CONSTRUCTION OF A SUPPORT TOOL FOR THE DESIGN OF THE ACTIVITY STRUCTURES BASED COMPUTER SYSTEM ARCHITECTURES

By

Sabah Mohamad Amin MOHAMAD,
B.Sc, M.Sc., Dip.Comp, AMBCS, MACM

Thesis submitted in fulfillment of the requirements of the degree of Doctor of Philosophy in Computer Science.

Department of Computer Science,
BRUNEL The University of West London,
Uxibridge, Middlesex,
ENGLAND
1986

To the Lord for his guidance,
To the memory of my father, for his efforts and sacrifices,
To my wife for here patience, encouragement and love,
To my mother who taught me the love of knowledge,
To my brothers and sisters.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to

Dr. Ladislav J. Kohout

for his time spend in guiding me in this research and for demonstrating that cooperative research can be so enjoyable. Without his unfailing energy and enthusiasm, I would not be able to complete this thesis.

I wish to thank

Dr. C. F. Reynolds

for his most helpful comments and encouragement.

I would like also to thank *Professor W. Bandler* of Florida State University Computer Science Department for his interest in my thesis.

Special thanks to my friends Dr. M. Ohiorenoya of Benn University, Institute of Computer Science, Mr. H.T. George of Brunel University, Electrical Engineering Department, and Professor J.C. Cavoras of Glasgow University for their helpful discussions and encouragement.

Finally, I would like to express my sincere gratitude for the financial support of the Iraqi Government for the period of this research.

The typesetting, formatting and art-work of this thesis have been prepared with LaTeX the document preparation system running on an IBM PC/XT and was produced on an IMAGEN Imagestation laser printer.

LaTeX is a trademark of Addison-Wesley Publishing Company, TeX is a trademark of the American Mathematical Society, IBM PC/XT is a trademark of International Business Machine Corporation, IMAGEN is a tradmark of Imagen Corporation.

Brunel University, Department of Computer Science, S.M.A. Mohamad, "Construction of A Support Tool for The Design of Activity Structures Based Computer System Architectures".

ABSTRACT

This thesis is a reapproachment of diverse design concepts, brought to bear upon the computer system engineering problem of identification and control of highly constrained multiprocessing (HCM) computer machines. It contributes to the area of meta/general systems methodology, and brings a new insight into the design formalisms, and results afforded by bringing together various design concepts that can be used for the construction of highly constrained computer system architectures.

A unique point of view is taken by assuming the process of identification and control of HCM computer systems to be the process generated by the Activity Structures Methodology (ASM).

The research in ASM has emerged from the Neuroscience research, aiming at providing the techniques for combining the diverse knowledge sources that capture the 'deep knowledge' of this application field in an effective formal and computer representable form. To apply the ASM design guidelines in the realm of the distributed computer system design, we provide new design definitions for the identification and control of such machines in terms of realisations. These realisation definitions characterise the various classes of the identification and control problem. The classes covered consist of

1. the identification of the designer activities,
2. the identification and control of the machine's distributed structures of behaviour,
3. the identification and control of the conversational environment activities (i.e. the randomised/adaptive activities and interactions of both the user and the machine environments),
4. the identification and control of the substrata needed for the realisation of the machine, and
5. the identification of the admissible design data, both user-oriented and machine-oriented, that can force the conversational environment to act in a self-regulating manner.

All extent results are considered in this context, allowing the development of both necessary conditions for machine identification in terms of their distributed behaviours as well as the substrata structures of the unknown machine and sufficient conditions in terms of experiments on the unknown machine to achieve the self-regulation behaviour.

We provide a detailed description of the design and implementation of the support software tool which can be used for aiding the process of constructing effective, HCM computer systems, based on various classes of identification and control. The design data of a highly constrained system, the NUKE, are used to verify the tool logic as well as the various identification and control procedures. Possible extensions as well as future work implied by the results are considered.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation

The human-computer interface is becoming the major component of the success or failure of computer systems. Its improvement is an objective for the fifth-generation (Treleaven and Lima 1982) and the sixth-generation (Gaines and Shaw 1986, 1986a) computing development programmes. Since the start of these programmes, there has been a growing acceptance of opinion in the computer science community that the traditional sequential control flow fourth-generation computers will be superseded in next decade by a new generation of general-purpose computers (see Table 1.1).

There are many factors supporting the adoption of a radically new generation of general-purpose computers. Firstly, the computional requirement is changing from a sequential and centralised type to a parallel and distributed type. Secondly, the processing tasks performed by the computers are becoming more "intelligent", moving from scientific calculations and data processing to artificial intelligence.

However, today's fourth generation computers are still based on the old von Neumann architecture; all that has happened during this period is that the software systems have been repeatedly extended to cope with the increasingly sophisticated applications.

To overcome the above limitation we had to consider the Human-Computer interaction in the case of the computer design. We are posed with the question,

Can we achieve a relativistic synthesis in the field of computer systems design

Generations	*Example*	*Bare H. W.*	*Functional Features*	*Problems*
FIRST 1946-1956	UNIVAC I	Valves	Bootstrap Software, Machine Code Software	No resource sharing, User could destroy the bootstrap Software, Bad performance (Speed: 10 kips; Space: 1 kbyte)
SECOND 1957-1963	IBM 7000	Transistors	Monitor Software, Logical I/O devices, library functions, user at a time, overlayed memory, H.L. Languages	Poor performance (200 Kips; 32 kbyte), Monoprogamming, No protection assumed
THIRD 1964-1981	IBM 360	IC	Operating System, Multiprogramming, Protection, Timesharing	Low performance (5 Mips; 2 Bbyte),
FOURTH 1982-NOW	IBM 370	VLSI	Concurrency,	Low reliability Good performance unless protection is enforced (50 Mips; 8 Mbyte)
FIFTH Proposed 1979	Not Implemented	ULSI	Functional Programming, Natural Languages, Speech, Vision,	Empirical Models of Human-Computer Interaction(HCI)
SIXTH Propossed 1986	Not Implemented	ULSI	Better design using theoretical models of HCI from diverse fields neurology, psychology, linguistics, cognitive science, system science	No apparent problems

Table 1.1: The six generations of computer systems

via the paradigm of general or meta systems methodologies?

This was the point of departure for several researchers of the development of effective foundations of research in this area (notably *Antonin Svoboda, Brian Gaines, Ladislav Kohout and Wyllis Bandler*) which led me to develop a new approach for designing general/meta systems-oriented computer models. The fundamental element of these theories is the triplet

⟨SYSTEM MODEL, DESIGN PROBLEM, DESIGNER⟩

The essential ideas that have been used in the thesis for the construction of computer system models, which can include both the computer user behaviours and the designer activities, were given by **Ladislav Kohout** in a series of research papers (starting in 1974 and still evolving, e.g. Kohout 1986) establishing a methodological approach for the study of actions of natural and artificial systems (named in 1979 as the *activity structures methodology*).

Historically, the research in Activity Structures methodology emerged from Neuroscience research, aiming at providing techniques for capturing the diverse knowledge sources and can be used to model/analyse the "deep knowledge" of this application field in an effective formal and/or computer representable form.

The view in this thesis is that *activity structures framework* is an approach which can be used to construct *total*[1] information processing systems which support the *self-regulating*[2] architectures whose processing environments can be dynamic and operate under *maximal constraints*[3]. Note that this definition is not identical to the original definition of Kohout:

> "The design framework of the Activity Structures provides the means for identifying the necessary processing environments. It also provides the structures, as well as their linkage interfaces, that are seen to be essential for supporting a successful

[1] A **total system** can be described as a finite set of rules which integrates the design and constraint **functional structures** with the **substratum structures**. The **functional structures** represent coherent goal-oriented segments of behaviour. Each functional structure identifies a specialised domain of the total system activities. These can be classified into primary structures of activities (**design modules/structures**) identifying the essential activities of the primitive design, and the secondary structures of activities (**constraint modules/structures**) identifying the constraints imposed on the primary activities c.f. Kohout 1986). The **substratum structures** represent fragments of the code and data used for the realisation/simulation of the functional structures (c.f. Kohout 1986).

[2] **Self-regulating** is the behaviour which reaches a **stable condition of satisfactoriness**. The machine environment may reach a stable condition of satisfactoriness, when initially the performance of the machine environment oscilates (because of the interaction with the user environment) between the acceptable and non-acceptable, with respect to some predefined thresholds, but it must finally become acceptable and remain so. In this condition we may say that the machine environment has become "adapted" to the user environment activities. The user environment may aslo behave in a self-regulating manner. In a similar way we may define the user environment self-regulation.

[3] In a **maximally constrained system** the interaction between the user environment and the machine environment is restricted to one degree of freedom only, i.e. only the cooperation behaviour is allowed between these two environments. The reason for such a restriction is given by the fact that with cooperation, the self-regulating behaviour can be ensured simultaneously within both environments. This is in contrast to the effects of the competion behaviour in which only one environment may behave in a self-regulating manner. However, a distiction should be made between a maximally constrained system in which cooperation behaviour is enforced at all times, and a **highly constrained** system; the type of interactions at the initial stage do not matter and only after the system leaves the initial stage the interaction must be forced to be of the cooperation type. A slightly different definition of the criterion of maximal constraints is given by Rosen (1986).

> design" (Kohout 1984).

However,

> "the growing area that concerns the design of total systems still needs a broader and formal foundation, more automation, and a higher level of integration into the overall design and realisation process." (Roman et al 1984)

The purpose of this thesis is to introduce an effective approach for the construction of total computer systems based on the activity structures methodology. The reasons for focusing our attention on achieving this purpose came from the examination of current experience in the design of total computer systems operating under maximal constraints. A summary of results is given below:

1. Constraint structures enforcement is by no means an inconsequential problem. Constraint structures have become an important and challenging goal in the design of computer systems. Constaint structures comprise engineering and managment activities imposed on the functionalities of the basic design of a computer system. For example the enforcement of a protection constraint structure is legitimately of major concern.

   > "At present the effecient implementation of protection system presents difficulties and further research is called for." (Wilkes 1984)

2. There is no successful highly constrained computer system.

   > "To date, most systems designed to include constraints such as protection in computer systems have exhibited either slow response times or awkward user interface or both." (Landwehr 1981)

   > "Although variety of improvements have been introduced to the structure of certain advanced computer systems (e.g. by using parallel architectures), the resulting systems have suffered from complexity in both the use and understanding." (Bic 1982)

3. A total system design framework is needed to overcome the system inadequacies that might be the fault of the designers who did not look at the whole picture including the constraint structures at each design stage.

   "To be successful, a design must merge hardware and software functionalities into a single, unified perspective." (Roman et al 1984).

4. Not every design framework can be effective.

   "The form of the design framework representation may affect the various design tasks." (Gaines 1974)

5. Design must start right from the first steps.

   "Many computer systems (e.g. B6700, ROLM 1664) failed to meet the user requirements owing to the inadequacy of the original system representation." (Dietz and Szewerenko 1979)

6. Design must not concentrate only on the actual products but it should consider the human factors.

   "An adequate design methodology for designing computer systems should capture the basic design trajectory, involving the three design relationship between computer, problem, and designer." (Gaines 1973)

Figure 1.1 illustrates Gaines design triangle.

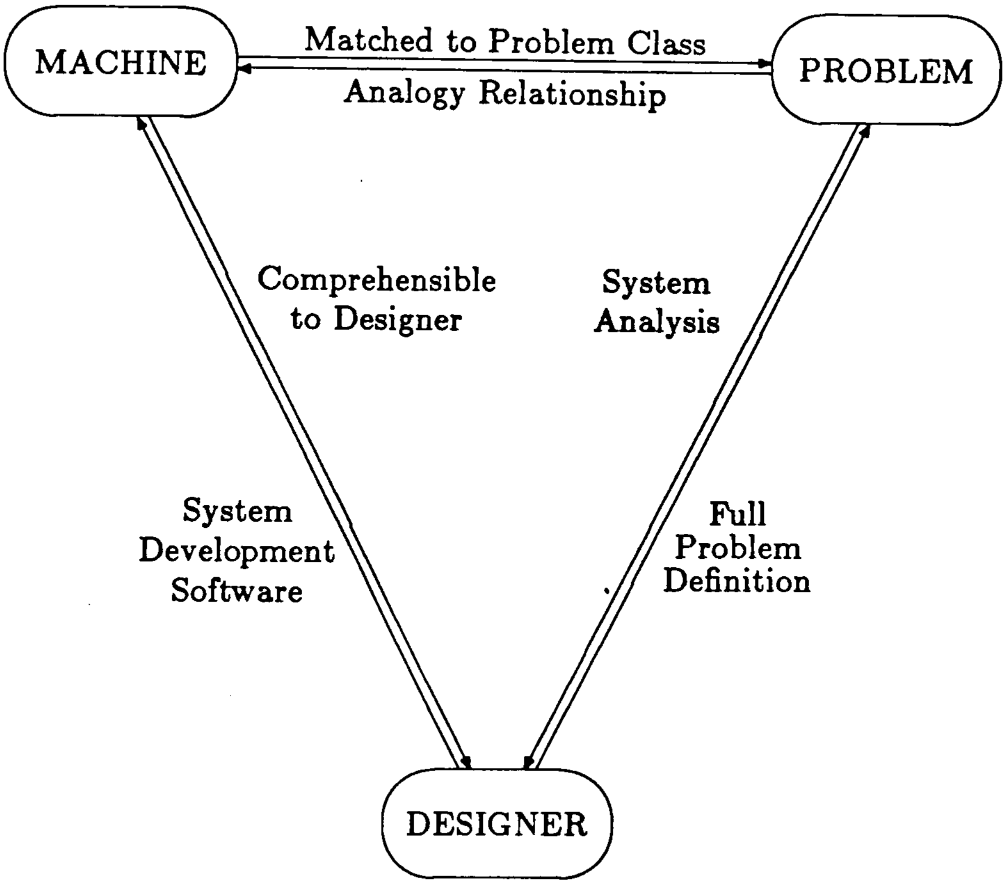


Figure 1.1: The essential elements of successful computer design methodologies.

7. To delegate the design activities in 6,

> "numerous schemes must be formalised first; to a considerable extent we must formalise those human activities that contribute to the design." (Mills 1985)

Such efforts, however, have been largely fragmented within the *science paradigm*[4]

> "there has been little discourse between mainstream computer scientists, AI

[4] The **science paradigm** can be described as a 'learning system' characterised by reductionism, repeatability and refutation (Checkland 1976).

> theorists, system developers, communication engineers, and so on. A general theory of design, and its relationship to human knowledge and activity, would help us to relate these theoretical fragments and to judge their worth, but no such coherent methodology or theory has emerged." (Tully 1985)

Capper (1986) expressed similar opinion.

8. In contrast to 7, the success of applying the activity structures approach which is based on principles similar to the more present *general/meta systems paradigms*[5] [6]), motivates me to use and develop the activity structures approach in the realm of effective computer systems design.

Hence, in this research project, we have developed a software tool to simulate many activity structures based computer systems. This is a detailed simulation which replicates the essential capabilities of modern computer systems and provides those activity structures mechanisms that are required for self-regulation. This tool supports a methodological approach for designing maximally constrained, high-performance computer systems that provide elegant solutions to several problems which previous attempts have handled only in an ad hoc fashion.

Indeed, in implementing the support tool for activity structures based computer systems, the aim is not only to show the applicability of the method presented. We also have to demonstrate the feasibility of writing a support tool for such systems in a high-level language, to develop a comprehensive and effective methodological approach to the construction of computer system architectures, and devise some techniques for aiding the process of exploring and evaluating these architectures.

The software tool described in this thesis consists of the following major components:

[5] The **general systems paradigm** takes into account the indivisibility of systems domains where organised complexity prevails. It originates from concerns that the science paradigm, which was designed to deal with the physical world, breaks down when faced with living systems (Gigch 1979). However, the **metasystem paradigm** originates in the premise that one cannot arbitrate deficiencies among systems in other than a meta language that is in the language of a metasystem which lies above that of the systems whose is sought (Gigch 1979).

[6] Examples of such sucesses in the construction of sophisticated information processing systems include expert systems (Kohout and Bandler 1982, Kohout et al 1984, Mohamad et al 1983), a decision support system (Kohout et al 1985, Ohiorenoya and Mohamad 1983).

1. **preprocessor** for eliciting the design information from the designers wishing to construct activity structures based computer architectures,
2. subsystem simulating the user intentions and his/her learning capabilities for the purpose of **generating the user interaction environment.**
3. a subsystem simulating the machine activities (i.e. traps and interrupts) and learning, for the purpose of **generating the machine environment.**
4. a highly parameterised shell which provides the designer with the essential design and functionality constraint modules (called the **functional structures**). These modules are linked in a distributed fashion (i.e. message-passing) and consist of:
   (a) *Design Modules* : These represent the essential functional structures needed for producing the primary activities of the computer system. These are:
      i. Knowledge representation structures,
      ii. Inferential structures,
      iii. Control Structures,
   (b) *Constraint Modules* : These represent the secondary activities imposed on the primary activities. These consist of the following functional structures:
      i. Protection structures,
      ii. Communication structures,
      iii. Interpretive structures.

The structures (4(a)i, 4(a)ii, 4(a)iii and 4(b)i) were introduced and used by Kohout (c.f. Kohout 1986, Kohout and Bandler 1986) in the designs based on his activity structures framework; communication structures he used only in the context of protection, and I extend their use in this thesis to other domains. Interpretive structures constitute my original contribution to the Activity structures framework.

The flexibility of the proposed tool allows the designer to explore the variations of any given design by direct experimentation, in order to force it to behave in an interesting way (i.e. to have high-performance).

1.2 Terminology and Definitions

This section contains the essential definitions used later in the text. The reader is also directed to the index.

- **ACTIVITY STRUCTURES METHODOLOGY:** The *activities of the designer* that are essential for the construction of *activity structures based computer systems.*
- **THE DESIGNER ACTIVITIES:** The process of dealing with the problem of the design and construction of computer systems. This process involves four design steps: *selection*, *decomposition*, *realisation*, and *exploration* (see section 4.6).
- **ACTIVITY STRUCTURES BASED MODEL OF A COMPUTER SYSTEM:** A *total* model of a computer system which incorperates the user and the machine environments (c.f. footnote number 1 of page 3 of this chapter). In this model the interaction between the *user environment* and,the *machine environment* is *maximally constrained.* The implementation of this model is called the *complete shell.*
- **MAXIMALLY CONSTRAINED BEHAVIOUR:** The *cooperation behaviour* between the *user environment* and the *machine environment* which is restricted to one degree of the freedom of interaction.
- COOPERATION BEHAVIOUR The interaction which reaches a *stable condition of satisfactoriness.*
- USER ENVIRONMENT A *user oriented possibilistic automaton* that updates its demand probabilities on the basis of the resulting machine environment *performance,* so that it chooses asymptotically the optimal *demand.* The implementation of this environment is called the *outer shell.* The updating criterion is referred to as the user environment self-regulation (c.f. foot note number 2 of page 3 of this chapter).
- **MACHINE ENVIRONMENT:** A *machine oriented possibilistic automaton* that controls its performance on the basis of the required user environment demand, so that it asymptotically reaches the optimal performance. Its implementation is called

the *inner shell.* The control criterion and in this context is referred to as the machine environment self-regulation (c.f. footnote number 2 of page 3 of this chapter.

- **USER POSSIBILISTIC AUTOMATON:** A highly-parameterised *user model* stochastic automaton which generates the user demands and is connected in a feedback loop with the machine environment.

- **MACHINE POSSIBILISTIC AUTOMATON:** A highly-parameterised and extensible *computer system model*stochastic automaton which is connected in a feedback loop with the user environment.

- **USER MODEL:** This is a statistical model which represents a quadruple $\langle r, \alpha, p, T \rangle$ where:

r	total number of user demands.
$\alpha = \{\alpha_1, \cdots, \alpha_r\}$	set of demands of the user environment,
$p = [p_1, \cdots, p_r]$	demand probability vector of the user environment,
T	an updating operator

If $\alpha(t)$ represents the demands chosen by the user environment at time t $(t = 0, 1, \cdots)$, then[7]

$$p_i(t) = \Pr\left[\alpha(t) = \alpha_i\right]$$

$$\textstyle\sum_{i=1}^{r} p_i(t) = 1 \qquad \text{for all t}$$

$$P(t+1) = T[P(t), a(t), b(t)]$$

$\beta(t)$ is the binary set of performance reactions from the machine environment the same as input to the user environment such that

$$\beta(t) \in \{0, 1\}$$

- The input to the user environment $\beta(t) = 1$ is called the penalty performance input and $\beta(t) = 0$ the reward performance input. The penalty probability vector

$$c = [c_1, \cdots, c_r] \qquad \text{has the following property}$$

$$c_i = \Pr\left[\beta(t) = 1 \mid \alpha(t) = \alpha_i\right]$$

[7] Note: These probabilities are produced in our implementation by two types of distributions and the user model is simulated by two routines PARTICIPANT-ONE and PARTICIPANT-TWO

The average penalty $M(t)$ that the user environment receives from the machine environment is given by

$$\begin{aligned} M(t) &= E\left[\beta(t) \mid p(t)\right] \\ &= \sum_{i=1}^{r} p_i(t) c_i \end{aligned}$$

The operator T represented by an algorithm called the updating (or user adaptive) algorithm. The user adaptive algorithm can be expressed by the formula

$$M(0) = \frac{1}{r} \sum_{i=1}^{r} c_i$$

such that

$$\lim_{t \to s} E\left[M(t)\right] \leq M(0)$$

where s is the observation time limit.

- **COMPUTER SYSTEM MODEL:** This is a simulation model which is partly driven by itself and partly driven by the outer shell. This model represent a quadruple ⟨ FS, ST, UD, PCR ⟩, where

  FS: FUNCTIONAL STRUCTURES

  ST: SUBSTRATUM STRUCTURES

  UD: USER DEMAND

  PCR: PERFORMANCE AND CHANGES REQUIREMENTS.

