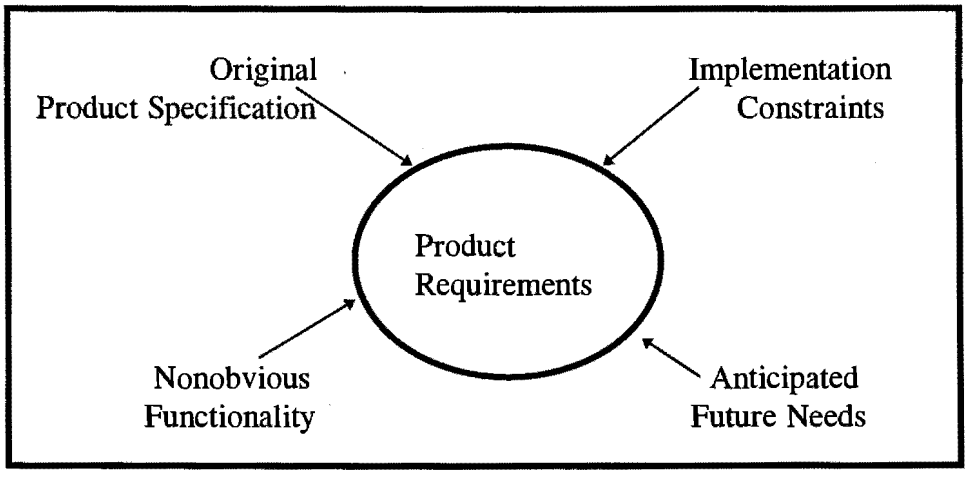


Figure 15. Factors Driving Product Requirements Identification

- *Nonobvious functionality*. Certain requirements may not be identified by traditional requirements analysis methods.

- *Implementation constraints*. The viability of a product may depend on requirements because of implementation constraints.

- *Anticipated future needs*. The product must often evolve to meet needs that are different from the needs that shaped the initial requirements.

In order to establish as complete a requirement as possible, it is very important to model the existing system be it manual or computerised. To achieve this objective, it is desirable to hold what is termed a Joint Application Development (JAD) session. The attendees should all be users of the current system and proposed users of the desired system. This could be a number of people, so ideally a part of the product should be handled at a time. Each session should be of a limited time span, perhaps a morning but it should be a continuous process. That is, it should be continuous sessions every morning for a week or two, depending on the scope of the product.

The selected methodology, ROOM, has the ability to capture high level architectural abstractions which makes it possible to identify and preserve features that must remain intact if a product is to evolve successfully. Also, the ability to use executable models provide an objective basis for requirements elicitation and negotiation involving customers and developers, often leading to the early uncovering of missing requirements [Selic1994].

3.2.3. Product Development Activities

The product development and product modeling activities are commonly categorised as analysis, design and testing. Within a SEE, it is desirable to have these activities flow one into the other smoothly. That is, the output of one phase must feed into the next phase as input.

In traditional modeling techniques, the activities of analysis, design and testing tend to be interleaved as suggested in Figure 16 [Selic1994]. This tendency is enhanced by the use of ROOM. The availability of single modeling notation encourages cycling back and forth between analysis and design activities and the existence of executable models tends to spread testing more evenly throughout the development process.

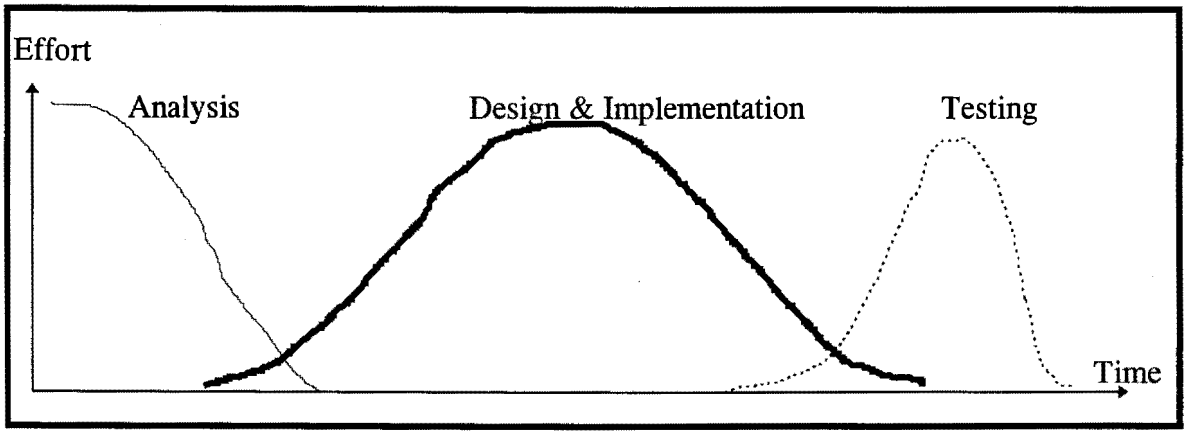


Figure 16. Distribution of Activities in Conventional Development

Figure 17 suggests the nature of the interleaving of the product development activities of analysis, design and validation using ROOM as contrasted with the conventional distribution of activities of Figure 16.

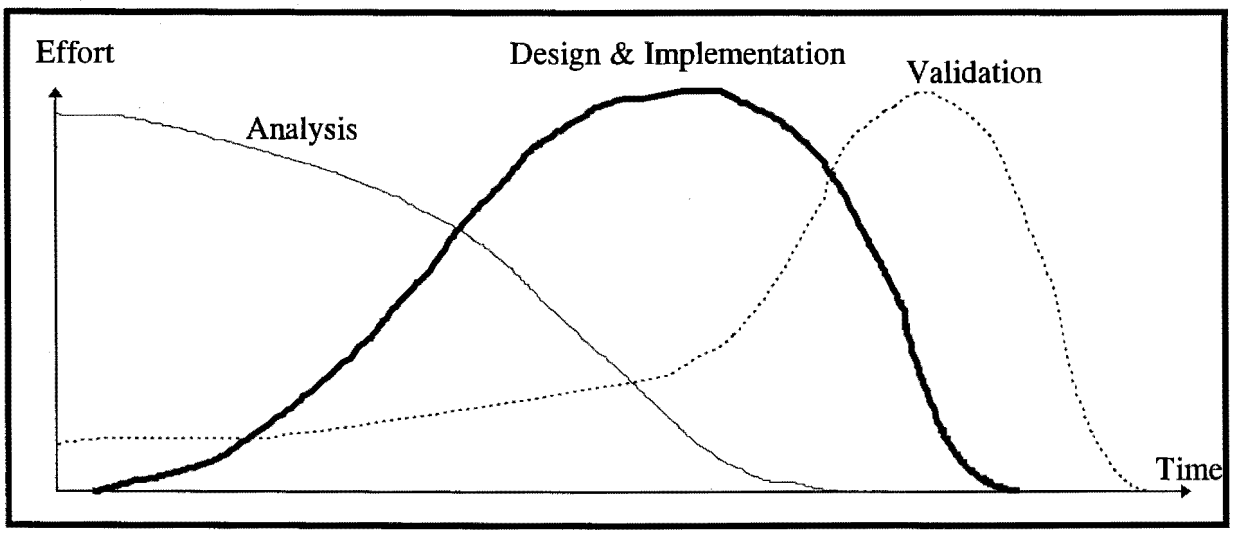


Figure 17. Interleaving of Product Development Activities in ROOM

3.2.4. Deliverables

Traditionally, the deliverables from most phases of the life-cycle was paper based documentation. To assure the quality of this documentation proved to be quite a tiresome task. In addition, text based documentation is known to contain ambiguities. It would be preferable to have a combination of both graphical and text based models. With the rapid advances in technology, it is now becoming possible to achieve this. Some methodologies take it one step further, that is executable models are generated in the early stages of the life cycle. ROOM is one such methodology.

ROOM's approach is that the executable models produced during development can have a much larger impact than traditional non-executable models. In particular, it is possible that the large narrative documents often defined as intermediate deliverables of a product development project can be replaced by much shorter documents accompanied by executable models. A narrative document, at best, can describe the features of a system under development to an interested party. An executable model, when provided with a suitable graphic interface can be used to prototype the features of a system under development to a customer or end-user.

3.2.5. Universal Model Relationships

A number of methodologies include many iterations of the product model during its life cycle. These iterations are necessary as omissions are discovered, or ambiguities are clarified or when there are additional requirements. Normally, this product model is a single model which is constantly modified to accommodate these iterations. Obviously, with each iteration the model is more than likely going to increase in complexity and size. The maintenance of this model will be quite cumbersome. Figure 18, is an example of such an incremental approach.

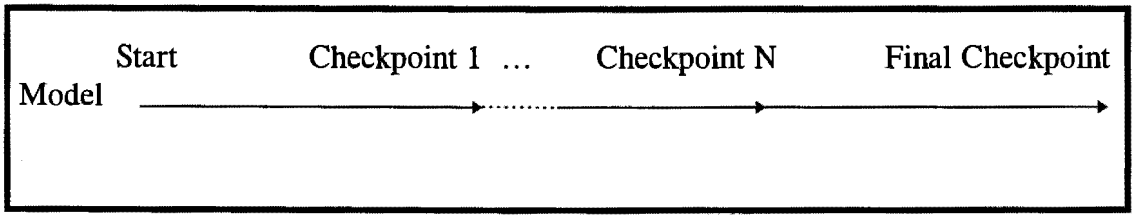


Figure 18. Development via Stages of a Single Model

The potential impact of ROOM on the generic modeling strategies just described is to change the cost-benefit ratio [Selic1994]. In general, the increased productivity caused by the use of ROOM can be "re-invested" in preserving and maintaining intermediate models. More specifically, two features of ROOM aid in the preservation and maintenance of intermediate models. First, the use of a single notation means that models can be preserved simply by evolving the original. Second, the fact that a ROOM model consists of a library of class definitions means that some of the class definition can be part of two or more models.

3.2.6. Project Team Organisation

According to Booch[1991], the skills of the members of the development team are as much as four times more important than the methods and tools used on a project.

One way of increasing productivity is to decrease the size of the development team for a given project. A small number of developers can mean simpler communication channels among developers and fewer levels of project management. Given a fixed pool of developers from which team members are drawn, a smaller team also means that, with judicious selection of team members, the average skill level of the team can be higher [Selic1995]. There is a shift in industry to a flatter structure within organisations. This means that staff will become multi-skilled, that is they will no longer be put in "pigeon-holes". They could have skills for which they have no qualifications but could perform better than those qualified. These skills will be exploited. The staff themselves will be happier as there will be more variety in their work. All staff will be viewed as peers regardless of whether they are managers or

not. The focus of this new structure is to **deliver**. Computer systems are notorious for being late on delivery, can make use of this new paradigm to try to uplift the computer sections' image

The advantages of having smaller teams is that each person has a specific task/responsibility and has to produce; there is no getting lost in the crowd. Subtle peer pressure is also evident in smaller teams.

The shift in project activities caused by ROOM also suggests that the specific mix of skills required on a team may be different. For example, the increased importance of architectural thinking suggests that developers with the relevant experience should be explicitly incorporated into the team as *architects* from early in the project, so that they can "live" with the architecture as it evolves, continually steering it based on continuous feedback from the team.

People of different cultures and personalities have different input into a team. According to research on peoples personalities/characters, there are nine categories (e.g., dictator, analyst, creator, etc.)[Prekel1995]. It is believed that each team should have the appreciate mix of these personalities, The trend now is not only to have the right mix of peoples' skills in a team but also the appropriate types of personalities.

Finally, a transitory (but extremely important) team issue is the management of the culture change introduced by the paradigm shift from traditional functional development to the use of OO executable models. Providing for education of team members and managing anxiety about or resistance to, the new approaches is an important aspect of project management.

3.2.7. Project Management and Tracking

One of the reasons that computer sections have such bad images is that when a request is made from a user, the request is acknowledged and then the development team seem to disappear into a huddle. The next instance the user sees this team is when the product is completed. Undoubtedly, without the users' involvement this product is bound to contain anomalies and may not necessarily meet the users' expectations. The majority of the users' of today are computer literate and would prefer to be involved in as much as is possible. It is quite important that the users feel part of the development team, because she is eventually the custodian of the system. This is a good political step to get the users' buy-in.

On the other hand, with the users' involvement her expectations could be raised quite high on seeing the rapid delivery of prototypes. The developers could be swayed into releasing the product before it is ready for release, the product could have not undergone proper quality assurance. This only worsens the situation. Therefore, there has to be the right balance of users' involvement.

Using OO technology to track projects is relatively new and the tracking of projects that combine OO and executable modeling is even newer. Some suggestions from Selic [Selic1994]are:

In general, executable modeling permits intermediate models to be used as project milestones. The progress of a project ought to be closely related to the number of scenarios that can be demonstrated to be satisfied by the two models. The progress of a project ought to be closely related to the number of scenarios that can be demonstrated to be satisfied by the existing models.

Two figures of merit, may be useful for fine grained tracking of project convergence. The first of these is the *rate of flux of high level, architectural actor and protocol classes.* Since these easily identified classes are used to capture

architectural abstractions, a reduced rate of change in them should indicate that the system architecture is stabilising.

Similar information about the progress can also be obtained from the *depth of its class hierarchies*, particularly when monitoring key architectural classes. Namely, deep class hierarchies may be indicators of mature, well understood abstractions and hence of a stable design.

3.3. *Meta Primitives of the Environment Aspect*

Careful consideration must be given to configuring the environment within which a process control target system will reside in (hardware considerations) as well as to its interfaces (interface considerations) to the real-world. It is necessary to focus on the environment because it forms the basis for constructing a model of the behaviour of a system. This is illustrated in Chapter 4. The meta primitives are:

- Hardware Considerations
- Interface Considerations.

3.3.1. Hardware Considerations

Process control applications operate in real-time. Time is a critical factor in these systems and there are a few hardware issues that can affect the response/timing of an event, e.g. event scheduling, synchronising of clocks. These issues are discussed.

Special hardware support for collecting run-time data in process control systems has been considered in a number of recent papers [Son1995]. These approaches introduce specialised co-processors for the collection and analysis of run-time information. The use of special-purpose hardware allows non-intrusive monitoring

of a system by recording the run-time information in a large repository, often for post analysis. The under-utilisation of a central processing unit due to the use of scheduling methods based on the worst case execution times of tasks is addressed by the use of a hardware real-time monitor which measures the task execution times and delays due to resource sharing. The monitored information is fed back to the operating system for achieving an adaptive behaviour.

It is difficult to detect violations of timing assertions in an environment in which the co-operating real-time tasks run on multiple processors, and timing constraints can be either inter-processor or intra-processor constraints. Since timing constraints may be imposed on events across multiple processors without physical shared memory to store event histories, several additional issues must be addressed.

- First, it is desirable to detect potential constraint violations at the earliest possible time. It may allow the system to take corrective action before the violation actually occurs. This is done by deriving and checking intermediate constraints from user-specified end-to-end constraints.

- Second, since events happen on different processors and timing constraints can span processors, some form of inter processor communication is needed to propagate this information. Minimising the number of extra messages is crucial for reducing overhead.

- Third, when an event occurs there must be a way of recording the occurrence time of the event. The granularity of timestamping determines the minimum observable spacing between two consecutive events on a processor. Timestamping is typically done by reading the clocks on the local processor. A distributed process control system must also deal with the fact that clocks on different processors are not perfectly synchronised. The processor clocks, however, can be kept synchronised within a known maximum bound on the deviation between them. Clock synchronisation allows controlled comparison of time stamps from different clocks. In particular, one must take into

consideration the deviation between clocks when evaluating a timing assertion at run-time.

A difficult problem is the determination of tight timing information on instructions and code sequences for contemporary computers; pipelining, caching and a host of other performance-enhancing features seen to hinder timing predictability. By selecting appropriate granularities for the higher level language elements, these and other hardware issues, such as exactly how and where to incorporate worst-case effects of memory and bus contention, can be handled practically.

3.3.2. Interface Considerations

The interfaces to a process control application can be either humans or devices. An example of a devices is a PLC. There is continuous interaction between the application and the PLC which the system must be capable of handling. Simultaneously, one or more operators may be attempting to access the application. There is therefore, a need for the application to be capable of handling all the PLC as well as the operators transactions, i.e., need for concurrency.
The interface between the operator and the system is on-line and interactive. Hence, the response time must be fast. If the same data is being accessed/updated by both the PLC and the operator, the access/update must be synchronised. If there is no synchronisation there is a potential for inconsistent data to be stored on the system.

3.4. Meta Primitives of the Information Aspect

Process Control systems utilise real time databases. As mentioned in Chapter 2, data in real time databases have to be logically and temporally consistent. Some of the timing constraints on the transactions that process real-time data come from this need. These constraints, in turn, necessitate time cognisant transaction processing so that transactions can be processed to meet their deadlines.

A real time database system provides database features such as data independence and concurrency control while at the same time enforcing real-time constraints that applications may have.

The meta primitives of the information aspect are:

- Representation of Information with attributes of data independence, binding and efficiency.
- Data Organisation with the data models (network, hierarchical, relational and object) as attributes.
- Manipulation of Information with attributes such as read, write, delete.
- Interpretation of Information with attributes such as timing, priority, periodicity.

3.4.1. Representation of Information

3.4.1.1. Data Independence, Binding and Efficiency

There is generally a single globally accessible shared data area, whose organisation is defined by the data definition facilities of the chosen programming language; every task in the system that shares data is compiled with the definition of the entire shared area and thus has access to all the data at any time. There are several problems with this scheme:

- First, no protection is provided for any of the data elements; if a data element exhibits an incorrect value there is no indication as to which task modified the element since all tasks have access.

- Second, any change in the structure of the stored data will lead to recompilation of all tasks in the system. Since all tasks assume the same structure for the data, then if any one task changes its view, all tasks must follow suit. Typically this is time consuming and error-prone which leads many programmers into taking "short cuts", such as finding an unallocated area of memory and making private agreements on the use of this area between the tasks that share it.

- Third, data access mechanisms are specific to each task, which leads to duplication of effort and further dependence of the present data structure.

Clearly, this scheme has significant disadvantages; however it is fast. References to data locations can be resolved at compilation time so that data elements can be accessed simply by a reference to an address.

In its attempt to overcome these problems, a key objective of data management technology is the achievement of data independence; that is each task should be unaware of the storage structure and access mechanism of the data accessed. If data independence is achieved, stored data can be re-organised with no impact on the code of each task. Data management technology achieves data independence by interposing one or more layers of system software between the application task and access to the data itself. In this way, a call is made to a system software task which then uses a data description to access the data item. These "additional" layers of code may introduce inefficiencies into the implementation, which causes problems in a process control system.

The efficiency and independence objectives are clearly in conflict. The two objectives can be studied together by introducing the concept of binding. In the case of a single globally shared data area whose definition is compiled into each process, the names are bound to locations at compile time or at link/load time. However, the use of a procedure to access a data item will delay the binding until run time. It is

the execution of the binding at run time that causes the use of data management technology to be less efficient yet more independent [Ward1985].

Figure 19, from [Ward1985], summarises the independence and efficiency characteristics of different binding times.

	Binding Time	Impact of Changing Data Definition	Impact on Run-time Efficiency	Data Location Associated With
Increasing Data Independence	Run-Time	None	Slows down Processing	Task Independent Data Description
↑	Initialise Time	Restart Task	Slows down Initialisation	Task Independent Data Description
↕	Load/Link Time	Relink/ Reload Task	None	Data Structures in Loaded Program
↓	Compile Time	Recompile Time	None	Data Structures in Compiled Program
Increasing Efficiency	Code Time	Re-code Task	None	Processing Instructions in Compiled Program

Figure 19. Characteristics of Various Strategies for Stored Data Access

3.4.2. Data Organisation

There are two faces to data organisation implementation; the view that is presented to the designer and programmer and the actual physical organisation of the data on the chosen storage media. The designer and the programmer will want to perform operations on the data structures according to an external view of how the data is grouped and organised; this can be separated from the internal data structure chosen for the data. It is the business of any data management software to provide mechanisms to manipulate the external data structure which can then be translated into operations on the internal data structure.

There are four basic approaches to organising data structures, each of which applies to both the external and internal models. Often it is natural to use the same approach for both models; though it is by no means necessary.

These approaches are:

- the relational approach, in which the basic data structure is a table, or
- the hierarchical approach, in which data is stored in hierarchical trees and there is only one path to a given data category, or
- the network approach, in which data is stored in a network of nodes and there may be several paths to a given data category, or
- the object approach, in which the data is stored as objects and communicated with by messages.

The approach chosen will affect the operations that can be carried out on the data structures.

Regardless of the type of data base selected, databases are useful for the process control systems because they combine several features that facilitate :

1. the description of data,
2. the maintenance of correctness and integrity of the data,
3. efficient access to the data and
4. the correct executions of query and transaction executions in spite of concurrency and failures.

Specifically,

- database schemas help avoid redundancy of data and its description,

- data management support, such as indexing, assists in efficient access to the data, and

- transaction support, where transactions have atomicity, consistency, isolation and durability properties, ensures correctness of concurrent transaction executions and ensures data integrity maintenance even in the presence of failures.

However support for real-time data base systems must take into account the following:

- First, not all data in a real-time database are permanent; some are temporal.

- Second, since timeliness is sometimes more important than correctness, in some situations, precision can be traded for timeliness.

3.4.2.1. Characteristics of Data in Real Time Database Systems

A process control system consists of a controlling system and a controlled system. The controlled system can be viewed as the environment with which the computer and its software interacts.

The controlling system interacts with its environment based on the data available from various sensors. It is imperative that the state of the environment as perceived by the controlling system, be consistent with the actual state of the environment to a high degree of accuracy. Otherwise the effects of the controlling systems' activities may be disastrous. Hence, timely monitoring of the environment as well as timely

processing of the sensed information is necessary. In many cases the sensed data is processed to derive new data.

In addition to the timing constraints that arise from the need to continuously track the environment, timing correctness requirements in a process control system also arise because of the need to make data available to the controlling system for its decision making activities.

The need to maintain consistency between the actual state of the environment and the state as reflected by the contents of the database leads to the notion of temporal consistency. Temporal consistency has two components:

1. Absolute consistency - between the state of the environment and its reflection in the database. This arises from the need to keep the controlling system's view of the state of the environment consistent with the actual state of the environment.

2. Relative Consistency - among the data used to derive other data. This arises from the need to produce the sources of derived data close to each other.

3.4.3. Manipulation of Information

Process Control transactions are characterised along three dimensions:

1. The manner in which data is used by the transactions.
2. The nature of the time constraints
3. The significance of executing a transaction by its deadline or more precisely, the consequence of missing specified time constraints.

Further temporal consistency requirements of the data can lead to some of the time constraints for the transaction.

Real-time database systems employ all three types of database transactions, i.e.,

1. Write-only transactions
2. Update transactions
3. Read-only transactions

The above classification can be used to tailor the appropriate concurrency control schemes.

Some transaction time constraints come from temporal consistency requirements and some arise from requirements imposed on system reaction time. The former can typically take the form of periodicity requirements: For example,

Every 5 seconds, sample the reservoir level.

System reaction requirements typically take the form of deadline constraints imposed on aperiodic transactions. For example,

IF Water_Flow > 100
add Chlorine within 10 seconds.

In this case, the system's action in response to the high Water_Flow must be completed by 10 seconds.

Transactions can also be distinguished based on the effect of missing a transaction's deadline. As discussed in the section on the Temporal paradigms the terms of hard, soft and firm transactions have been discussed. This categorisation shows the value imparted to the system when a transaction meets its deadline.

The processing of real-time transactions must take their different characteristics into account.

3.4.4. Interpretation of Information

In the process control domain, a real time database consists of a set of data objects representing the state of an external world controlled by a real-time system. The data objects are interpreted as two types: continuous and discrete.

Continuous data objects are related to external objects continuously changing in time. The value of a continuous data object can be obtained directly from a sensor (image object) or computed from the values of a set of image data objects (derived object) within a regular period. Discrete data objects are static in the sense that their values do not become obsolete as time passes, but they are valid until update transactions change the values.

Different from non-real time data objects found in traditional databases, continuous data objects are related with the following additional attributes:

- A timestamp indicates when the current value of the data object was obtained.
- An absolute validity duration is the length of time during which the current value of the data object is considered valid.
- A relative validity duration is associated with a set of objects used to derive a new data object.

A continuous data object is in a correct state if and only if the value of the object satisfies both absolute and relative temporal consistency while a discrete data object is in a correct state as long as the value is logically consistent.

Note that there is only one "writer" for each continuous data object and that its value can be used as long as it maintains temporal consistency. Thus, serialisability

and recoverability of transactions, on which most conventional databases depend to maintain their correctness may not be necessary for these kinds of data objects.

Generally, a real-time transaction can have the following attributes:

1. Arrival Time.
2. Periodicity.
3. Timing Constraints.
4. Priority.
5. Execution Time Requirement.
6. Data Requirement (Read/Write).
7. Criticalness.
8. Value Function.

As the RTDBS utilises the unique characteristics of real-time data and transactions, it can make more efficient decisions in processing transactions and thus improve overall system performance.

3.5. *Meta Primitives of the Systems Engineering Aspect*

This aspect focuses on those engineering principles relevant at the operating system and networking level [Steenkamp1995]. The space/time trade-off, representing a particular set of options in the form of parameters, must be determined.

The salient properties which form the meta primitives characterising the systems engineering aspect of process control systems are:

- *Timeliness of function.* This is the most common attribute of real time systems of any kind. By definition, a real time system is required to perform its function "on time", whatever that happens to mean in a particular context.

- *Dynamic internal structure.* Many real time systems are required to exercise control over an environment whose properties vary with time. This requires that the system components dealing with the particular aspects of the environment must be dynamically reconfigured to match the dynamic of external environments. Because of limited resources (memory, processor capacity), this typically entails the dynamic creation and destruction of software components.

- *Reactiveness.* A *reactive system* is one that is continuously responding to different events whose order and time of occurrence are not always predictable.

- *Concurrency.* This is a feature of the real world in which a real time system is embedded. At any given time, multiple simultaneous activities can be taking place in the real-time system. When this is combined with the need for real time response, the usual result is that the real-time system itself must be concurrent.

- *Distribution.* A distributed computing system is one in which multiple computing sites co-operatively achieve some common function. Distribution is either inherent (as is the case for communications systems) or it may be driven by the need to increase throughput, availability or functionality [Selic1994].

3.5.1. Timeliness Issues

In process control systems, time is clearly a dominant issue. For example, if a remote reservoir reaches a "high high" level an alarm needs to be sounded immediately and not a few hours later. The timeliness issue boils down to reducing the following two specific time intervals:

- *Service time.* This is the net time taken to compute a response to a given input. It is primarily a function of the algorithm used in the computation and is often deterministic and predictable.

- *Latency*. This is the interval between the time of occurrence of an input and the time at which it starts being serviced.

The sum of these two intervals for a given input represents the overall *reaction time* for that input. Naturally, this time interval should be shorter than or equal to the deadline specified for this type of input. Ensuring that a system always meets its deadlines is complicated by the presence of variable delays.