- **THE COMPUTER SYSTEM FUNCTIONAL STRUCTURES:** These are coherent goal-oriented segments of behaviour. Each functional structure represents a specialised domain of computer system activities. This leads to communication and exchange of knowledge between the domains. Each functional structure represents an algorithm that updates some activities. In addition to the module that traps the user demand and generates the intention steps (i.e.the external interrupts) and is called the PROCESS GENERATOR, there are other two main types of functional structures used here, the *design structures and the constraint structures.*

- **DESIGN FUNCTIONAL STRUCTURES:** These represent the essential functional structures needed to produce the primary activities of a computer system. These are

  1. the Information Structures (c.f. section 5.3.1),
  2. the Inferential Structures (c.f. section 5.3.2), and
  3. the Control Structures (c.f. section 5.3.3).

- **CONSTRAINT FUNCTIONAL STRUCTURES:** Represent the secondary activities imposed on the primary activities. These consist of the following structures:

  1. the Protection Structures (c.f. section 5.3.4),
  2. the Communication Structures (c.f. section 5.3.5), and
  3. the Interpretive Structures (c.f. section 5.3.6).

- **THE SUBSTRATA STRUCTURES:** These are the fragments of code and data simulating the implementation dependent responses (in absence of the real hardware). These responses in our implementation(i.e. internal interrupts) are generated by the **job scheduler** routine.

- **USER DEMAND:** This is the average user demand, expressed by the number of concurrent tasks or processes placed by the user environment. This measure implicitly identifies the user environment average panalty (demand / number of active terminals).

- **PERFORMANCE AND CHANGE REQUIREMENTS:** These are the measures used to monitor the performance of the modelled computer system model activities via the software probes. There are two types of measures that can be used to assess the average panalty of the machine environmen. These are: the average response time for an interactive system, and the average system throughput for the general system. The change requirements include the designer changes on the computer system model introduced in order to achieve a target computer system. The change that involve the algorithmic *non-parametric* changes on the original computer system model produce a *constelation*. The changes that involve the non-algorithmic changes (i.e.*parametric changes*) on the original model or on a constelation produce a *general system family*. The repeated changes on the general system family produce the *admissible data* that are used to tune the general system to reach the stability state. This tuning procedure is called the *performoact modelling*.

1.3 The Thesis Synopsis

1.3.1 Presentation of the Thesis:

This thesis is concerened with the following topics:

1. outlining the problems encountered in some contemporary computing systems that need solution,

2. presenting a new design method that can deal with the outlined problems,

3. designing and implementing a design support tool based on this new method, and

4. validation of this proposed method.

We concentrate on the design of highly constrained multiprocessing computer systems. The aim is to produce well- protected systems of high-performance, that can achieve stable dynamics of user-computer interaction. A more precise statement of the problem and further discussion of the origin of the problem, its significance and its motivations are discussed in **chapter 1**.

Chapter 2 critically surveys the current design techniques that are available within the three major computer science design fields. Namely, software engineering, knowledge engineering, and computer architecture. **Chapter 3** critically reviews the current theoretical approaches, dealing with computer design, and pinpoints their inadequacies.

Chapter 4 presents the conceptual framework which forms the foundation of the proposed design method. This framework is arrived at by analysing the activities of the designer by the means of Activity Structures approach. **Chapter 5** outlines new essential structures which we need in the design of the complete shell of the tool, the design of which is also derived here. and **Chapter 6** describes the details of the implementation of the design tool. **Chapter 7**presents the so called *performoact modelling*, a new framework for evaluating a computer performance and for selecting those sets of the design data thatforce acceptable performance. Within this framework, the evaluation of the design tool is carried out.

1.3.2 CHAPTER 2:

INVESTIGATING THE EXISTING TECHNIQUES FOR CONSTRUCTING COMPUTER SYSTEMS

In an effort to deal with the problem of designing effective maximally constrainted computer systems, we surveyed the *"computer architecture"*, *"software engineering"*, and *"knowledge engineering"* fields as they were considered to be relevant to the overall problem (see section 2.1). These fields provide the design techniques relevant to the construction of effective complex computer system designs, since they capture the results and accomplishment of research in various parts of the computer science (see sections 2.2, 2.3, and 2.4). In this chapter we investigate the role of each field in constructing effective computer systems and conclude that the techniques provided by these fields are largely fragmented and there is very little discourse between the workers of these three fields. Hence, there is no widely accepted or even practiced methodology that outlines the construction discipline which would link effectively together the methods of these three fields (see sections 2.5 and 2.6).

1.3.3 CHAPTER 3:

AN OVERVIEW OF THE EXISTING THEORETICAL APPROACHES FOR DESIGNING COMPUTER SYSTEMS

In .this chapter we investigate the role of the existing theoretical approaches in the area of designing effective computer systems. We conclude that there is no effective design theory. Indeed, the current conventional building blocks of scientific theory, the theoretical and methodological constructs as well as the mathematical formalisms, have made it difficult to conceive of the possibility of a general account of the nature of computer systems design and its organisation. We have had to rely on the essential notions of physics (time, length, etc.) or else on descriptive analogies taken from ordinary experience in order to describe intelligibly natural phenomena (see section 3.1). The advance of "*statistical*" sciences (operational research, queueing theory, statistical mechanics and operational analysis, mean-value analysis, etc.) have led to the new ways of simplifying complex phenomena within an empirically meaningful framework (see section 3.2). Successes in these areas suggest that behind the formidable complexity of nature there is actually a surprisingly small number of simple relations governing interactions; the difficulty has been to refine several theories to the point where such relations could emerge clearly. Hence, the road to understanding the behaviour and predicting the performance of computer systems has been, and still is, arduous. Sections 3.2.1, 3.2.2, and 3.2.3 review the main theoretical models based on the Queueing theory for designing computer systems.

1.3.4 CHAPTER 4:

ACTIVITY STRUCTURES AS A METHODLOGICAL APPROACH FOR CONSTRUCTING EFFECTIVE COMPUTER SYSTEMS

In this chapter we introduce a methodology approach for designing effective computer systems. This approach is based upon the use of the activity structures design concepts. This methodological approach provides total computer system designs which support self-regulating architectures whose processing environments are dynamic (i.e. changing) and operate under maximal constraints (see sections 4.1- 4.5). Section 4.6 presents the main design steps of the activity structures. Finally section 4.7 presents the main meta-definitions that are used in the process of the design and evaluation of activity structures based computer systems.

1.3.5 CHAPTER 5:

AN ABSTRACT COMPLETE SHELL FOR THE ACTIVITY STRUCTURES BASED COMPUTER SYSTEM DESIGNS

The design issues are discussed here which lead to the subsequent abstract description of an activity structures based complete shell of the support tool. The shell design represents a conceptual model, from which a variety of computer systems can be built. In section 5.2 the cooperation activities of both, the outer shell concerning the user shell, and the inner shell concerning the computer machine environment are described. In section 5.3 we present the abstract design features of the inner shell using the notion of the functional structures. Section 5.4 presents the main performance probes. finally, section 5.5 discusses the problem of selecting a suitable programming language for the implementation of the complete shell.

1.3.6 CHAPTER 6:

THE IMPLEMENTATION DETAILS OF THE SIMULATION OF AN ACTIVITY STRUCTURES BASED POSSIBILISTIC GENERATOR

In this chapter, we present the implementation details of the internal structure of the activity based complete shell that was outlined in chapter 5. The complete shell can be used for generating extensible (i.e. possibilistic) computer systems. Sections 6.1-6.3 and part of section 6.4 present the first three design steps of the designer. In section 6.4.1 the generation of the cooperation environment activities are presented. Section 6.5 describes the implementation details of the various functional structures used in the design.

1.3.7 CHAPTER 7:

EXPLORING THE DYNAMIC BEHAVIOUR OF THE ACTIVITY STRUCTURES BASED POSSIBILISTIC GENERATOR OF COMPUTER SYSTEMS

In this chapter, we develop a special theoretical framework, the *performoact*, which captures the trends of behaviour of interest to the designer and selects those admissible trends that can be used to tune the activity structures based designs. This framework helps to preserve two important criteria, the self- regulation and concurrency (see sections 7.2 and 7.3). The parameters (of a similar, in substrata) of a highly constrained system, the Nuke, are assigned to the possibilistic generator in order to verify the activities of the generator (see section 7.4). The verification process is used to carry out several case studies in order to analyse the effects of various changes within the user interaction environment, the changes within the machine environment, different addressing policies, and many other parameters having influence upon the performance indices (see section 7.6). The purpose of establishing these case studies is to understand the contribution of those design parameters and functionality changes that are responsible for producing effective computer systems. Finally, the chapter discusses the validation process of the possibilistic generator (section 7.7).

1.3.8 CHAPTER 8:

CONCLUSIONS AND FUTURE RESEARCH

In this chapter we summarise the main research contributions of this thesis and discuss the new research problems suggested by this work (see section 8.1). Suggestions for expansion of this work were presented (see section 8.2).

Chapter 2

AN OVERVIEW OF THE EXISTING TECHNIQUES FOR CONSTRUCTING HIGHLY CONSTRAINED COMPUTER SYSTEMS WITHIN THE SCIENCE PARADIGM

2.1 General Discussion

This chapter presents an exposition of current problems and issues associated with design, development and evaluation of highly constrained computer systems. This strives at:

1. Outlining the particular difficulties associated with the application of the existing techniques of the science paradigm to the problems in which their solutions are sought in the subsequent chapters.

2. Outlining the major areas of concern of which one should be aware of in the construction of highly constrained computer systems.

3. Establishing a certain outlook, or an overview, towards the phenomenon of protection; one of the major constraints required for constructing the highly constrained computer systems.

Since the early days of the computer industry, there has been considerable interest in the construction, and performance evaluation, of computer systems. The most common goal has been obtaining better insight into their behaviour and improving their performance.

> "During the last decade, we have seen the development of a large number of computer systems. In most cases, these systems have failed to meet [1] the performance objectives predicted during the initial design. During the same period, the complexity of these systems has increased tremendously with the introduction of multiprogramming, protection, multiprocessing, virtual memories, etc. It has thus become more difficult to understand the behaviour of these systems in a qualitative sense, let alone quantitatively predict their performance" (Graham 1984).

There are many difficulties in constructing a general-purpose computer system. Problem decomposition, component connections, and interprocessor communication are some issues which can pose significant obstacles to the successful application of such systems.

Therefore, the road to understanding the behaviour and predicting the performance of protected computer systems has been, and still is, arduous. Many people have realised this (Downs 1984, Fabry 1974, Lampson and Sturgis 1976, Wilkes and Needham 1979) and have attempted to investigate the problem of constructing effective highly constrained computer systems, and to proceed towards the development of superior tools. Here we should note that we are concentrating on both construction and evaluation tools, since

> "it has been proven that construction without evaluation is usually inadequate" (Cantrell and Ellison 1968).

The tool that should be developed must not only provide the design primitives but also it should enable the designer to monitor performance and determine (*dynamically*) where restrictions or bottlenecks occur.

There are certain typical questions that the designer can easily answer with a design tool that provide performance information, for example, the time spent by the processor running or waiting for a task, or the time spent in communication.

In efforts to deal with the problem of designing effective highly constrained computer systems, the expressions "*computer architecture*", "*software engineering*", and "*knowledge*

[1] Notable examples on such failures are: the CAP computer system designed by Wilkes and Needham (Needham 1977), its performance degredation reported by Watson (1978), NYU Ultracomputer by Gottlieb et al (1983), and its performance degredation reported by Maples (1985); and the NEPTON system by Evans (1981) its performance degredation reported by Newman and Woodward (1981).

engineering" designate the relevant solution fields. These fields capture the results and accomplishment of research in various parts of computer science (Gaines and Shaw 1986, Tully 1985, Capper 1986), that contribute to the problem of constructing suitable highly constrained computer systems.

According to the results of investigation presented later in this chapter, there is no widely accepted or practiced methodology for the construction of highly constrained computer system, that outlines clearly the construction discipline, based on the three design feilds. The conclusion reached was that the existing attempts of constructing effective highly constrained computer systems can be described as *"black-box"* designs (c.f. Figure 2.1).

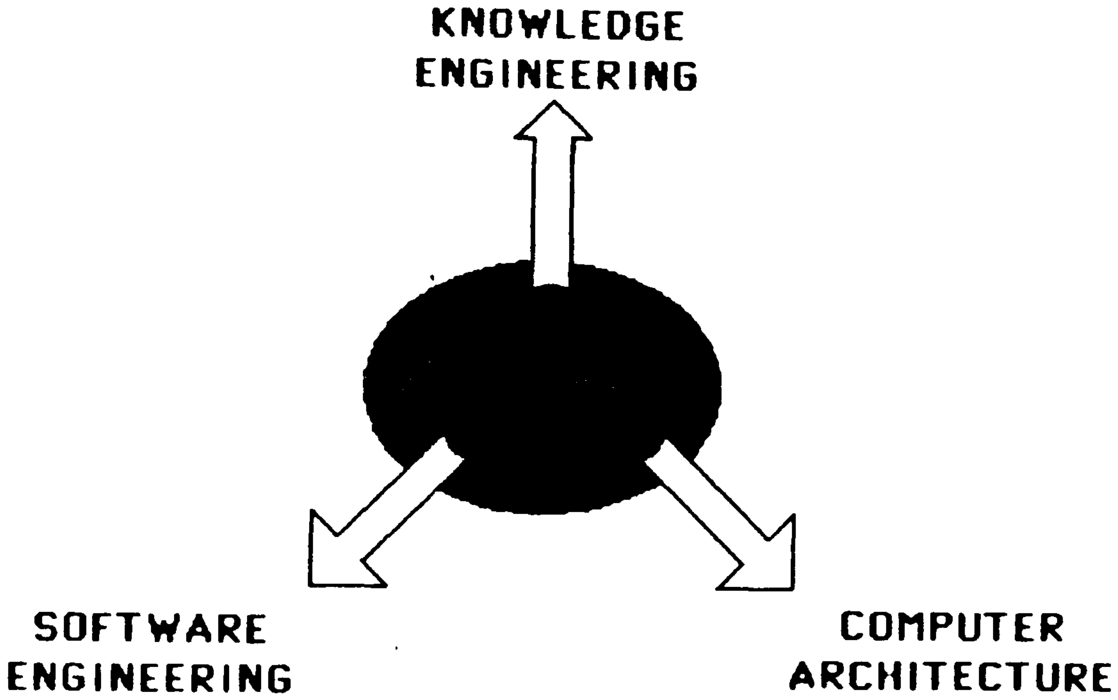


Figure 2.1: The black box: The Traditional Design Methods

The general consensus is that many problems in systems construction and development are caused by the following:

1. The lack of a consistent construction methodology that can offer a the framework which captures the different processing environments and provides effective structures (in hardware, firmware, and software) as well as the integration constraints for holding them together. This is quite an important issue, since we noticed the repeated failures of many of the existing highly constrained computer systems (c.f. Bic 1982) and a large amount of cosmetic techniques used. For example, while designing highly

constrained computer systems using mainly **software engineering techniques** (e.g. Madsen 1981) one could face several performance difficulties and might require the system to be redesigned with the use of, e.g. *performance engineering* (Smith 1980), or the *transparency design technique* (Parnas and Siewiorek 1975). All these steps are certainly expensive and do not ensure the optimal results (Mamrak and Randal 1977).

Similarly if **computer architecture** is used to design highly constrained computer systems (e.g. Dennis 1980) there are variety of other problems, such as the *"von Neumann bottleneck"* and the *"semantic gap"* problems.

The von Neumann bottleneck refers to the type of interaction between the CPU and the computer memory: the huge content of the store must pass, one word at a time, to the CPU and back again. In other words, the von Neumann bottleneck stems from the fact that at the machine language level, any access of a data object requires first the fetch and execution of an appropriate instruction. This is aggravated by the fact that only the elementary (scalar) data objects exist (c.f. Backus 1985). The semantic gap is defined by Myer (1978):

> "The semantic gap is a measure of the difference between the concepts in high-level languages and the concepts in the computer architecture."

Further descussion on the semantic gap is provided by Jones (1977) and Flynn (1980). Here we summarise our findings about the semantic gap issue. It is known that the data objects of typed high- level languages may be defined as a triple:

$$\text{DATA OBJECT} ::= \langle \text{IDENTIFIER, VALUE, TYPE} \rangle$$

In a computer, the data object is represented by the content of some memory location (or a number of consecutively addressed memory locations). If the computer is of the von Neumann variety, then only the component VALUE is represented in the memory. Consequently, TYPE is not an attribute of the data object any more but becomes the attribute of an operation (see Wulf 1981). This discrepancy produces a large semantic gap. The semantic gap of the von Neumann architecture (e.g. VAX architecture) motivates me to search the better operational principles that are matched to the requirements of the support software tool and that provide better performance. Current solutions of the semantic gap minimisation are obtained by introducing the capability addressing of the objects that are encapsulated into memory segment. This allows for typing and access right control at the granulation of the memory segments, at the cost of aggravating the von Neumann bottleneck. This is so become since every single memory location is accessed through at least two levels of indirections.

Mohamad (1982) suggested the introduction of the descriptor-oriented architecture which deals with the semantic gap minimisation. In this scheme, each data object (elemantery or complex) is presented to the hardware by a descriptor that containes all the information needed to enable the communication structure to carry all data transports between the main memory and the CPU of the system. Briefly, it states:

> "the larger the semantic gap the higher the performance degradation is".

A variety of solutions have been reported which include methods such as the use of *direct execution architectures* (Chu 1977), using more *powerful architectures* (e.g. MPP (Potter 1985), CLIP (Duff 1985), Helix (Fridrich and Older 1985), Transputer-oriented architecture (Inmos 1985), Mach-1 (Baron 1985), Manchester Data Flow Machine (Bohm, Curd and Sargeant 1985), Caltech Hypercube Computer (Fox 1985) and the Connection Machine (Hillis 1986)), or using the *vertical migration* technique (Stockenberg 1978). These represent an array of partial solutions of rather arbitrary character. A complete satisfactory solution still remains a substantial research issue.

Finally, using the **knowledge engineering approach** alone (e.g. *adaptive technique* (c.f. Vick et. al 1980)) does not provide a direct answer to the question whether both the performance will be enhanced and the system integrity will be ensured (c.f. Reiner 1980). Indeed, it is an established fact that there is a need for a specialised powerful architectures which can be used to enhence the parrallelism (i.e. performance) of the knowledge-based systems (e.g. the Columbia University Parallel Production Machine (Reeves 1985) and the Fifth Generation Computer (Moto-oka and Stone 1984)), but there is no quantitative assessment on what type of parallelism is the most effective for the knowledge-based systems. Since there are many potential levels of parallelism in any knowledge-based system, for example, the system level, the language level, the search level, the rule level, the subrule level (c.f. Douglass 1985). This issue remains largely a research issue.

2. The second major concern is: How can the user requirements be realised and partitioned into functions to be converted into software, firmware, and hardware components in such a way that sufficient flexibility is retained, in order to coordinate the resulting components at different levels of implementation. This has been treated formally in Kohout (1983,1986). An abstract mathematical formulation of this for parallel computetional structures in terms of abstract logics and generalised topologies was present first in Kohout (1978). In this paper the functional and substatum structures are treated as a pair of adjiont mappings connected by means of so called *galor's* connections. For application in computer science see Roberts (1986), Sharp (1984), Kohout and Bandler (1986).

   It should be noticed that the term **functional** is used in a rather restricted maner (e.g. the function-level programming (Backus 1985), the LISP-oriented programming (c.f. Backus 1981), the blackbord (Craig 1986)). The latter approaches link the functionality to either substratum or behaviour but do provide the explicit link of these two conceptually distinct structures. For the criticism of the confusion that prevail in computing with respect to functionalities see Kohout (1983,1986).

However, in sections 2.1.1 and 2.1.2 we critically review the existing design attempts for constructing one type of highly constrained computer systems which essentially include some protection structures and protection mechanisms (from both the abstract and implementation point of views). These systems are reviewed because they represent a current research issue (c.f. Wilkes 1984) and they provide some examples and design issues that can be compared to our protection functional structures discussed in chapters 5 and 6. and in sections 2.2, 2.3, and 2.4 we report on the problems associated with the most notable attempts. These are discussed according to their relevant design theme (i.e. software engineering, computer architecture, and knowledge engineering). Finaly in section 2.5 we present some design hints that can be used to construct effective highly constrained computer systems.

2.1.1 Abstract Features of Computer Systems Enforcing The Protection Constraint

The development of computer systems that utilise protection functionalities has progressed slowly during the last 15 years. Designers have had great difficulty in determining the best way of supporting a protection policy. Even when they can formulate an architectural approach, they must then face the rather complex problem of making it work correctly and efficiently (Downs 1984).

To a large extent, our ability to create and modify easily any given protected system is determined by its basic underlining abstract architecture. The predominant structure taught today is the simple *reference monitor* organisation (Lampson 1969). The reference monitor acts as an agent checking the legality of every reference of a *subject* to an *object*. Three important concepts unite in the idea of a reference monitor: *mediation*, *isolation*, and *verifiablity*.

The reference monitor must mediate every access to all protected objects, no matter what the situation is. It must be isolated and protected from the rest of the system and from the users. If any user can change the reference monitor, the monitor's ability to mediate all references can be nullified. Furthermore, the reference monitor must be verified to work correctly as a monitor to implement the protection policy.

Based on Lampson's reference monitor scheme, several modifications have been carried out recently. Here I shall survey these modifications, presenting them in an abstract context. This way will enable us to capture the important features that help in creating sophisticated protected computer system designs. However, with the recent introduction of the IBM/38 and the Intel 432, two of the most sophisticated protected computer systems, much attention has focused on the systems that support tickets (capabilities) concept. Most of the early experimentation with tickets was done in universities, but become more active in the development of such systems. In this section, the important designs leading up to and including the ticket-oriented, are reviewed. In fact, the methodological approach of this thesis utilises a modified version of tickets, that is referred to as the ***interpretive descriptor-oriented architecture*** (refer section 6.5.2).

2.1.1.1 Access restriction control:

This type of organisation restricts the operation, each user is allowed to perform upon an object. I assume that the rules of access are specified in some suitable form, so that it is possible to tell whether a user has, or has not, the permission to invoke an operation on an object. There are two methods of enforcing access restriction. These are called list-oriented and ticket-oriented schemes.

Using a list-oriented scheme, the system maintains a list of triples consisting of

⟨USER NAME, OBJECT NAME, OPERATION NAME ⟩

The chief characteristics of this scheme are the maintenance of the list of allowed operations, and validation of each operation by searching through the list. Any of the existing systems having an access control list is list-oriented, although variations can exist in the form of organising the list.

A ticket-oriented scheme uses a protected ticket (usually called a capability or a descriptor). A ticket contains the pair

⟨OBJECT NAME, OPERATION NAME ⟩

A special mechanism, such as tagging words is provided, to ensure that the tickets cannot be changed by any one, other than the protection system. They can however be moved around as data, and passed from one process to another. The ticket-oriented scheme may be further subdivided to **direct** ticket approach (Fabry 1974), and **indirect** ticket approach (Dennis and van Horn 1966).

2.1.1.2 Domains restriction control:

This scheme claims more flexibility for protecting sensitive information. It is desirable to partition a user's computation into several compartments. Programs in a compartment are prevented from directly manipulating data structures residing in other compartments. This is a useful technique to prevent malfunctioning parts of a computation from damaging other parts. Bugs are thereby localised. Several mechanisms for such partitioning have appeared in the literature (Needham 1972, Schroeder and Saltzer 1972, Spier et. al. 1973). Protection rings, domains of protection, regimes of protection are the terms used, which correspond to the compartments. This type of scheme may be further subdivided into **domains in ticket-oriented** system or **domains in list-oriented system**.

2.1.1.3 Type extension control:

The concept of creating abstract data items which are manipulated by associated operations has been found very useful in program development. Type extension scheme involves the following functions (Short 1980) provided in order:

1. To validate the invocation of an abstract operation,
2. To ensure that the components are accessible only in procedures that implement abstract operations,
3. To maintain the correspondence between an abstract object and its components, and
4. To maintain the correspondence between an abstract operation name and its implementing procedure so that control can be transferred to it when the operation is invoked.

Notice that 1 and 2 are aspects of protection in the sense that they are validations of accesses to objects under different circumstances. Once these validations are done, 3 and 4 which are book-keeping operations, can be done by a separate type extension module. This type of protection scheme has always been implemented within the language level of the system.

2.1.2 An Overview of The Ticket-Oriented Protection Constraint

The history of tickets or capabilities can be traced back to the original Rice University computer, designed in 1956. This machine introduced *"codewords"* to designate regions of main storage accessible to a process. The objective was to support more naturally the abstract idea of an array. This concept was first mentioned in the literature by Iliffe in 1961 (Iliffe 1961). Some time later, Robert Barton, a computer designer for the Burroughs Corporation, adopted this abstraction, renamed it the *"descriptor"* and used it in the design of the Burroughs B5000 computer (Burroughs 1961).

Jack Dennis and Earl van Horn at MIT first described capabilities in their 1966 paper (Dennis and van Horn 1966). In their design, each process has a single capability list, containing capabilities for all accessible resources. Dennis and van Horn's paper has had substantial influence on the design of many systems. Most notably, capabilities were incorporated into the design for a computer at the university of Chicago Institute for Computer Research. This computer, later called the Chicago Magic Number Machine, was the first attempt to build a hardware capability mechanism (Fabry 1967). This project was never completed, but much was learned about the general properties of capabilities and their addressing mechanism (Fabry 1974).

At the computer centre of University of California at Berkeley, concepts from Dennis and van Horn and from the Chicago project were incorporated into the design for CAL-TSS, a time-sharing system for the CDC 6400 (Lampson and Sturgis 1976). The CAL-TSS system provided an additional level of indirect addressing. That is, capabilities specified the location of an entry in the Master Object Table, a single data structure maintained by the kernel, that held addresses for all accessible objects.

However, the first two capability based hardware systems to be completed here in U.K. were built by Plessey Corporation, and by the University of Cambridge. These systems were strongly influenced by the Chicago and the MIT work. Maurice Wilkes of Cambridge University, had visited Chicago during the Magic Number project, and had included a description of capability hardware in the 1968 version of his book on time-sharing (Wilkes 1968). At that time, Jack Cotton at Plessey decided to include the idea of capabilities in the System 250. The Plessey system 250 was a commercially available multi-processor system, designed for the use in telephone switching systems (England 1974).

Shortly after the Plessey system 250 was designed, Maurice Wilkes and Roger Needham at Cambridge University began a hardware and software research project. Since the early seventies it was possible to include a reasonable amount of micro-control store in a computer, Needham and Wilkes decided to build a system with implicit capability registers. They call this system the CAP. CAP is running today and is connected to the Cambridge Ring distributed system (Wilkes and Needham 1979). It was not until 1980 that a major computer manufacturer would announce a product that used a capability addressing mechanism. Examples of such systems announced for a commercial market are the IBM System/38 and the Intel 432. Table 2.1 reviews the most common capability machines created until recently.

For more detailed description of many of the machines described here, the reader is referred to (Dennis 1980, Gehringer 1979).

Capability System	*Capability Implementation*
Chicago Magic Number	Hardware
CAL-TSS	Software
Project SUE	Software
Plessey System 250	Hardware
CAP	Firmware
Hydra	Software
cm*	Firmware
EPN	Hardware
Horton	Firmware
PSOS	Hardware
ORSLA	Hardware
SWARD	Hardware

Table 2.1: Various Capability Designs.

However, the literature reveals an additional ticket- oriented architecture called the descriptor-oriented architecture, which is used for more than one purpose. Descriptors are a popular feature in the design of new computer architectures but little has been written about them. Experience with writing and investigating compilers for the two structured architectures, the Burroughs B6700 and the ICL 2900, has shown that the terms used in relation to descriptors often have contradictory meanings (Bishop and Barron 1981). Descriptors as implemented on these machines, are often not the blessing they were made out to be.

A common problem with ticket-oriented systems is the problem of unsatisfactory performance, and most of the systems described, have been substantially slower than the contemporary traditional architectures (Gehringer 1979). Tickets provide fine-grained protection, allowing for the system to be constructed out of a large number of isolated components. Such protection mechanisms have the potential of increasing system reliability at the cost of performance, since frequent changing of these protection domains requires additional processing. Here, we should mention that Brian Gaines reported the design of a high-performance descriptor-oriented mini computer system (MINIC S) that provide an environment for information protection (Gaines et. al. 1974, 1975) However, the discriptors of the MINIC S were implemented in hardware, which still suffer certain performance degradation at the higher system levels of abstraction (i.e. semantic gap). The author's main concern is to investigate the possibility of finding a protection architecture for enhancing the overall performance of the system at the various design levels. The author reported the possibility of finding a solution to the above problem by using the interpretive descriptor-oriented architecture (Mohamad 1982, Mohamad and Cavouras 1984) (refer to section 6.5.2).

2.2 The Software Engineering Approach

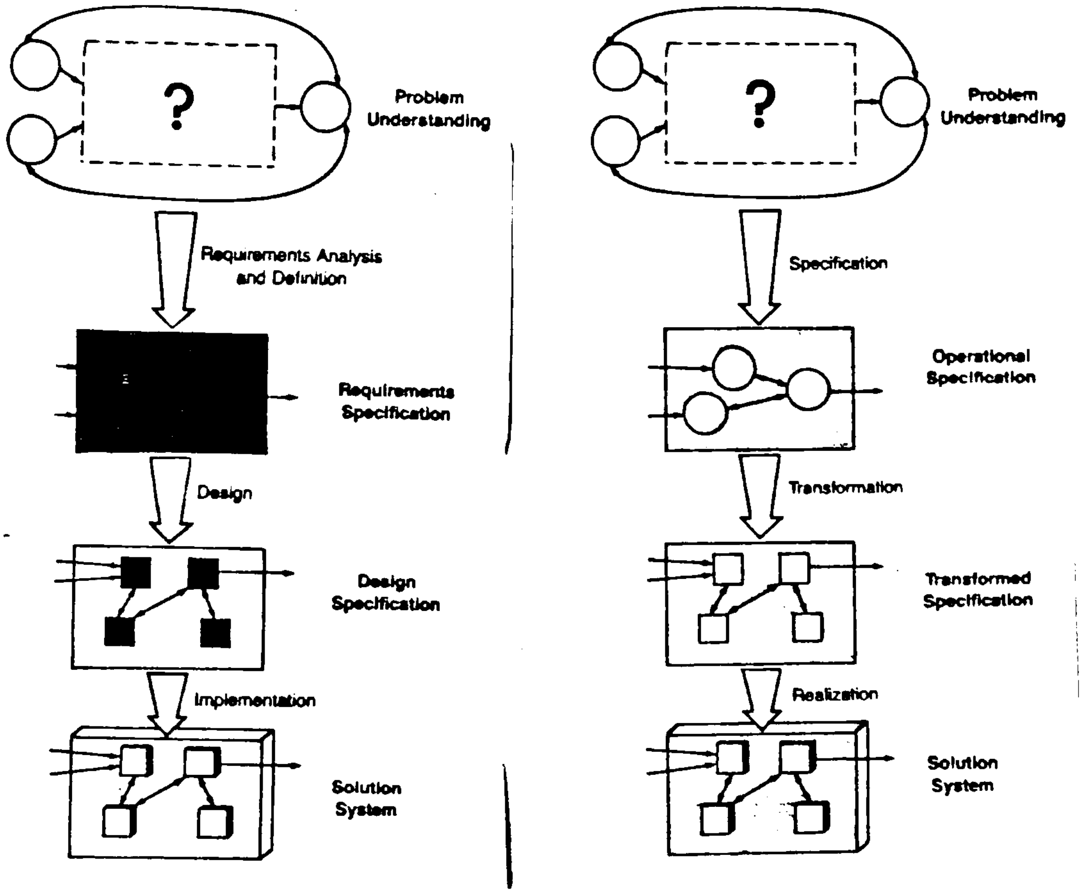
Software engineering is a collection of techniques for constructing and developing large software systems. This simply means that software engineering can be used to develop software tools for the design and construction of protected computer systems. In this section, the present author investigates whether the software engineering approach is capable of producing effective protected computer systems.

Software engineering is considered to be one of the three main techniques that make up system construction and development; these techniques are outlined below, together with some brief comments:

- A *design methodology* that encompasses the techniques used to design the system. The goal of such methodologies is to integrate the design techniques into a rigorous software engineering process that reduces the user specification to a computer-based information system possibly through a number of design levels.
- The *development cycle* which defines the reporting stages through which a project proceeds. In this cycle, the task at each stage together with their inputs and outputs are defined and the documentation is subsequently used in reviews that precede approval of the management to commence the subsequent stages.
- A *project management system* to monitor the progress of a project under development and to take corrective actions whenever some problems arise. The project management system is closely integrated with the system development cycle as it uses the reports produced at each stage of system development.

The main approaches used for the detailed realisation of any of the above techniques are: the **hierarchical approach** and the **operational approach** (Zave 1984). The hierarchical approach is based on the principle of top-down decomposition of black boxes, and all its features can be derived from that philosophy (see Figure 2.2a). On the other hand, thè operational approach is based on separation of problem-oriented from implementation-oriented concerns, and all its features can be derived from that philosophy (see Figure 2.2b).

Based on these two main approaches many system construction techniques have been developed (referred to as software engineering construction tools). Here we list some of them:



(a) The Hierarchical Approach (b) The Operational Approach

Figure 2.2: The conventional approaches of software engineering that can be used for computer systems construction.

- The **participative** technique (Mumford et. al 1978),
- The **life cycle** technique (Shooman 1983),
- The **structured design** technique (Gane and Sarsons 1979),
- The **data analysis** technique (Shuey 1986), and
- Others such as **BSP, SADT, ISAC, MOS, SASO, NIAM, BC, Wrnier-Orr, Jackson, PSL/PSA** (see Blank and Krijger 1983).

The main problem with both of these approaches is that *system description process is completely separated from the system evaluation process.* System performance can not be determined accurately in advance. The main reason for the difficulty of performance estimation of software system arise from the fact that it is difficult to assess the performance of their interface with human users. Recently many researchers expressed the opinion that software engineering techniques must be utilise within artificial intelligent schemes (c.f. Simon 1986, Lindquist 1985). Indeed, there are many issues beside the human interface problem which are not treated properly by the software engineering techniques and may play an important role in producing effective software system (c.f. Goguen 1986). Examples of such issues include the following:

1. Which modules should be kept uncompiled and which should be compiled?
2. What techniques for module composition should be used?
3. How do we best identify the software components most relevant to a particular user need?
4. How do we construct families of related programs?
5. How do we integrate such facilities with other software environment parts (module test, linkage, and interpretation facilities)?
6. How do we best present information to users?
7. What experiments would test the viability of various approaches to these problems?

On one hand, some researchers simply believe that a solution to this problem can be obtained by adopting the idea of **software performance engineering** or SPE (Smith and Browne 1982) which attempt to incorporate set of procedures and metrics along with the software system development process. The SPE is of quite recent origin, awaiting major research developments.

On the other hand, other researchers believe that in order to enhance performance it is necessary to abandon the pure outside- in approach (the two mentioned above) and adopt some additional procedures which are actually of an inside-out or bottom up nature (Parnas and Siewiorek 1975). The typical stages used in the bottom up synthesis starts with a well defined lower level or the base machine. The set of abstractions performed on the lower level will result in higher levels called the virtual machine. The design process is called **transparent** and assumes the base machine activities (i.e. sequence of states) are obtainable. This approach, however appealing it might seems, proves to be quite complex (c.f. Habermann et. al. 1976). Specially in the design and construction of a sophisticated systems such as the operating system, in which the coordination of many concurrent activities is required, and the detailed base machine reaction and capabilities are extremely difficult to obtain.

Furthermore, some other researchers believe that the idea of **reconstructibility** should be incorporated from the beginning of the system design process (Cavallo and Klir 1981). Although this approach initially started as a formal development, we are noticing some success reported in certain simulation experiments (Klir and Way 1985). But indeed, still this solution awaits further research and experimentation as well as metrication of its effectivity to be adopted for (successful) computer system design and construction.

However, in reality there are few computer system design methodologies that have been developed mainly upon the software engineering approach. The **family of system** models by Parnas (1976) seems to be the most notable method. It is based on the concept of *hierarchy of uses*, and it is an extension to the work of Price and Parnas (1973). It was extended in a somewhat different direction by Habermann and Cooprider (1976). Habermann's approach is based on the concept of **incremental machine design** and is similar to Dijkstra's approach in the *"T.H.E."* system (Dijkstra 1968) (see Figure 2.3). There are, however, several related attempts that can be sighted in the literature: the software factory (Bratman and Court 1975), the Boaing Software Design Tool (Carpenter and Tripp 1975), the Dejong System Building System (Dejong 1973), the Habermann System Design (Habermann 1977), and the Sofware Engineering Database (Irvine and Brackett 1977).

Another notable software engineering-based method is the **object oriented design**. This method is basically used for data protection. Abstract data type languages such as CLU (Liskov 1977) , ALPHARD (Wulf 1976), and ADA (Wiener and Sincovec 1984) can be explained well by this scheme. Also, Cm*/StarOS (Jones 1979) and iAPX432 (Khan 1981) can be considered as computing systems that are based on this method. The architecture designed using this scheme employs the concepts of capability-based addressing mechanism

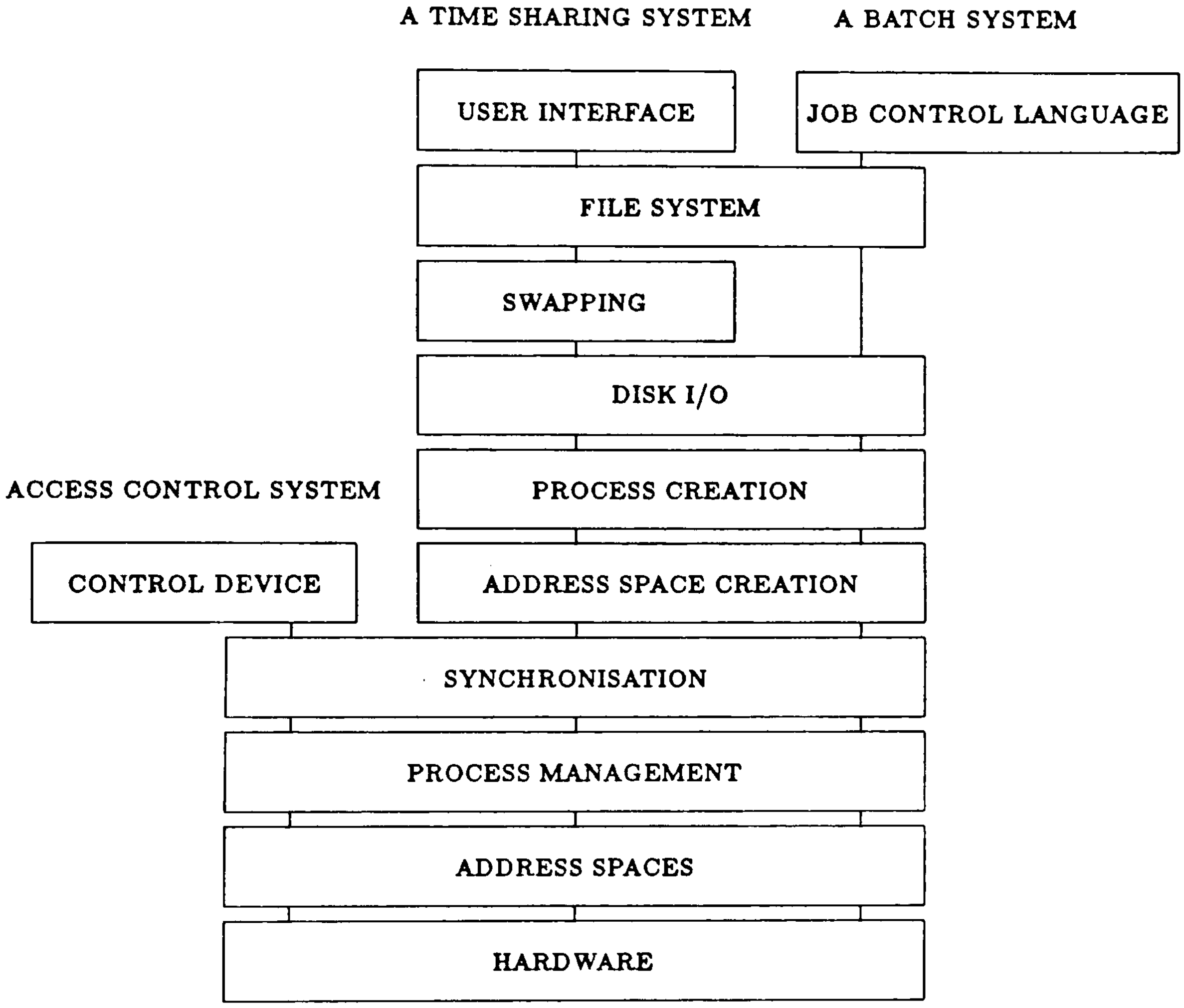


Figure 2.3: The Hierarchy of Uses: Software Engineering-based Design Method

(Fabry 1974). The capability-based addressing mechanism provides a method for identifying an object with an authorised operation. More suitable designs can be developed using the concept of descriptors (Bishop and Barron 1981) (Mohamad and Cavouras 1984) (refer to section 4.6.3, 5.3.1).

Both methods show certain advantages, but it is very difficult to choose among them. Object oriented design concentrates on the real world aspects of the problem via abstraction and information hiding. Hierarchy of uses however, supports factoring which allow us to share system modules, and more importantly, it makes such modules easier to comprehend. Object-oriented design identifies the objects and their operations and groups them together in order to yield cohesive modules. However no provisions have been made to further divide these modules.

There exist no approach for the successful design of highly constrained computer systems based solely upon a software engineering approach. Hierarchy of uses employ transaction and transformation analysis as its strategy for implementation. However this still leaves the design largely an art. Object oriented design, on the other hand, appears to be providing such an approach only until we get to attempt to establish the interfaces. Hence again at this stage, the designer requires a great inspiration for achieving effective construction results.

To conclude, we cannot adopt the software engineering approach as the sole solution to the problem of constructing and developing protected computer systems not only because the aforementioned weaknesses but also because it concentrates on producing only the functional structures of the design. According to Ross and Schoman (1977) there are always problems in the construction of functional designs based on a software engineering approach, since the physical structure is seldom identical to the functional structure (see Figure 2.4). To arrive at an effective construction and development methodology we need to capture the construction requirements elements mentioned earlier (sec. 2.2). The software engineering approach manages to represent the functionality element via many successful design methodologies; such as Gane and Sarson method (1979), the MASÇOT (Simpson and Jackson 1979), the process oriented method (Floyd 1981), Resource Monitors (Pashtan and Unger 1984), and the object-oriented design (Jamsa 1984). All fail to match the operational requirements of the bare hardware or physical structures through the successive treatments (e.g. decomposition) of the functional structures.