Different systems will have different requirements for meeting deadlines. Some systems where even missing a single deadline is considered unacceptable, is the so called *hard* real time systems which is discussed under Paradigms for Modeling Temporal Behaviour.

3.5.2. Dynamic Internal Structure

A common source of complexity in process control systems is the need for reconfiguration of a system as it is running, based on the dynamic changes in the external environment. Resource management leads to several difficult problems. Resources that are shared, such as memory or disk has be managed effectively and efficiently. Algorithms that manage these resources must be fast as there is normally a "hard" time constraint on the response time. Further, as the environment is real-time, there is a constant "state of emergency" as critical events are waiting for access to the shared resource. The complication increases when there is a need for dynamic creation and destruction of system components and relationships. The cherry on the top of all this is when an exception situation occurs, i.e. an error condition. This leads to not only proper resource management but also priority management on the events

One of the requirements of a modeling language that arises from the dynamic internal structure, is the ability to create models of systems whose internal structures can be dynamically modified at run-time.

3.5.3. Reactiveness Issues

The process control environment is classified as a reactive system which means that there is a continuous interaction with the environment. In the process control environment the interest is in a subclass of reactive systems that conform to the following definitions:

- *Nondeterminism*. The system has no control over the relative order of time of occurrence of input events.

- *Real-time response*. The system must provide a timely response to all input events.

- *State dependence*. The response of the system to a given input depends on previous inputs and time.

Nondeterminism is simply a reflection of the unpredictable nature of the real world in which such a system functions. In some systems, it may be possible to process events in the most convenient order rather than in the order of occurrence, [Selic1994]. However, in process control systems, this is generally not possible because of the timeliness requirement. For example, when a valve fails, it may not be acceptable to postpone the recovery action for a later time. A term commonly used to describe this kind of system is called *event driven.*

3.5.4. Concurrency Issues

A concurrent system contains two or more simultaneous threads of control that dynamically depend on each other in order to fulfil their individual objectives. This interaction between threads is at the heart of most difficulties in dealing with concurrent systems. The problem lies in the inability to construct effective mental models of dynamic relationships. Specifying the behaviour of concurrent systems

requires simultaneous awareness of multiple entities and their progression relative to each other, as well as with respect to absolute time. This is complicated by the fact that threads may progress at different speeds (particularly in distributed systems) so that the number of possible combinations quickly becomes very large and intractable to the human mind. Part of the reason is that time, as humans perceive it, is a continuous quantity with an infinity of values and no recognisable boundaries.

The two primitive forms of interaction between threads are

- *Synchronisation.* Synchronisation involves adjusting the timing of the execution of an action step in a thread based on the execution state of other threads. Synchronisation may be required either to achieve non-interference between threads (mutual exclusion) or to ensure proper interaction between them.

- *Communication.* It is often necessary to pass information from one thread to another. This can take on many different forms, including global shared memory, message passing, remote procedure call and rendezvous.

3.5.5. Distribution Issues

By far the most difficult aspect to handle, is that of distribution. There has been discussion on distribution in the previous sections, but emphasis has to be placed on this issue. Process control systems by nature are distributed. In general, the following are major design issues that accompany distributed systems:

- *Concurrency*. This issue is being handled by the newest operating systems which have incorporated such functionality into it.

- *Unreliable communication media (lost messages, corrupted messages, and so forth)*. Needless to state, this has caused the most problems. One way of handling this is by incorporating strict error checking mechanisms within the messages. In

terms of having a lost communications line, most organisations are installing redundant links between their critical sites. An example of this is, if a telephone link is used as the primary transport mechanism, the router will be programmed to make use of perhaps a microwave link if it detects that the telephone line is unavailable.

- *Prolonged and variable transmission delays.* Those organisations that have a need for rapid transmission of data will tend to use more modern technology to achieve the "hard" real time constraints. A telemetry system which is radio based is therefore unacceptable in such a situation. However, there is a price to pay for the new technology; the costs and sufficient skills base to manage it.

- *The possibility of independent partial system failures.* This is a complex situation as data could be corrupted and could infiltrate the entire distributed system. One approach would be to isolate the failed system and disconnect it from the distributed network until the problem on it is resolved. The current trend in database development is that data is buffered at the distributed sites if a communications break is detected. Once the communications link is restored, all the outstanding information is transmitted to the database that was "disconnected".

3.6. Meta Primitives of the Software Engineering Aspect

This is the one of most important aspects within the process control domain. The success of a software system is highly dependent on the selection of the appropriate software engineering method and supporting techniques.

This section is loosely based on Selic [Selic1994], and the reader is pointed to the original text for further detail.

Some of the criteria used for the selection of ROOM include capabilities to:

- handle real-time events,
- handle concurrency,
- manage distributed databases,
- communicate across different platform, i.e. support different protocols on the same system, and
- support object orientation.

The modeling concepts of ROOM can be classified in either of the following two ways:

- According to the scope or granularity of objects that are under consideration.

- According to the modeling concept that they address.

The first classification scheme leads to the Abstraction Levels paradigm, in which concepts are classified as addressing either low-level detail concerns (the Detail Level) or higher-level distributed system concerns (the Schematic Level) [Selic1994].

The second classification decomposes the modeling space into three independent dimensions representing different degrees of freedom in modeling:

- Structure - which deals with the static aspects of a system,

- Behaviour - which deals with the dynamic aspects, and

- Inheritance - which cover abstraction and reuse characteristics.

The combination of the two classification schemes is referred to as the conceptual framework of ROOM (see Figure 20) [Selic1994].

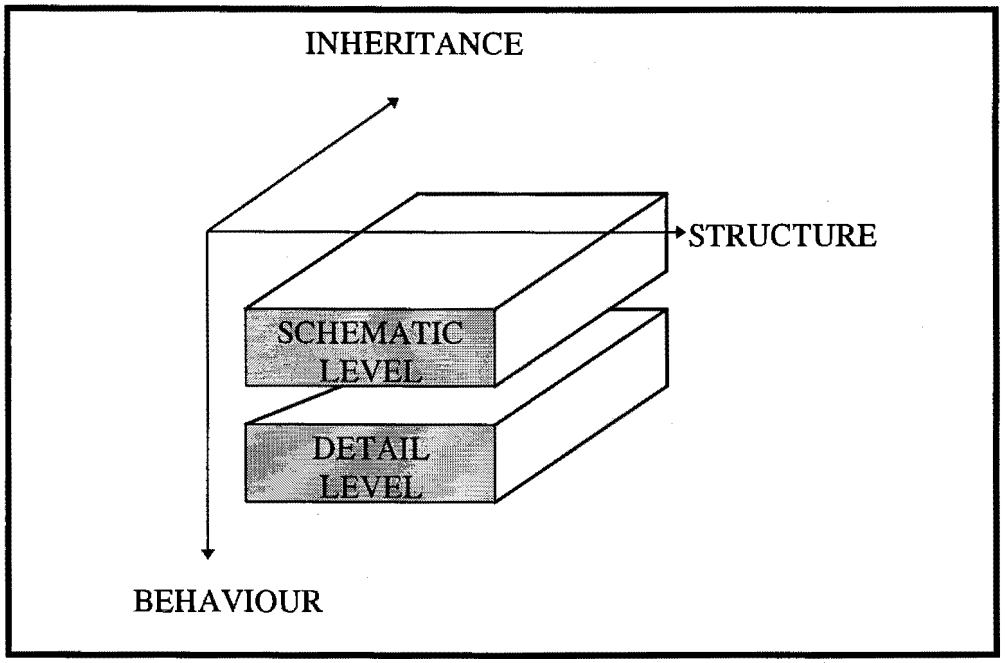


Figure 20. The Conceptual Framework of ROOM

The separation of structure from behaviour provides a convenient way for organising modeling activities. Structure captures primarily the static aspects of a system whereas behaviour addresses the dynamics, resulting in two separate dimensions in some abstract "modeling space" (see Figure 21).

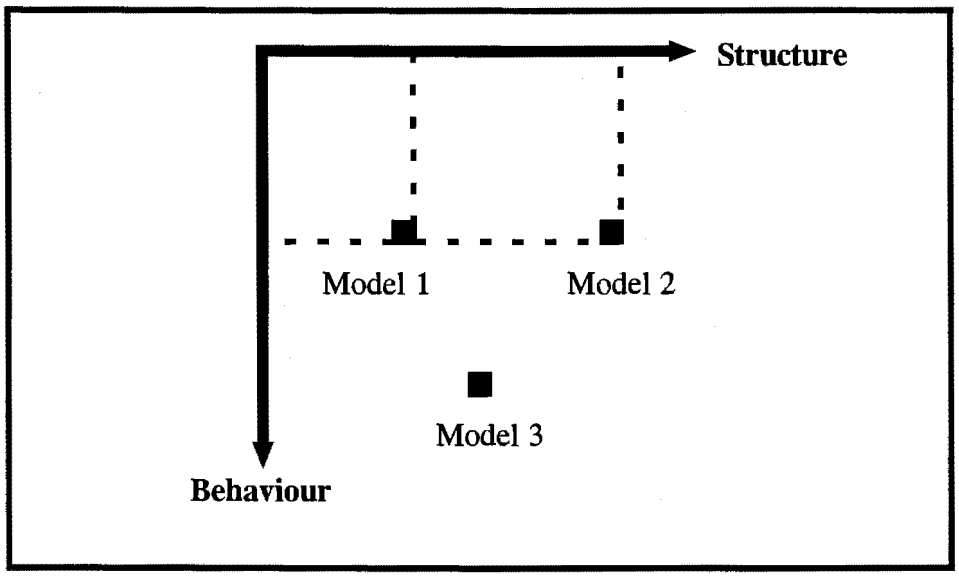


Figure 21. The Dimensions of the Modeling Space

A particular model represents a point in this space. For example, the models identified by points labelled Model 1 and Model 2 in Figure 21 are two different models that have common behaviour but different structures. To fully specify an operational model, both the structure and behaviour must be specified.

The meta primitives of the software engineering aspect are:

- High Level Structure Modeling with attributes as follows:
  - ◊ Actor, Actor Interface Components.
  - ◊ Communication Types - Messages and Protocols.
  - ◊ Communication Relationships.
  - ◊ Actor Structures - Generic, Composite, Functional and Multiple Containment.

- High Level Behaviour Modeling

- High Level Inheritance Modeling

- Detail Level with attributes - Exception Service, Timing Service, Frame Service and Communication Service.

3.6.1. High Level Structure Modeling

In terms of the ROOM modeling framework, the high level structure lies primarily along the structural dimension of the Schematic Level as shown in Figure 22 below [Selic1994].

As discussed in Chapter 2, reusability is fundamental to the OO paradigm. ROOM has adopted this reusability strategy at the High Level Structure dimension. A special concept defined as an actor is its basic modeling concept.

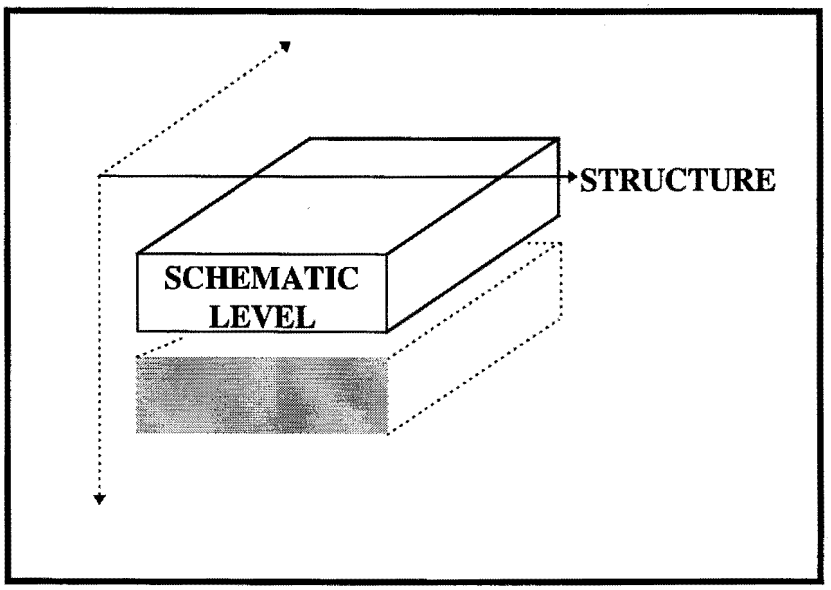


Figure 22. The ROOM Conceptual Framework: High Level Structure

In order to achieve reusability, each actor has to have one specific purpose with well defined interfaces to other components. An actor can have more than one interface as it can communicate with different types of actors. With this structured architecture, actors can co-exist with other actors concurrently in its domain. The process control domain is composed of concurrent objects and it is this functionality which makes ROOM a suitable choice.

An actor makes use of encapsulation to maintain its primary purpose. The reasons for using encapsulation is that it is necessary to ensure that the coupling between actors is restricted to only the interactions across the interfaces.

In order to satisfy the communications between actors, ROOM has defined three actor interface components:

1. ports,
2. service access points (SAPs)
3. service provision points (SPPs).

Ports are used for communication across objects in the same layer while the other two are used for communications across objects in different layers.

The graphical representation of an actor is illustrated in Figure 23.

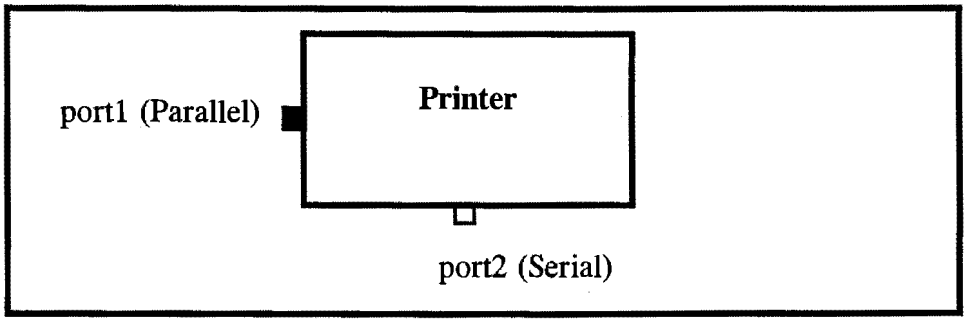


Figure 23. Graphical Representation of an Actor (Printer)

The label (Printer) corresponds to the name of the reference and must be unique in context. If the corresponding actor class has ports (Parallel/Serial), then these are represented by small squares on the perimeter of the rectangle. These may be filled in black depending on whether their type is based on the unconjugated or conjugated protocols receptively. The names of the ports corresponds to the names of appropriate ports in the actor class definition. As was mentioned previously, actors can have multiple ports as different actors may view the same actor in different ways. This is extremely useful in real-time systems in which the most concurrent objects simultaneously collaborate with two or more other actors.

The significance of individual ports may depend on the specifics of the application in which the actor appears (actor specifications are potentially reusable in multiple applications). It is possible that in some applications one or more ports of an actor may be intentionally left unbound, i.e. it will not be connected to another actor. This indicates that the activities normally undertaken through these ports are not relevant for this application. On the other hand, if a port is bound that means that the interaction achieved through that port is critical to this application.

3.6.1.1. Communications

When employing OO in the process control domain, the system is not viewed as a set of transformations to be performed on the data but rather as a collection of concurrently active communicating components across distributed networks. Communication within ROOM is achieved via message passing and protocol passing either as synchronous (blocking) or asynchronous (non-blocking) communication. These two techniques, messages and protocols are handled next.

Messages

The composition of the ROOM message has two compulsory fields, that of:

1. Signal attribute which is a symbolic value.
2. Priority which determines its importance between two objects.

An optional field is that of a data object, Examples of these two types are illustrated in Figure 24.

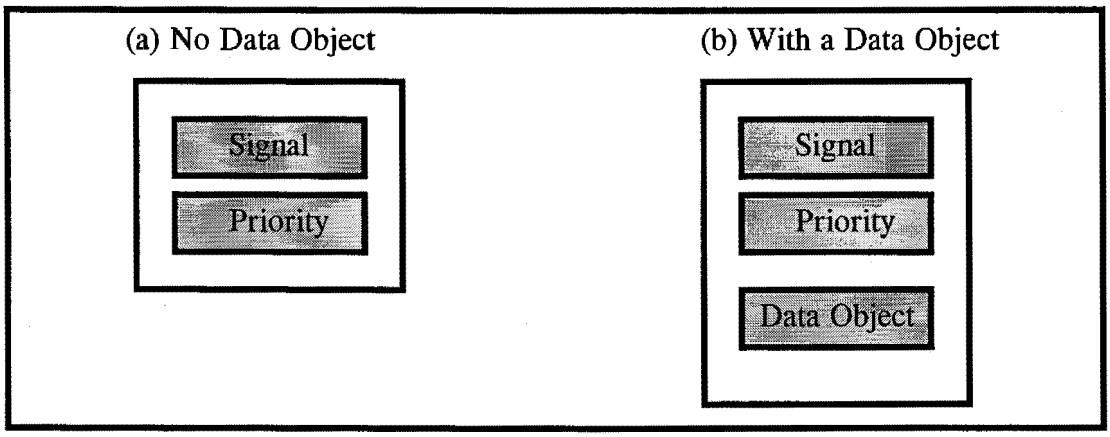


Figure 24. ROOM Messages

Protocols

Communications between software or hardware objects require a means of translating the incoming messages. This need arises because actors could belong to different types/layers of architectures, e.g. if a PLC was communicating with a workstation, they are at different layers, have different interfaces and functionality. However, they can understand one another if "they speak the same language", e.g. TCP/IP (Transmission Control Protocol/Interface Protocol).

This common language has a very structured format. It has a fixed number of bits (e.g. 16/32) called a frame. The frame is further split into groupings of bits where each group has a specific meaning, e.g. the first 4 bits are the header, the next 6 bits are the address, etc.

In a scenario as described above, viz. the PLC communicating with the workstation, the protocol pattern for transmission and receipt of messages is the same and is therefore called symmetric. ROOM has a combination of symmetric and asymmetric communications. The distinction between transmission and receipt is called outgoing and incoming messages. Incoming and outgoing are defined with respect to the actor to which the interface component is attached.

It may appear that each protocol has to be defined twice (once for each side.) However, this is avoided by introducing the concept of a conjugated protocol. A conjugated protocol for a protocol P has the same definition as protocol P* except that the incoming and outgoing messages sets are interchanged. The ROOM graphical notation uses a white square (as opposed to a black one) to indicate that a port reference is conjugated.

3.6.1.2. Communication Relationships

There are two types of communication relationships specified within ROOM; bindings and contracts.

Actors that are directly connected to one another can influence each other. ROOM refers to these explicit communication paths as bindings which is really relationships between objects. An informal simplistic illustration of where an actor cannot influence another is in Figure 25.

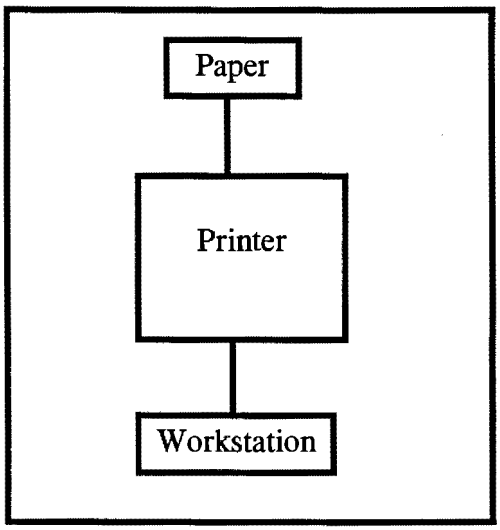


Figure 25. An Informal Illustration of Actor Relationships

The Workstation actor cannot influence the Paper directly, it can only do so indirectly, by influencing the Printer which can influence the Paper, e.g. if a report is being printed from the Workstation, it could issue a form feed request first, before the report can start printing. This instruction is interpreted by the Printer which will enable the form feed. Similarly the Paper cannot influence the workstation, however it can influence the Printer, e.g. if while printing, there is a paper jam, the Printer has to buffer the report in its memory, while the Workstation will be oblivious to the paper jam.

The graphical notation for a binding is a continuous line that interconnects two ports. An example of a formal ROOM diagram with a binding is shown in Figure 26. Here the binding connects two ports, a1 and b1, on actors ActorA and ActorB, respectively.

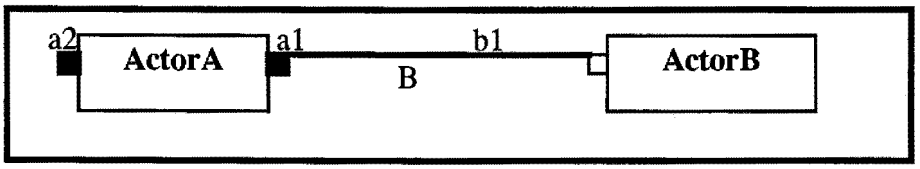


Figure 26. A Binding

An example of the second type, contracts is shown in Figure 27 where Client and DataBase are the interface components and ClientServer the binding. Formally, a contract consists of a binding and the two interface components that it connects. It is said that an actor satisfies a particular contract if it has an interface component that is part of the contract.

Note that ActorA has no responsibility regarding a2 in Figure 26 as that port is not contracted.

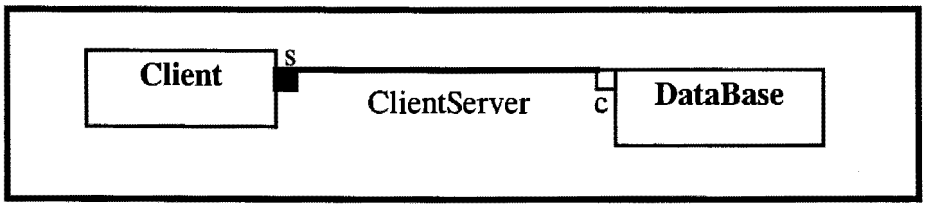


Figure 27. A Simple Entity Relationship Represented as Two Actors and a Binding

3.6.1.3. Actor Structures

Actors can be composed into one of the following structures:

- Generic
- Composite
- Functional
- Multiple Containment

Each of these structures are discussed next.

Generic

An actor is constituted of both the externally and internally visible components. The internal is made "invisible" by the use of encapsulation. The external components which are really the interface components, define an observable behaviour. A generic actor structure can be defined as any actor class that satisfies the contracted type of an actor reference could be put in its place without affecting the overall functionality of the aggregate.

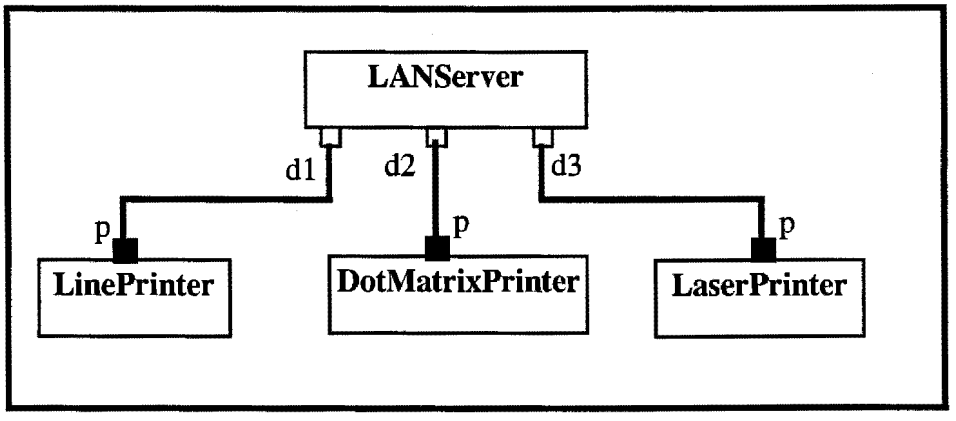


Figure 28. A Generic Actor Structure

The contracted type of a component actor in an application context is defined by the subset of all its external interface components involved in contracts. For example,

consider Figure 28, in this case, the responsibilities (obligations) of any component (actor reference) in the structure is determined by its contracts. The LANServer in this example has a responsibility for each of the different types of printers.

To control the use of genericity, in ROOM it is necessary to explicitly designate an actor reference in a structure as being substitutable. This substitution takes place when the actor is instantiated. The graphical notation for substitutability is a "+" icon placed in the upper left hand corner of the actor reference. See Figure 29.

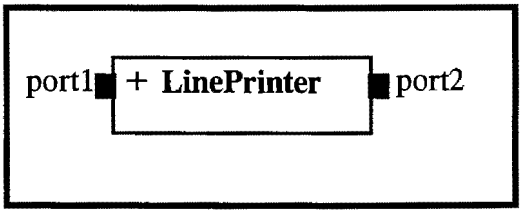


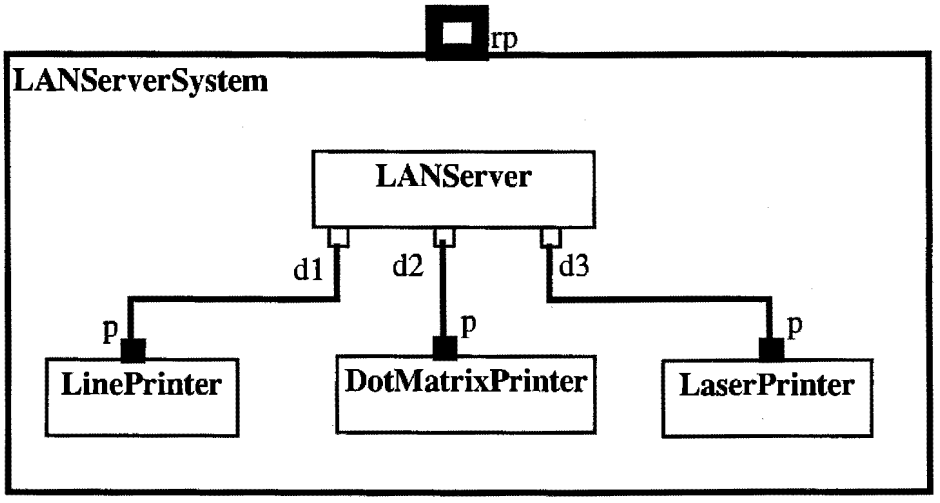
Figure 29. Graphical Notation for a Substitutable LinePrinter

In the case of a substitutable reference, the class of that reference specifies the default class to be instantiated in that position unless explicitly overridden.

Composite

The concept of composite actors uses the OO principle of abstraction. A composite actor can be nested within other composite actors in multiple nested levels. Its notation is similar to that of the "regular" actor. For example, Figure 30 (a) represents the composite view of the actor of Figure 28, seen as a class view and Figure 30 (b) depicts how the composite actor is viewed from the outside (just like a regular actor).

(a) Class View



(b) Reference View

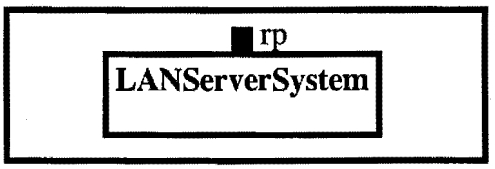


Figure 30. A Composite Actor

The different types of printers, line, dotmatrix and laser are referred to as peer actor as they are at the same level of abstraction. The port, rp is a relay port, which is the port through which communications with the other ports occur. It is the boundary between the outside world and the composite actor. It has two sides, the outward that considers the environment and the inward side that considers the composite actor, (see Figure 31) [Selic1994]. The protocol P that is associated with a relay port is always defined with respect to the outward side. That is, incoming messages for the relay port is always defined to be messages that flow into the actor. Note, that on the inward side, the polarity of the protocol is reversed so that the inward side uses the conjugated protocol.

Functional

In process control systems there are a number of concurrent processes with tight integration between them in order that they achieve their functionality. ROOM facilitates this integration through its relay ports, as illustrated in Figure 32.

Different systems, e.g. a WordProcessor package and a SpreadSheet Package can utilise the same LANServer System.

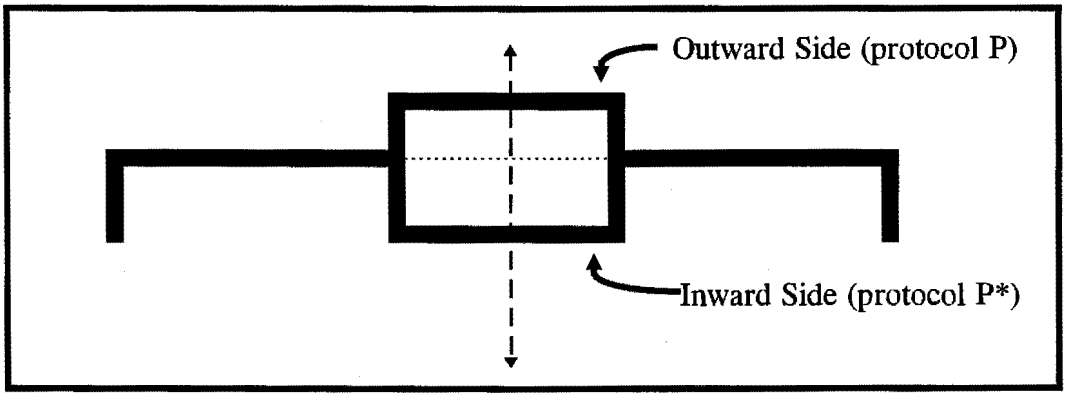


Figure 31. The Two Sides of a Relay Port

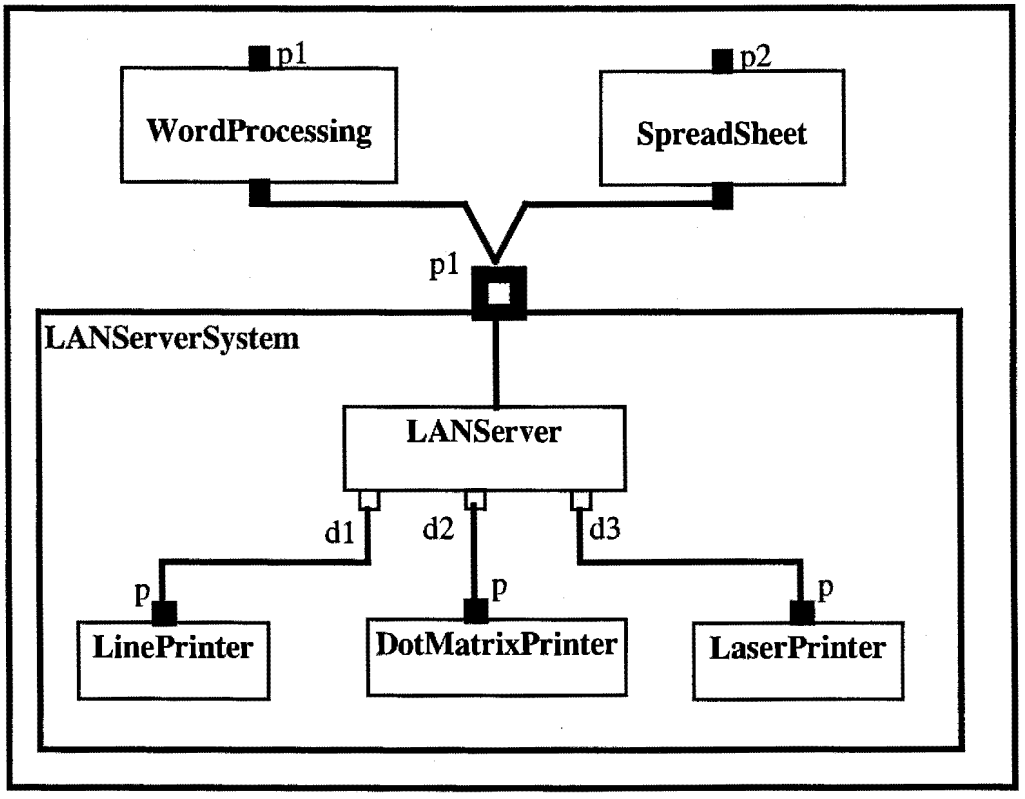


Figure 32. Combining the Functionality of Different Actors

Multiple Containment

In order to support overlapping abstractions it is necessary to allow actors to belong simultaneously to more than one composite actor. This feature is called multiple containment.

As the number of elements becomes large, the graphics get unwieldy. For this purpose, ROOM provides the concepts of replicated actors and replicated ports. These are used for models in which multiple references to an actor or protocol class are included repeatedly based on some regular pattern. The regularity of form is exploited to provide a practical and compact way of representing large populations graphically. Replication is, in fact, a form of abstraction.

3.6.1.4. The Relationship between Structure and Behaviour

Within the process control domain, actors are continuously created and destroyed due to the dynamic nature of the environment. ROOM has introduced a component called the behaviour component which captures this nature. By definition only leaf actors include behaviour.

The behaviour of an actor has the following distinguishing properties:

- It is created and destroyed automatically with its containing actor.
- It is "pure" behaviour and has no internal high-level structure.
- It can share interface components with its containing actor.
- It can directly reference all its peer actors in its decomposition frame.
- It is not rendered explicitly in the graphical ROOM notation for actor structures.

Consider the shaded area in Figure 33, in this informal diagram, the behaviour is viewed as another component in the structural decomposition. It can be seen from this illustration that both the Line Printer and the Laser Printer share a common behaviour component called idle.

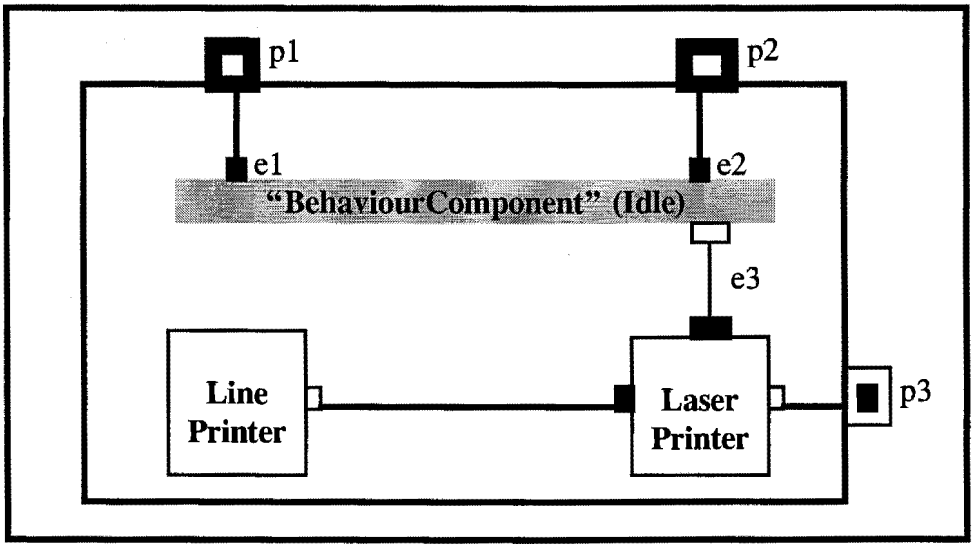


Figure 33. An Informal Representation of the Behaviour of Printers

The ROOM graphical notation uses several shortcuts in order to minimise the visual clutter of structure diagrams. These are illustrated in Figure 34.

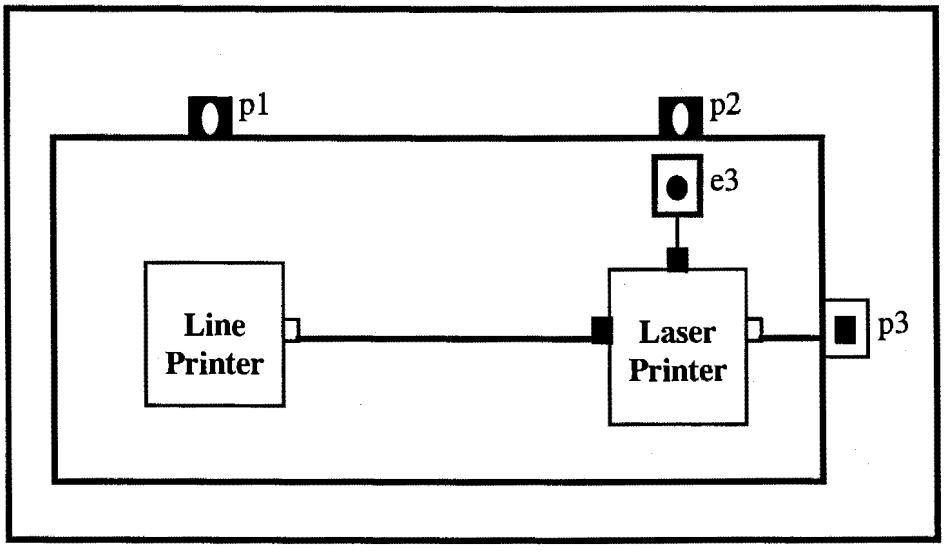


Figure 34. The Actual Representation of the Behaviour of Printers

The new notation introduced in Figure 34 is called an end port and is the link between the structure and the behaviour since they are visible in both modeling dimensions.

As mentioned previously the process control domain is extremely dynamic and the structure that has thus far been defined are static structures. In order to depict the dynamic structures ROOM has introduced "optional actors" and "imported actors". Optional actors do not have any fixed bindings so they can be easily destroyed without leaving behind any dangling links. These dynamic links are time related, and are said to have temporal properties.

The graphical notation for an optional component actor is shown in Figure 35 (actor reference OptActor).

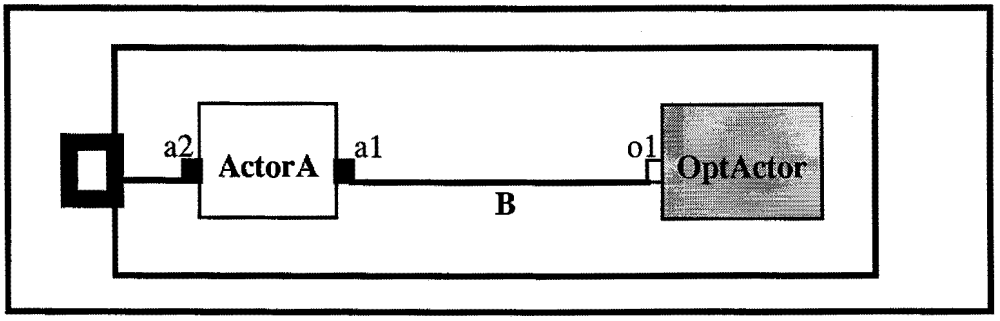


Figure 35. A Composite Actor with an Optional Actor Reference

Creation and destruction of optional actors is driven by the behaviour component of the containing actor, since it is the only component that can reference other components in its framework. For this reason, it is not possible to destroy the behaviour component.

When an optional actor is created, any bindings that are structurally linked to it (for example, binding B) are activated, so the underlying communications channels are ready to accept messages as soon as the actors are created.

Another form of dynamic structure occurs when an existing actor must enter into a dynamic relationship with another actor. For example, consider a prototypical client-server system in which a single server is shared among a number of clients. Assume that the server can only serve one client at a time, and that the need for service occurs unexpectedly. This relationship can be modeled by a "service usage" actor that incorporates the server and the current client, as shown in Figure 36. The problem, however, is that one cannot predict in advance which particular client will be involved in the relationship at a given time.

ROOM has introduced the imported actor concept which is a typed "placeholder" into which it is possible to insert an actor that already exists in some other framework.

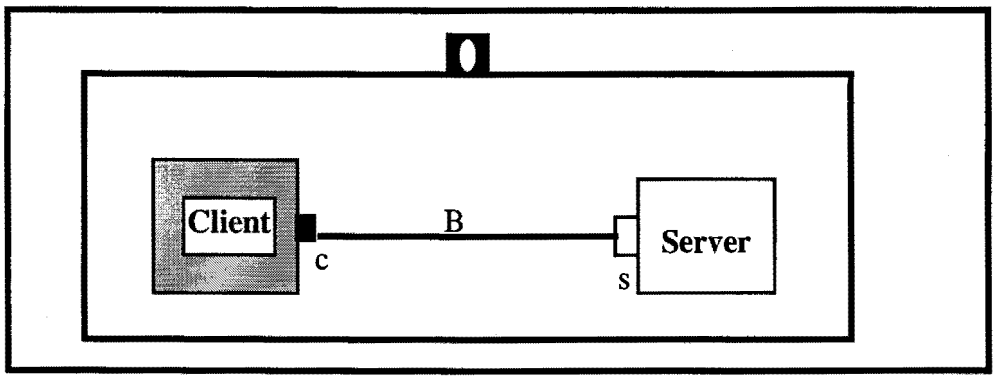


Figure 36. Imported Actor Reference

3.6.2. High Level Behaviour Modeling

The portion of the conceptual space identified by the behaviour dimension of the Schematic Level, see Figure 37, is discussed next [Selic1994]. This domain is characterised by the presence of two complex and difficult phenomena, concurrency and distribution

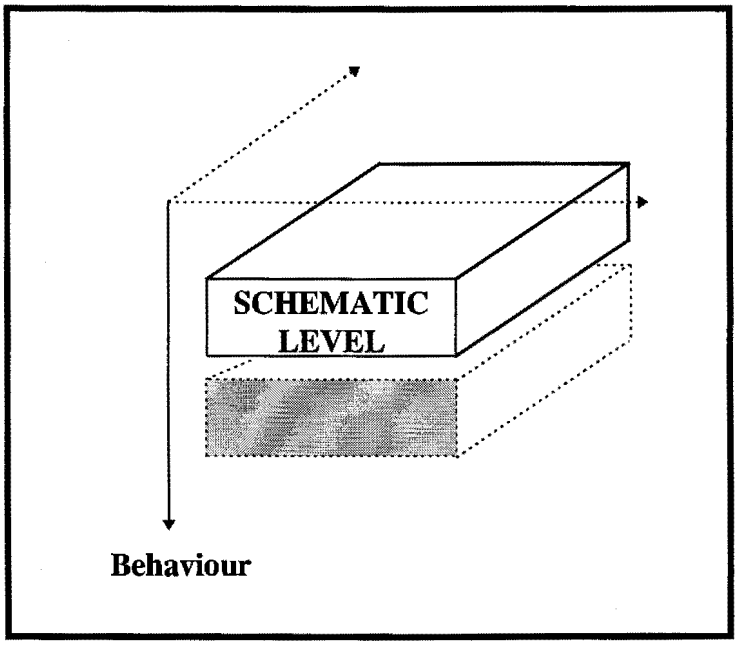


Figure 37. The ROOM Conceptual Framework: High Level Behaviour

3.6.2.1. Events

There is no clear cut distinction between messages and events and they are often used interchangeably. Formally an event is related to time, i.e. it has temporal properties and it is generated from outside the ROOM environment. The ROOM virtual machine, on receipt of an event translates it into messages and it is interfaced into the ROOM environment via the SAPs.

There are two general approaches to the handing of events:

1. Run-to-Completion
2. Pre-Emptive.

Consider Figure 38, and that event e1 is currently being processed. If event e2, of higher priority than e1, arrives at the same time. In the run-to-completion approach e2 has to wait for e1 to complete it execution. This approach is obviously simple to

handle from a development point of view. However, e1 could take long to complete and e2 although being of higher priority cannot influence the environment and effect the necessary changes that it needs to, i.e. the reason for its high priority.

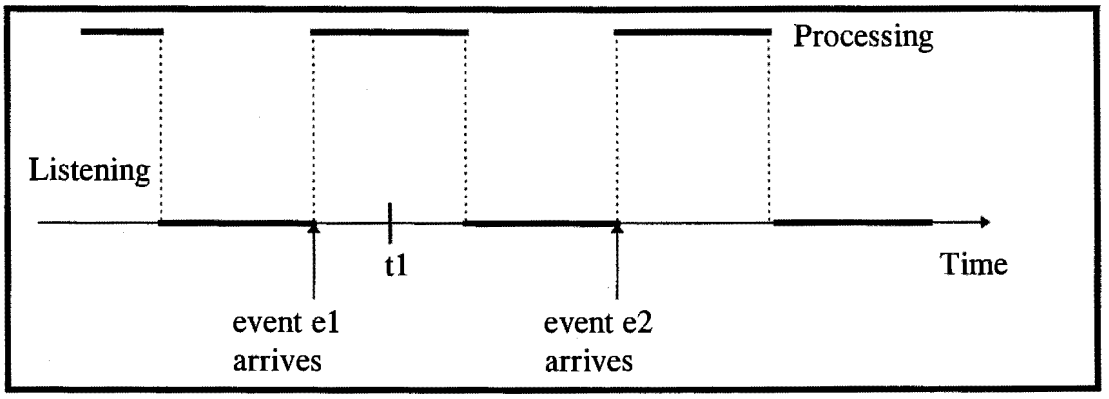


Figure 38. Event Driven Model

With pre-emption, e1 immediately saves the statuses of its current variables and procedures. It then relinquishes control of the processor and e2 can execute. However, e1 could have been in the midst of updating a database of variables and e2 during its execution updates the same variables. As can be seen, this leads to incorrect and inconsistent data.

ROOM has opted for the run-to-completion approach. The two approaches are depicted in Figure 39.

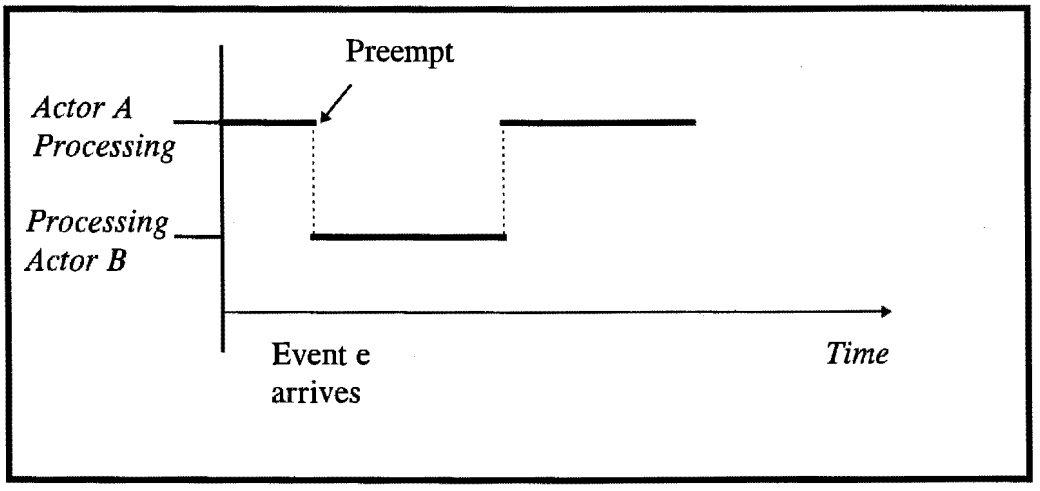


Figure 39. Pre-emption and Run-to-Completion

In order to achieve a short execution time of e1, protracted event processing sequences can be broken up into a number of shorter chunks. However, this can significantly complicate the implementation.

In terms of priorities, the ideal would be to have one single priority. However with the distributed process control environment this is not possible. It is thus recommended to have some level of urgency to be provided. This would cater not only for the high priority tasks but also the low priority tasks. Frequently, the low priority tasks are disregarded during busy periods.

3.6.2.2. State Machines

The technique used in ROOM to specify behaviour components was inspired by the state charts formalism Harel[1987]. In recognition of this influence the formalism is called ROOMCharts.

ROOMCharts is introduced using the simple example of "name server" actor whose structural form is shown in Figure 40. The purpose of this actor is to handle requests from clients to map a symbolic name into a communication address. The actor can handle up to two clients through its client service ports (clientA and clientB). It also interfaces to a controller through port master. The controller is responsible for supplying the server with the initial version of the name mapping table and for subsequent table update requests.

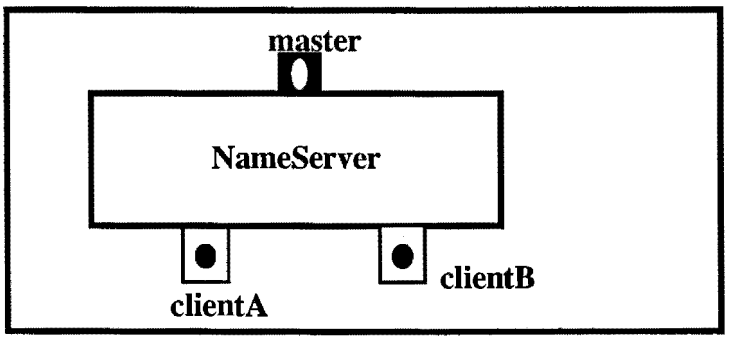


Figure 40. The Structure of the Name Server Actor Class

The state machine diagram for the behaviour of this system is depicted in Figure 41 (in this diagram, transition labels match the names of the signals that trigger them.) During its lifetime, the name server progresses through two basic phases. On creation, it enters the Uninitialised state, during which it waits to receive the initial version of the name table. Once it has received that data, the server stores it in an extended state variable (nameTable) and enters the Operational state in which it processes service requests in the order in which they occur.

For simplicity it is assumed that the nature of the requests is such that each one can be handled by a single action.

Obviously, as with all state machines, a state change can only occur if there is an outgoing transition from the current state. A transition takes the behaviour from one state to another. Each transition may have an action associated with it. An action consists of a set of action steps that are limited to operating over any of the following types of objects:

- Extended state variables

- Temporary variables that are used to store intermediate results during the execution of an action

- Behaviour interface objects (end ports, SAPs) that are accessed to communicate with other actors.

The message that caused the current event is automatically stored in a predefined extended state variable so that it can be accessed if necessary, [Selic1994].

A single transition can have more than one simple trigger. Run-to-completion event processing offers only one message at a time to the behaviour so that it is not possible to construct trigger conditions based on conjunctions of two or more simultaneous events.

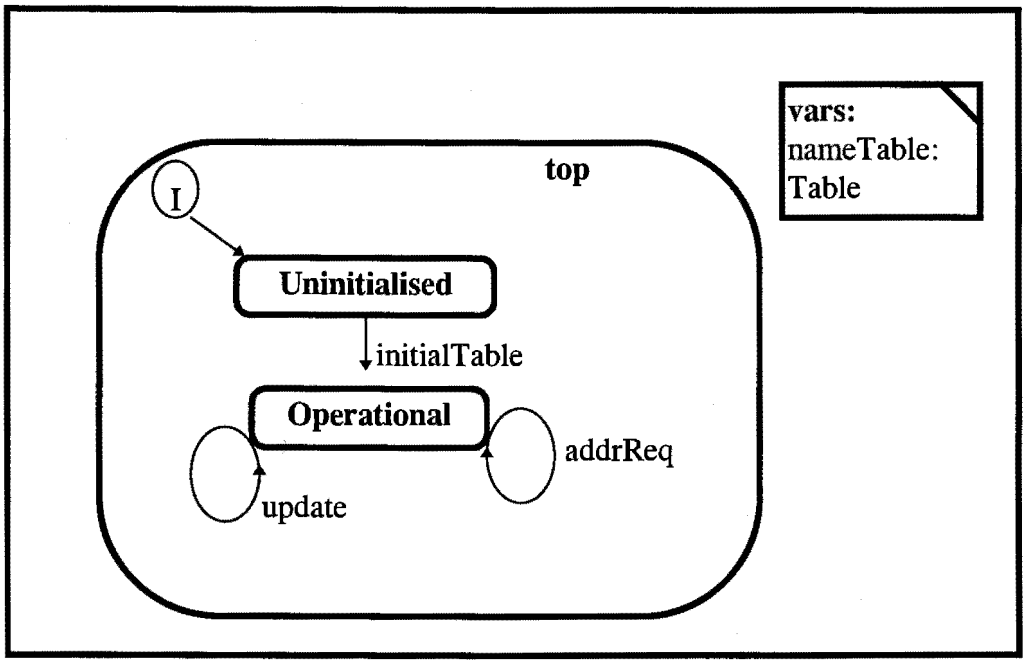


Figure 41. The Behaviour of a Name Server Actor

A composite state is defined as a state in some state machine that contains a lower level state machine or sub-machine. In Figure 42 there is an example of a state machine with two composite states S1 and S2 [Selic1994]. The current state of such a system can be described by a nested chain of states called a state context. For example, if the system is in state S12, then its context is defined by the chain S1-S2-top. The behaviour is simultaneously "in" all of these states depending on the level of abstraction under consideration.

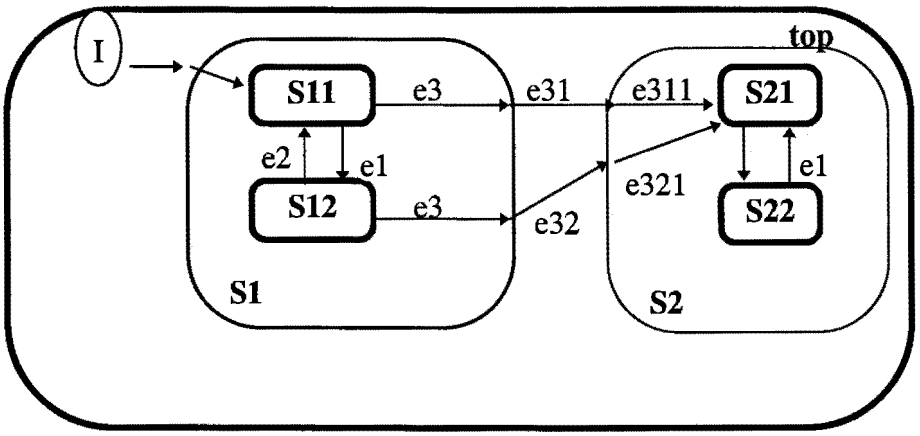


Figure 42. A State Machine with Composite States

An alternative representation is one in which each state machine is drawn separately. In that case, the state machine in Figure 42 would be modeled by three related diagrams: one for the entire state machine (that is, the top state), one for state S1 and one for state S2. The state transition diagram for the top-level state machine is shown in Figure 43.