2.3 The Computer Architecture Approach

In the pre-LSI era, computer design had to be carried out under the postulate of hardware cost minimisation; a postulate that was satisfied best by the von Neumann architecture. However, the systems that is based on this type of architecture suffers from performance

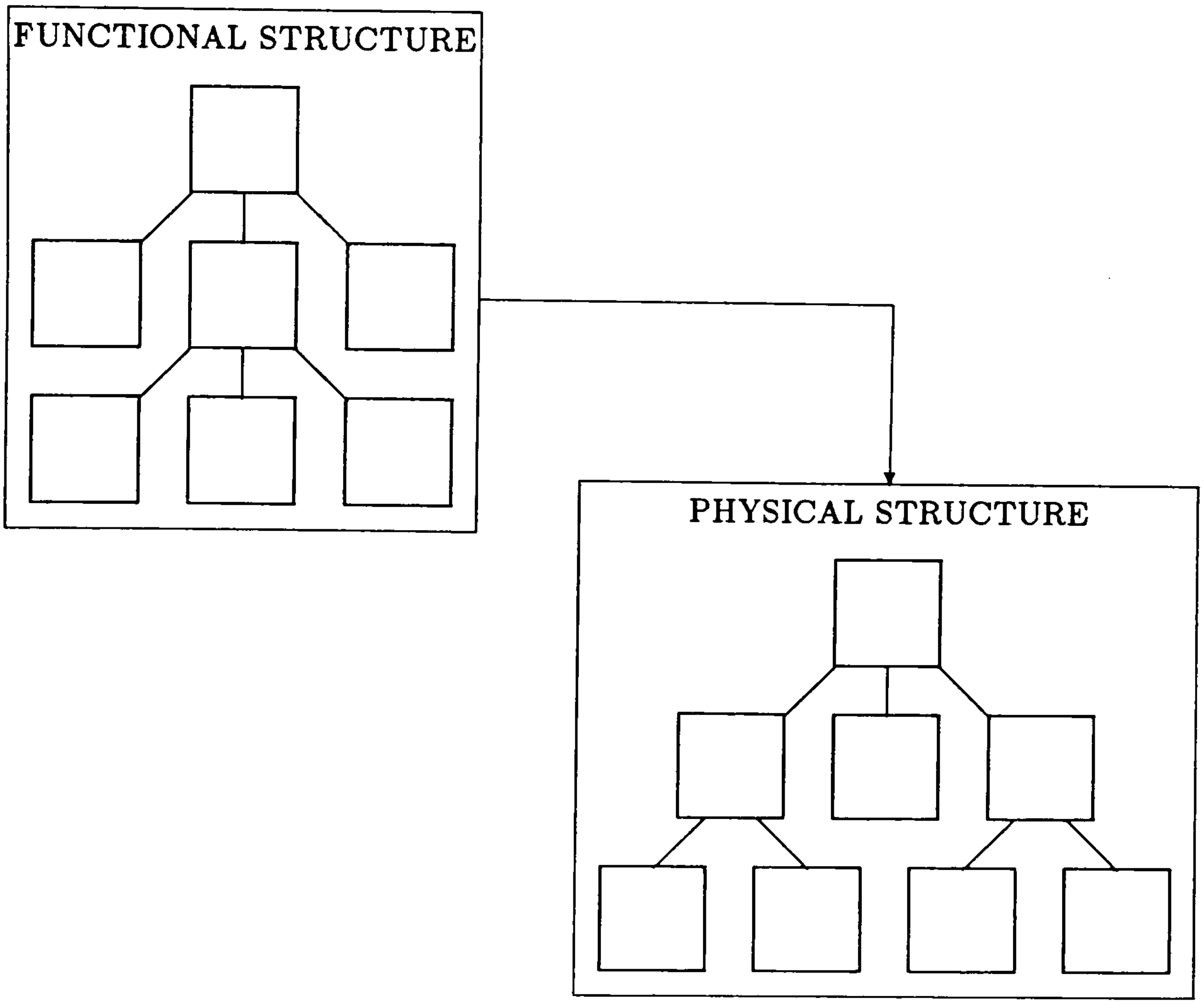


Figure 2.4: The Matching Problem Between Structures

degradation or as Myer put it, the *"Von Neumann bottleneck"* (Myer 1978).

Consequently, many attempts have been made to overcome the performance limitations of the classical von Neumann architecture. Flynn (1972) proposed different types of architectures that can be used for *performance enhancement*. Figure 2.5 illustrates these types of architectures.

Several computer designers believe that Flynn's architectures can be used directly with little software support. Basically the software support utilises certain concurrency control mechanisms that are based on one of the many suggested mechanisms for concurrency enforcement: the concept of software monitors (Hoare 1974, Brinch Hansen 1972), the concept of message-passing (Lauer and Needham 1979), condition queues (Holt et al 1978), coroutines (Marlin 1980), semaphores (Dijkstra 1968), or the rendezvous (Gammage and Casey 1985). Although many software support techniques have been suggested, there are very few

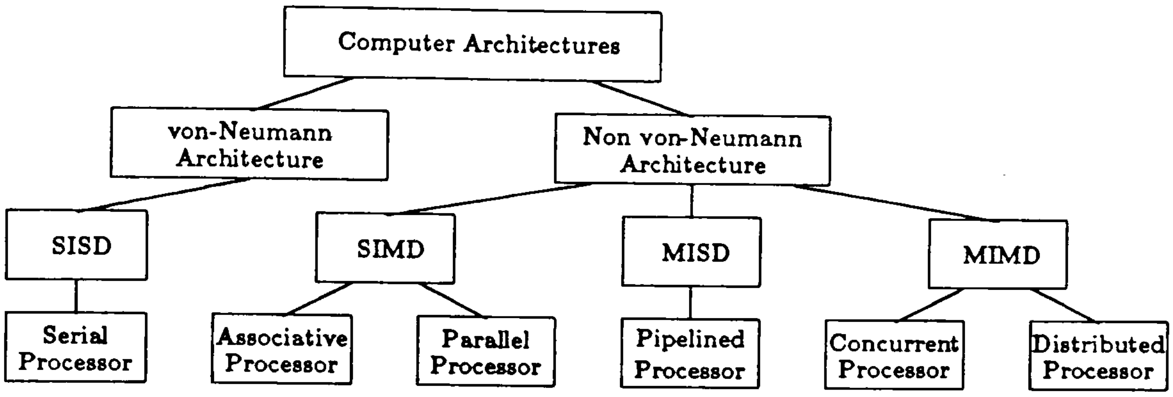


Figure 2.5: Flynn's Computer Performance Architectures

successful implementations available in reality (e.g. OCCAM programming language for the Transputer-oriented computers (c.f. Jones 1985), Parallel Pascal for the MPP computers (c.f. Reeves 1985)). Figure 2.6 illustrates the current status of research in seven of the leading US research centres on supercomputers (i.e. highly parallel computers) as surveyed by IEEE Software journal (1985). The main conclusion of this survey pointed out that there is a great need for innovative software development techniques that can ensure the effectivity of highly parallel computer architectures.

Center	Application Development	Libraries and Algorithms	Compiler Techniques	Operating Systems	Programming Environments	Memory Hierarchy	Debuggers	Hardware
Center for Supercomputing Research and Development	✓	✓	✓	✓	✓	✓		✓
Supercomputing Research Center	✓	✓						✓
John von Neumann Center	✓							
San Diego Supercomputer Center	✓							
National Center for Supercomputing Applications	✓				✓			
Cornell	✓		✓		✓			
Supercomputer Computations Research Institute	✓	✓	✓	✓				

Figure 2.6: Major research efforts at the US supercomputers centres

However, the research experience prove that this sort of software support certainly fails to deal with the addition of extra functionalities that may be added on top of the 'less parallel Flynn architecture' (e.g. distributed systems of the von Neumann computers), such as protection. The fact is that the proposed solutions of software techniques of concurrency support software does not effectively address protection which any effective synchronisation technique must enforce (c.f. Lundstrom and Lawrie 1986, Rennels 1980). In order to overcome this problem, two different approaches have been proposed as a means to achieve protection when it is associated with the support of concurrency: **recovery block** (Randell 1975) and **design diversity** (Avizienis and Kelly 1984).

In the recovery block approach, software routines are organised in a manner similar to the hardware technique of dynamic redundancy. This approach is dependent on the effectiveness of the acceptance test, which judges whether or not the routine has been executed successfully. Such effectiveness is often quite difficult to measure.

In the design diversity approach, a number of independently designed and programmed software routines for a given function are executed concurrently. The results of these routines are compared, and the preferred result is identified by majority voting. This approach, which is called N-version programming, has been experimented upon, and the results have been discussed (Chen and Avizienis 1978). The success of this approach is governed by the degree of independence among redundant software routines.

However, Flynn's architectures refer only to the hardware features of the system. Consequently, they lack the discriminating power to be able to represent the other major features that is important for the construction of effective protected computer systems. The missing features are primarily functional, such as:

- The representation of control structures,
- Information representation in the machine,
- Access mechanisms,
- Communication structures representation, etc.

Here we should make a clear distinction between *computer architecture* and *computer* *organisation*. Computer architecture refer to the functional structures of the system (see section 4.6.2) whereas computer organisation refers only to the firmware and the hardware features of the system (i.e the substrata (see section 4.6.3)), employed to realise the functions generated by the system abstractions.

Missing this distinction, some designers of highly constrained computer systems believe that an effective highly constrained computer system can be constructed only by choosing an effective computer organisation (Dietz and Szewerenko 1979). Using this effective computer organisation, constraints can be built on top of it in order to produce a successful highly constrained computer system (McLaren and et. al 1981)! According to the same belief, the US Army/Navy Computer Architecture (CFA) committee tried to construct or select a well protected computer system to be their military computer (Fuller et. al. 1977). The committee started a series of experiments to select the suitable computer organisation for military purposes. The initial list of candidate systems is given in Figure 2.7 (Burr and et. al 1977).

The results established from Table 2.2 lead to the selection of three 'final candidate' systems: the IBM S/370, the Interdata 8/32, and the DEC PDP-11.

System	*Quantitative Score*	*Pass-Fail Criteria*
Interdata 8/32	1.68 (BEST)	MINOR UNCERTAINTY WITH STATE AFTER TRAPS
PDP-11	1.43	PASSED ALL
IBM S/370	1.36	PASSED ALL
AN/GYK-12	0.94	FAILED FLOATING POINT
ROLM 1664	0.92	FAILED VIRTUAL MEMORY
B6700	0.92	FAILED PROTECTION
SEL-32	0.86	FAILED VIRTUAL MEMORY
AN/UYK-7	0.46	FAILED VIRTUAL MEMORY
AN/UYK-20	0.46 (WORST)	FAILED PROTECTION

Table 2.2: The CFA Selected Computer Organisations

This scheme has been proven by many researchers to be unreliable. Bic (1982), for example, proved that protected computer systems constructed using the data flow computer organisation suffers mainly from complexity and performance degradation, failing to meet all the expectations. Newman and Woodward (1981) proved that protected computer systems suffer from performance degradation even when multiprocessor computer organisation is selected. McLaren and MacEwen (1981) explained the reasons why building a protection subsystem on top of a successful computer organisation will not produce an effective highly constrained computer system both in the degree of maximal constrained achieved and from the performance point of view.

The problems associated with Flynn's architectures have been slightly enhanced by adding certain functional features to the computer organisation and design. Figure 2.7 lists the most notable attempts of the modified approach that are specially made for highly constrained computer systems construction (Mohamad and Cavouras 1984).

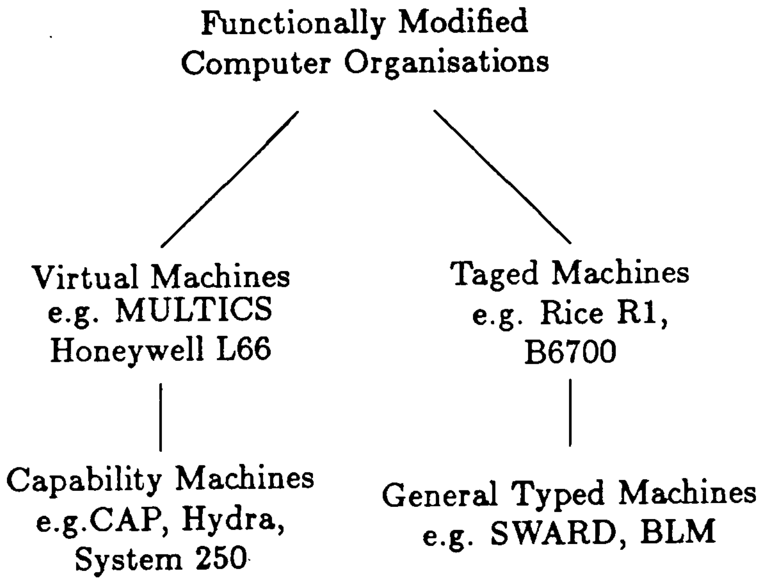


Figure 2.7: Functionally Modified Flynn's Architectures

Although these modifications may appear to be good, they add serious problems to the research aim of constructing effective highly constrained computer systems. These problems are:

- Existing methods of performance evaluation are not suitable for these new architectures.

- The resulting systems should be dedicated to the computer organisation that is associated with.

The problem of finding a suitable performance evaluation method has been studied recently by using yet a different technique, the **software science methods**. Several researchers demonstrated that by using the software science we can extract certain architecture quality measures (Kavi and Krishnamohan 1984), the simple Halstead measures (Pashtan 1985), or certain empirical approaches (Lunde 1977). However, even using the conventional evaluation techniques certain modified computer organisations are found to suffer from performance degradation. The system developed by the Cambridge University group (Wilkes and Needham 1979) has been developed to CAP2 after conducting a performance study on the original CAP and finding that it suffers from major performance limitations (Cook 1978). But we still need a performance analysis method that can *evaluate* computer designs with respect to the *constraints parameters* as well as *hardware configuration* and *workload.*

The second problem has been partly solved by proposing that the design of highly constrained computer systems should be based upon certain high-level computer architectures in which the constructed system features are **directly executed** within the high level language itself (Flynn 1980) (Chu 1981) or by using a design language that allow us to enhancethe performance via techniques such as the **vertical migration method**.

Using the direct execution approach, there exist two options: the use of the **notational design languages** or the use of the **hardware description languages**. The notational design languages treat the design of computer organisations from formal point of view, such as **AHPL (Hill 1975), ISP (Baebacci 1977), CDL (Chu 1965), DDL (Duley et. al 1969), RTS (Piloty 1975), and PMS** (Gordon et al 1971). These tools are convenient for studying the overall configuration of a computer system (see Figure 2.8).

Indeed the notational representation of a system's structure is a powerful documentation tool. But, it is difficult for a machine to interpret it. In addition, it lacks information about the behaviour of the system components, thus limiting its applicability (Djordjevic

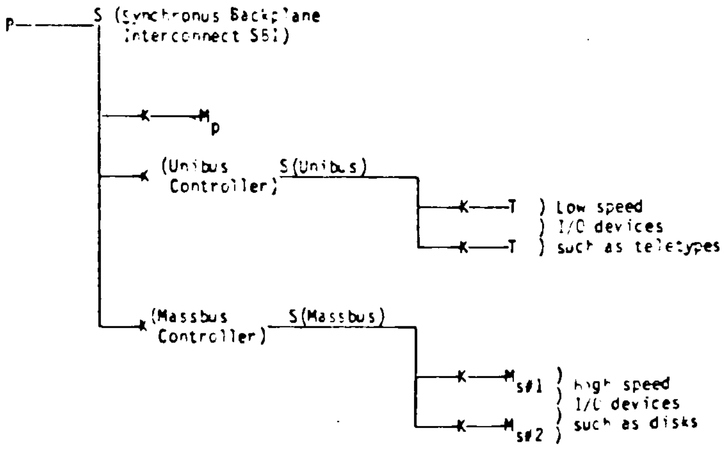


Figure 2.8: The PMS Description of the VAX-11/770 computer system.

et al 1985). However, we should point out that the ISP notation is used to describe the computer system components behaviour. This encouraged some researchers to develop it into a very useful formal language, which has been called ISPS (Barbacci 1977). But, a complete separation between the specification of the structure and the behaviour of a computer system is not an easily realisable or even a desirable goal. Structure and behaviour go hand in hand and it is a measure of the power of a language that is able to enhance one aspect over the other.

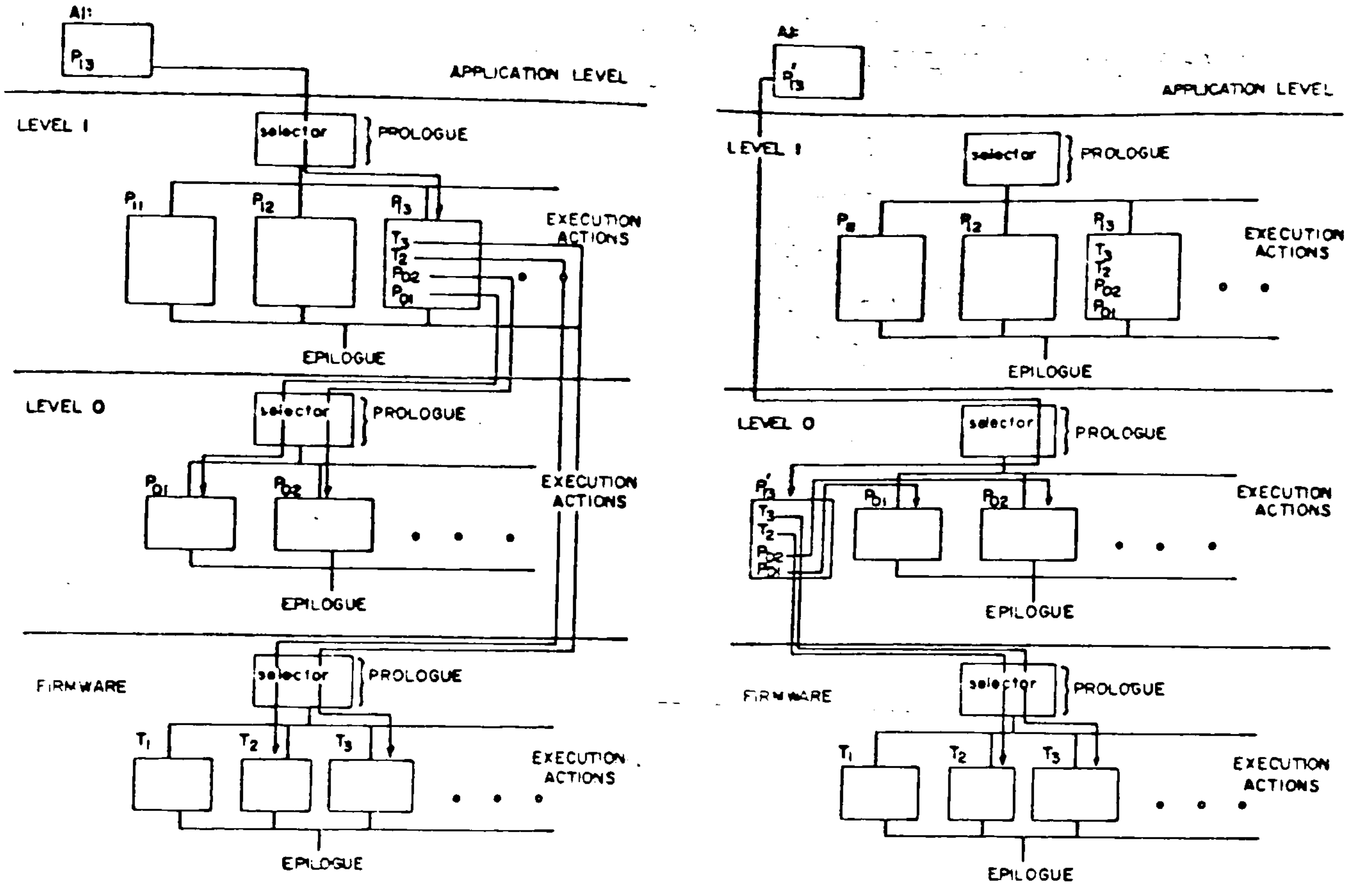
Computer Hardware Description Languages (CHDL's), on the other hand, have been used in computer design since early 1960's, as can be seen in (Schlaeppi 1964) (Schorr 1964). According to Barbacci (1975) a description language can be procedural or nonprocedural. Procedural languages impose an explicit ordering of execution of the statements describing target machine activities. A statement is executed after the completion of the preceding one. The nonprocedural languages attach no meaning to the lexicographical ordering of the statements describing the target machine. Examples of these hardware description languages may include: the CDL language by CHU (1965) nonprocedural type, the DDL by Duley and Dietmeyer (1969) nonprocedural type , and AHPL by Hill (1975) procedural type .

Over the years, many CHLD's have been defined and some of them implemented with so many additions and developments (see Fernandes 1982) (Barbacci and Uehara 1985). However, the primary aim of these languages is to describe the computer hardware that enables us to realise the target machine. Indeed this is not our primary interest in the design of highly constrained computer systems. What we need is a *description and synthesis tool* that allow us to *define* the *computer architecture* (hardware, firmware, and software), *that is machine independent* which *captures the essential structural and behavioural elements* of the system as well as it can be used for *predicting the system performance.*

Vertical migration method, however, is a systematic, partially automated method for the performance improvement of a dedicated application or a class of applications in a multilevel firmware- software hierarchical system which aim at reducing the CPU overhead (Stockenberg and van Dam 1978). Each level has an associated execution time overhead. The execution time overhead of a level is lowest for the hardware and increases for each level as you proceed up the hierarchy from hardware to the application program level. This is because the higher levels typically make use of the lower levels, incurring the overhead of the lower levels in addition to their own overhead.

The method for reducing overhead involves reimplementing either entire functions, or paths through them, which are CPU intensive on lower levels, for example, reimplementing an OS-level 1 function as an OS-level 0 function, or reimplementing an OS-level 2 in the firmware. Exactly what the overhead consists of is described as part of the model below. Figure 2.9 illustrates a typical example of using the vertical migration to enhance the performance of a hypothetical multilevel model.