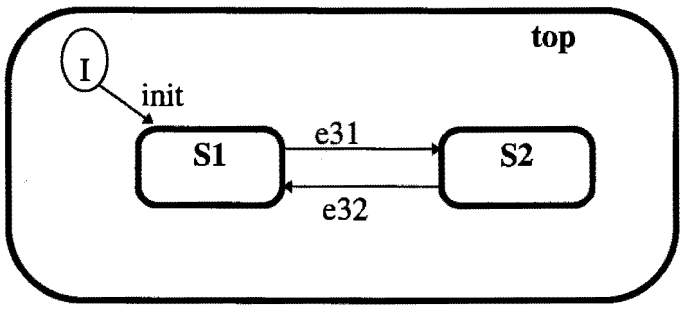


Figure 43. The Abstract View of the Top Level State Machine from the Figure above

Transition points provide a formal basis for correlating the different segments of a transition that spans multiple contexts. Transition points are located on the boundary of a state and represent either the source or the destination of a transition segment. For convenience, a transition point is labelled with the name of the external transition. The icon for an incoming transition point is a circle enclosing a diagonal cross, while the icon for an outgoing transition point is an a circle enclosing a diamond shaped polygon. Figure 44 depicts the individual state transition diagrams for the two composite states of the example being discussed and also shows the placement and usage of the transition point icons.

Group transitions mean that that transitions apply equally to all the sub-states. In the ROOMChart notation, such common transitions are indicated by a transition that originates from the composite state. This is illustrated in Figure 45. A group transition acts like a high level interrupt. The behaviour for the nested state machine

is suspended as there is a move to a new state. For this reason, group transitions often model high-priority events that cannot be left untended for too long.

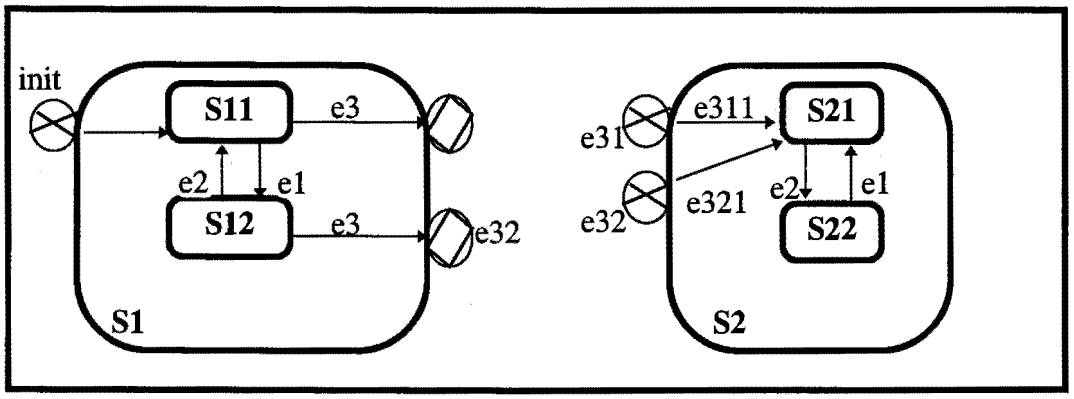


Figure 44. Transition Point Icons

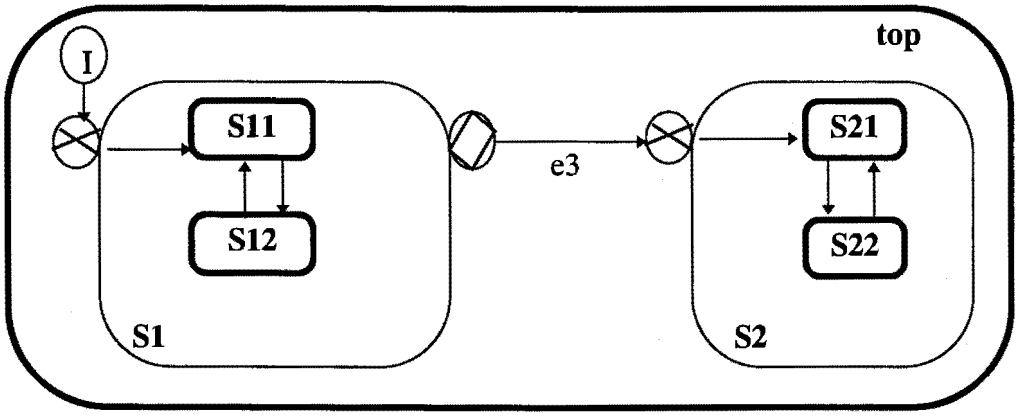


Figure 45. A Group Transition

The search order for event triggering rules are:

1. The search starts with the scope defined by the innermost current state.

2. Within a given scope, triggers are evaluated sequentially. If a trigger is satisfied, the search terminates and the corresponding transition is taken. The search order is arbitrary but deterministic. If two or more triggers in the same scope are satisfied by an event, it cannot be predicted during design which one will be

evaluated first. However during execution, the search order will not change over time. This means that, under equal circumstances (that is, the same state), repeat occurrences of the same event always will trigger the same transition.

3. If no trigger in the current scope is satisfied, the search is repeated for the next higher scope (state) and the behaviour remains unchanged.

4. If no triggers are satisfied even at the topmost scope, then the event is discarded and the state of the behaviour remains unchanged.

Often there are decisions that need to taken, but there various options that could be followed depending on the evaluated statement. A typical example of this is :

```
If (expression) then
        S1
else
        S2
endif
```

The ROOMChart facility through which this is achieved is called a choicepoint and is shown in Figure 46.

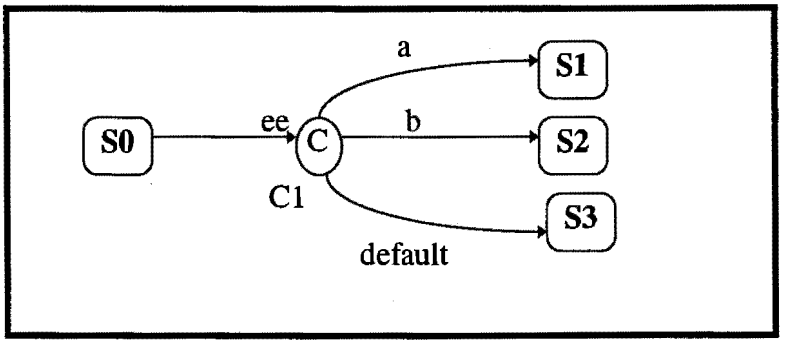


Figure 46. A Choicepoint

3.6.3. High Level Inheritance

In this section, the third dimension, inheritance, in ROOM is examined. See Figure 47.

ROOM views inheritance primarily as an abstraction mechanism that helps to deal with complexity by allowing detail to be introduced gradually.

With the abstraction based approach, abstract classes play a pivotal role. The definition and preservation of the integrity of abstract classes is the primary concern in dealing with inheritance. Each abstract class should symbolise a well-defined abstraction with a clear meaning.

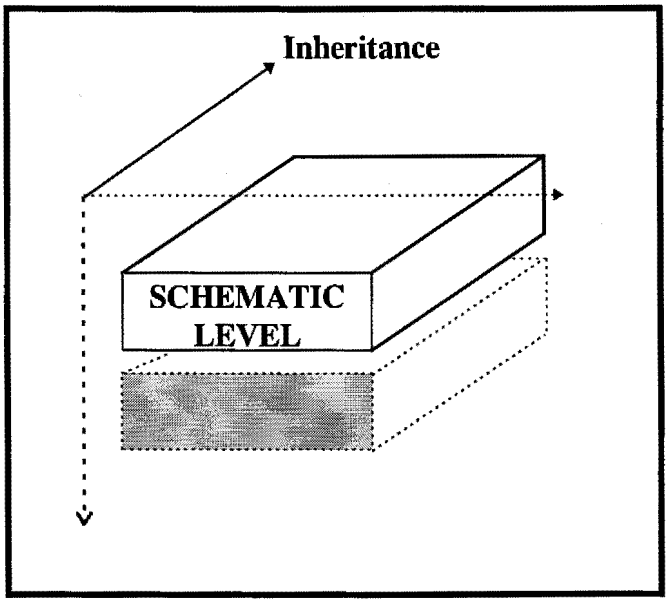


Figure 47. The Conceptual Framework: High Level Inheritance

In ROOM, inheritance plays a fundamental role, since all designs are specified as classes. Three different class hierarchies are supported: the actor hierarchy, the protocol hierarchy and the data object hierarchy. The inheritance rules of data objects are determined by the Detail Level language used in modeling. Actor classes allow both high level structure and high level behaviour to be subclassed, providing for a much higher form of reuse than is available in traditional OO programming

languages. Only single inheritance is supported for actor classes and exclusion and overriding of attributes is allowed [Selic1994].

Abstract structures contain abstract versions of interface components and components that can be refined by subclasses. The designer should strive to make all abstract classes meaningful by ensuring that they are complete and executable. An executable abstract class is called a simulation class, since it can serve as a lightweight substitute for a concrete class while evaluating larger designs, Subclassing involves the substitution of abstract attributes by more refined ones.

3.6.4. The Detail Level

This section examines the issues and techniques used to specify the Detail Level aspects of a ROOM model and how these fit in with the concepts introduced in the previous sections. See Figure 48.

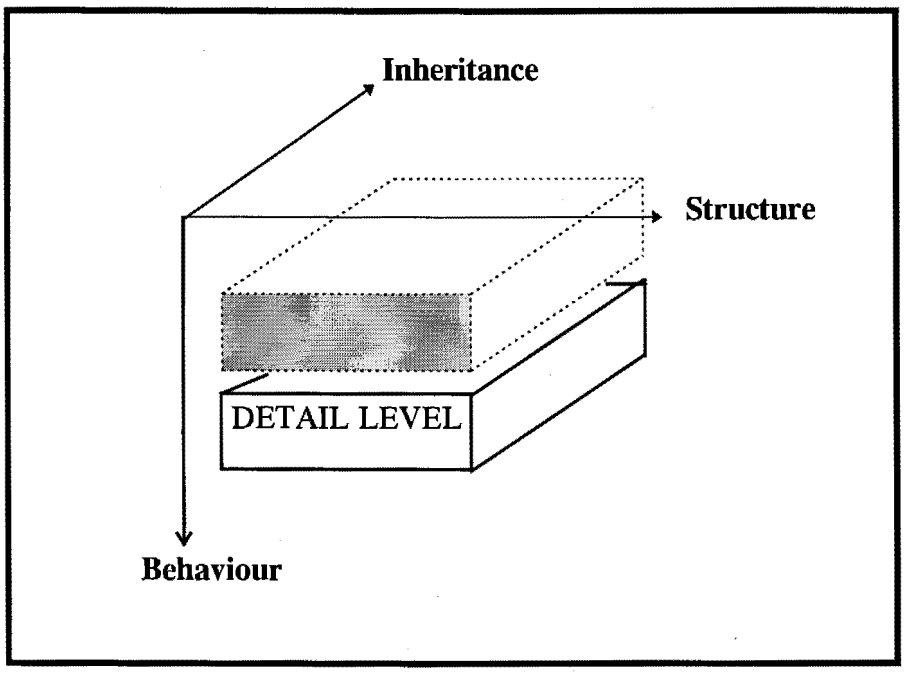


Figure 48. The ROOM Conceptual Framework: The Detail Level

The detail level deals with fine grained actions and fine-grained objects. All elements of concurrency and high level architectural issues are filtered out and dealt with at higher abstraction levels. As a result the complexity of the specifications at this level is greatly reduced. Detail level actions capture the behaviour that occurs during transitions of a state machine from one state to another. Fine-grained objects are used either to capture the extended state of a state machine or as information units that can be transferred between actors [Selic1994].

The Detail Level provides a few services:

- Exception Service
- Timing Service
- Frame Service
- Communication Service

These services are explained in the next section, However briefly the:

Exception Service

This service is specific to low level exceptions. There are two levels of handling; either through the language or through ROOM. The language specific handling is applied first, if it fails then the ROOM handling is initiated. If neither succeed then the behaviour component is placed in a special exception mode from which it can be recovered by a containing actor.

Timing Service

As there is concurrency between the actors in a real-time system, it is obvious that a number of actors would require real-time facilities. This service provides the shared Timing service and its basic usage is a service request that is submitted through a SAP on the Timing Service. The request may specify a time interval or an absolute time of day. This results in the creation of a dedicated timer within the service. Each

request has a different timer so that multiple parallel requests can be made. When the service detects that the appropriate moment has arrived, it sends a special time-out message to the SAP through which the request was made. This message can then trigger a transition just like any other message. The Time-out message is queued and scheduled like any other event and depending on its priority and the current processing load, there may be additional delay before it is actually received.

Frame Service

This service is responsible for the dynamics of the ROOM methodology, i.e. the creation and destruction of the dynamic relationships as well as the dynamic actors.

Communication Service

This service is connected to all the interface components of an actor, i.e. (endports, SAPs, SPPS) Both synchronous and asynchronous communication modes are supported, but the receiver of the message is generally unaware of which mode was used.

3.7. Process Control Reference Model

According to Brown [1992], a reference model is a conceptual and functional framework which helps experts to describe and compare systems. It allows experts to work productively and independently on the development of standards for each part of the reference model. A reference model is thus not a standard itself; it should not be used as an implementation specification, nor as the basis for the conformance of actual implementations. The reference model for the ROOM virtual machine and its relation to the process control environment is discussed in this section.

This section introduces the reference model for the ROOM virtual machine and an OO conceptual model for process control software. The virtual machine is introduced in this section to provide an understanding of its functionality before it is used in the next section. The OO model that is presented is merely a conceptual model of the process control software and not of the process control domain.

The next section describes the reference model for the process control domain. It is thereafter presented in terms of ROOM's notation.

3.7.1. Reference Model for the ROOM Virtual Machine

The ROOM virtual machine is a hypothetical device capable of directly executing ROOM model specifications. This means that it must directly support the high level ROOM concepts such as actors, contracts or ROOMcharts. While it is certainly conceivable to construct such a device in hardware, it is usually more practical to implement it in software so that it can be easily ported to a variety of processing platforms and environments. This also allows one to take advantage of the latest technological advances in hardware.

The Target System Reference Model for the Process Control application domain is presented in this section in terms of the ROOM virtual machine. In this section this virtual machine is described and the mapping of ROOM specifications into the traditional real-time environment is considered. An OO conceptual model for process control software is also described.

The ROOM virtual machine is situated as a single layer between the target environment and the executing application, as shown in Figure 12. The top level structure of the virtual machine is shown in Figure 49 [Selic1994]. It has just two components.

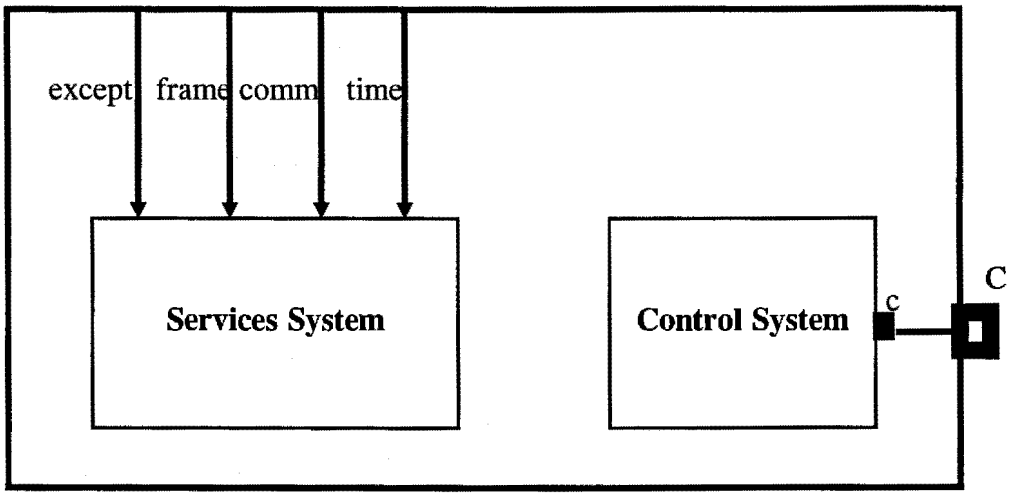


Figure 49. The Top Level Structure of the ROOM Virtual Machine

For clarity, there is a legend on the ROOM methodology in Appendix A.

- The Services System implements the functionality of at least the elementary services. These services are exported by way of a layer connection, to the application. The application has Service Access Points (SAPs) that may be attached to the corresponding Service Provision Points (SPPs) of the virtual machine.

- The second major component is the Control System which takes care of controlling and co-ordinating the operation of the services and indirectly of the application itself. To allow external control of the entire system, the Control System has a port (port c) through which it can communicate with an external higher level control system. The precise linkage of this port to an external control system is specific to each target environment.

3.7.1.1. The Services System

The Services System of a minimal virtual machine consists of four essential services, which are mutually dependent. The Services System is illustrated in Figure

50. It can be seen that each of the services has a port used for control purposes, i.e. port c. This port is bound to the Control System.

- The Frame Service is responsible for realising the structural aspects of a ROOM specification. This includes the creation and destruction of actors, importing and exporting and the imposition and removal of contracts (bindings and layer connections).

- The Communications Service is responsible for the dynamic construction and destruction of communication channels, as well as the actual transport and delivery of messages.

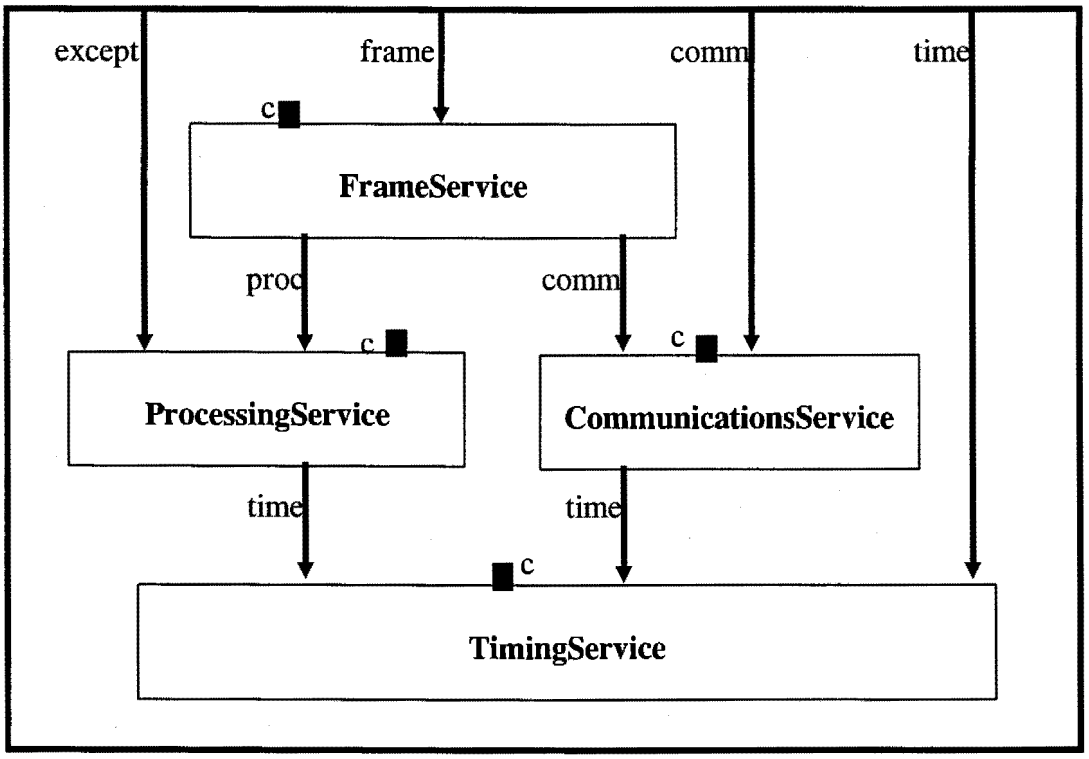


Figure 50. The Services System

- The Processing Service handles the scheduling, dispatching and execution of behaviours including the detection and handling of run-time exceptions.

- The Timing Service provides absolute and interval timing.

3.7.1.2. The Control System

The Control System is responsible for co-ordinating the activities of the individual services and also ensuring that those are synchronised with the activities of an external control system. The internal structure of this system is shown in Figure 51 [Selic1994]. The service components shown in this diagram are the same ones shown in the Services System diagram (multiply contained actors).

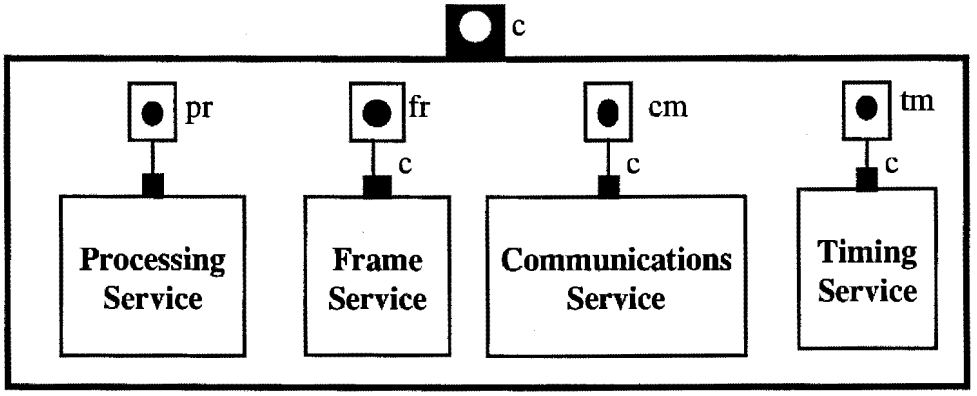


Figure 51. The Control System

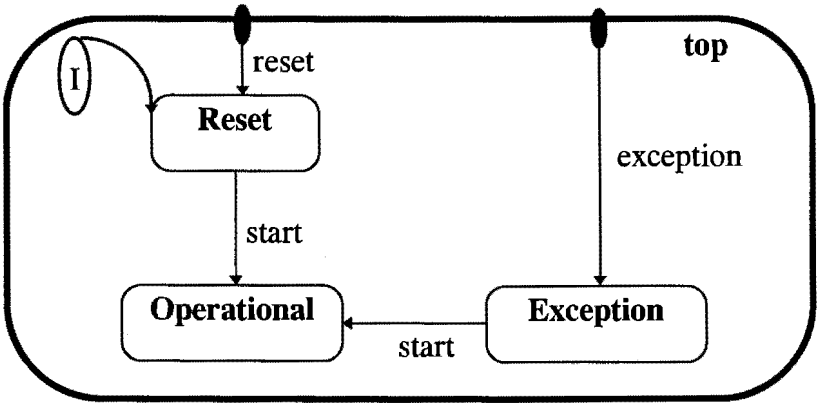


Figure 52. The High Level Behaviour of the Common Controlled Component

There are four services within the Control System. To have different interfaces to each of the four would increase the complexity of the interface and would be a managerial nightmare. ROOM has approached this by having a common control protocol for any of the four services. This is captured defining an abstract actor class that embodies such a component. The simplest version of this class has just

one port through which it receives control signals. A basic variant of the high level behaviour of such an abstract component is shown in Figure 52.

This figure depicts a typical computer set-up, where by using the reset button, regardless of where you are in the system, the machine is rebooted. The two group activities, Reset and Exception can occur while the component is in any "state". Once in a Reset state, a start request will trigger the component to move into the Operational state. Similarly, when an Exception has occurred, a start signal from the Controller must be received in order to move back into the Operational state.

Clearly all services are candidates to be subclasses of the abstract controlled component class. Furthermore, since the virtual machine is a controlled element, it makes sense to make the Control System a subclass of the same abstract class. The Control System then acts to relay commands to its minions. For example, when told to reset itself, the Control System will not only reset itself, but will also reset all the services.

3.7.1.3. The Timing Service

The operation of the Timing Service is very simple. It has a SAP through which it accepts services requests. Depending on the request, it creates or destroys individual timers. A timer is typically destroyed when its timer has expired or when it is cancelled. The relatively simple internal structure of the Timing Service is shown in Figure 53.

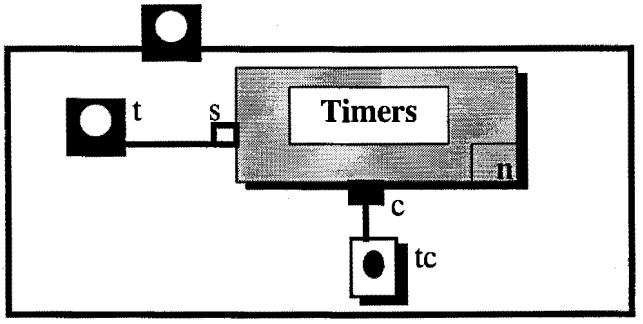


Figure 53. The Structure of the Timing Service

3.7.1.4. The Processing Service

The responsibility of this service is the management of the usage of the processor. It maintains one queue per priority and schedules the access of the processor. If the current execution is incomplete, and there are tasks waiting in the queue, the processing service suspends the current one by issuing a yield message.

3.7.1.5. The Frame Service

The Frame Service is perhaps, the most complex of the system services. It is responsible for creating and destroying actor instantiations and maintaining the dynamic structural relationships between them.

Each application actor is represented by a corresponding meta-actor within the Frame Service. The life span of the meta actor is dependent on the application actor's life span. The Frame service maintains a dynamic data structure to manage the dynamic containment relationships. The structural model of the Frame Service is shown in Figure 54.

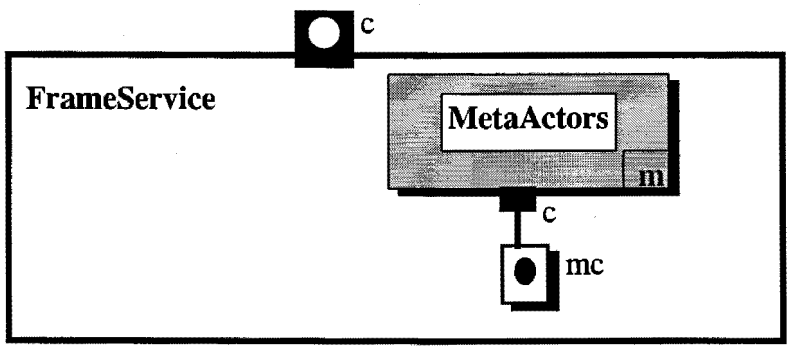


Figure 54. The Structure of the Frame Service

3.7.1.6. The Communications Service

This service provides two types of services.:

- It is responsible for establishing, maintaining and removing connections between all actor interface components.

- The second type of service is the transporting of messages between interface components (the comm SPP). All application-actor interface components are attached to this SPP, so that when they ask for a message to be delivered, the Communications Service relays it through the established connections and delivers it to the final destination. If the destination is a behaviour interface, then the corresponding meta-actor is notified.

3.7.2. Mapping ROOM specifications into the Traditional Real-Time Environment

The techniques for mapping a ROOM model into an implementation targeted at a standard real-time kernel and a traditional block structured imperative programming language can be summarised as follows:

- In mapping hierarchical state machines, an exact mapping of the model to an implementation must be ensured. This means that the full expressive power of ROOM state machines can be used in the model, regardless of the underlying target environment.

- In mapping the structural aspects of a model, the approach is more constrained. It is recommended that certain restrictions on the modeling concepts and techniques be used Selic[1994]. This will enable an easier transformation from the resulting specification to implementation. These restrictions depend on the capabilities of the target implementation environment but are in most cases, relatively minor and do not detract significantly from the modeling power of ROOM.