The vertical migration as described deals exclusively with performance issues. There is no concern for the complexity of the mapping or execution actions, or the types and numbers of interconnections between modules (except that modules be interconnected in a hierarchical fashion). Indeed the complexity issue is of great concern for the design of highly constrained computer systems and cannot be ignored if we want to achieve the aim of producing effective designs.



(a) before migration

(b) after migration

Figure 2.9: An Example of a Vertical Migration Process.

2.4 The Knowledge Engineering Approach

The current interest in knowledge engineering has been stimulated by the announcement (in 1981) of the Japanese programme of research and development into the "*fifth generation computing* ". This generation of computers is characterised by the view that "*knowledge*" rather than "*data*" is the essential raw material to be processed. Knowledge engineering has been intensively used in the area of artificial intelligence (AI) and Expert Systems. Certainly, it is commonly missing from operating systems and computer design. However, recently some researchers started to develop "*smart kernels*" of operating systems using certain **adaptive strategies** that represent a tool box for knowledge engineering. Reiner (1980) used his adaptive strategy to improve system performance through dynamic modification of system control parameters (c.f. Figure 2.10). Lantz et. al. (1982) has implemented the RIG an *"intelligent"* distributed operating system based on similar ideas of Reiner.

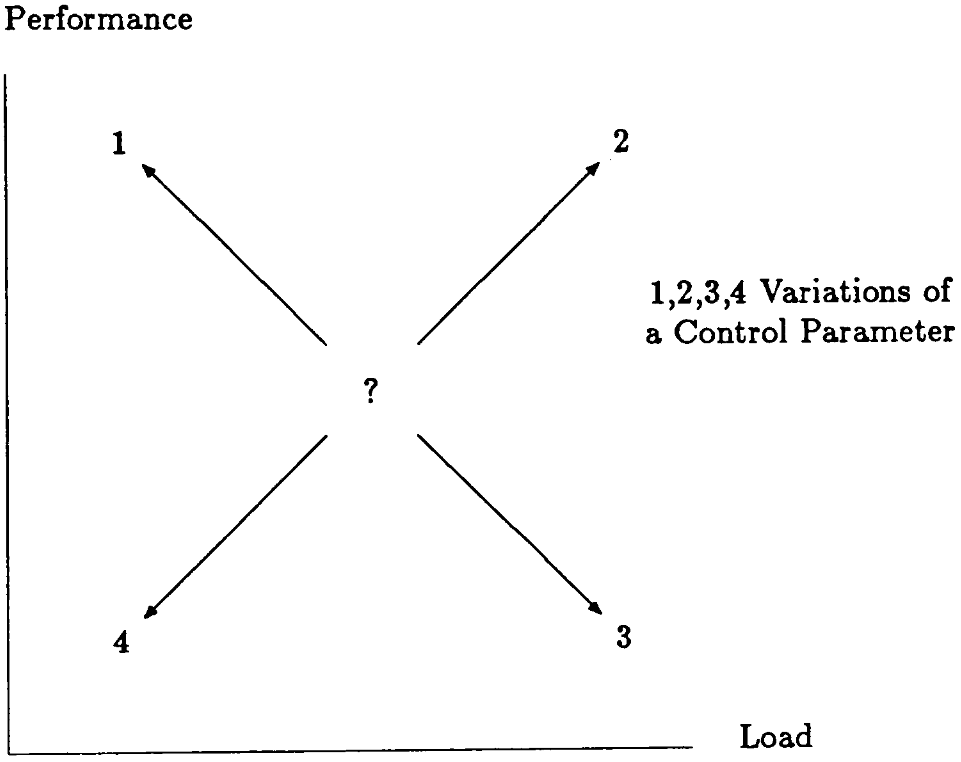


Figure 2.10: Reiner Adaptive strategy.

The problems with Reiner's approach (or indeed any similar attempts such as the scheme of *adaptable architectures* by Vick et al (1980)) are quite serious. While a parameter may be easy enough to modify, the effects of a change may be unclear or difficult to observe. This is particularly true for large systems, where workload and resulting performance variations are sufficiently large to observe changes in the performance caused by resulting control parameters. The wrong choice of corrections may cause the system to become unstable, oscillate, or even crash. Furthermore, frequent changes to control parameters may include transient effects which degrade overall system performance. Clearly, guesswork and unstructured attempts to introduce adaptive control have little chance of success with complex systems.

The author believes that more effective approach should be achieved by employing the following knowledge engineering steps:

- The use of *iterative design* (Mohamad 1981),
- The use of *extensible structures* (Mohamad and Cavouras 1984),
- The use of *'smart' scheduling policies and inferential techniques* (Mohamad 1982).

These design steps will be incorporated within our proposed methodology in chapter 4.

2.5 Hints For A Successful Computer System Design

> "Designing a computer system is very different from designing an algorithm: the external interface–that is, the requirement–is less precisely defined, more complex, and more subject to change; the system has much more internal structure–hence, many internal interfaces; and the measure of success is much less clear. The designer usually finds himself floundering in the sea of possibilities, unclear about how one choice will limit his freedom to make other choices or affect the size and performance of the entire system. There probably isn't a best way to build the system or even a major part of it. Much more important is to avoid choosing a terrible way and to have a clear division responsibilities among the parts" (Lampson 1984).

The most important hints, and the vaguest, have to do with obtaining the right functions from a system. Most of them depend on the notion of an **interface** separating an implementation of some abstraction from the clients who use the abstraction (Britton et al 1981).

Defining interfaces is the most important part of system design. Usually, it is also the most difficult, since the interface design must satisfy conflicting requirements:

- An interface should be simple,
- It should be complete, and

- It should admit a sufficiently small and fast implementation.

Indeed, defining interfaces is a part of the whole process of system design and synthesis that corresponds closely to analytic modelling in many other fields. Construction of a model usually starts with observations, followed closely by formation of hypotheses about principles or axioms that explain the observation. These axioms are used to derive or construct a model of the observed system. The parameters or variables of the model may be derived from the axioms or they may be estimated from observation. The model is then used to make new predictions. The final step is to perform experiments in controlled or well- understood environments to determine the accuracy and robustness of the model and of the axioms. This cycle of hypothesising and validating models is then continued with additional observations.

However, designing computer systems involves a series of design activities of a designer and design tools. The design activities consists of manipulating design objects according to some rules derivable from the design objectives. The activities range from the first specification of the functionality expected from the final design, through various types of analysis, synthesis of implementation, to synthesis of the final implementation, realisation and testing. We can coarsely describe the design process as composed by some major design steps as illustrated in figure 2.11

Design tools and techniques are often associated with the specification of systems intended for implementation as a mixture of hardware, software, and firmware. These tools generally consisted of two main parts; a language independent part and a language definition and handling mechanism part (see Figure 2.12).

Traditional approaches are based on technological premises that are no longer valid.

> "The new *"ground rules"* make possible new relationships between architecture and language" rm (Flynn 1980).

Indeed modern design tools contribute to what Flynn pointed out, and they provide a non-restricted design paradigm.

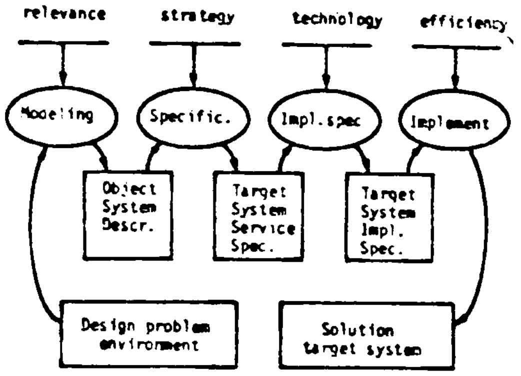


Figure 2.11: The main phases of the design process.

Hence, the ***interaction between the computer language design*** and the ***system model structures*** (architecture) have serious implications for the overall computational cost and efficiency. In this section we investigates those interactions, and classify them into four distinct categories that lead to four distinct approaches. For the purpose of developing a general purpose design and construction tool, the most non-restrictive approach should be adopted. The details and advantages and disadvantages of each approach is given below:

2.5.1 Dedicated Language Dedicated Architecture Approach:

This approach is called traditionally the *Direct-Execution Architecture* approach (i.e. is a language-directed computer architecture). It can accept a high-level-language program and executes it directly without compilation, assembly, linkage editing or loading. It offers a means to eliminate compilers, loaders etc. and attacks the problem of mounting software cost (Chu 1977). The history of this type of architecture returns back to the year 1963 when Mullery et. al. (1963) designed a problem-oriented symbol processor called ADAM and concluded that a high-level language could be implemented with a reasonable amount of hardware.

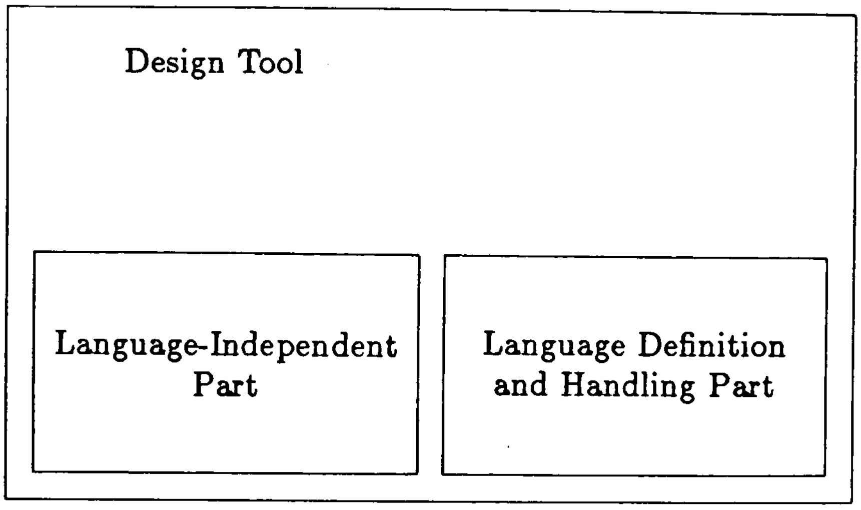

Design Tool
Language-Independent Part
Language Definition and Handling Part

Figure 2.12: Design and Synthesis Tools Structure.

There has been considerable research directed toward the use of direct-execution architectures (especially in the area of hardware description languages (CHDL)) in the automated design of digital hardware, but attempts to produce efficient hardware design in this way have had little success (Boulton and Goguen 1979). This is primarily because previous high-level CHDLs could not represent the design of large scale digital system in a way that could be related directly to a low level machine hardware realisation.

Some solutions to the above problem have been provided, for example, Shimizu and Sakamura (1983) decided to use the a knowledge-base (MIXER) which has the relevant information on a family of target architectures which can be used later by the CHDL to produce effective matching descriptions directly on a particular hardware. This approach however, is still very restrictive to be adopted for computer system design purpose.

2.5.2 Non-Dedicated Language Dedicated Architecture Approach:

It is the fundamental premise to this approach that the purpose of the resulted computer systems is to provide a cost-effective solution to a particular set of problems. That solution is best attained through the use of dedicated system architecture (Bose and Davidson 1984). The supporters of this approach argue that machines should be designed from historical base. Consequently only rarely does a new architecture appear in a real world of computer design. Most computer architectures are variations on the same theme: a simple von

Neumann machine. Therefore, dedicated computer architectures appear to be very advantageous, since the computer's instruction set and its gross information flow have been chosen to make the hardware simpler, more *"efficient"*, or to invoke some obvious optimisation. An excellent example for this approach is the *systolic array architecture* approach (Sorasen et. al 1983). Indeed we may found some dedicated architectures that can be ported within similar environments, those architectures may include the *SCAPE* (Lea 1983) and the *EPSILON* (Hayes 1983). The porting criteria of this sort of architectures is not defined and well reported in the literature and is indeed still greatly being researched.

Further, some researchers have realised that certain computer architectures (e.g. the von Neumann architecture) do not provide adequatetranslations for the constructs that occur in common programming languages. This type of shortcoming is attributable to a phenomenon known as the *"semantic gap"*. [2]

> "Most current systems have undesirably large semantic gap in that the objects and operations reflected in their architectures are rarely closely related to the objects and operations provided in the programming languages. This large semantic gap contributes to software unreliability, performance problems, excessive program size, compiler complexity, and distortions of the programming languages; all of these contribute negatively to the efficiency and cost of the resulting computer system" (Myer 1978).

However, in the case of pararllel architectures (i.e. non von Neumann) it is rather a problem to use them effectively with non-dedicated languages, since humans tend to think sequentially rather than concurrently. The human programmers, hence, tend to develop their programs in a sequential language such as Fortran. While the resulting programs are usually very effecient on a von Neumann machine, they often incapable of directly making effective use of the parallel machines. Since it seems clear that the next generation of computers will be based on the parallel paradigm, this poses a potential roadblock in the full use of these parallel computers. A typical solution to this problem, which represent the

[2] See section 2.1 for further information on the nature of the semantic gap.

current practice, is provide a set of very simple machine calls which can be incorperated in any language in order to support the concurrency criterion (c.f. Allen and Kennedy 1985). As a result, the programmer is responsible for explicitly handling all synchronisation. The problem with this approach is that concurrent programming is unnatural for many programmers. Not only is writing such a program tedious, but it is also presents many opportunities for creating bugs that are almost impossible to find; such as deadlocks and programs that produce different results on the same data. Hence, it is necessary for the programmer using non dedicated language with a dedicated architecture to understand the of the details of the given dedicated architecture in order to use it optimal parallel capacity. For this reason, this approach seems quite limited.

2.5.3 Dedicated Language Non-Dedicated Architecture Approach:

Present problem specification languages contain very few constructs about dynamics such as synchronisation between processes, dynamic allocation of resources, or timing of events that are useful for modelling dynamics. In particular, the constructs of dynamics are generally not adequate, or complete enough to permit the construction of highly constrained computer systems. The objective of this approach is to define concepts and a language to describe certain constructs that can as the primitives for computer system construction. SODAS is an example of such an attempt (Parnas and Darringer 1979).

Recently researchers like Hac (1982) expressed opinion that any computer language can be transformed and be used for computer system design and construction tool (for example Pascal). Ambler and Hoch (1977), however, believes that constraints such as protection can be enforced from the language level. Although this approach appear to be very attractive for highly constrained computer systems construction, it suffer from a major problem that it concentrate on the representation of the system features using the high level language constructs without giving proper consideration to the interpretation of these constructs and to the performance of the resulting design. Berg (1977), along with Cavouras and Davis (1981) pointed out this problem and reported the need for an ideal solution that consider the representation of hardware, firmware and software features of the system in an effective and integrated way.

2.5.4 Non-Dedicated Language Non-Dedicated Architecture Approach:

This is the most non-restrictive approach that can be used effectively to design highly constrained computer systems. However, to date there are very few successful attempts that can achieve computer system design in such non-restrictive way. Here we distinguish the scheme of *activity structures* applied within the general meta systems framework (Kohout and Gaines 1976). This scheme is used for capturing the diverse features (functional and substratum (hardware)) in natural or artificial systems. Because of generalised formulation (generalisation) of the activity structures constructs, the activity structures based machine can be transported across disciplines and environments. This presents the opportunity to provide a knowledge domain independent, but purpose oriented empty *shell.* These features let the activity structures scheme to be an excellent approach for integrated and systems design and construction. Activity structures scheme was used to construct effective medical (Kohout et. al. 1984) and technological (Kohout and Bandler 1981) as well as certain social systems (Kohout et. al. 1984a). This approach not only uses flexible functional structures but it also utilises extesible substrata, such as coroutines (can be programmed in any programming language) which can operate in a parallel or sequential fashion depending on the host bare architecture. However, this approach may be divided into two sub-approaches; namely, the *bottom-up* sub-approach, and the *top-down* sub- approach. The main reason for introducing this type of classification is to ensure the design flexibility of this approach.

2.6 Concluding Remarks

It is hoped that the preceding discussion has indicated some of the practical problems of highly constrained computer systems design. It was not intentend to present any methodological solution in this chapter, rather, I have attempted to present and discuss the ideas from the current *"science paradigm"* literature which may be considered as possible approaches to solutions of some of these problems. In this respect, my primary concern was to show how these approaches fail within the relevant design fields to capture the main construction requirements of highly constrained computer systems. I have demonstrated that the existing techniques from the relevant design fields (software engineering, computer architecture, and knowledge engineering) are largely fragmented to be used for constructing highly constrained computer systems. There is a need for a methodlogical approach which integrates the design advantages of the relevant design fields.

Chapter 3

AN OVERVIEW OF THE EXISTING THEORETICAL APPROACHES FOR CONSTRUCTING HIGHLY CONSTRAINED COMPUTER SYSTEMS WITHIN THE SCIENCE PARADIGM

3.1 Milestones

Since the early days of computer industry, there has been considerable interest in the theoretical design and performance analysis of computer systems. There are three practical goals related to theoretical design and performance analysis: selection of the best among several existing systems; design of not-yet existing system; the analysis of an existing accessible system.

The mathematical analysis of congestion in telephone systems pioneered by the Danish engineer A.K. Erlang (Brockmeyer 1948) was a major contributor to performance assesment and the design of "computer systems". The problem tackled by Erlang is the relationship between the number of connected telephone subscribers, the probability of making a call, the probability of the call requiring various lengths of time, and the number of *"trunk"* lines that should be installed by the telephone company. However, not until 1957 a realistic theoretical design approach had started. That year Jackson published his (queueing theory or queueing network) analysis of a multiple device system wherein each device contained one or more parallel servers and jobs could enter or exit the system anywhere. In 1963 Jackson extended his analysis to *open* and *closed* systems with local load-dependent service rates at all devices. In 1967, Gordon an Newell simplified the notational structure of these results for the special case of closed systems. Baskett et al. (1975) extended the results to include different queueing disciplines, multiple classes of jobs, nonexponential service distributions.

The first successful application of the queueing network analysis to a computer system came in 1965 when Scherr used the classical machine repairman model to analyse the MIT time sharing system and the CTSS system (Scherr 1967). However, the Jackson-Gordon-Newell theory remained dormant until 1971 when Buzen introduced the central server model and fast computational algorithms for these models (Buzen 1971, 1973). Working independently, Moore (1971) showed that queueing network analysis could predict the response times on the *Michigan Terminal System* (MTS) within 10Extensive validations since 1971 have verified that these design models reproduce observed performance quantities with certain accuracy percentage (not yet remarkable !) (Hughes and Moe 1973, Denning and Buzen 1978).

However, most of the current computer systems design theories (including the queueing theory) provide only certain ***specialised design models*** (c.f. Klienrock 1985, Lundstrom and Lawrie 1985). This fact can be depicted from the tremendous effort spent by the researchers working on the widely used computer design theory; *the queueing theory*. These efforts are critically reviewed in sections 3.2.1, 3.2.2, and 3.2.3. Table 3.1 illustrates some of the less widely used theories for constructing and analysing certain specialised computer system models.

Until now the building blocks of scientific theory; the theoretical and methodological constructs as well as the mathematical formalisms– have made it difficult to conceive of the possibility of giving a general account of nature of computer systems design and their organisation. We have had to rely on the essential notions of physics (time, length, etc.) or else on descriptive analogies taken from ordinary experience in order to make intelligible natural phenomena. The advance of *"statistical"* sciences (operational research, kinetic theory, queueing theory, statistical mechanics and thermodynamics, quantum mechanics, etc.) have led to new ways to simplify complex phenomena within an empirically meaningful framework. Successes in these areas suggest that behind the formidable complexity of nature there are actually a surprisingly small number of simple relations governing interactions;

	Modelling Theory	*Computer Model*	*Reference*
1	CONTROL THEORY	Distributed Systems	(Kramer et al 1984)
2	OPTIMISATION THEORY	Interactive Systems	(von Mayrhauser 1979)
3	RELATIONAL PRODUCTS THEORY	Protected Systems	(Kohout et al 1981)
4	MATHEMATICAL SYSTEMS THEORY	Symbolic Systems	(Pichler 1983)
5	INFORMATION THEORY	Communication System	(Usher 1984)
6	AUTOMATA THEORY	Distributed Systems	(Strak 1984)
7	POSSIBILITY THEORY	Protected Systems	(Rine 1978)
8	FORMAL MODELLING THEORY	Simulation Models	(Zeigler 1972)
9	CATEGORY THEORY	Formal Models	(Bandler 1978)
10	CYBERNETIC MODELLING THEORY	Interactive Systems	(Iyenger et al 1980)
11	PERFORMABILITY ANALYSIS	Reliable Systems	(Meyer 1980)
12	SYSTEM CONNECTION ANALYSIS	Performance Models	(Yuval 1980)

Table 3.1: Some Less-Used Computer Systems Modelling Theories.

the difficulty has been to refine several theories to the point where such relations could emerge clearly.

Without a general design theory, the road to understanding the behaviour and predicting the performance of protected computer systems has been, and still is, arduous. Many people have realised this and have attempted to investigate the problem of designing and analysing the performance of highly constrained computer systems, and to proceed to develop superior theoretical models and tools ("yet only fragments!" (Tully 1985)).

3.2 Critique of Analytical Modelling

Any system design, any measurement project or any resources allocation strategy is based on some conception of environment in which it operates. That conception is a model. It is beneficial to have such models explicitly stated so that they can be explored, tested, criticised and revised. Even better, though not often achieved to the extent desired, is a formal analysis of the model.

Theoretical models and methods of computer systems design and analysis vary greatly (Table 3.1). While most will argue that the goals of such models are inherently worthwhile and must be pursued, there is widespread dissatisfaction with the current state of theoretical paradigm. Basically, there are three areas of dissatisfaction. First, the models are generally oversimplified in order to make them mathematically tractable. This obviously makes the results questionable and brings us the second major failt which is that analytical results are often not validated by measurement or simulation. Moreover, in cases where system evaluation studies are carried out, the existing models do not seem powerful enough to provide a uniform basis for measurements. The third major criticism is that most of the literature on analytic modelling is a collection of analyses of specialised models. This points out the lack of very general powerful models which would allow analysis to become an engineering tool. As it is now, each new situation almost always requires a separate analysis.

Although *queueing theory* (c.f. Kleinrock 1975, 1976) is not the general design theory agreed upon by all the researchers (including Klienrock 1985), designers, and manufacturers, it was the only theory employed widely for the design and evaluation of computer systems within various classes of the design models. Indeed the queueing theory fails to accurately describe distributed computer systems (one of our main design requirements). However, the analytical or the application models based upon the queueing theory treat various types of the pre-assumed computer settings or networks (closed or open), treat various job classes, and employ certain approximations which relax some of the restrictions necessary for the application of the queueing theory.

Mohamad and Cavouras (1982) classify the models that utilise the queueing theory into three categories (see Figure 3.1.

1. *analytical models*,

2. *simulation models*, and

3. *empirical models*.

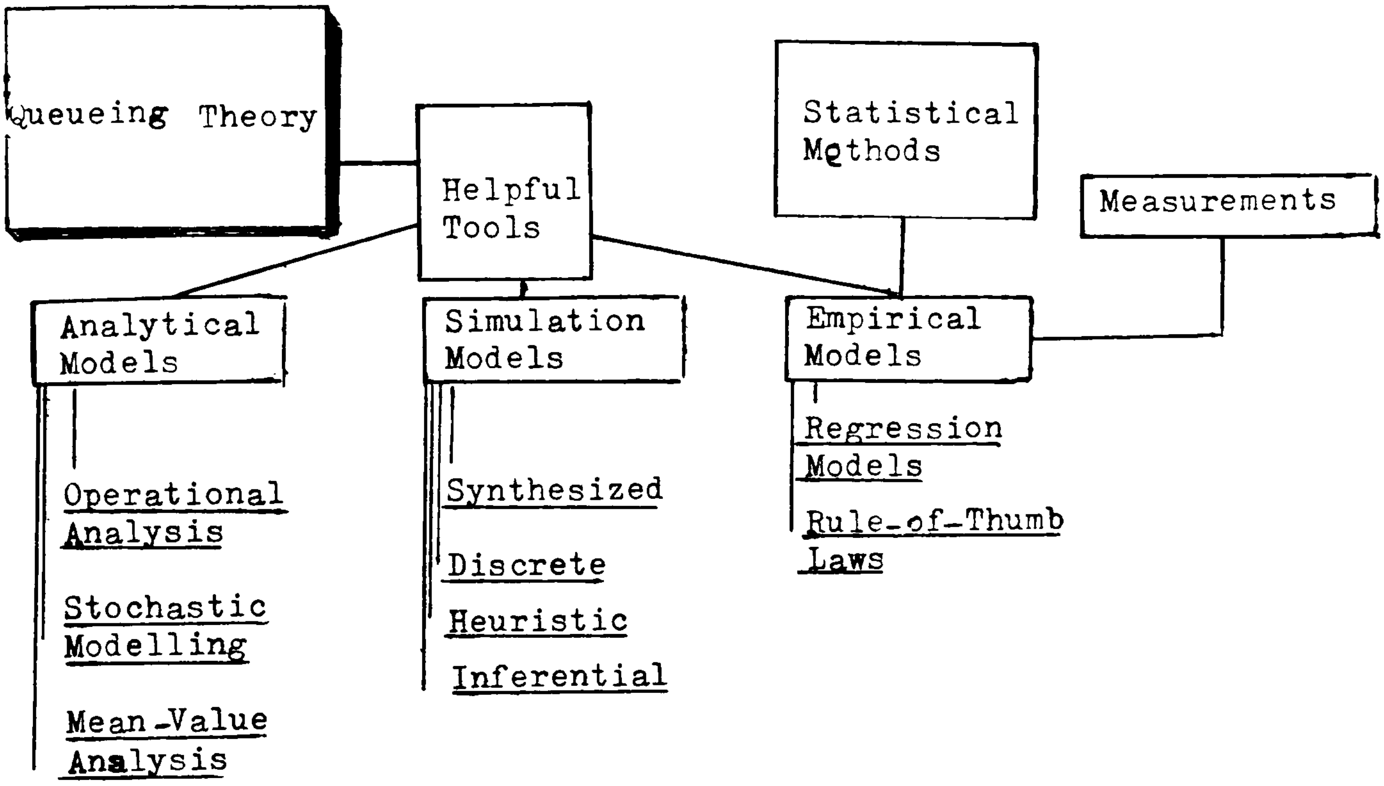


Figure 3.1: Queueing Theory Based Models

In the following section, we investigate the use of these models for computer system design as well as their associated problems.

3.2.1 The Analytical Models of The Queueing Theory

Analytical models represent system design and evaluation parameters strictly in mathematical terms. Indeed, certain researchers prefer this approach (c.f. Kobayashi 1978), for the following reasons:

- It is an economical method compared to simulation,
- It can be used to optimise the design variables, and
- It is quicker to produce results than simulation models.

This modelling approach, however, may have the following disadvantages (Farrari 1978):

- limited in scope,
- difficult to develop and build, and

- not easy to test the simplification assumptions.

The notable theoretical models derived from the principles of the queueing theory are:

1. **stochastic modelling** (c.f. Chandy and Sauer 1978),

2. **operational analysis** (c.f. Buzen and Denning 1980), and

3. **mean-value analysis** (c.f. Riser 1979).

Brief critical review of these attempts are given in the following sections.

3.2.1.1 Stochastic Modelling

This modelling technique considers the system as consisting of service centres among which jobs circulate. This analysis may also be called stochastic modelling or probabilistic modelling, since the servicing time of a job at a servicing centre is taken to be a sample from a specified distribution and the frequency by which the job will move to another servicing centre is controlled by a specified probability distribution. The stochastic modelling technique concideres the following definitions and hypotheses (Ferrari 1978):

Definition 1 *A stochastic process $x(t)$ is a function of time t whose values are random variables. The value of $x(t)$ at time t^* represent the state of the stochastic process at t^*. If each random variable take only a finite or a countable number of values, we have a* discrete-state process *or* chain. *Otherwise, we have a* continuous-state stochastic process.

Hypothesis 1 *The behaviour of the real system model during a given period of time is characterised by the probability distributions of the stochastic process if and only if the following assumptions hold (refer to Sevcik and Klawe 1979):*

1. *the system is modelled by a* stationary stochastic process,

2. *jobs are* stochastically independent,

3. *successive* transitions *among service centres are* independent,

4. *The system* reaches equilibrium,

5. *the system is* ergodic *(i.e. long-term time averages converge to the values computed for stochastic equilibrium), and*

6. *the network model must be* operationally connected *(i.e. each device must be visited at least once by some job during the observation period).*

If 1 and 2 were assumed and if the service time distribution at each centre is exponential then the system state (i.e. the number of jobs at each service centre) is a **continuous Markov process** (Kobayashi 1978). If hypotheses 4 and 5 were assumed then the system is at a ***steady-state equilibrium***, and long term performance measures can be computed.

Based on these hypotheses, a stochastic model can be defined and used for designing a computer system. Observable aspects of the real system model– e.g. states, parameters, and probability distributions– can be identified with quantities in the stochastic model and equations relating these quantities can be derived. Although formally applicable only to the stochastic process these equations can also be applied to the observable behaviour of the system itself (i.e. limited time), under suitable limiting conditions (Buzen 1978). The parameters of the stochastic process, representing the operation of the system, must be estimated from observations during a finite time interval. The specific formulae depend on what measurement data is available and on the amount of detail in the queueing network model.

In order to validate the model, the estimated parameter values are substituted into the performance measure formulae, and the results are compared to the corresponding observed values for a specific observation period. The most common purpose for which models are created is to obtain an indication of how a system will behave in the future, either after its configuration has been altered or its workload has been changed. In order to accomplish this, it is possible to employ the same computational formulae as in the validation of the model, by using modified parameter values in order to reflect the altered circumstances anticipated in the future. Once the future values of the model parameters have been estimated, the obtained formulae are used to calculate the performance measures. These are then interpreted as equilibrium performance measures of a stochastic process.

Stochastic analysis has, however, certain disadvantages (Denning and Buzen 78):

1. It is impossible to validate the stochastic hypothesis and conditions, hence an analyst can never be certain that an equation derived from a stochastic model can be correctly applied to the observable behaviour of a real system.

2. Stochastic analysis is an *inductive mathematical tool*: (it estimates unknown values from the projection period from values observed in the baseline period). Thus, one faces the problem of uncertainties in estimation of variables. (Note: this problem is not present in operational analysis, since operational analysis is a *deductive mathematical tool*).

3. Stochastic analysis can be applied to study a fairly simple and special class of computer systems design because the type of assumptions used by this analysis cannot be easily found in real systems (e.g. the assumptions of equilibrium or stochastic independence of successive service times).

4. Stochastic modelling may not be so easy to understand.

5. Stochastic modelling cannot be relevant to a real system. For example, in real systems transactions between devices do not follow Markov chains or processes, and service time distributions are not generally exponential (Von Mayrhauser 1979).

On the other hand, Stochastic models bestow certain benefits. Independent and dependent variables can be defined precisely, hypothesis can be stated succinctly and a considerable body of theory can be called on during analysis (Denning and Buzen 1978).

3.2.1.2 Operational Analysis

> "Operational Analysis is a framwork for studying the design performance of systems during given periods of time. The system may be real or hypothetical, and the time may be past, present or future" (Buzen and Denning 1980).

This kind of analysis was recently invented, about 1976 (Buzen 1976), to construct a precise mathematical tool to meet the following objectives:

1. Relate existing measurement data to other quantities that were not measured but which could, in principle, be empirically determined.

2. Verify the internal consistency of existing sets of measurement data.

3. Predict the effect that certain modifications to the system or the workload would have on measured quantities.

4. Be simple and easy to understand.

5. The tool should be based on testable assumptions.

The general idea of operational analysis (or operational method) can be shown in the following diagram (see Figure 3.2:

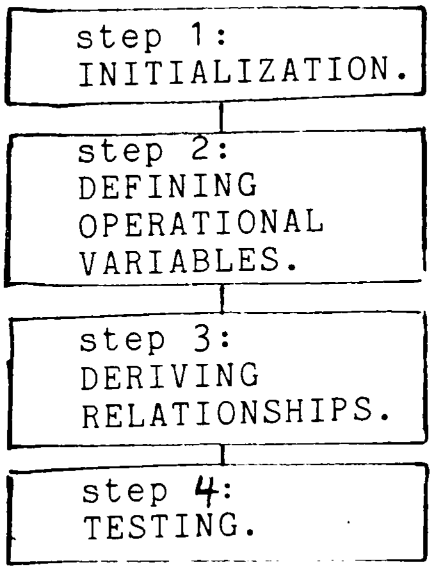


Figure 3.2: The Operational Method.

- *step 1: INITIALIZATION* In this step an observation interval is obtained: an interval of time during which system behaviour is monitored and measurement data is collected. The measured or computed quantities within the observation interval are called operational variables.

- *step 2: DEFINING OPERATIONAL VARIABLES* Defining the operational variables that directly affect the performance indices of interest.
- *step 3: DERIVING RELATIONSHIPS* The behaviour of the system is specified in this step by deriving the relationship between the operational variables. These relationships are represented by mathematical equations.
- *step 4: TESTING* At this step, the mathematical relationships are tested against the original objectives.

This method is considered by many researchers as equivalent or as an alternative to the traditional method of stochastic analysis (or Stochastic modelling) (Buzen 1976, Buzen 1978, Buzen 1976a, Denning and Buzen 1978). Other researchers find that this approach has several advantages to the traditional approach. These advantages can be summarised as follows (Sevcik and Klawe 1979):

- *Relevance to actual system*: The fact that operational analysis is based on observable quantities and testable assumptions makes it easier to relate to system measurements.
- *Understandability*: Operational analysis can be understood easily, even for large systems.
- *Breadth of applicability*: Since operational analysis depends on testable assumptions, it has a wide applicability as a modelling technique. Its major application areas are (Denning and Buzen 1978).

    1. Performance Calculation: Operational results can be used to compute quantities which have not measured.
    2. Consistency checking: A failure of data to verify a theorem or identity reveals an error in the data, a fault in the measurement procedure or a violation of a critical hypothesis.

3. Performance Prediction: Operational results can be used to estimate performance quantities in a future time (or indeed a past time) for which no directly measured data are available.

- *Testability of Assumptions*: Most of the assumptions of Stochastic analysis can neither be verified nor disproven in any finite period. While the assumptions of operational analysis can, in principle, be tested in finite time intervals.

To give an example on how the operational analysis treats computer system design and evaluation, we provided the following equations of a single server queueing system (c.f. Denning and Buzen 1978) (see Figure 3.3):

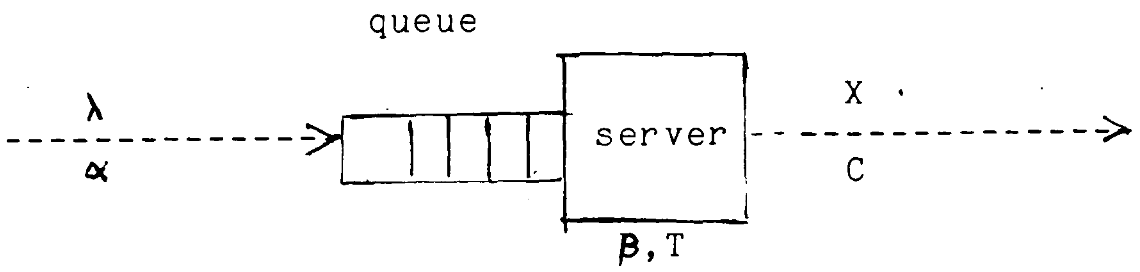


Figure 3.3: Single Server System

- *Primary Design and Evaluation Indices*

  T The length of the observed period

  α The number of arrivals occurring during the observed period

  β The total amount of the time that the system is busy during the observed period

  ν The number of completions occurring in the observed period

- *Derived Indices*

  $\lambda = \alpha/T$ the arrival rate (jobs/second)

  $\chi = \nu/T$ the output rate (jobs/second)

$\rho = \beta/T$ the utilisation (fraction of time system is busy)

$S = \beta/\nu$ the mean service time per completed job

- *Operational Equations*

Utilisation Law	$\rho_i = \chi_i \times S_i$
Little's Law	$\overline{n} = \chi_i \times R_i$
Forced Flow Law	$\chi_i = V_i \times \chi_o$
Output Flow Law	$\sum_{i=1}^{k} \chi_o = x_i \times q_{io}$
General Response Time Law	$R = \sum_{i=1}^{k} V_i \times R_i$
Interactive Response Time Law	$R = \frac{M}{\chi} - Z$

However, some researchers do not find this approach suitable for parameter estimation and anticipated design and modification (Muntz 1979, Sevcik and Klawe 1979, Buzen 1979) they express the opinion that,

> "the estimation problem is not really an integral part of either operational analysis or stochastic modelling. It is crucially important but an entirely separate issue"

At the same time, Buzen believes that the performance analysis offers major advantages over stochastic modelling in performance prediction.

Operational analysis uses queueing theory, in which case it is called *Operational queueing network theory* (Denning and Buzen 1977). The important reason why queueing theory should be used, is the speed with which performance quantities are computed using queueing network formulae. The operational queueing network theory may use some assumptions - e.g. flow balance, one-step behaviour and homogeneity, but these assumptions (as mentioned previously) can be tested for validity in any observation period.

3.2.1.3 The Mean Value Analysis

This is a new mathematical tool, used to calculate some important performance indices, such as mean response time, throughputs and queue length in closed queueing networks. A primary advantage of mean-value analysis over the traditional approach (i.e. Stochastic Analysis), is its improved numerical stability (Buzen and Denning 1980). This analysis uses the Sevcik and Mitrani (1978) arrival theorem to calculate the mean-value for successively larger loads N.

Riser (1979) found queueing networks with product-form solution[1] remarkably *robust*[2] with respect to routing and service time distributions. This robustness leads to the new mathematical explanation called Mean-Value analysis. Mean-Value analysis uses some basic equations which can be applied iteratively for any value of N.

Let

i	device number
K	number of devices
N	number of jobs
Q_i	overall mean queue length at device i
Qa_i	mean queue length seen by arriving customer at device i
$R_i(N)$	mean response time of device i $i = 1, \cdots, K$ given N jobs
$R_o(N)$	mean response time of the system given N jobs
$X_o(N)$	mean system throughput given N jobs
V_i	mean number of visits per job to the device i
$S_i(N)$	mean time between completions

definition

$$Qa_i(N) = Q_i(N-1) \quad \text{Sevcik-Mitrani theorem.}$$

Then the basic mean-value equations are

$$R_i(N) = S_i \times (1 + Q_i \times (N-1)) \tag{1}$$

$$X_o(N) = N / \sum_{i=1}^{k} V_i \times R_i(N) \tag{2}$$

Using the forced flow law, we get

$$X_i(N) = V_i X_o(N) \textit{forced flow law}$$

Where

$$X_i(N) = \text{ throughput at device i given N}$$

[1] gives the joint queue-size up to a normalisation constant. This constant has a simple analytic expression in the case of open queueing networks but is a sum of product terms of closed system

[2] statisticians call a system robust if only the mean enters into the solution

we get

$$Q_i(N) = R_i(N) \times V_i \times X_o(N) \qquad (3)$$

$$Where \quad i = 1, \cdots, K.$$

Equations (1), (2) and (3) can be used iteratively, once the values V_i and S_i are given. The iteration begins with N=1 and the boundary condition $Q_i(0) = 0$.

It is clear that this type of analysis uses no normalisation constant to calculate the important performance indices, and hence the formulae have a simple mathematical structure. This criteria is not available in the two previous analytical methods (i.e. Operational analysis and Stochastic analysis).

Some ideas of extending Mean-Value analysis were given by Buzen and Denning (1980) and by Riser and Lavenberg (1980), to which the reader is referred to for further information.

3.2.2 The Simulation Models of The Queueing Theory

Queueing theory models involve generally a large amount of computations and the support of simulation software tools are unavoidable, so every new queueing model is followed by its simulation counterpart. Indeed with simulation, we avoid tremendous difficulty in which many theoretical equations results are obtained within a very reasonable time (e.g. normalisation constant calculation in product-form models). For this reason, in our opinion, more efforts were recently directed towards more effective computational (i.e. simulation) algorithms for already existing theoretical models than towards creating conceptually new theoretical models (see Figure 3.4).

Although, the simulation models can be classified according to their theoretical counterpart, they do differ in way they represent workload. Mohamad and Cavouras (1984, 1982) introduced such type of classification, which consist of the following elements:

1. *Discrete Activity (event) -Oriented*: model parameters, including the workload, are derived from probability distributions (Overstreet and Nance 1985),

2. *Heuristics-Oriented*: model parameters based on the heuristic approaches are determined based upon certain prediction formulae or using certain basic rules such as

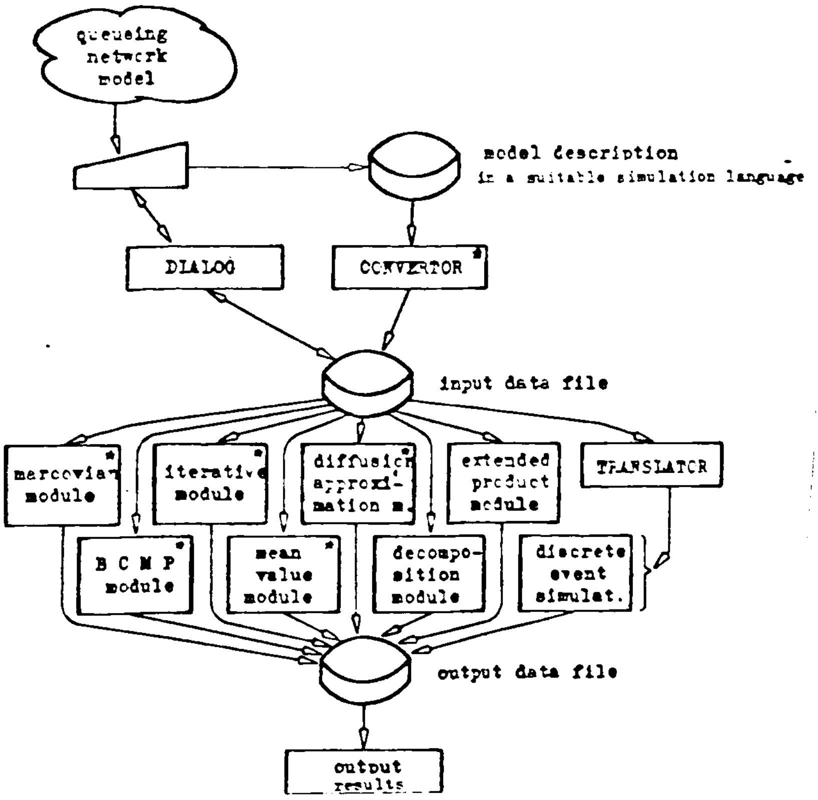


Figure 3.4: An Example of Queueing Networks Simulation Tools

system comparison to determine the more desirable alternative (Kimbleton 1975),

3. *Synthesised-Oriented*: model parameters (the workload in particular) are determined according to the features found in a real-life sample (Haring et al 1978, Curnow and Wichmann 1975).

4. *Inferential-Oriented*: model parameters are determined according to a the basic information in a knowledge base and the inferences made by the inferential unit. This is quite new approach proposed but not yet implemented by Harvard University (Levine 1984).

We do not want to discuss the advantages of each type of simulation separately, since they are basically similar in being based upon the same theoretical ground (the network models of the queueing theory). But, it is important to survey the notable attempts that have been made to simulate the different kinds of the queueing networks.