3.7.3. An OO Conceptual Model for Process Control Software

The Process Control environment is precisely the type of complex, highly concurrent system for which the ROOM methodology is intended.

This section discusses the conceptual model for the process control environment and mentions its mapping to the ROOM virtual machine.

3.7.3.1. General Aspects

Process Control Software usually is related to one of the three levels shown in Figure 55 [Pirklbauer1994].

The *hardware control level* covers all processes for controlling machines on a low level. These processes, which are responsible for one or several machines, receive commands from the level above and send special commands to their machines. In most cases, these processes are implemented on special hardware (typically a PLC) that usually is delivered with the controlling software.

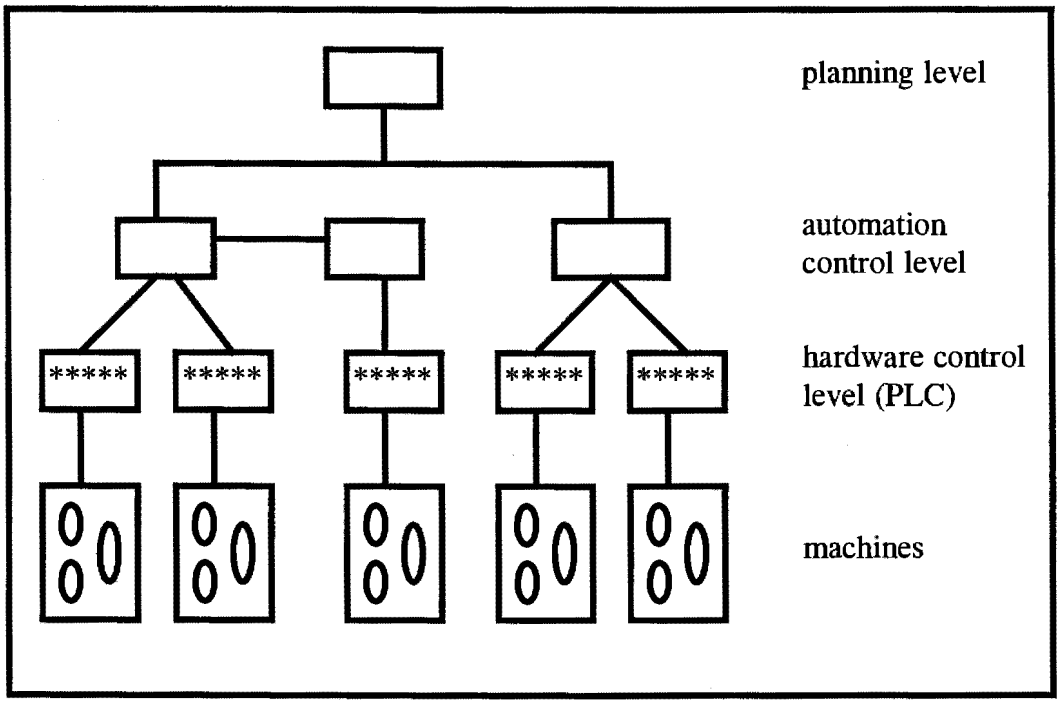


Figure 55. Levels within the Process Control Domain

The *automation control level* covers all processes for controlling production. This level is usually structured according to the tasks rather than the machine structure. The automation control level processes communicate with the machines and with the user at the control station.

The *planning level* is at the top level of an automation system. It is the connection of all automation control level components working on the same product or using common data. Processes on the planning level control power consumption, make machine-capacity calculations and plan the production.

The discussion below concentrates on the first two levels.

Some special requirements that a process control has of an OO system:

* Several tasks working in parallel must be controlled by a process control system that is often distributed throughout the entire plant. This requirement leads to a *distributed system* as the base architecture for process control software.

* Processes on the automation control level must exchange data with each other and with hardware and planning level processes. Thus in an OO implementation of process control software, the basic class library has to provide *communication* services.

* Automation systems require an error-tolerant implementation. Production should not be stopped if an error occurs. Processes that died because of an error must be restarted and broken communication lines must be reconnected, for this reason, *supervisor processes* must observe the automation process and act in case of errors.

* Visualisation of the current state of production and of machine occupation is required at the control panel.

3.7.3.2. Special Requirements for Process Control Automation

In the process control environment, e.g., a water treatment SCADA environment, quality is a very important aspect. Historical information, typically, flow of water through a pipeline is required for:

* demand analysis / predictive analysis to be performed by the Corporate Planning Staff,

* daily water balancing by the Plant Superintendent,

* determination of a leak in a pipe, by the Maintenance Staff, etc.

Thus, large amounts of data must be shared between the automation processes. This necessitates a mechanism for *data management.*

It may also, sometimes be necessary for manual troubleshooting. This is accomplished by sending manually entered commands from the control panel to the hardware control level processes and by entering missing data in the case of communication breakdowns or other problems. The intensive user interaction is also necessary because the production sequence and the processing steps cannot be foreseen exactly.

3.7.3.3. Process Control Environment

Based on the requirements above, an OO environment consists of the following class libraries as shown in Figure 56:

* Communication library - provides easily configurable and flexible communication among processes.

* Data management library - provides mechanisms for the management of a large amount of process data.

* Process management library - provides mechanisms for process management which includes synchronising and warm boot-up of crashed processes.

* Process control library - provides process control classes for controlling the treatment processing steps.

* Control panel library - provides visualisation of process data in conjunction with the design of a modern GUI for a control panel.

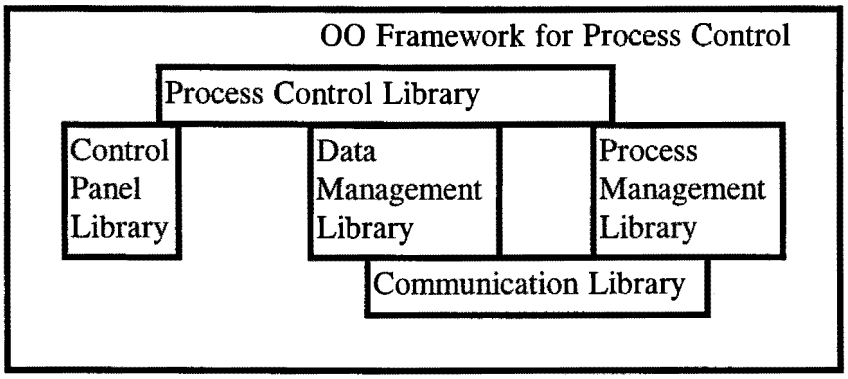


Figure 56. OO Conceptual Model for Process Control

The communications library, process management library and process control library have counter parts in ROOM's Services System and Control System. There is a direct mapping of the process management library and communication library to the ROOM control system. The process control library is a component of both ROOMs, services and control, systems while the data management library and control panel are part of the structural dimension of ROOM.

In the following sections, the components of the process control OO conceptual model are discussed in more detail. With each component, the class library and synopsis is described.

3.7.3.4. Communication Library

Class Library

A distributed system requires flexible communications between its components as described by Pirklbauer[1994]. On the one hand, fast communication lines are needed as well as slow but reliable communication lines. On the other hand, connections on one machine and connections crossing machine boundaries are needed. A flexible communications class library should allow the use of all these services, with the different interfaces of these services hidden from the user. It is also desirable to be able to change communications services without changing the source code of the programs using them. As an additional requirement, there should be the capability to send objects as well as conventional data structures (i.e., integers, strings, etc.) to allow OO design of the whole system.

In distributed applications, the flow of messages and data is of vital importance to understanding how the system works. A tool to monitor the communications lines provides a comfortable extension to the communication library.

Synopsis

The benefits of using OOP in this area are easier configuration and a gain in extensibility and flexibility. To change the communication service by which a particular process can be addressed, only an entry in a configuration file must be modified. There is no need for recompilation or relinking. Even unforeseen communication services can be added in this way, although with relinking. This is achieved by using abstraction (a common superclass for all services), see Figure 57, and meta information about the system (to determine which subclasses of a certain abstract class are available).

The application of the OO methodology discussed in Chapter 4 shows that an OO approach produces a clean, well-understood design that is easier to test, maintain and

extend than non-OO designs because the object classes provide a natural unit of modularity.

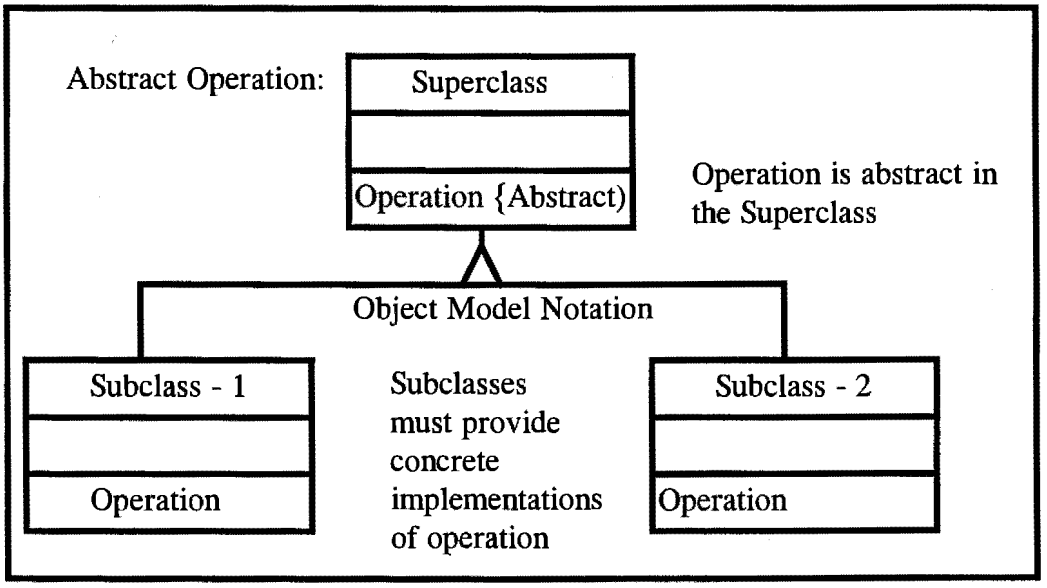


Figure 57. Object Notation

3.7.3.5. Data Management Library

Class Library

Distributed control process applications often require mechanisms for the management of a large amount of data shared among processes. Consistency and security of data are important, as is a sufficient level of performance.

Synopsis

Due to the use of abstraction, the underlying data management facility can be changed without affecting any clients. This results in greater flexibility and easier extensibility of the whole system.

3.7.3.6. Process Management Library

Class Library

Distributed systems consist of a number of co-operating processes that communicate and share some kind of resources (e.g., data). A mechanism must be provided for the co-ordination and supervision of processes. Distributed process control systems need to be highly reliable and, at the least, must provide mechanisms ensuring the integrity of the entire system after breakdowns of parts of the system.

Generally, a distributed process control and a SCADA system consists of a number of supervisory processes and other processes connected by communication lines using the communication class library.

Every supervisory process (at least one on each computer) controls a number of other processes. It is responsible for starting the controlled processes, synchronising them with the rest of the system and controlling them during execution. A supervisory process is also responsible for automatically restarting and resynchronising crashed processes.

3.7.3.7. Process Control Library

Class Library

The real aim of process automation is to monitor and control product treatment. The control task must be restartable so that a system crash cannot confuse the product treatment or data consistency.

Synopsis

This part of the automation system is the most specific part for the current application. There are not as many general and reusable classes as in the other parts. Nevertheless, it is believed that in these areas, class libraries for special purposes will reduce implementation work.

3.7.3.8. Control Panel Library

Class Library

The control panel serves as the interface between operators and the automation system. To control the product treatment, the processing states and the corresponding data have to be visualised at the control panel. For the interaction between operators and automation system, a modern GUI is required.

OO programming has been used for user interface implementation for a long time. So, there are many class libraries supporting user interface construction.

A primary aim is to design a nearly mode-less interface where the user can decide what to see or do next. If the user has to attend to a certain matter, dialogues pop up. Like all modern user interfaces, the user can see all possible commands in menus.

3.8. Reference Model for Process Control

This section presents the reference model for the process control domain in terms of the aspects of the Target System Reference Model. The meta primitives associated with each aspect is also depicted. The reference model is represented as a conceptual model first and is then instantiated in terms of ROOM.

3.8.1. Conceptual Reference Model for the Process Control Domain

The four aspects that have been defined as part of the Reference Model has been discussed within this chapter. Each aspect has a set of meta primitives that defines its role. The four aspects are:

1. Environment.
2. Information.
3. Systems Engineering.
4. Software Engineering.

Figure 58 represents the conceptual model for process control.

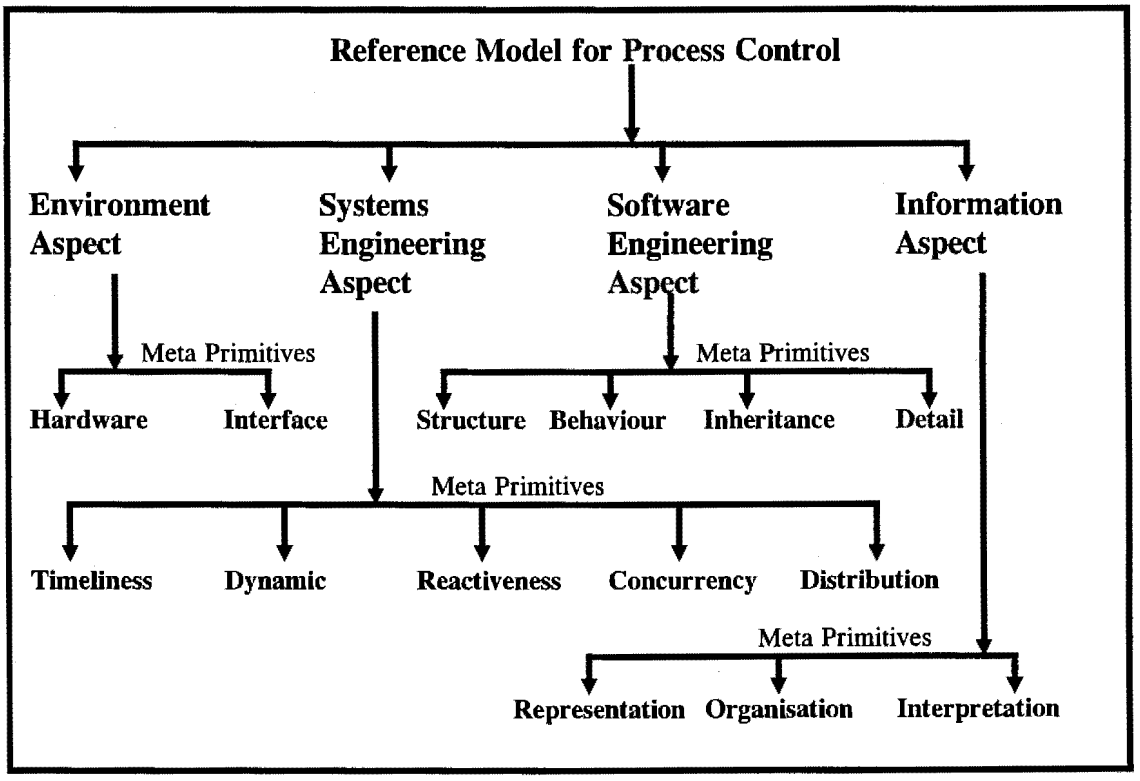


Figure 58. Reference Model for Process Control

3.8.2. Conceptual Model in terms of ROOM

The top level model of the reference model is illustrated in Figure 59. It is shown using ROOM's notation. There are basically two key components, the process control application and the co-ordinating system. The composition of both components are the aspects of the Target System Reference model.

The function of the co-ordinating system is to ensure that the events are controlled appropriately in terms of timing whilst taking into consideration issues such as concurrency and priorities. It may appear that there is an overlap between the co-ordinating system and the systems engineering aspect. The co-ordinating system is responsible for the external control whilst systems engineering is responsible for internal control. It has a port c through which it communicates with other high level control systems.

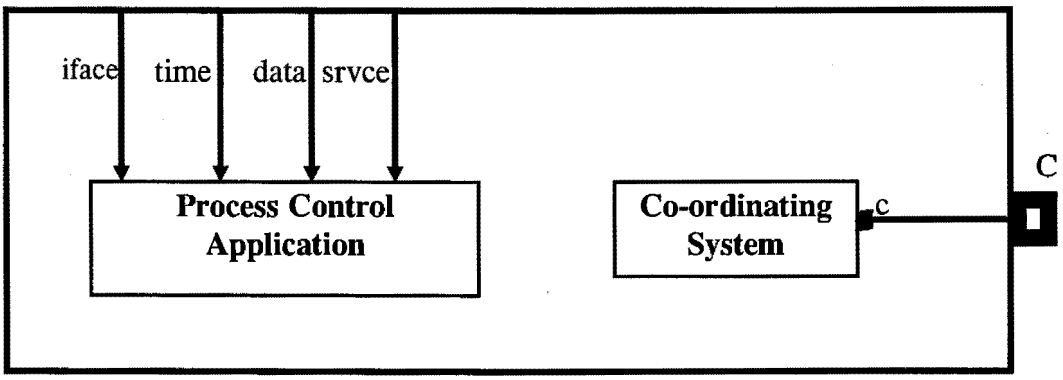


Figure 59. Reference Model for Process Control in terms of ROOM

3.8.2.1. Process Control Application

The four aspects in relation to one another and the co-ordinating system is shown in Figure 60.

The Environment Aspect is responsible for interfacing to the external environment. On receipt of an event from the environment this component forwards it to the

Software Engineering Aspect. One of the functionalities of the software engineering component is to ensure that the correct protocol is used for communications within the application. This component does not handle the timing in respect of events, hence it forwards it to the systems engineering component. Depending on the criticality and the type of communication required the environment component could interface directly to the systems engineering component. On processing the event, the data has to be interpreted, and stored or manipulated, therefore there is an "info" trigger.

Each component has a port c that is used for interfacing to the co-ordinating system.

It may seem misleading that only the time meta primitive of the systems engineering component is used as a trigger. The reason is that the others primitives are dependant on time.

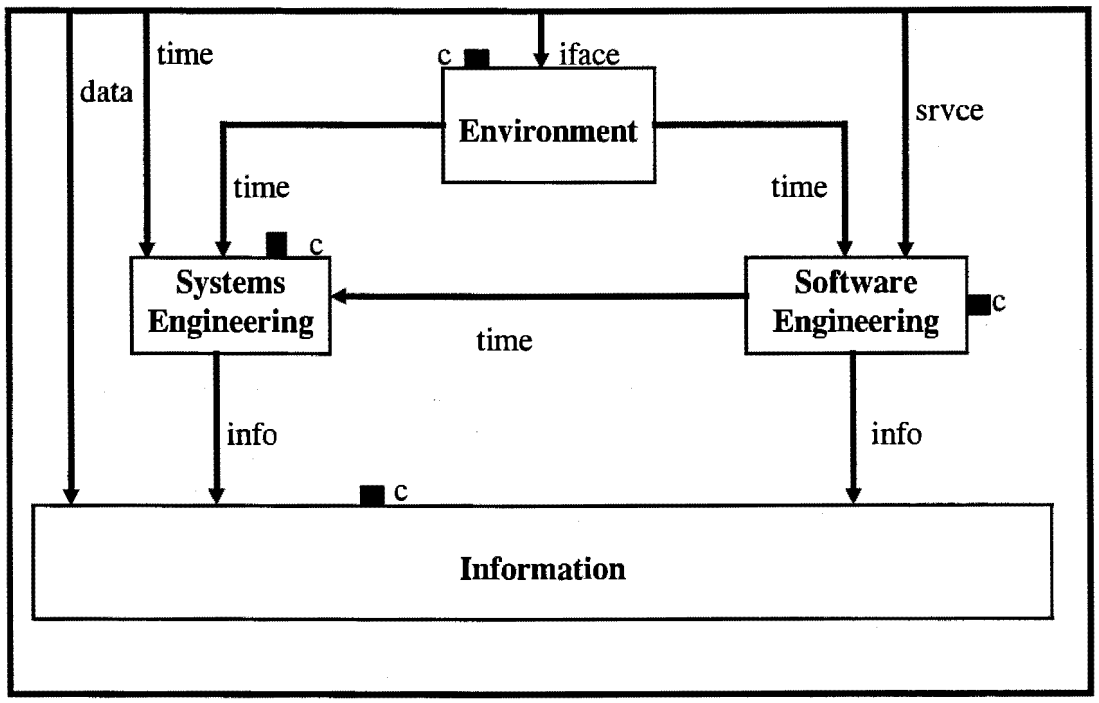


Figure 60. The Process Control Application

Similarly there are a few meta primitives of the software engineering component, but the attributes of the detail level are the triggers to the component. These

attributes provide a service, e.g. exception handling, timing of actor incarnations and communications between actors.

3.8.2.2. Co-ordinating System

The co-ordinating system is similar to the control system of the ROOM virtual machine. Essentially, they are providing the same functionality, that of co-ordinating the activities of the components and ensuring that there is synchronisation with the external control system.

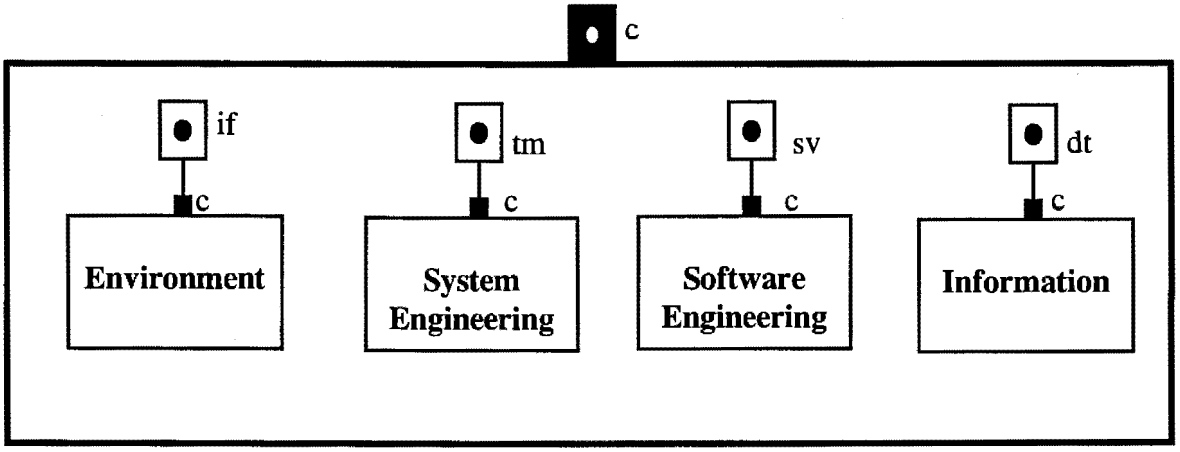


Figure 61. The Co-ordinating System

3.9. Conclusion

The environment, information, software engineering and system engineering aspects cannot exist in isolation. There is a close relationship between them as their boundaries are not well defined and there is clearly an overlap between each aspect's discipline.

The Reference Model proposed is meant to serve as a guideline in the development of process control applications. Due to the applications' inherent complexity, it is necessary to have a frame of reference that encompasses all the relevant issues in its

development. The reference model itself is pertinent to the development within the application domain, however there are external influences on the development, therefore the section on work organisation has been included in this chapter .

The aim of OO implementation in a process control environment is to increase the reuse of software in the automation software area, design flexible and reusable software and achieve a highly extensible product, especially if the same applications with probably even the same configuration is to be used at various plants.

Classes for communication, data management, process management and visualisation are highly reusable in any system where communicating processes and shared resources are involved. Only the reusability of the process control class library is restricted to automation software for similar apparatus in a plant.

In addition, the OO paradigm permits high extensibility and flexibility of the components. The use of meta information also increases the flexibility.

According to Pirklbauer [1994], experience has shown that OO is very powerful in cases where general behaviour and common structure can be factored out. This has been identified by ROOM and discussed in this chapter. Although it is more difficult to find general behaviour in the process control area than in GUIs, it has been shown that in some implementations, the former does involve general behaviour and that the concepts of OO programming and application frameworks in particular, are demonstrably very valuable in this domain.

CHAPTER 4

DEMONSTRATION OF CONCEPT

4.1. Introduction

4.2. Problem Statement

4.3. Application of ROOM to the Problem

4.4. Conclusion

4. CHAPTER FOUR: DEMONSTRATION OF CONCEPT

4.1. Introduction

The previous chapter discussed the ROOM modeling approach, which provides a foundation for a real time methodology. This chapter describes an iterative approach to applying the ROOM methodology to a chosen application problem. It also discusses associated fundamental heuristics, such as how to pick actor classes in the domain of discourse. The intention of this chapter is really to demonstrate the suitability of the implementation of ROOM within the process control domain.

This dissertation is one of limited scope, therefore a small application is demonstrated. It is nevertheless a proper implementation within a process control environment, requiring input from an operator.

4.2. Problem Statement

Umgeni Water is a water purification organisation in Kwa Zulu/Natal. It is a rapidly expanding company, with new plants, pump stations, reservoirs etc. being constantly constructed as well. The company's growth is also attributed to the number of "take overs" of existing treatment plants from the local authorities. Its area of supply extends in the Coastal area, along the North and South Coast and in the Inland area as far as a little town called Howick which is located close to Umgeni's Head Office in Pietermaritzburg.

Recently, a new reservoir was constructed at one of the treatment plants in Pietermaritzburg, to service the increased demands from consumers. This new reservoir is fed from the plants' reservoir (Clearwell) to the new (outgoing)

reservoir. Signals (such as reservoir level, alarm statuses, etc.) are transmitted from the reservoir to a PLC that is local to the reservoir. This PLC then reroutes the signals to the central PLC, called the Plant PLC. The Plant PLC then passes the information to an existing computer system, (SCADA), for the use of the plants' shift operator, see Figure 62.

A single remote controllable valve was installed at the inlet to the outgoing reservoir. The requirement for automation on this reservoir was to provide the facility for the shift operator to open or close the valve remotely from the existing SCADA package. The opening condition is not necessarily a full open position, e.g. the valve can be opened to 65%. Such a valve setting is referred to as a setpoint. The procedure is that the shift operator selects from the relevant mimic, on the SCADA system, the inlet valve. A "pop-up" screen relevant to that valve is displayed. The shift operator selects the setpoint and waits for acknowledgment from the remote site that the value has been received. Only then is the changeover relay bit sent which initiates the valves' motion. Once the valve is within a 5% range of the setpoint it stops moving and feeds back its new position to the shift operator.