GPSS uses a queuing network representation, as do the activity- cycle-based languages (Hutchinson 1975). Interestingly, Nygaard and Dahl (1978), in discussing the development of Simula, state that early in its design Simula was to be a queueing network- oriented simulation language. This approach was dropped, however, when the developers become convinced of its lack of generality.

Several authors have suggested the process concept (Blunden and Krasnow 1967) or Simscript's entity attribute set approach as basis for modelling the queueing networks (Markowitz 1979). While neither provides a theory supporting the simulation process, both provide powerful representational and conceptual tools for model specification.

Zeigler's work is the most significant effort to provide a sound theoretical basis for simulation (including the queueing networks). Based on general system theory, which in turn is based on finite state machines, this approach provides powerful conceptual tools for dealing with the dynamics of simulation process, including the concept *"model state trajectories"* . Also, Zeigler's *"experimental frames"* provide both theoretical basis and some practical guidance for dealing with model validation (Oren and Zeigler 1979, Zeigler 1984).

Kindler's set-theoretic approach provides a basis for a categorisation of models, systems, and simulation models, although the impact on the practical issues of model development, validation, and verification has yet to be developed (Kindler 1979).

Program generators have been used for more than a decade to assist in model implementation. A program generator typically consists of a component to build a model specification which is then used by another component to generate code in a particular simulation language. Mathewson's DRAFT systems (1977) uses a family of generators (one for each target simulation language) to produce programs based on activity-cycle diagrams. Davis' approach is to build a

> "simulation-independent description of a situation" (Davies 1976).

. Support systems for model development range from the simple expedient of programmer checklists (McLeod 1973) to GASP (Pritsker 1974) and Visontay's DOCUM program for Simula (Visontay 1979). Zeigler et al have an interactive system to assist a modeller in the construction of model object descriptions (Zeigler 1980). Oren's GEST language (Oren

1984) provides a clear separation of model specification from the monitoring of the simulation study. The latest version of the language (Oren 1984) refers to Zeigler's experimental frames (Oren 1979). The most ambitious attempt in this area is the Delta project, which seeks to allow a modeller to develop a complete executable simulation program (Holbaek-Hanssen 1977).

Several authors discuss a formal simulation model specification and documentation language (SMSDL), first defined in (Nance 1971). Kleine describes an SMSDL which, by progressive refinement, is intended to lead to executable Simscript programs (Kleine 1977). Frankowski and Franta propose a process (and Simula) oriented SMSDL (Frankowski 1980). As with Kleine, a specification evolves into an implementation; the same simulation language is used for both.

Little evidence of analytic techniques to assist in construction of efficient simulation model implementation is found. DeCarvalho and Crookes describe analyses to improve the efficiency of an activity-scanning time flow mechanism and to identify components whose output can be saved and reused in subsequent executions (DeCarvalho 1976). Schruben analyses *"event graphs"* in order to simplify a model specification and to identify other properties of the model (Schruben 1983).

As the domain of model specification encompasses that of software specification, they are closely related. Advances in either area are likely to benefit both. The similarity is particularly strong in approaches such as that of the JADE software development project. JADE uses a development methodology based on the modelling and simulation of a proposed system; the model is refined until it becomes the software system (Unger 1983). Balzer, Cheatham, and Green argue for a new paradigm for software development which is based on the use of a high-level formal software specification which is then transformed, at least partially automatically, into an implementation (Balzer 1983). In Lehman's categorisation of programs, all model implementations fall into the more difficult 'A' classification (Lehman 1980).

To conclude, several packages appeared in the literature based on these methods and techniques that simulate variety of queueing networks (a list is given without references because they are simply quite allot): RQA, MARCA, QSOLVE, ASQ, BCMP, QNET4, SNAP, PNET, CHW, CADS, IQNA, QSIM, APLOMB, RESQ, BEST/1, QNAP. Diethelm (1977), Ross (1976), and Bhandiwad and Williams (1974) validated the accuracy of predictions for most of the above simulation models and prove that 6 to 28may involve in their use.

3.2.3 The Empirical Modelling of The Queueing Theory

This method represents an alternative to the modelling techniques described above in the last two sections. These methods are appropriate when performance or measurement data of (an) actual system (s) are available. Statistical methods use these data to forecast future performance. A perfect example of this approach is given by Gomaa (1976) in which he defined several queueing network laws using the regression analysis.

Empirical data can be obtained through measurements, may be from an actual system or from a queueing model of a system. The collection of these measurements can be performed with hardware monitors, software monitors (or probes) and accounting packages. The reader interested in measurement techniques is referred to Brad (1971), Williams (1972), Lunde (1977), Robinson and Torsun (1977). This approach have limited success according to the area where the measurement (probes or monitors) are concentrated.

3.3 Conclusions

To conclude, there is no general-purpose theoretical approach that can be used for modelling highly constrained computer systems within the science paradigm. In particuler, there are many design difficulties associated with the queueing theory; the widely used design theory. Indeed, there a are vast number of science-paradigm theories, but these are either unknown or can not be used for modellling general-purpose computer systems. The problem in the auther opinion can be solved by the development of a methodological approach which uses some theoretical and practical computer system design notion. The reason for this conclusion can be depicted from the quotation of Butler Lampson (1984)[3]

[3] A senior consultant designer of several Sucessful computer systems

"Designing a computer system is very different from modelling an algorithm; the external interface– that is, the requirement– is less precisely defined, more complex, and more subject to change; the system has much more internal structure– hence, many internal interfaces; and the measure of success is much less clear. The designer usually finds himself floundering in a sea of possibilities, unclear about how one choice will limit his freedom to make other choices or affect the size and performance of the entire system. There probably isn't a best way to build the system or even a major part of it. Much more important is to avoid choosing a terrible way and have a clear division of responsibilities among the parts."

Chapter 4

INTRODUCING ACTIVITY STRUCTURES: A METHODOLOGICAL APPROACH FOR CONSTRUCTING HIGHLY CONSTRAINED EFFECTIVE COMPUTER SYSTEMS

4.1 Introduction

As seen in the last two chapters, current computer systems design methodologies have evolved dynamically from the experiences of the past forty to fifty years and represent a motley collection of nearly isolated theoretical methods and techniques, linked together through an experience-based, but otherwise arbitrary, sequence of much discussed process phases within the science paradigm. Hence, there is a sameness to the design of all computers based on these isolated methods and techniques so that

> "only rarely does a new design methodology appear in the real world of computer design" (c.f. Allison 1977).

Obvious answer is to get rid of the von Neumann architecture and build a more homogenous computing machine in which memory and processing are combined. It is not difficult today to build a machine which hundreds of thousands or even millions of tiny processing cells which have a raw computational power that is many orders of magnitude greater than the faster conventional machines. The problem is in how to couple the raw power with the application of interest, how to program the hardware for the job. How do we decompose our application into hundreds of thousands of parts that can execute concurrently? How do we coordinate the activities of a million of processing elements to accomplish a single task? This chapter provides an answer to these questions.

In this chapter the important consideration in the construction of computer systems is the entire *construction environment.* In its most general sense, the construction environment includes the human design activities, the technical methods, the management procedures, the computing equipment, the problem elicitation, the requirement realisation, and the automated tools to support the construction environment.

At the hart of the environment is a construction methodology, which deals with the construction of a *system* through its specification, design, development, operation and evolution, including human design activities. The construction point-of-view that has been used in this chapter to deal with the identification of effective construction methodology for computer systems is based on the Activity Structures scheme (c.f. Kohout 1986).

Activity structures scheme provide several useful design concepts/constructs that are derived not only from the science paradigm but also from the general/meta systems paradigm. The main contribution of this scheme is that it provides a total design framework for constructing effective design environments. Our methodological approach, however, represents an extension to the original scheme of activity structures, which is found to be effective for designing highly constrained computer systems. After a few sections intended mainly to introduce certain basic definitions and analogies, we shall give a brief account of the basic assumptions within which the design of highly constrained computer systems can be realised.

4.2 The Cornerstones

> "Computer design may involve selection from among competing designs (especially in the case of an experienced designer) and requires identification which involves consideration of the system's structure and function (activity)" (Davis et al 1983).

In general, solutions to identification problems can be either selected from a set of pre-enumerated alternatives (for known conditions) or constructed (for novel problem or the ones that combine multiple, interacting disorder in an unforeseen way). While computer design is often thought of as a constructive problem-solving process, identification is typically thought of as a selection or a classification problem. But the solution methodology is not inherent in the task itself. Instead, it depends on the problem solver' design knowledge, requirements for customisation, and the like (i.e. human behaviour computer design model).

The complexity arising in the construction of human behaviour design models is due mainly to the lack of our knowledge about its constructive mechanisms (Kohout 1976). The study of the design process underlying human activity initiated mainly by researchers operating within the general and meta-systems design paradigms (Svoboda 1964, Kohout and Gaines 1976, Gaines 1977, Bandler and Kohout 1979). It was a research issue originated from many diverse, otherwise unrelated fields, such as the studies of human movement control, linguistics, psychology, neurophysiology, scientific and engineering system theoretic studies, etc. The most fruitful ideas for the construction of design models that can include human behaviour were given by Ladislav Kohout in a series of research papers (started from 1974 and still evolving, e.g. Kohout 1986, 1986a) on the establishment and the identification of a methodological approach for studying human actions (named in 1979 as the *activity structures methodology*). Originally, Kohout's ideas were presented as a formal framework for the representation of actions both at an intentional and detailed perceptual-motor coordination level (refer to Kohout 1976). Later this methodology has been extended and applied to conceptual and structural design of several sophisticated information processing systems, e.g. *expert systems* (Kohout 1982, Kohout et al 1984, Mohamad et al 1983), a *decision support system* (Kohout et al 1985, Ohiorenoya and Mohamad 1983), a *library transaction system* (Kohout et al 1984). All these designs highlight the generality and importance of the activity structures design methodology. Hence, it is our intention in this thesis to extend and use the activity structures methodology for designing computer systems that fulfil our motivation (c.f. Chapter 1). Indeed, for the author who worked previously over three years in designing effective computer systems (c.f. Mohamad 1981), activity structures represent solutions for for several problems associated with the current techniques and theories of modern computer systems design.

4.3 General And Meta Systems Paradigms Versus Activity Structures

General systems theory (von Bertanlanffy 1968) and the *Metasystem design* frameworks (Beer 1972, Klir 1976, Kickert 1980, van Gigch 1984) have had much influence, but have not really been concerned with practical systems design.

> "The general/meta systems approaches have been developed long before they achieved their importance in computer science" (Zemanek 1980).

Historically, these approaches have been originated from the research e.g. in telecommunications technology and biology. For telecommunications technology the general/meta systems theory was intoduced by Karl Kufmuller (1949), whereas for biology, it was introduced by Ludwing Bertalanffy in 1945 (see Bertalanfy 1968). These approaches represent an attempt to come to terms with, and to understand, the nature of systems. They are really methods for theoretical model building used for the explanation of the behaviour of complex and diverse systems.

Since their onset many researchers have tried to apply their techniques for solving practical problems. Some researchers believe that it can provide many fruitful design and synthesis paradigms (van Gigch 1979). Other researchers reported moderate success (Checkland 1975), but the majority of practical applications have been notably unsuccessful (Lilienfeld 1978). The reason for this lack of success is that the very generality of these design paradigms makes it difficult to use them, and to develop a concrete methodological solution; and where occasionally a good solution is arrived at, it is often one that requires a technological revolution to implement. It is not a process which would permit small incremental changes but one which more usually results in the complete reassessment of structures, roles and behaviour (Wood-Harper et al 1982). Thus a system designer considers the application of the general systems theory and the metasystem framework too impractical and wide ranging for this purpose.

However, activity structures have sought to come to terms with this problem and to make the general systems theory and the metasystem framework more practical for problem solving. They have striven to convert these paradigms into a practical methodology by *firstly*, breaking down the process into a number of defined steps to be followed and *secondly*, seeking to limit the range of the alternative solutions by introducing notions such as the identification of certain *general constructs* within which the problems must be set (Kohout and Gaines 1976, Kohout 1976, Kohout 1981).

4.4 From Neuroscience Research To Computer System Design

Historically the research in Activity Structures methodology has emerged from the Neuroscience research, aiming at providing the techniques for combining diverse knowledge sources that capture the *"deep knowledge"* of the application field in an effective formal and computer representable form (Kohout 1976, 1977).

The question that may be asked here is what are the reasons behind selecting the activity structures schemes knowing that it has been originated from neuroscience research an how such kind of research field can help in designing effective computer systems? Part of the problem that concerns the design of computer systems is that we do not yet fully understand the algorithms of thinking[1] (c.f. Palm 1982)). But part of the problem is the use and control of the speed. One might suspect that the reason the conventional computer system is slow is that its electronic components are much slower than the biological components of the neurological system or the brain, but this is not the case. A transistor can switch in few nanoseconds, about million times faster than the milliseconds switching time of a neuron. A more plausible argument is that the neurological system has more neurons than the conventional computer has transistors, but even this fails to explain the disparity in speed. As near as we can tell,

[1] "Design belongs to those human activities which cannot be totally described because their roots are in the unconscious processes in our brain" (Zemanek 1980).

> "the neurological system has about 10^{10} neurons, each capable of switching no more than a thousand times a second. So the neurological system should be capable of about 10^{13} switching events per second. A modern digital computer, by contrast, may have as many as 10^9 transistors, each capable of switching as often as 10^9 times per second. So the total switching speed should be as high as 10^{18} event per second, or 10,000 times greater than the brain. Thus the sheer computational power of the computer should much greater than that of the human brain. Yet we know the reality to be just the reverse" (Hillis 1985).

Similar opions to Hillis's have been expressed by several other researchers (Lamport 1985, Rosen 1986, Grossberg 1982, Kleinrock 1984).

In this thesis we assume that computer system design can benefit from the experience of methodologies and formal approaches, modelling the neurological information processing system, such as the original formulation of the activity structures scheme. Design by analogy has been quite common practice both, in the old days and at present. Turing (1936), for example, who was motivated by the knowledge in biology and psychology, delimited the behaviour of any computer, in the sense of a human making calculation according to some well-specified rules. It was a good analogy and a good start, but not so effective for designing effective computer systems for the present days. However, the activity structures represent not onlly anologies but also formal isomorphisms of some of the Neurological and computer models. Recently some researchers have expressed believe that possibly, by returning to the classical approach we will gain construction effectivity. For example, Alfred Spector and David Gifford (1986) advise us to look at the civil engineering and its success in designing bridges and take this analogy as a cornerstone for developing effective design principles for new computer systems. But this classical analogy proves to be wrong in physics and other fields, and we expect that it will fail in the area of designing effective computer systems– it does not provide, for example, the self-regulating criteria.

However, there are two successful *new* computer design schemes that are based upon neurological science understanding. The first utilises a biological tissues to design what is called *molecular computers* (Conard 1985, Friedland and Kedes 1985), and the second starts a new area in computer design which incorporates the brain research as one of its fundamental design concepts, it has been called the *sixth generation of computer systems* (Gaines 1978, Gaines and Shaw (1986, 1986a)). Indeed, activity structures design approach can be considered as a class within the second scheme, since it is based originally upon the concepts of the brain research. The next section introduces the basic concepts of activity structures as it was compiled by the present author to suite computer systems design.

4.5 Activity Structures versus The Other Neuroscience Modelling Disciplines

Indeed, the discipline of designing effective systems based on the knowhow of the neurological science is still very young and in common with most other emerging disciplines it occasionally enters periods of radical self examination and re-thinking. The reason for the current turmoil in this discipline is the emergence over past few years of a number of new approaches and methodologies. The author feels that we are in the midst of such a phase at present; new ideas abound, arguments rage, and the development of technology is a powerful impetus to the re- examination of ideas. However, we believe that the existing approaches inherited the drawbacks of their originating paradigms (i.e. either the metasystem or the general systems) and represent a confusing array of modelling methods. This judgment can simply be formed by looking at a short list of the neurological science based modelling approaches:

- The Field Theory of Self-Organising (Amari 1983),
- The Extended Automata Theory (Arbib 1975),
- The Cybernetic Modelling (Nurmi 1978, Carlsson 1979),
- The Stability Analysis (Perlis and Ignizio 1980),
- The Structural Modelling (Lendaris 1980),

- The Structural Decomposition of Dynamical Systems (Jacak et al 1985)
- The Functional Modelling (Baylin 1984),
- The Dynamical Inferences (Jugeli 1980),
- Brain Modelling For Robotics (Andreae and Cleary 1976),
- Simulation of Human Thoughts (Szymanski 1980),
- Statistical Simplification of Neural Nets (Zeigler 1975),
- Casual Structures in Brains and Machines (Rosen 1986),..., etc.

It is the author's view that these approaches are not simple alternatives, but that each approach seeks to do different things than the other ones. However, we believe that the methodology of activity structures represent quite superior to all these approaches. Activity structures is a methodology designed to provide an integral system to support the technology architectures whose processing environments are changing. The design framework of the activity structures provides the essential design and construction steps, the essential structures, the main interfaces, as well as many other features that are essential for the success of any and all applications.

4.6 The Concepts of The Design Methodology of Activity Structures

An activity structures-based design provide an total information processing system which support *self-regulating architectures* whose processing environments are dynamic and *operating under maximal constraints.* The design framework of the Activity Structures identifies the necessary processing environments and provides the structures, as well as their linkage interfaces, that are seen to be essential for the success of any and all applications.

Figure 4.1 schematises the role of activity structures in the construction of (a knowledge representation) for some target system within a given domain of application.

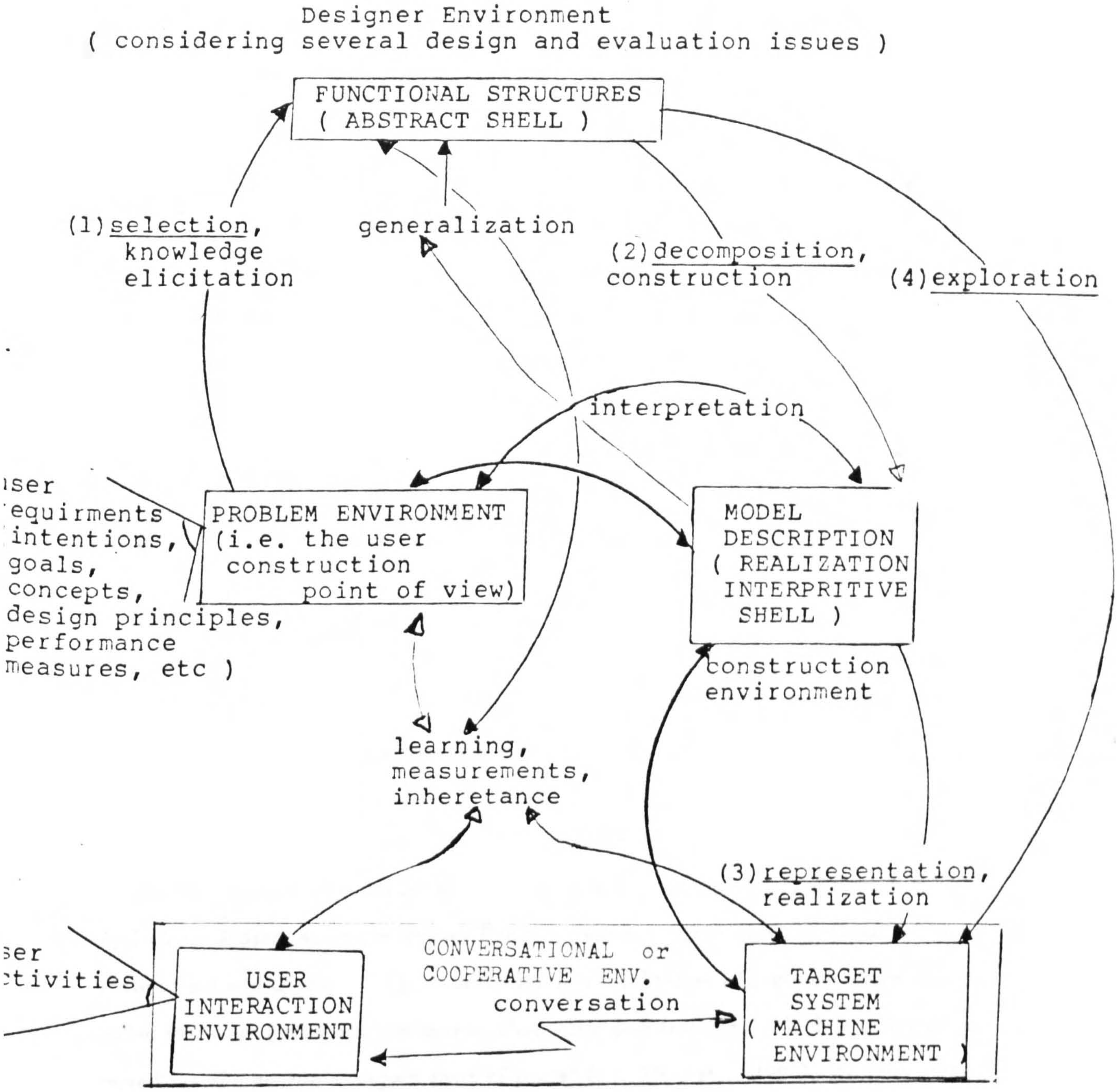


Figure 4.1: The Design Scheme of Activity Structures

Imagine being given a new system to design, in which the system structure is unknown (only the given requirements are known). The process of dealing with it can be broken down into four phases which, although different conceptually, usually overlap in practice. The breakdown seems to correspond to what we do intuitively when presented with a strange problem to solve. This process involves four design steps: *selection*, *decomposition*, *realisation*, and *exploration*.

4.6.1 The Selection Step:

This step involves three types of the designer activities. *Firstly* to elicit the design requirements from the problem environment. *Secondly,* to select those user requirements that are judged to be relevant, and discard the irrelevant ones. *Thirdly* to select the relevant design functionalities (sometime called strategies, metaphors, constraints or missions (Baylin 1986, Carroll and Thomas 1982)) that can achieve the user requirements and simultaneously enforce the designer constraints (these are referred to as the *functional structures*, behavioural models or the structures of behaviour (Kohout 1976)).