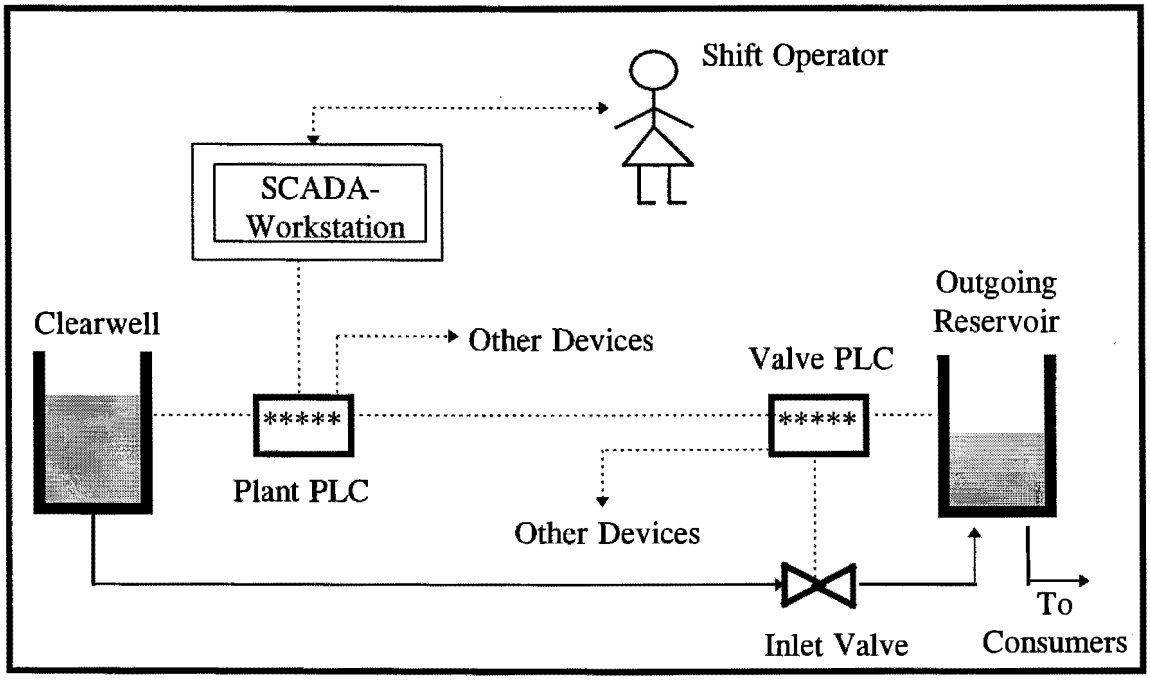


Figure 62. Plant Layout

The SCADA package resides on a workstation which has a LAN connection to the plant PLC. The plant PLC has a fibre optic link to a PLC at the valve as illustrated in Figure 62.

The problem is to model the functionality of such a system with emphasis being placed on the Plant PLC's program. In order to understand and validate the requirements an operational model is constructed.

4.3. Application of ROOM to the Problem

Before commencing the modeling, it will be useful to include an overview of the steps involved :

1. Distinguish between the system and environment components.
2. Model the system boundary.
3. Model each component of the system.
4. Identify interactions between components.
5. Model the interactions.
6. Model the behaviour of each component (now identified as an actor).
7. Re-iterate and Validate the model at all stages.

Step 1

As a starting point to the model, there has to be a distinction between the components of the system and the environment. This is necessary to establish the system boundary.

From Figure 62 above it is obvious that the SCADA is inside the system. The PLCs are responsible for sending the appropriate signals between the valve and the SCADA are also inside the system. The shift operator forms part of the environment

as she is only a user of the system. The valve also falls within the environment as it is a device that is acted upon. The Clearwell and Reservoir does not form part of the control system and is not included. Figure 63 illustrates the system boundary.

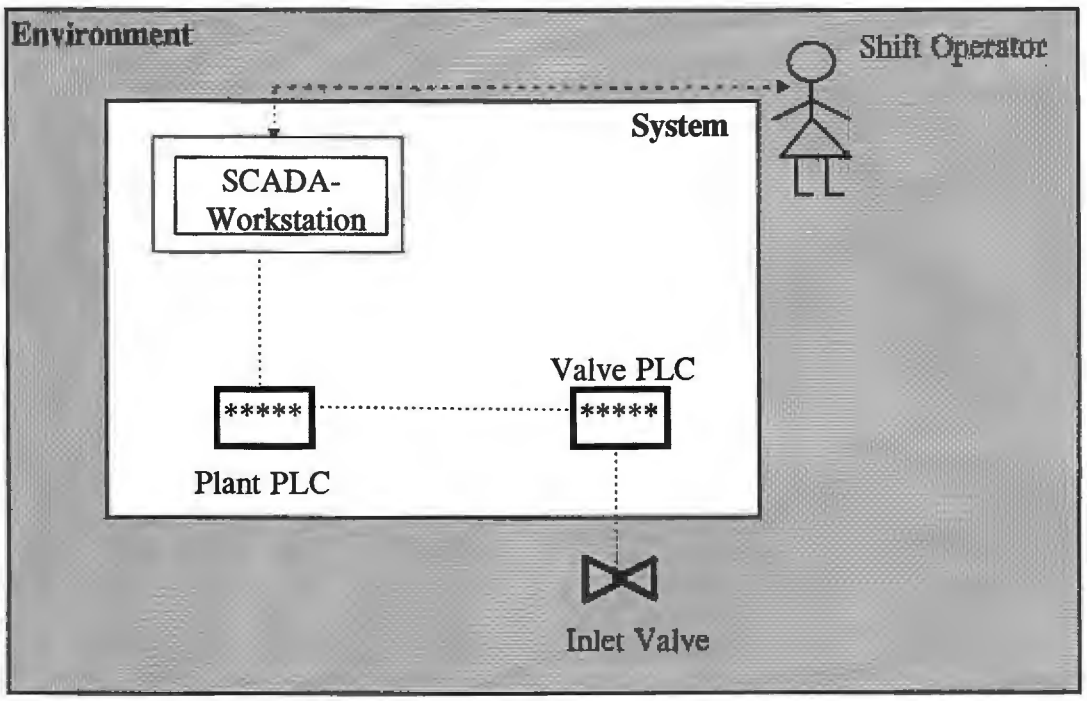


Figure 63. System Boundary

Step 2

The initial ROOM model that depicts the system and the environment is shown in Figure 64. The valve and shift operator are modeled as actors as they are obvious concurrent physical objects.

To keep track of the primary purpose or responsibility of each actor class, it is a suggestion to annotate each class with a description and its purpose (enclosed in quotes).

- Shift Operator: "A person using the SCADA system." One of the functions is to control the inlet valve.
- Valve: "The inlet valve which is controllable."
- System: "A collection of hardware and software that provides control of the inlet valve."

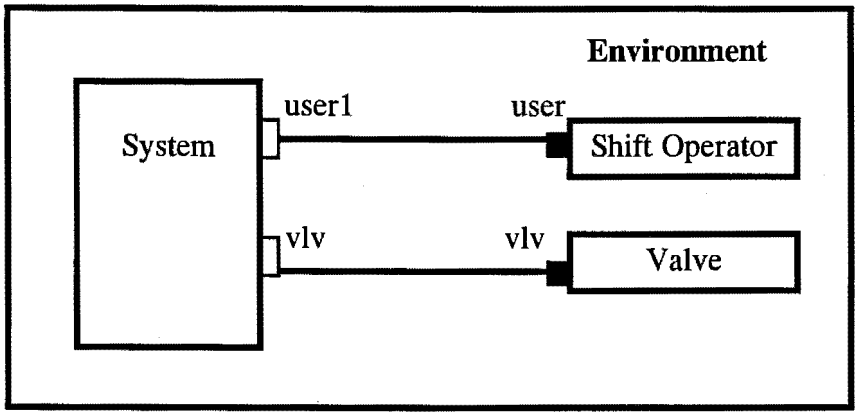


Figure 64. Initial Model: System Boundary

The previous step outlines only the high level classes of the model and did not address their interaction protocols. To do this, all possible scenarios within the domain should be considered. However this can get very complex especially if the specification is not complete, as new requirements could emerge.

Rather than representing the scenarios textually it can be beneficial for understandability purposes to represent it as a message chart. This information is not inherently message based but can still be modeled as abstract messages. For example, in the message sequence in Figure 65, *Idle*.

From the message sequence, a protocol class from the viewpoint of the Valve can be defined:

protocol class VlvInteraction

in: {{ Send %, null}}

out: {{Send Ack, null},{Feedback new position, null},{idle, null}}

and for the Shift Operator is:

protocol class ShiftOperatorInteraction

in: {{Highlight Valve, null},{Wait for Ack, null}

out: {{Select Valve, null},{Select % to open ,Setpoint},
{Send Relay, null},{Close Valve Menu, null}}

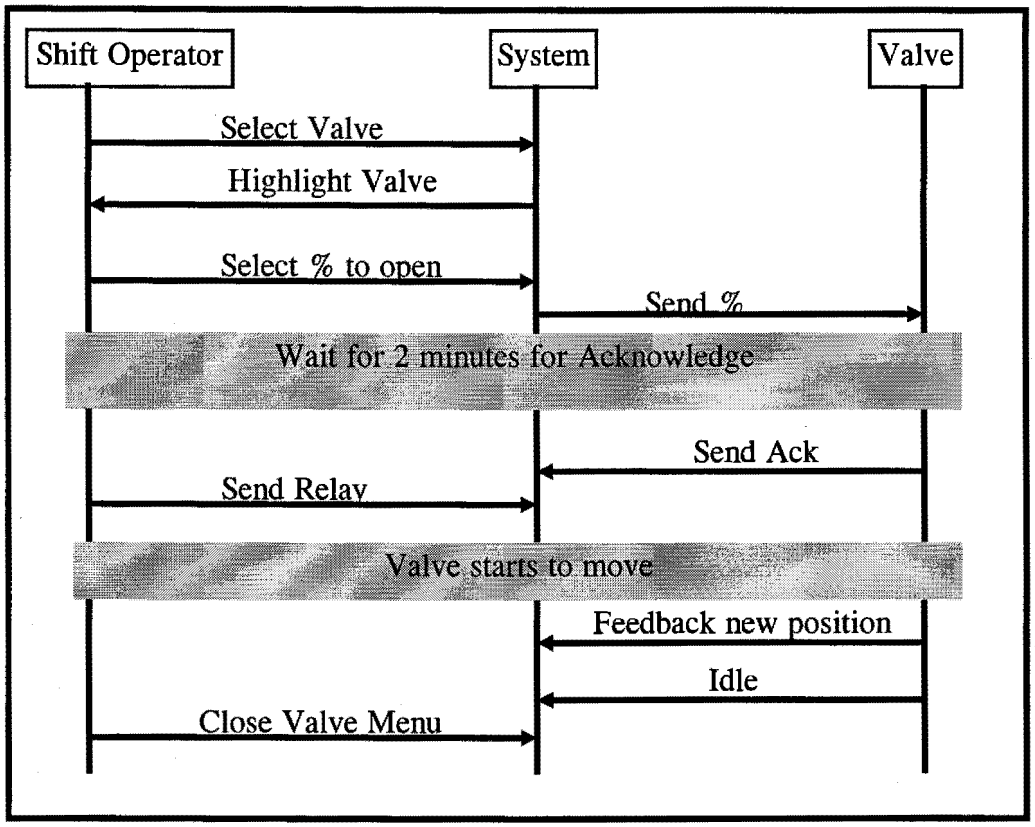


Figure 65. Valve Control

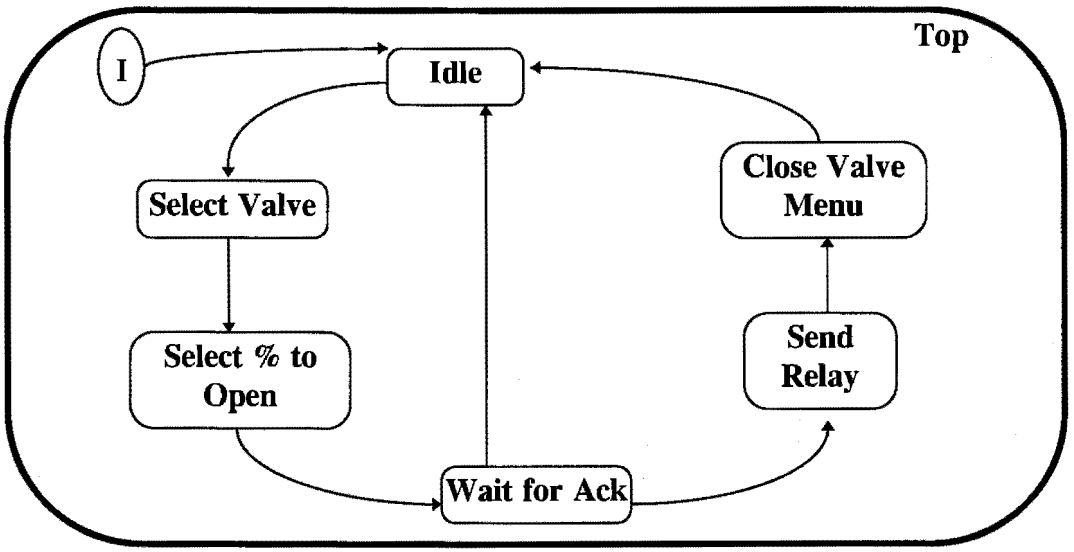


Figure 66. Shift Operator Behaviour

The behaviour has not been specified yet, so the model is not executable. Protocols are closely associated with behaviour. A preliminary version of the Shift Operator's behaviour is shown in Figure 66 to get a better understanding of the scenarios. Transitions are not specified as this is just to get a quick impression of the states.

Later when the shift operator is employed to validate the system a formal approach to define its behaviour can be taken.

Step 3

After modeling the environment the next step is to model the components of the system. The most obvious components are the physical objects from the requirements specification. These are modeled as actors contained by the system actor (which starts to look recursive):

- SCADAUI: "The software system which allows an interface to control the valve".
- Valve PLC: "Sends and receives signals from the Plant PLC and interfaces to the outgoing reservoir and the inlet valve".
- Plant PLC: "Sends and receives signals from a number of plant devices, including the Valve PLC".

These actors are included in Figure 67. The Valve PLC has two interface components:

1. Vlv - that is a reference to the VlvInteraction protocol and
2. PLC - that is based on PLCInteraction protocol class which is yet to be defined.

The SCADAUI actor also has two interface components:

1. Shift Operator - that is a reference to the ShiftOperatorInteraction protocol and
2. Dev - that is based on the DeviceInteraction protocol which is yet to be defined.

These protocols are needed to assert that there is some connection between the Valve PLC and the Plant PLC and the SCADA User Interface (SCADAUI) and the Plant PLC.

The line between the Valve PLC and the Plant PLC and the SCADAUI and the Plant PLC are shown as bindings since it appears to be a straightforward communication link. The Valve PLC is connected to the PLC interface on the Plant PLC which is based on the conjugated PLCInteraction protocol. The SCADA actor is connected to the Dev interface on the Plant PLC which is based on the conjugated DevInteraction protocol.

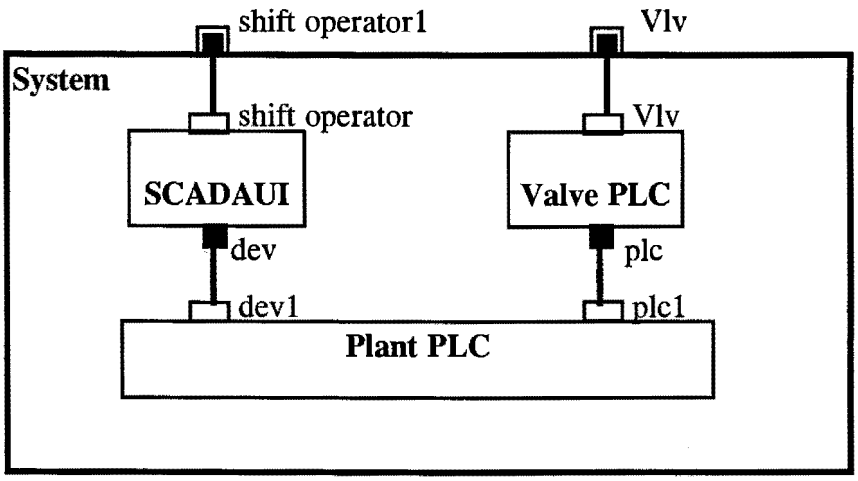


Figure 67. System Structure

Step 4

In order to better understand the PLCInteraction and the DevInteraction it is realised that more detail is required from the "specifier" (the person who provided the problem statement). As a consequence of this a message sequence chart shown in Figure 68 is drafted. There are some implications in the protocol e.g. if there are a number of PLCs connected to the Plant PLC, the Plant PLC knows which PLC is the Valve PLC.

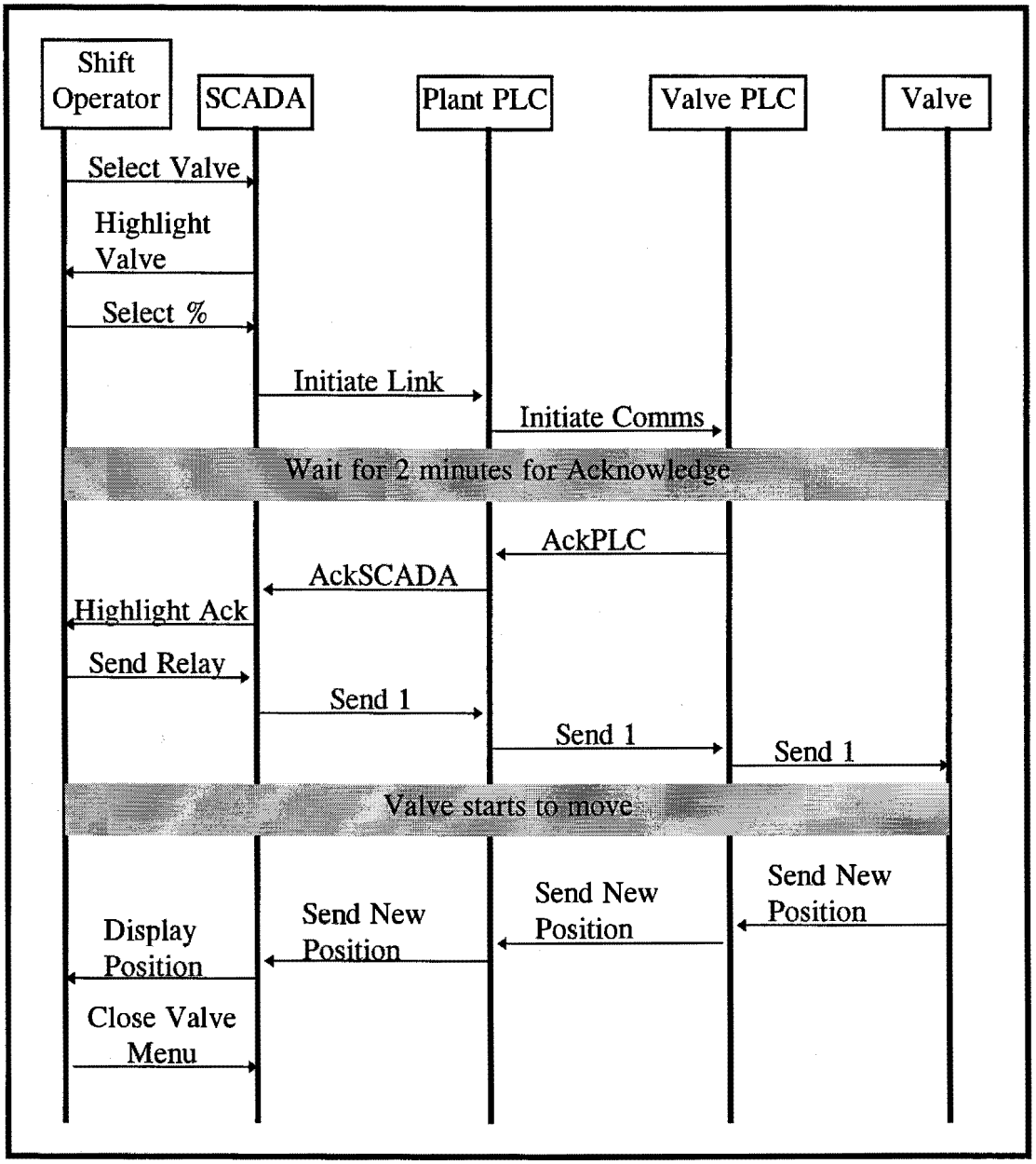


Figure 68. An Instruction to Close the Inlet Valve

The signal definitions are:

- Select Valve: "Select from a global mimic of all the outgoing reservoirs, the inlet valve".
- Highlight Valve: "The SCADA system highlights the selected valve".
- Select %: "Select the required setpoint".

- Initiate Link: "A request to initiate communications between SCADA and the Plant PLC".
- Initiate Comms: "A request to initiate communications between the Plant PLC and the Valve PLC".
- Wait for Ack: "The entire system awaits the acknowledgment that the Valve PLC has received the setpoint".
- AckPLC: "The Valve PLC acknowledges the Plant PLC that it has received the setpoint".
- AckSCADA: "The Plant PLC acknowledges to the SCADA that the Valve PLC has received the setpoint".
- Highlight Ack: "The SCADA system highlights the setpoint as an indication that it has been acknowledged".
- Send Relay: "The SCADA system sends a relay (bit 1) to initiate the valve motion".
- Send 1: "The bit 1 is sent from the SCADA system to the inlet valve which it recognises as an active bit, and it starts to move".
- Send New Position: "The new valve position is fed back to the shift operator".
- Display Position: "The SCADA displays the new position".
- Close Valve Menu: "The menu that would have been opened when the shift operator selected the inlet valve is shut".

Step 5

From the above it is now possible to define PLCInteraction and the DevInteraction protocols from the viewpoint of the Valve PLC and the SCADA respectively:

protocol class PLCInteraction:

in: {{Initiate Comms, null},{Send 1, null}}

out: {{AckPLC, null},{Send New Position, null}}

protocol class DevInteraction:

in: {{AckSCADA, null},{Send New Position, null}}

out: {{Initiate Link, null},{Send 1, null}}

At this stage the modeler has a basic understanding of what the Plant PLC must do. A preliminary sketch of the solution is posed. It is important to note that the solution devised at this early stage is the purpose of gaining deeper insight into the requirements and does not represent a commitment to any particular design alternative.

The total functionality of the Plant PLC could be modeled by a single actor but this could result in quite a complex state machine. It is therefore preferable to partition the functionality amongst component actors. The Plant PLC actor forms an abstraction container for these components.

It is useful to formalise the coordination relationships between objects. In the SCADAUI sending a setpoint, it has to wait for acknowledgment before sending the relay to initiate the valve motion. Also the Valve PLC wishes to respond to the Plant PLC, both must be connected to allow them to communicate. Therefore an abstract coordinating actor is defined to encapsulate this functionality:

- Link: "The relationship between 2 devices, including initiating communications to them, connecting devices and closing down communications".

It is realised that the SCADAUI as well as the Valve PLC could change. Therefore, to shield the Link actor, the SCADAHandler and ValveHandler actors are introduced. The resulting structure is illustrated in Figure 69.

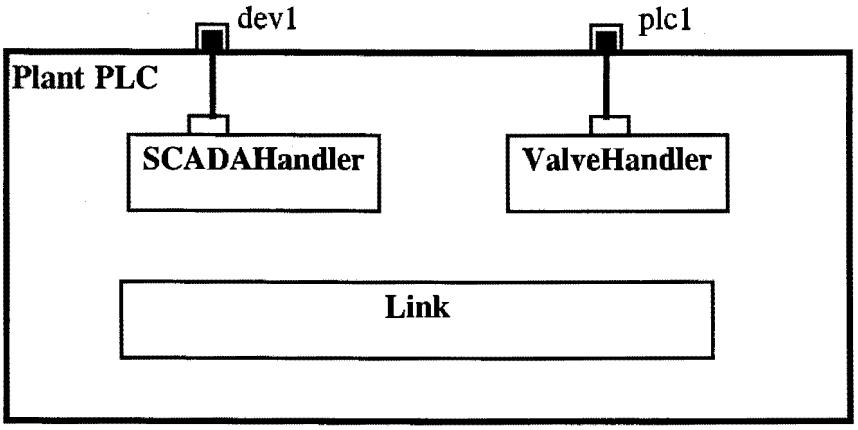


Figure 69. Initial Plant PLC Structure

However there are omissions (planned!!!) and the structure has to be refined. The omissions are that there has to be communications between the SCADAHandler and the ValveHandler for the operation of the valve. It is possible to reuse the PLCInteraction and the DevInteraction protocols between the Valve PLC and the Link and between the SCADA and the Link respectively. Figure 70 shows the result of the model capture.

At this point it appears that the decomposition into components appear simple enough and that their functionality can be captured without further decomposition. In order to check the validity of the system it is necessary to specify the actors' behaviours.

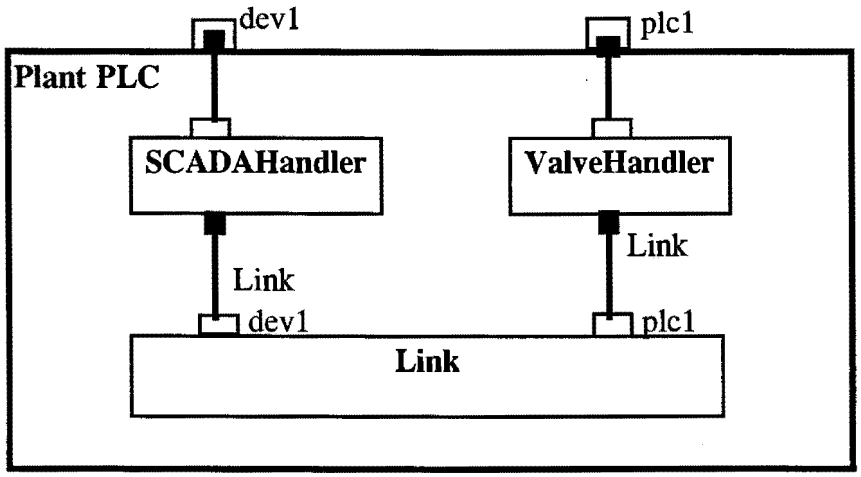


Figure 70. Final Plant PLC Structure

Step 6

The capture and subsequent execution of behaviour often uncover deficiencies in model requirements or design. The behaviour of the Shift Operator, depicted in Figure 66 was very loosely approached.

The Plant PLC actor is considered first as it does not require an explicit state machine since all its external interactions are handled by its components. Its behaviour is a result of the actors it contains. The ValveHandler is considered first. The valve is initially in the *Idle state*, as illustrated in Figure 71. It is assumed that the valve will only start moving once it receives a *start moving* (hence the transition) signal from the PLC. Note that even if there is no remote communication between the PLCs, i.e. the valve is under local control, it still is operated under the Valve PLC. At this stage the state transition code to be captured is that the Setpoint figure is stored in the NewPosition variable.

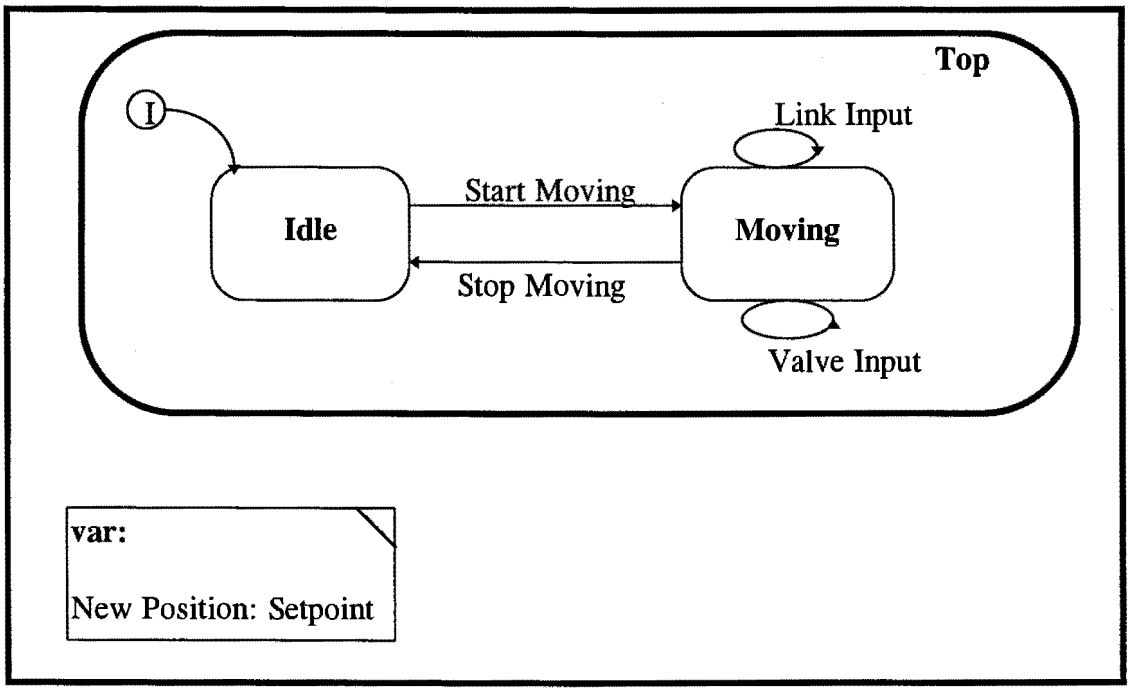


Figure 71. The ValveHandler Behaviour

There has to be some communication between the Valve PLC actor and the Plant PLCActor and thus there is a self transition on all the messages coming from the

Valve PLC (from the PLCInteraction (AckPLC and Send New Position)); this is termed *ValveInput*. The *LinkInput* does the same in the reverse direction (from the PLCInteraction (Initiate Comms, Send 1)).

The ValveHandler behaviour is now complete and it can be validated, by compiling and loading the actor into a model execution environment.

To capture the remaining detail, it is important to note that it is important to achieve a consistent level of completeness among the components that interact with each other. The behaviours of the SCADAUI, ValvePLC, SCADAHandler are shown in Figure 72 and the Link actor is shown in Figure 73.

The SCADAUI actor class is very simple. It relays information between two interfaces. It does not need a state machine. An internal binding to the actor is used between its two interfaces to simulate its relay-like function..

The SCADAHandler is similar to the ValveHandler.

The Valve PLC actor is also quite simple and self explanatory.

The Link actor is more detailed but its trigger from the idle state is a request from the SCADA to initiate a link between the SCADA and itself. A link to the Valve PLC has to be established and an acknowledgment of receipt of the setpoint is waited for. The time waited for is 2 minutes and will be captured within the t:{timeout, timer} service. On receipt of the acknowledgment, a one is fed to the Valve PLC which is transmitted to the valve and it will recognise as an action bit and will start moving. When the new position or as close as to the setpoint is reached the valve will stop moving. This will result in the Release transition and it will ensure that all links established for the duration of the valve motion is removed. The Link actor is illustrated in Figure 73.

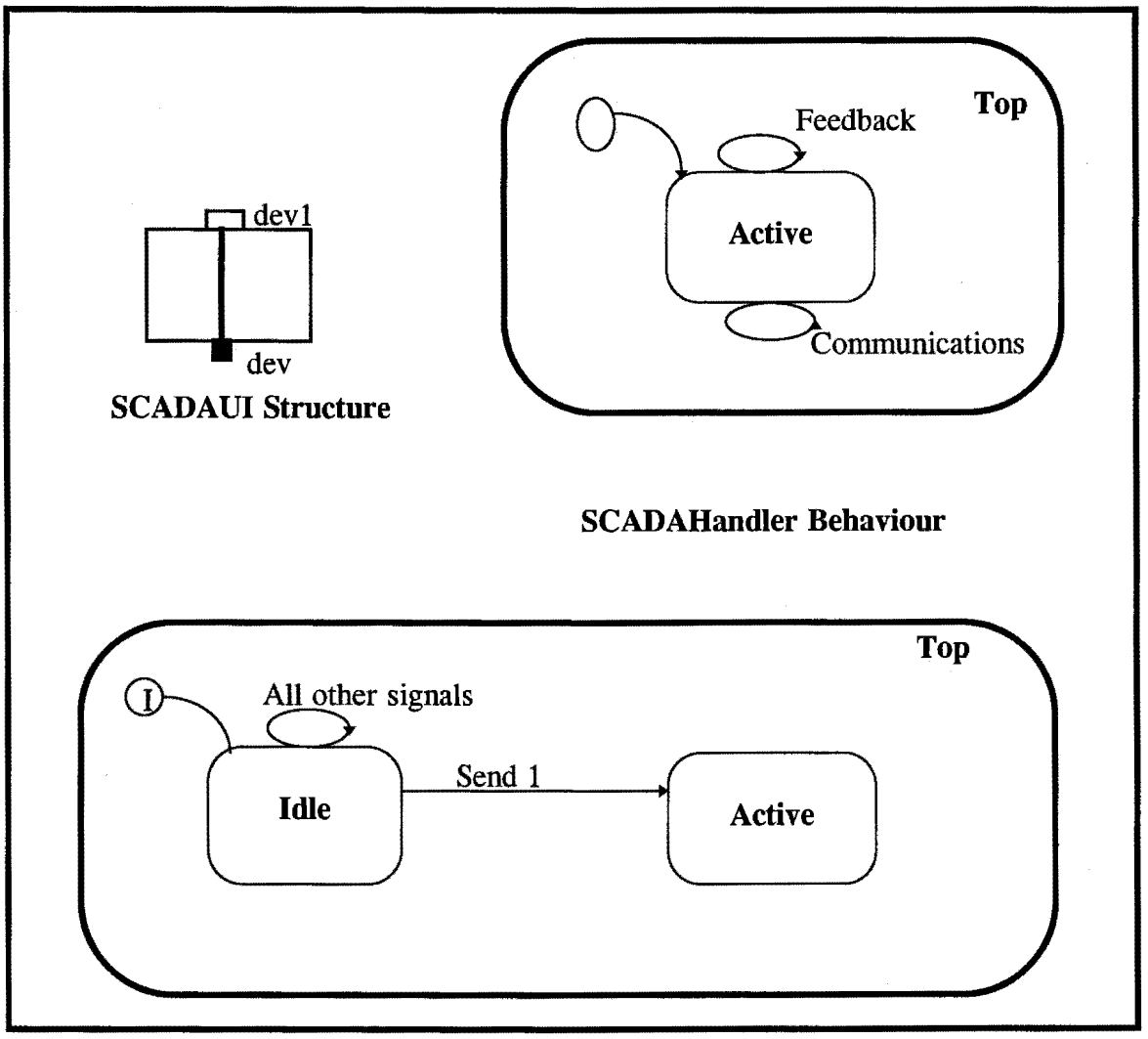


Figure 72. SCADAUI, SCADAHandler and ValvePLC Behaviour

The behaviour components for all the Plant PLCs' components have been completed. It can be validated by manually injecting messages into the actors but it is preferred to take a more automated approach. For this purpose, validation components have to be created.

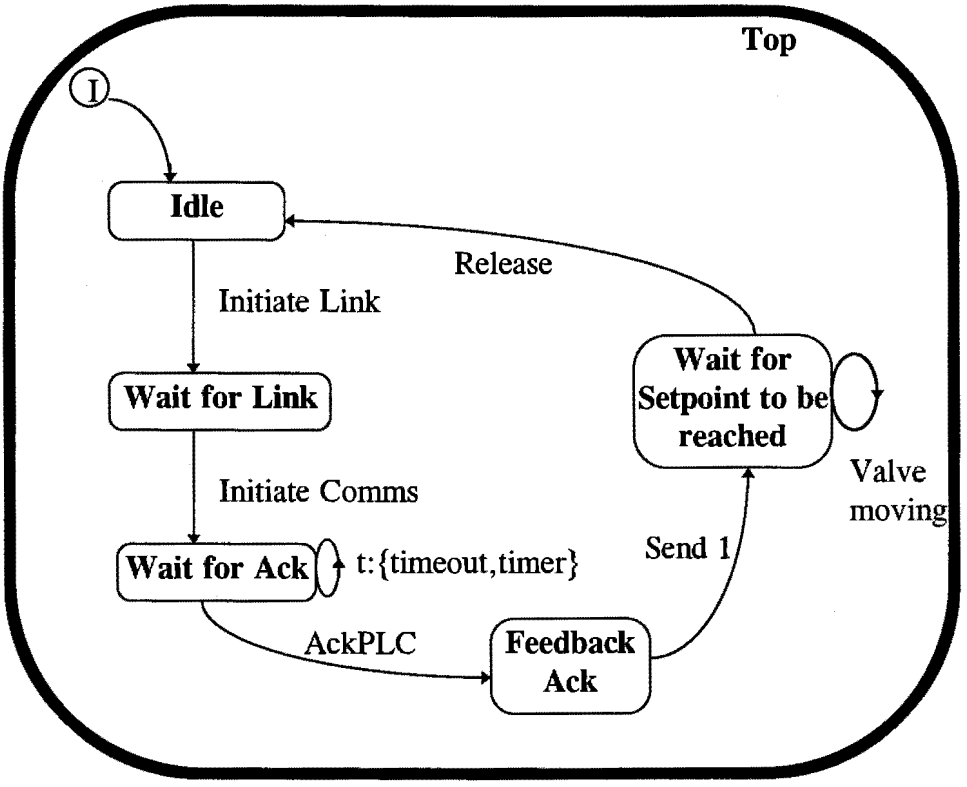


Figure 73. Link Actor Behaviour

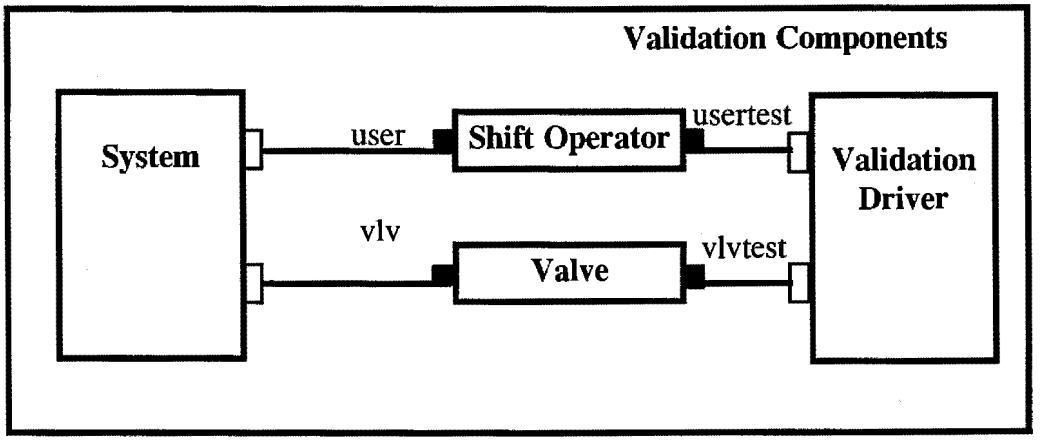


Figure 74. System and Validation Components

Step 7

To validate complex scenarios affecting several components it is preferable to create explicit validation components. Validation components are like any other part of the model - they are constructed using the ROOM modeling language. The best place to start is in the systems environment, as a basis for validation. Figure 74 shows the validation components added to the model.

During early construction iterations of the model, deficiencies in the requirements have been identified. In validating the possible scenarios, it appears that a sequential operation to open/close a valve will operate properly. However the concept of two shift operators trying to open the valve concurrently has not been handled. Worse still if one operator is trying to open the valve while another is trying to close it. This implies that the requirements and specifications were not clarified when the valve control was being discussed.

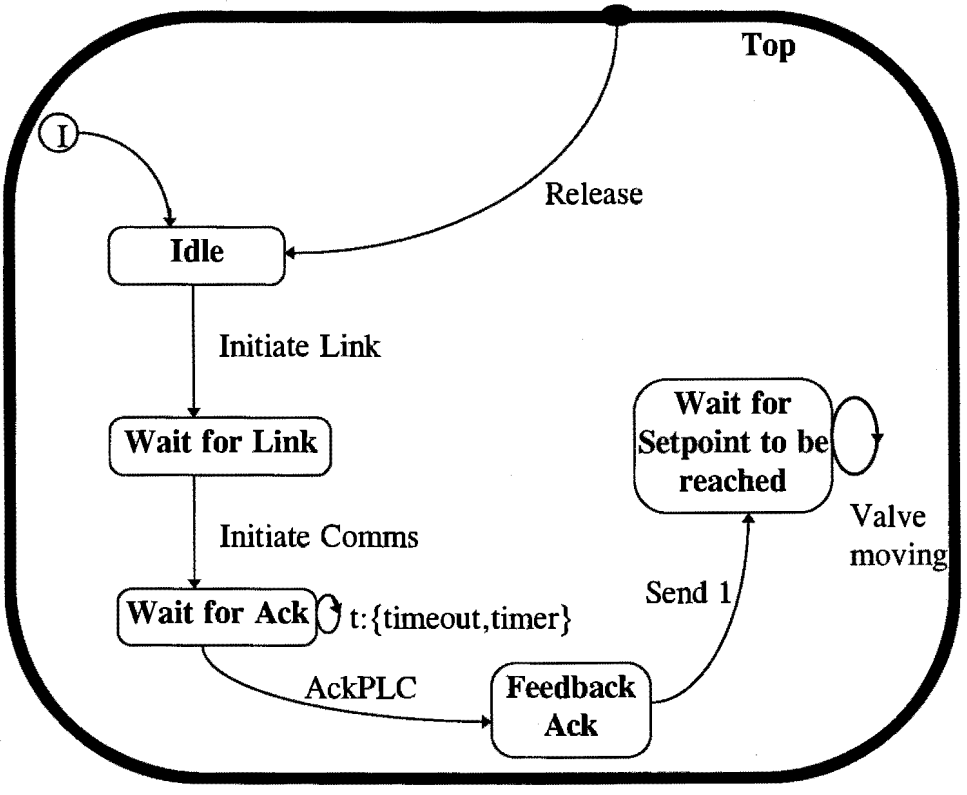


Figure 75. Link Actor Behaviour (Refined)

Another subtle defect is: what if after the Valve PLC has acknowledged that it has received the setpoint, the operator decides not to open the valve and thus does not send the relay/send 1? The Link actor will remain in the Feedback Ack state forever. The Link actor is therefore modified, a group transition triggered by Release to take the state machine back to idle. This is illustrated in Figure 75.

In distributed systems individual components can fail as can communications paths between them. These problems are frequently overlooked in specifications given to development teams, since they are considered as "implementation" concerns. It is often left to developers to apply their domain knowledge to address these issues. A quick review of potential distribution effects uncovers the following scenario:

Transmission Link Failure: While the Valve PLC is feeding the Ack back to the Plant PLC, there is a link failure.

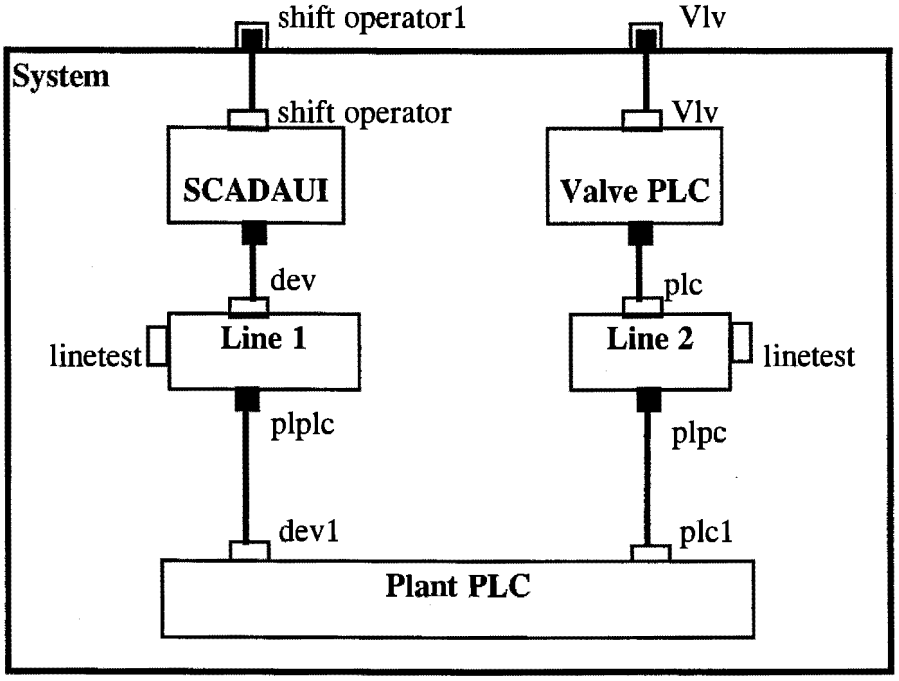


Figure 76. Iteration of System Structure

To model this problem, the validation approach has to be enhanced. The simple bindings between the Valve PLC and Plant PLC as well as between the SCADAUI and the Plant PLC should be replaced with an actor that mimics the characteristics

of real lines. This new *line* actor is inserted in the system structure as shown in Figure 76. The same actor can be used for both the SCADAUI and Valve PLC as its functionality is the same.

- Line: "The communication link between the System Components."

Note that this is a validation component inside the system. The *linetest* interface is used to instruct the *Line* actors to simulate communication link failures. The Line actor's behaviour either relays or ignores the messages on the *dev, plc* and *plplc* interfaces, depending on the state of the link.

Often it is useful for abstraction purposes to encapsulate all components that are strongly related to each other. For this reason, a new *Validation System* actor is created, as shown in Figure 77.

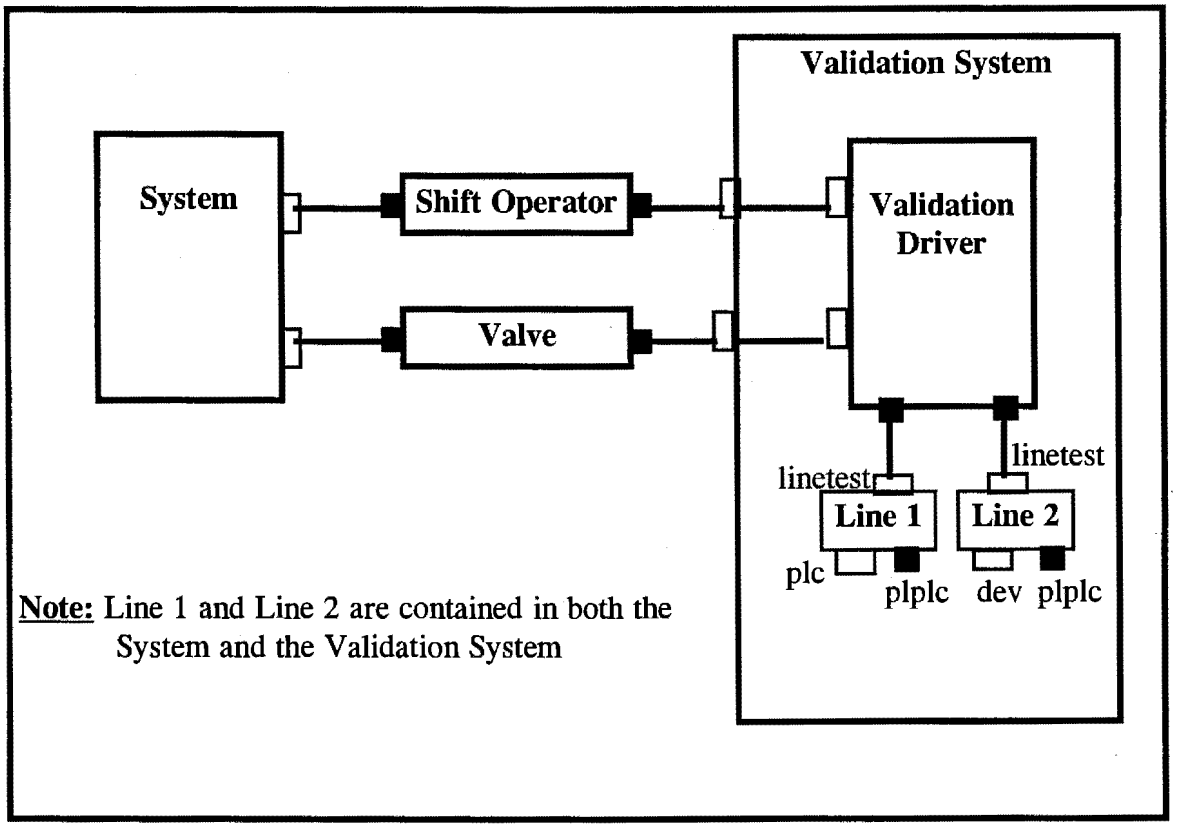


Figure 77. Iteration of the Validation System Structure

The *Validation System* contains all the components strongly related to validating the system, but may not be considered to be directly interacting with the system. This includes the *Line* actors, which are contained in both the *System* and the *Validation System*. Multiple containment has been used because the *Line* actors are logically part of two systems simultaneously.

There may still be aspects of functionality overlooked but it is considered that the concept has generally adequately demonstrated. The complete structure of the Plant PLC is depicted in Figure 78.

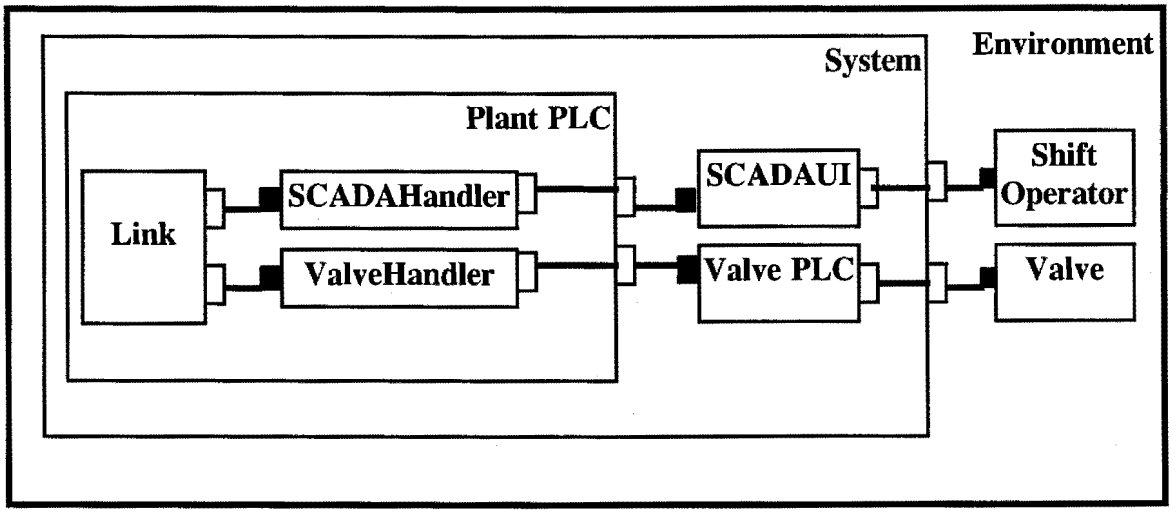


Figure 78. The Complete Structure of the Plant

4.4. Conclusion

In order to understand and validate the requirements for the system, a boundary was defined to distinguish between the system and the environment. Using scenarios it was possible to derive the key protocols.

Next an early view of the system structure was captured. The initial set of actors were implied by the concurrent physical components described in the scenarios. As

the actors, protocol and data classes were defined, the purpose for each class was documented. An effort was made to balance the effort between structure capture, behaviour capture and model execution. Further, a particular aspect did not have to be defined before capturing other parts.

Abstract actors were defined for coordination and interface decoupling purposes. By incrementally capturing more structural details, the need for further protocol definition was uncovered. Assumptions were made at various stages to simplify the modeling and this uncovered specification deficiencies.

The model was validated at the earliest possible time. Rather than defining the behaviour for all actors in one step, a key actor was selected and validated. The behaviour for this actor was derived by studying the scenarios that it supported. The concurrent execution of the actors in the scenarios uncovered subtle problems that drove further iteration of the model.

CHAPTER 5

Summary of the Dissertation

5.1. Objectives of the Dissertation

5.1.1. Identify the Aspects

5.1.2. Identify the Meta Primitives of each Aspect

5.1.3. OO Methodology

5.1.4. Propose the Reference Model

5.1.5. Application of the Methodology

5.1.6. Evaluations

5.1.6.1. Constraints/Limitations

5.2. Suggested Extensions

5.3. Summary

5. CHAPTER FIVE: SUMMARY OF THE DISSERTATION

5.1. Objectives of the Dissertation

The objective of this dissertation was to propose a reference model within the process control domain of application. This reference model falls within the scope of the Target System Reference Model, which is one of the four reference models of the OOISEE project.

To propose the reference model, the following objectives had to be met:

1. Identify the aspects of the reference model.

2. Identify the meta primitives for each aspect.

3. Develop/Identify an existing OO methodology for application to process control. It must be capable of executing on a LAN that is open systems compliant.

4. Propose the reference model, conceptually and in terms of OO.

5. Apply the methodology.

6. Evaluate the methodology based on its application.

Each of these objectives are discussed.

5.1.1. Identify Aspects

The aspects were proposed by Steenkamp[1995], each was evaluated for its relevance to the process control domain. These aspects are:

- Environment.
- Systems Engineering.
- Software Engineering.
- Information.

Each of these aspects have been discussed in great detail in Chapter 3.

5.1.2. Identify Meta Primitives of each Aspect

Each aspect is characterised by its meta primitives. These meta primitives and its attributes are discussed in Chapter 3. This list is not intended to be all encompassing. It can be extended if deemed necessary.

5.1.3. OO Methodology

A literature survey of a few OO methodologies was conducted. The evaluation of each is listed in Chapter 2. In Chapter 1, it was stated that an attempt would be made to adopt the revised spiral life cycle model. It was found that it was not suitable for the process control domain, further explanations are in Chapter 2.

The ROOM methodology was selected for application within the process control domain. It provides a real-time modeling approach which is well suited to the process control environment. Major issues that distinguish process control systems from other systems are the handling of time, concurrency and distribution. ROOM handles these

issues very well. Further, it is not dependent on any programming language nor does it have specific hardware requirements, i.e. it is open systems compliant.

5.1.4. Propose the Reference Model

The reference model for the process control domain of application has been formulated in terms of the aspects and the meta primitives identified. The model has been represented conceptually as well as using the ROOM notation.

5.1.5. Application of the Methodology

ROOM was applied to a process control application. The application was to an extension of an existing system. Due to the limited scope of this dissertation, the full potential of ROOM could not be exploited.

5.1.6. Evaluation of the Methodology

ROOM is distinguished by the following major features:

- It is inherently OO, which allows it to fully exploit the advantage of this new paradigm.
- It has powerful modeling concepts that are specific to the real-time domain and which facilitate the construction of accurate (yet concise) system models.
- It provides for the explicit capture and documentation of system architecture.
- It provides executable models at all levels of abstraction, allowing early detection of requirements or design flaws.
- It supports an incremental and iterative development process that covers all aspects of development.

ROOM was organised around the following three key elements:

1. The operational approach.
2. A phase independent set of modeling abstractions.
3. The object paradigm.

A common theme that unites the three elements is the elimination of discontinuities in the system development process. Discontinuities are intrinsic to the nature of the process itself and cannot be completely removed. For example, the requirements definition, design and implementation activities are characterised by different thought processes, emphasis on different kinds of details and different verification criteria. Thus the choice of particular notations or model building strategies can introduce unnecessary, artificial discontinuities into a development project. ROOM with its modeling techniques eliminates these problems as there is a constant notation used throughout the cycle Selic[1994].

Executable models, especially those enhanced by validation components, can uncover further problems. These problems may not be found by a static inspection of the model. Further, users have early confidence in the model if it is executable.

The demonstration of concept in the previous chapter applied the ROOM modeling technique to a simple process control application and while there were benefits, some of which are listed above, there are a few limitations that should be noted.

5.1.6.1. Constraints/Limitations

- It is imperative to prepare some form of reference grid or framework when utilising ROOM as it is a formal language and spans a significant portion of the overall development cycle. (Hence the Reference model for the Process Control domain). It abounds in a variety of concepts, rules and guidelines. On first encounter even the most "adventurous" developer could be overwhelmed by the level of detail

required and some navigational assistance may be required. As the details of this conceptual space is described, no matter how systematic the traversal, there is a danger that "the forest will be obscured by the trees" as phrased by Selic[1994]. This of course does not lend itself to the usability of the model!

- The full benefits of the ROOM modeling language can be reaped only if suitable computer tools are available. This is a departure from tradition, since most software development methodologies are based on the premise that computer based tools are possibly useful (but not fundamental) "power boosters".

  Within the ROOM methodology as described previously there are three levels:

  1. Work Organisation
  2. Modeling Heuristics
  3. Modeling languages

- ROOM is part of level 2 and there is just *one* tool that can be applied to this level. This tool provides a modeling environment to support the ROOM language, including model capture and display, model analysis and model execution. (A model that can be executed and run). There is very limited support of the other two levels of the methodology. This is an area of concern as there is only one tool available (very specialised) and also that the methodology is not fully supported now and does not fit neatly into UNISA's objective of developing a complete OOISEE.

- Further, another area of concern is that as the complexity of systems increased, the approach taken by ROOM was to separate the concepts and notations used for different scopes or levels of granularity of the software system, Selic[1994]. The picture that emerged from this is a vertical stack of concept bases each with its own notation and each dealing with a different scope, see Figure 79 [Selic1994].

Currently ROOM only covers two of these abstraction levels, the Detail Level and the Schematic level. The Detail level deals with concepts for modeling the structure and behaviour of passive data objects such as strings, numbers etc. It was felt that it was simpler to utilise existing programming languages to handle this level rather than cater for it within ROOM. The Schematic Level provides concepts for dealing with higher-level phenomena (including concurrency and distribution) and ROOM handles this level quite adequately.

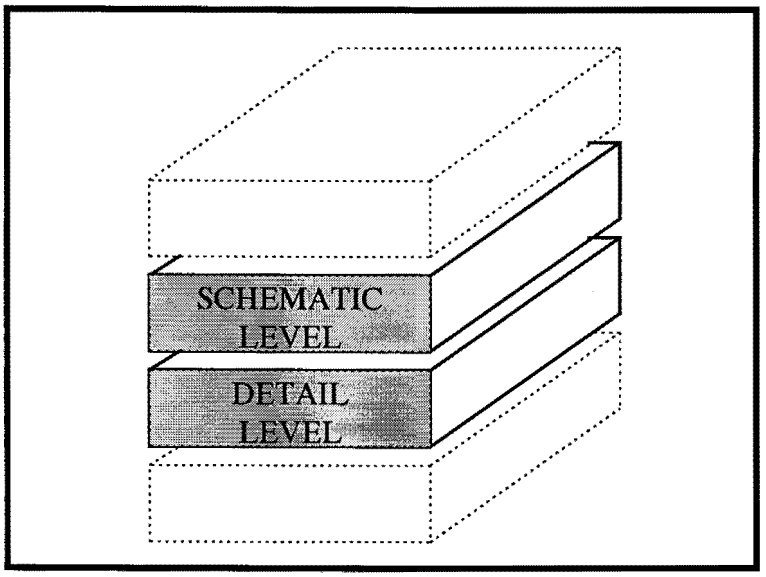


Figure 79. The General Abstraction Levels Paradigm

Despite the separation of concerns, the concepts at different abstraction levels in ROOM are still formally interrelated. The concern, here however is that there will be no continuity in terms of the software life cycle from design to implementation if the Detail and Schematic Levels are handled by different languages. This is in essence defeating the objective of developing a SEE.

- ROOM is very strong in using graphical representation and one of the traditional shortcomings of it is its impracticality for capturing detail. A graphic that is loaded with minutiae loses its synthetic quality and becomes as difficult to digest as an equivalent textual version, Selic[1994]. State transition diagrams are used a great deal in ROOM which are used as graphical renderings of state machines. The main

value of the graphical notation is that it provides a compact view of all possible behaviours of an object. Ironically, this is also one of the main weaknesses of the state machine models. When the state machine is viewed as a directed graph, there are usually many valid traversal paths through the graph (if the graph is cyclic as in Figure 80, then the number of traversals can be infinite.) In most systems however not all traversals are equally likely. There are so called "main paths" through a state diagram that are taken in the great majority of cases, while the remaining paths might only be taken in pathological or exceptional situations. The problem is that it is not possible to discern which are the main paths by simple inspection of a state diagram. Thus, to the uninitiated observer, the state transition diagram does not reveal which behaviour is common and which is exceptional.

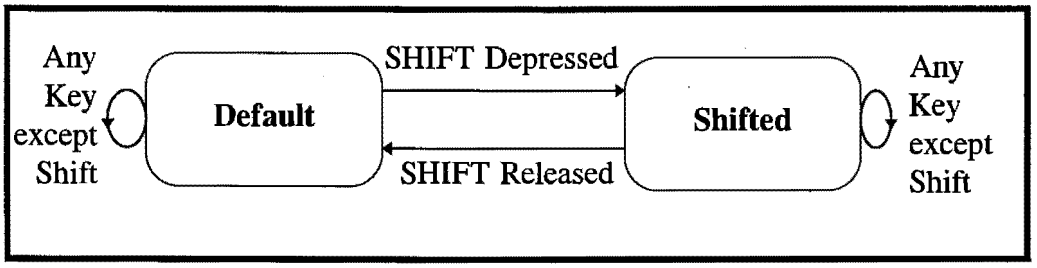


Figure 80. A State Transition Diagram for a Typewriter Keyboard

- The ROOM structural framework places the following constraints on behaviour:

  - ◊ All behaviour is located within the behaviour components of actors.
  - ◊ All communication between actors is achieved by the exchange of messages through end ports, SAPs and SPPs. Note that message based communications does not necessarily imply that the communication model must be asynchronous.

- The behaviour of an actor must conform to the combined set of protocols specifications of its behaviour interface (end ports, SAPs and SPPs). It almost seems as if this process can be automated. Unfortunately no such methods for automatic synthesis exist yet, so there is dependence on the creative abilities of human designers to synthesise the desired behaviour.

- ROOM adopts the run-to-completion semantics in terms of handling activities. In the process control domain as well as in the real-time environment certain activities are critical and could cause serious problems if there is no immediate interrupt in the current "non-critical" activities to handle the crises.

5.2. Suggested Extensions to ROOM

There should be more tools developed that support the ROOM modeling language so that there is greater flexibility and availability of such tools.

A concerted effort should be made in developing tools for the areas of Work Organisation and Modeling Heuristics. There is little documented experience of methodology support in these area in the real-world development situations.

The feasibility of utilising the pre-emptive semantics for handling event processing as compared to run-to-completion should be investigated. In as much as there will probably be a need for greater management of the "interrupted" variables, by handling a crises when it occurs it prevents the disaster from becoming worse. This could introduce complexity into ROOM but the advantages far outweigh the disadvantages as process control systems are "hard" time dependent.

The life-cycle adopted by ROOM does not conform to any of the traditional life cycles. Thus, at this stage it cannot be modeled against the revised spiral life cycle model, as described by du Plessis and van der Walt [1992]. The one aspect of the spiral model that is particularly important and is not apparent in ROOM is risk analysis. Thus, an area for expansion within ROOM is to perhaps incorporate risk analysis in its methodology, typically when the boundary between the system and environment is identified and when actors' behaviours are being modeled. A further suggestion is that various options/strategies should be considered when faced with a

problem statement as proposed by the architecture cycle in the revised spiral life cycle.

5.3. Summary

This dissertation is of limited scope and due to the time frame for the dissertation (limited scope) it was not possible to fully demonstrate ROOM's full functionality or to develop a prototype. The application was confined to a rather small example but the intention was to demonstrate the ease of use and ROOM's suitability to the process control domain.

The Target System Reference Model provides a frame of reference for viewing the various aspects (environment, information, systems engineering and software engineering) in support of the real world. The process control domain is by nature complex and it is essential to provide as much assistance as possible in ensuring that the development is painless.

The rest of this section summarises the features of ROOM.

The increasing complexity of large, process control systems impedes efforts to construct, co-ordinate and monitor these systems effectively as described by Goerner[1991]. Developing software paradigms and mechanisms with respect to the individual cohesiveness of subsystems and applications while allowing their effective and graceful extension, integration, synchronisation and control, is a great challenge.

ROOM was developed for such environments and it was designed from the practical experience of the authors and their colleagues in this environment, according to Selic[1994]. The executability of ROOM models has a major repercussion on the development process. In contrast to traditional "phased" models, development with

ROOM can proceed in a steady sequence of executable system models, progressing from the abstract to the detailed, with each model expressed using the same notation as its predecessor (thereby ensuring continuity and correctness). To minimise discontinuities in the process, ROOM also incorporates implementation-level concepts provided by traditional implementation languages. The final result is a high quality implementation that is produced in less time than would have been produced through a traditional method.

ROOM was devised to create effective models of real-time systems and to support common model building strategies by incorporating three key features:

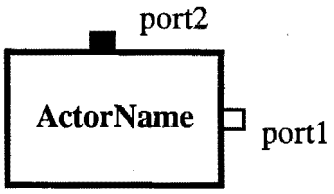
1. executable modeling
2. a single notation used throughout the development process and
3. the object paradigm.

ROOM has greatly increased productivity over traditional modeling approaches Selic[1994]. From the management point of view, this increased productivity is a dividend and the appropriate use of this dividend deserves some thought. While the dividend can be reinvested in the faster, cheaper creation of adequate products, it can also be reinvested in the creation of properly designed (that is evolvable) products. Thus, the productivity improvements made possible through ROOM can be used to redirect the systems development process from a project-oriented style of development to a product-oriented style.

It is important to note that a common theme in the ROOM model development heuristics is balance. This is expressed in the frequent alternation between model construction and model validation. It also applies to the specific approaches to the construction (for example, not modeling the structure too deeply to the neglect of the associated behaviour, or vice versa). A final balance involves the heuristics themselves. Although they point the modeler in the right direction, they will not be optimum for every particular situation. This must be considered and regarded as a general guideline but not a prescription to be followed inflexibly.

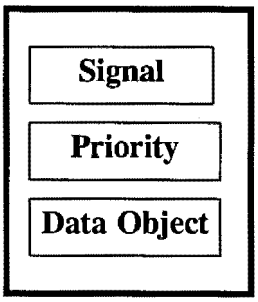
6. *APPENDIX A: LEGEND FOR ROOM*

Actor



The actor name is unique
port1 is a conjugated port
port2 is an unconjugated port
NB: An actor can have multiple ports

Message

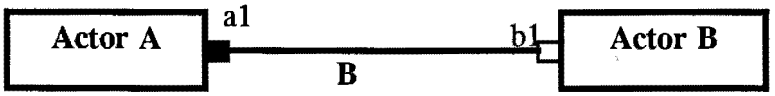


Signal is a symbolic value

Priority is relative to other messages in transit

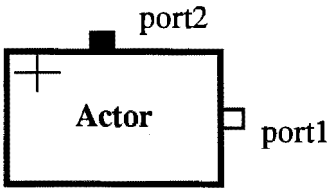
Data object is optional

Binding



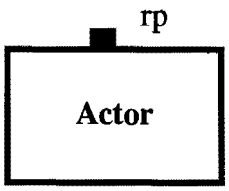
B is a binding, that is a continuous line that joins two points.

Substitution



An actor with a + indicates it can be substituted

Composite Actor



A composite actor represents its components. rp is a relay port found outside the composite actor - used as an interface to other actors

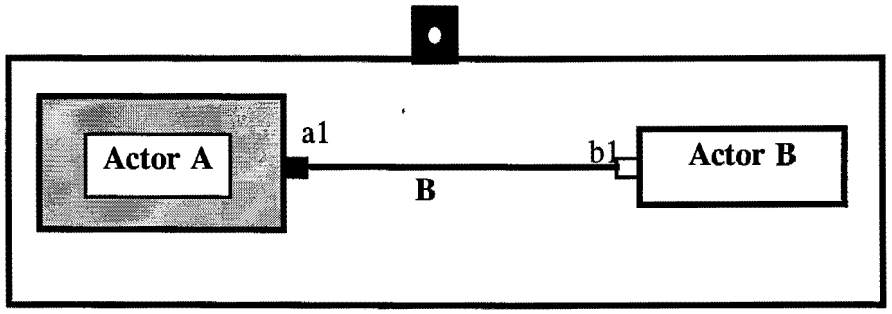
End Ports

conjugated end port

unconjugated port

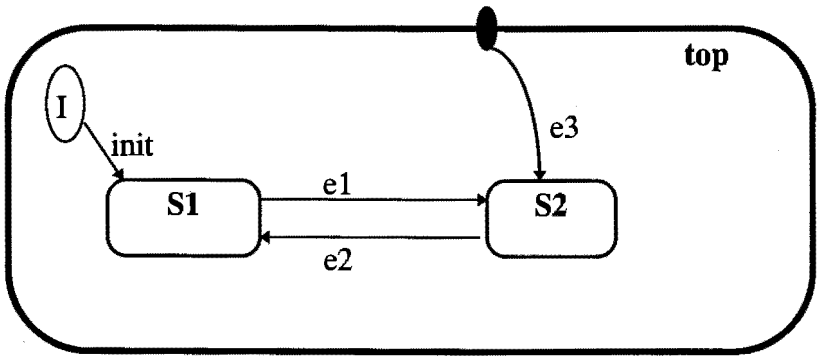
End ports are used as a link between structure and behaviour

Dynamic Actor Relationship



The dynamic actor structure occurs when an existing actor must enter into a relationship with another actor

State Machine



I is the initial state
Init, e1 and e2 are triggers that cause a state change from the current state to the next
e3 is a group transition and effects a state change regardless of what the current state is
S1 and S2 are states

7. APPENDIX B: EXAMPLES

A number of organisations are in the process of implementing OO in their process control systems. Some of the development is for in-house systems while others are for market sales.

The purpose of this section is to inform the reader of such systems. Most of these systems were encountered when the literature survey for this dissertation was conducted. A brief description of a few systems is presented.

7.1. CEBAF Data Acquisition System

The Continuous Electron Beam Accelerator Facility (CEBAF), is an electron accelerator currently under construction as proposed by Quarrie[1992]. The CEBAF data acquisition system is being developed in all experimental halls for systems with a large range of event sizes and data rates.

A decision was made to use OO techniques in developing the control software for the systems. After an intensive literature search the Eiffel language and development environment was chosen as the vehicle for the development.

Eiffel is a pure OO language. It is targeted at large software engineering projects requiring a high degree of robustness and low maintenance costs.

Major features of Eiffel are :

- A small language having a similar number of reserved words to Pascal.
- Both single and multiple inheritance.
- Static typing combined with dynamic binding.
- Deferred classes to specify behaviour without implementing it.

- Efficient handling of basic objects (INTEGER, REAL) and complex objects.
- Exception handling.
- Assertions (pre and post conditions).
- Automatic garbage collection.
- An extensive class library.

This covers many data structures such as hash tables and linked lists, together with graphics classes and classes for lexical analysis, as well as the availability on a large number of mainframe platforms and recently PC platforms. There are also tools for automatic generation of documentation from the source code and to aid in the design process.

7.1.1. Evaluation

Several important deficiencies in the supplied Eiffel graphical library classes were discovered, many of which could be solved using inheritance, but others, including the lack of figure dynamics such as dragging, could only be solved by direct modification of the supplied source code. Similarly, generation of PostScript code to enable printout of the created diagrams was also lacking.

The most severe problem during evaluation was that were sometimes there was a long turn around time for an edit/compile/link/run cycle and the lack of a true symbolic debugger, although the ability to inherit from the VIEWER class alleviated the latter to a certain extent. However, many of the programming bugs that would have only been found at run time with most conventional languages were caught either by the compiler or through the use of assertions. Assertions proved to be quite expensive in terms of performance, but the facilities provided for enabling or disabling them were easy to use.

The major drawbacks are being addressed by the next release of the compiler and it proved to be quite easy to implement a Motif class library. Furthermore, it was felt that the resulting programs were significantly more robust and maintainable than similar programs written in conventional languages such as C or FORTRAN. Additionally, these programs were easier to modify and enhance to cope with changing requirements than similar conventional programs. Finally, in most instances where a direct comparison was made, the number of lines of code required to implement a program using Eiffel was significantly less than the equivalent using C or FORTRAN. In combination these indicated that a useful gain in productivity was in fact possible.

7.2. An OO Operator's Interface for Real Time Process Control Expert Systems

As expert system technology evolved into a proven approach for extracting crucial information from today's deluge of data, the data rich domain of real-time process control has emerged as an obvious area of great potential benefit according to Adams[1992]. In evaluating commercially available expert system packages it was found to be unacceptable because they failed to provide functionality, which is crucial to the process control domain. A new expert system shell was therefore developed that traded off generality for power to the advantage of those specifically interested in the domain of real-time process control. The goal of this shell is to pave the way for creating real-time advisory expert systems to help operators do a better job of controlling complex processes. Expert system applications in a process control environment should be presented in a manner consistent with the existing applications Adams[1992]. An integrated interface is well positioned to provide the same man-machine interfacing techniques (buttons, lights, touch targets, etc.) that the operator has already mastered.

7.2.1. Overview of Functionality

There are two primary modules in the shell, the Operator's Interface (OI) and the Runtime module. The OI exists to get information from the operator back to the Runtime module. When new information is available, i.e., some new problem is discovered or new advisory or query is available to the operator, the Runtime module makes this information available to the OI's "display data base." This display data base is an area of common memory in which all currently active process situations, advisories, queries and other variables of interest are located. At this time, if the OI is not currently being displayed, a message is sent by the Runtime module to the control system's standard message facility requesting that the operator invoke the expert system interface for this application.

7.2.2. OI's OO Design and Development

The OI was designed and developed as an OO system, using Stepstone, Inc.'s Objective-C language on a Digital Equipment's' VAX/VMS computer. Because the components of on interface screen can be easily though of as objects, the OO paradigm proved very effective for this application. There is a smooth, simple mapping leading to a surprisingly robust, flexible and maintainable application.

The OI takes advantage of the structural and functional inheritance capabilities of the OO approach. When different kinds of objects share a certain level of structure and functionality, they are subclassed from a common class to consolidate and simplify the design and code. Each situation, advisory, query, scroll arrow, etc. on the OI is represented in code as an Objective-C object.

7.2.3. Summary

The OO paradigm proved highly successful for the OI because objects on the screen are so easily mapped to objects in code. Using this approach, a clear and sensible

design practically "falls out" from the requirements. Maintenance and enhancement efforts are simplified because the code architecture is as easily remembered as the screen layout. For those in a position to focus on the domain of real time process control, especially on a particular control system, there are tremendous benefits in terms of the powerful tools that can be designed for expert system development and deployment. Integrating an expert system OI's into the control system itself represents a positive evolution of the entire control system - one that both operators and management can feel confident about. The advantages of using the existing interfacing hardware and protocol help ensure an orderly and successful introduction of expert system technology into the control room for daily use.

7.3. Adroit™

With Adroit™, a Windows NT®, SCADA package, the concept of intelligent objects has been applied to a process control/SCADA context according to le Roux[1995]. Instead of containing lists of relatively unintelligent records to represent the tags in a SCADA system, Adroit™ uses intelligent objects known as agents, that embody the data as well as the rules for operating on the data.

The analogue agent type, for example has built-in knowledge about transforming raw plant values into scaled engineering ranges, performing high, low and rate of change alarm checks, etc. On the other hand, the expression agent types are able to perform sophisticated mathematical and logic calculations on real numbers. There is an ever-growing list - currently a dozen or more - of distinct agent types in ADROIT™, each of which encapsulates a unique part of a process control/SCADA application. The real user benefits of Adroit™, aside from the obvious ones like ease of use, open system interconnectivty, scaleability, performance, robustness, etc., however lie in the fact that it runs on Microsoft's strategic 32-bit Windows NT® platform.

Adroit™ is the trademark for Adroit Technologies

7.4. Smalltalk

Long viewed as a language for academics, Smalltalk is quickly proving to be a leading OO language for mainstream Manufacturing Information Systems departments Skerret[1993].

Texas Instruments have announced a Smalltalk-based product for developing process control systems and a number of telecommunication companies are using Smalltalk to build their next generation of applications. All of these companies are proving that Smalltalk is a successful development environment for mission critical applications.

7.5. PrintFlow

A UK company, Sentata, developed a process control software system, called PrintFlow Seybold[1993]. It uses an OO database custom developed by Sentata. According to them, they chose to develop their own database because they felt that the standard RDMS with SQL was too limited for the application. Printflow can be used for scheduling and managing all the operations of a printing plant. It can also be used for other applications such as workflow tracking in a newspaper environment. It is a client-server application.

8. REFERENCES

8.1. Books

Booch G: Object-Oriented Design with Applications. Benjamin/Cummings. 1991.

Brown AW, Earl AN & McDermid JA: Software Engineering Environments. McGraw Hill,1992.

Rumbaugh J, Blaha M, Premerlani W, Eddy F & Lorenson W: Object-Oriented Modeling and Design. Prentice Hall, 1991.

Selic B, Gullekson G & Ward PT: Real-Time Object-Oriented Modeling. John Wiley & Sons, 1994.

Schach SR: Software Engineering. First Edition. Aksen Associates Incorporated Publishers, 1990.

Son SH: Advances in Real-Time Systems. Prentice Hall, 1995.

Taylor DA: OO Technology: A Manager's Guide. Addison-Wesley Publishing Company, August 1994.

Ward PT & Mellor SJ: Structured Development for Real-Time Systems. Volume 1: Introduction and Tools, Yourdon Press, 1985.

Winblad A L, Edwards S D & King D R: Object-Oriented Software. Addison-Wesley Publishing Co. Inc., 1990.

8.2. *Articles*

Acuna A: Plant software: your questions answered. SA Instrumentation and Control, September 1995, Vol. 11, No. 9, pp. 55-54.

Adams J: An Object-Oriented Operator's Interface for Real- Time Process Control Expert Systems. SA Transaction, 1992, Vol. 31, Iss. 3, pp. 65-74.

Aschmann HR, Giger N, Hoepli E, Janak P & Kirrmann H: Alphorn: A Remote Procedure Call Environment for fault tolerant, heterogeneous, distributed systems. IEEE Journal, Oct. 1991, Vol. 11, Iss. 5, pp. 16-19, 60-67.

Beam K: Object Technology Object-Oriented Programming. I/S Analyzer, December 1993, Vol. 31, Iss. 12, pp. 1 - 15.

Bornman C & Du Plessis A L: Constructing Software Engineering Environments using the Systems Encyclopedia Manager System. Information Systems in Practice and Theory, 1988, pp. 220-235.

Carter E: DCS/PLC/SCADA. Pulse, August 1994, pp. 16.

Du Plessis AL & van der Walt E: A Revised Spiral Model for Object-Oriented Development. 1992.

Gilbert III JW & Wilhelm, Jr. RG: A Concurrent Object Model for an Industrial Process-Control Application. Journal of Object-Oriented Programming, November/December 1993, Vol. 6, No. 7, pp. 35-51.

Ghosh A & Rio R: Programming with Modules. Chemical Engineering, June 1991. pp. 82-93.

Goerner AA & Hines ML: Instrumenting Intelligent Manufacturing Applications: An Object-Oriented Approach using Probes and Generalised Daemons. University of Tennessee Space Inst. 1991, Vol. 2, pp. 192-199.

Gunderson D: LANs and PC Based Data Acquisition. SA Instrumentation and Control, July 1995, Vol. 11, No. 7, pp. 16-20.

Harel D: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, July 1987, pp. 231-274.

Hayes F & Coleman D: Coherent Models for Object-Oriented Analysis. OOPSLA, 1991, pp. 171-183.

Hill R & McKinnon M: Object-Oriented Programming in Process Control. Elektron, August 1993, Part 8, pp. 29-39.

Le Roux D: SCADA based on Win NT and Object. SA Instrumentation and Control, August 1995, Vol. 11, No. 8, pp. 41.

Meyer F: Object Based Systems Do Windows-Better. Intech, July 1993, Vol. 40, Part 7, pp. 39-41.

Pirklbauer K, Plosch R & Weinreich R: Object-oriented Process Control Software. Journal of Object-Oriented Programming, May 1994, Vol. 7, No. 2, pp. 30-67.

Prekel T: Seminar on Advanced Skills for Women Managers. Graduate School of Business, UNISA, 1995.

Quarrie DR, Heyes G, Jastrzembski E & Watson WA: Object-Oriented Run Control for the CEBAF Data acquisition. IEEE Journal - Transactions on Nuclear Science, April 1992, Vol. 39, Iss. 2, part 1, pp. 115-120.

Reid, G: Reuse Requires Removal of Cultural Barriers. Computing Canada, 27 April 1994, v20, n9, p23.

Roedner D: Client/Server and Object-Oriented Programming. The Computer Conference Analysis Newsletter, 25 July 1994, pp. 346-347.

Seybold: PrintFlow process control from Sentata. The Seybold Report on Publishing Systems, November 1993, Vol. 23, No. 5, pp. 33.

Skerrett I: Smalltalk makes its Mark. Computing Canada, 6 December 1993, Vol. 19, No. 25, pp. 45.

Steenkamp A L: Object-Oriented Information Systems Engineering Environment. May 1995, pp. 1-20.

Strydom, DB: Supervisory Control Systems: An Introduction and Overview of Recent Developments. SAIMM School: Process simulation, Control and Optimisation - 1993. pp. 1-8.