Formally, a functional structure can be represented by a single or a series of *relations* and the transformation can be performed on these relations using the *relational products* (Bandler and Kohout 1980). Indeed the concept of functional structure and their transformations has been highlighted as a very important idea in neuroscience based modelling by several researchers (c.f. Kohout 1976, Bernstejn 1967) and it is associated with the cortex structures and the behaviour the cortex generates. In this case a knowledge base located in neurons store information by using a sequence of deoxyribonuclic acid (DNA) molecules. Indeed, since the human Brain activities are goal-directed, these neurons cooperate to achieve that goal. The cooperation is a *mechanism of communication* that utilises coded messages (using the messenger ribonucleic acid (mRNA)) which is delivered and received via the neuron synapses (sort of ports) (c.f. Mori et al 1985). Activity structures, distinguish the following essential functional structures that were obtained from the analogy with the neurological system (Kohout 1976, Kohout and Bandler 1982, Kohout and Mohamad 1986)):

1. Design Structures:
    (a) the *knowledge representation structure*,
    (b) the *inferential structure*, and
    (c) the *control structure*,
2. Constraint Structures:
    (a) the *protection structure*,
    (b) the *communication structure*, and
    (c) the *interpretation structure.*

The role of these particular functional structures in designing computer systems as well as other details are given in the next chapter and will not be discussed anywhere in this chapter.

4.6.2 The Decomposition Step:

The designer second activity is to define an ordering of partitioning events that are necessary for the activity structures-based computer system development. While the activity structures methodology does not give any insight on how one *"thinks"* of a system being developed and partitioned (i.e. it does not define the intellectual building blocks used to construct a particular system conceptualisation), it does provide a description of the segmentation of various functional activities (i.e. the functional structures) and refinment transformations that occur during the development process and the way these functional activities interact.

The key point about the activity structures partitioning is that this partitioning segments the *development of the activity structures* based system into an optional number of sequential phases, depending on the level of design abstraction required. The activity structures methodology is distinct from any other particular design methodology in that its activity structures provide ordering on the optional phases (i.e. they describe what needs to be done in each abstraction level in order to define a system, and when it should be done).

The criteria that were used for defining the development rules as well as the optional phases consist of two requirements: the ***inter-phase independence*** and the ***intra-phase dependence***. Inter- phase independence requires that the functional structures (or the functional substructures) defined for each optional phase of an activity structures-based system, are independent of the functional substructures defined for any other optional phase of the activity structures based system design, except they are linked by an experience factor. If the optional phases are ordered by the experience factor then the top level will handle the goals and the following levels will handle other intrinsic duties such as the identifications of tasks, semantic, syntax, lexical, alphabetical and the physical structures (c.f. Nielsen 1986). Intra-phase dependence requires that all the functional substructures within a particular optional phase are related to each other. There are two reasons why we have chosen these two partitioning requirements. First, we want the designer to be able to segregate those design activities that can be performed in isolation from the other design activities. Secondly, by isolating only dependent design activities, we can define models, tools, and methodologies that can be optimised with respect to address restricted tasks.

The reader should note that, unlike some other logical structuring schemes (such as Constantine's Structured Design, c.f. Myer 1978), we do not suggest that each phase performs a single task. Instead, by grouping related tasks together, interaction is enhenced among common functional substructures. In fact, if one must err in devising system partitionings, we feel better to group unrelated tasks together in a single phase than to define phase boundaries too strictly. This is because, as a practical matter, it is better to foster inter-phase communication than to promote isolation between groups that should work together.

An interesting aspect of the segmentation of the optional development phases based on sequential ordering and phase dependence/independence is that feedback of system development should not occur between phases, but rather within a correctly- defined development optional phase.

In modelling an activity structures-based system, then, two activities of the designer needed to be considered for partitioning. **An individual phase model of development**

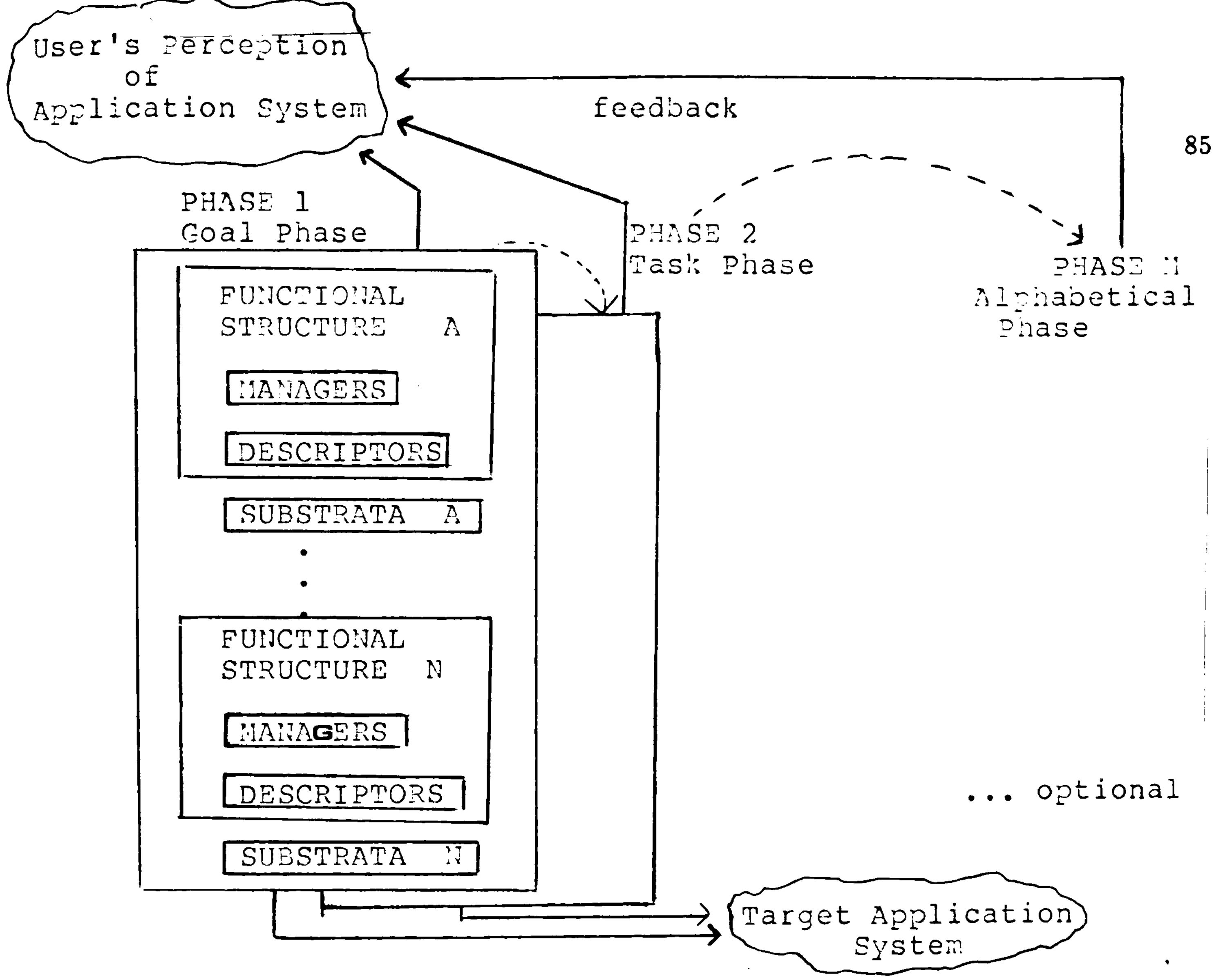


Figure 4.2: Partitioning the Development Cycle of Activity Structures.

will define how with a given user's perception of an application system progresses towards a particular implementation could be reached. At the same time a **macro-development model** exists (see Figure 4.2), which defines how systems functionalities are refined during the development process if further detail level of abstractions are needed for example by using more phases in the series. The choice of a phase within a model of system development corresponds closely to the definition and selection of scopes used in the development of *"programming environments"* (Osterwiel 1981) or the use of partitions in the development of *"blackboard systems"* (Craig 1986).

However, the task of *partitioning system within a given phase*, starts by decomposing the functional structures both semantically and syntactically into smaller functional components called the functional substructures. This decomposition process is recursive and it stops until no further decomposition is required or can be reached (terminal functional substructures). Indeed the non-terminal functional substructures are undefined entities open to refinment or decomposition. So an activity structures-based model before the decomposition

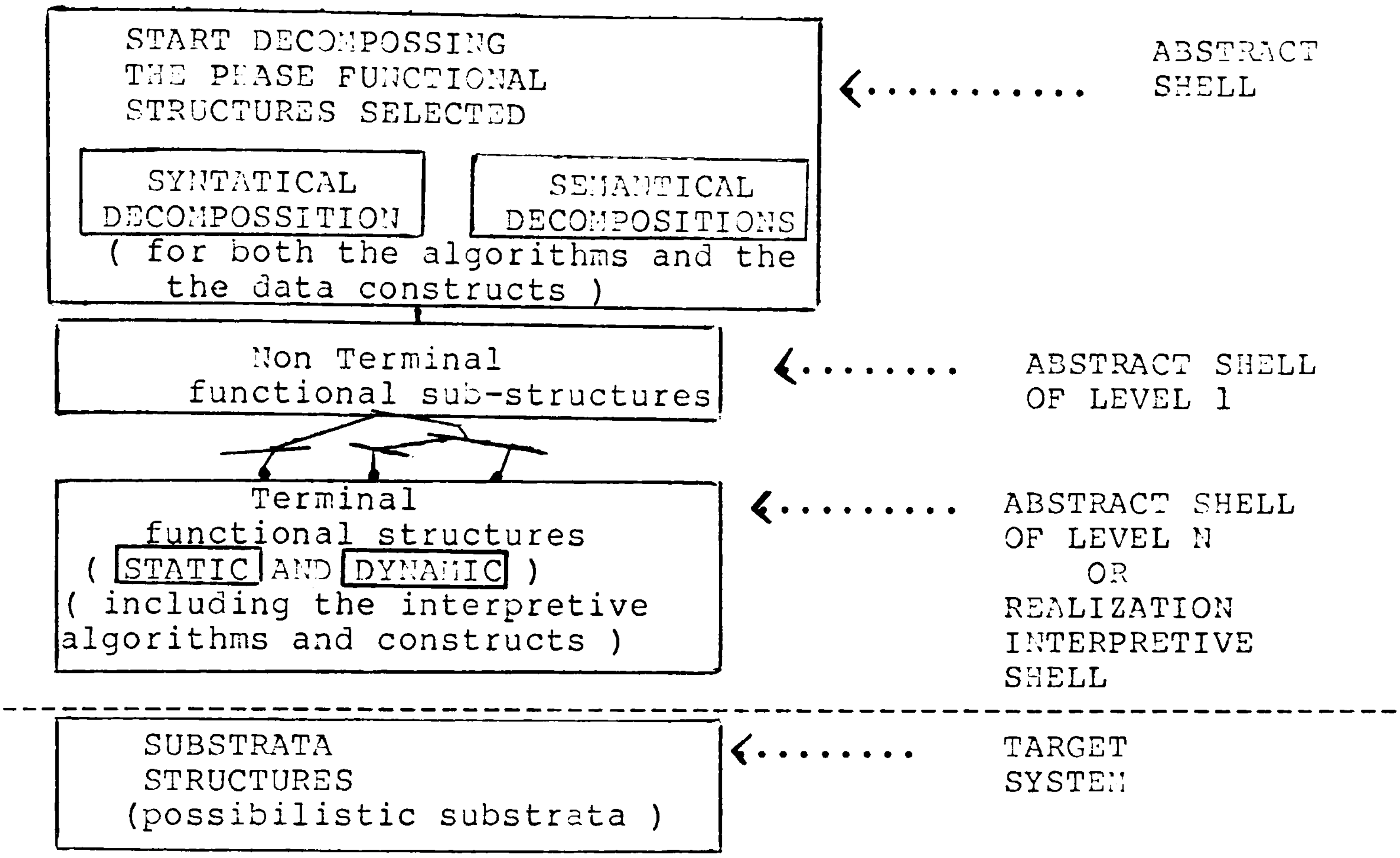

START DECOMPOSSING THE PHASE FUNCTIONAL STRUCTURES SELECTED
SYNTATICAL DECOMPOSSITION
SEMANTICAL DECOMPOSITIONS
(for both the algorithms and the the data constructs)
ABSTRACT SHELL
Non Terminal functional sub-structures
ABSTRACT SHELL OF LEVEL 1
Terminal functional structures
(STATIC AND DYNAMIC)
(including the interpretive algorithms and constructs)
ABSTRACT SHELL OF LEVEL N OR REALIZATION INTERPRETIVE SHELL
SUBSTRATA STRUCTURES (possibilistic substrata)
TARGET SYSTEM

Figure 4.3: Activity Structures Forms Before and After Realisation

step represent a partially defined skeleton with elements to be filled in (c.f. Kohout 1983).

When the decomposition step is performed the detailed algorithmic specifications of the system will be known as well as their interpretive constructs. These algorithms and their associated interpretive constructs will be interpreted in the next step to the machine environment and replaced at the end of the realisation (or the representation) step by the (possibly matching), machine dependent structures called the **possibilistic substrata structures**. All the terminal functional sub-structures form a **realisation interpretive activity structures shell**, whereas, in contrast, the top level non-terminal functional sub- structures form the **activity structures abstract shell**. We should note also that the terminal functional sub-structures can be further classified into **static functional sub-structures** and **dynamic functional sub-structures** depending on whether the realisation interpretive shell is operational (see Figure 4.3).

Kohout (1976, 1978) recommends Klir's epistemological hierarchy (Klir 1969) as a typical procedure for the decomposition step for each given development phase. For this purpose, the system is composed of several conceptually distinct levels:

Level 0 **A free system**: At this lowest level, a system is defined by a set of potential states (values) associated with each variable (source system) and by the description of the meaning of variables in terms of some attributes (object system). At this level, we define the basic *"alphabet"* of description and its semantics without any restriction imposed upon this alphabet.

Level 1 **A data system**: Here the free system of the zero level is supplemented by data (which restrict the range of possible states of variables).

Level 2 **A generative system**: At this level, a system is described by its intention, the generative component (i.e. a functional structure) is defined in terms of variables of the free system and component constraints upon their values. These generative structures are decomposed into several parts (functional sub-structures).

Level 3 **A structure system**: At this level, a system is described by a set of functional sub-structures of level 2 together with their interfaces. In each structure system a certain design 'level of abstraction' is reflected.

Level 4 **meta-system**: Here a system is defined as an inter-related collection of structure systems. Further higher levels are defined by recursion. At this level the functional sub-structures decompositions are completed.

Here we should note, that in contrast to the decomposition step, it is the reduction step which may be needed by the designer to polarise or concentrate upon identifying, with high resolution, some specific activities or constructs.

4.6.3 The Representation Step:

This is the third activity of the designer in which the description details of both, the algorithms and their constructs produced by step 2, need to be interpreted and realised in a suitable implementation dependent structure (i.e. using the substrata structures, for example via using some particular C programming language constructs, and certain data types). The most important condition of the choice of suitable substrata structures is that they must be

1. **extensible** (i.e. possibilistic substrata) to allow various kinds of interpretations to be performed (via the exploration step) and

2. provide a suitable distribution environment for the functional structures in order to let their behaviours to be performed concurrently.

These two substrata requirements represent certain performance factors. The flexibility criterion allows the designer to avoid the *"semantical gaps"* (c.f. Meyer 1981) that could be generated when the interpretive algorithms and their constructs are translated to the substrata structures. The distribution criterion speeds up the rate of executing the various behaviours resulting from the different functional structures, and hence improving the overall performance of the activity structures based system. Finally, the representation process must incorporate both the static and the dynamic descriptions of the interpreted algorithms and constructs. Also, for the purpose of the exploration, the substrata structures must contain certain monitoring probes within their description.

4.6.4 The Exploration Step:

In this step the designer tries to vary all the extensible or possibilistic substrata structures according to a *well-synthesised generated input sequence of events*, in order to force the target system to exhibit the interesting and required behaviours (i.e. reaching the desirable states). For this purpose, the designer should *simulate* the main activities of the conversational environments (i.e. the user interaction environment and the machine environment). The conversational environments activities should replicate the real word activities in that they must be random as well as having the capability of survival (using certain learning mechanisms).

The exploration step may force the designer to try different decomposition strategies in

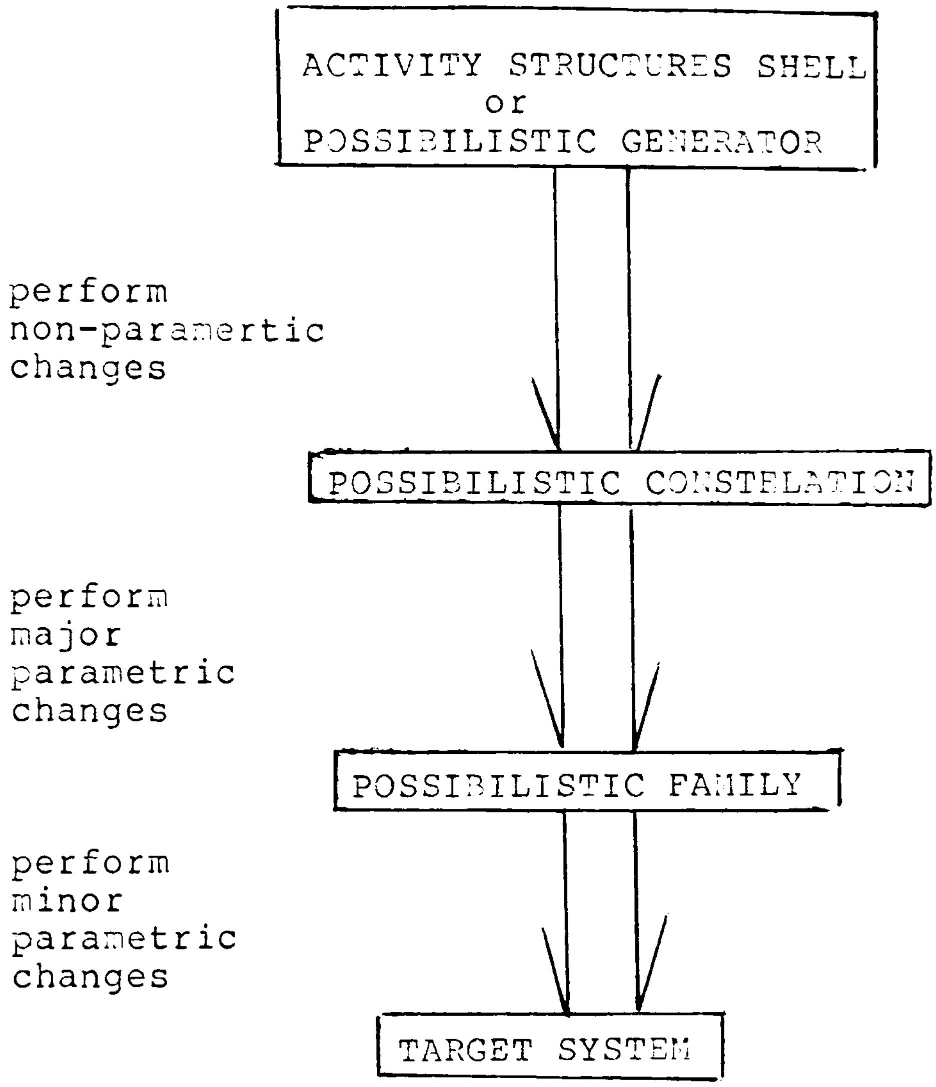


Figure 4.4: The various exploration changes on a shell

order to derive the resulting construction to behave in an interesting way (i.e. reperforming step 2). For the purpose of illustrating the various decomposition changes, we distinguish between the *algorithmic changes* which require the designer to perform certain major changes in the way the different functional substructures communicate (referred to as the **non-parametric changes**) and the *changes of the resources descriptions* which require the designer to perform certain changes concerning certain constraints changes and/or certain parameters alterations (these changes referred to as the **parametric changes**). Figure 4.4 illustrates an example of the different changes that are likely to be performed on a given activity structures shell.

The exploration step utilises the information gathered via measurement, inheritance, and learning that are monitored from both the user and the machine environments as well as including the additional experience gained by the designer. The exploration step is said to be **extensive** if all the possible behaviours of the target system have been analysed (c.f. Gaines 1972). The exploration step is said to be **intensive** if a particular behaviour of the target system has been studied (c.f. Gaines 1972).

Here we should note that for the designer wishing to formally define the activities of exploration, in particular the following notable research work have found to be of great help:

1. the implication operators and the relational products theory by Bandler and Kohout (1980),

2. the pioneering work of Svoboda (1964) especially his masking and activity matrix techniques,

3. Gaines' behaviour and structure identification scheme, in particular his complexity measures on the admissible models (Gaines 1976),

4. Mason's productivity theory that helps in defining general purpose performance measures (Mason 1979), and

5. Klir's reconstructibility theory which helps in defining certain complexity measures of the system relational structure (see Klir and Way 1985). Here we should note also that the formal approach to design activity structures-based systems is out of the scope of this thesis.

4.7 Describing Computer Systems via Activity Structures

In this section, I shall briefly outline the way that was used to produce an activity structures-based computer system description. By description we mean the design details of the activity structures based shell at any level of abstraction. This section outlines the main ingredients that are necessary for the construction of an activity structures shell along with stating the necessary substrata structures needed for realisation. Indeed, many researchers share our concern in constructing computer system models from the analogy with the Brain structure. These researchers use some very general models of concurrency, such as Petri nets or electrical circuits (Aleksander 1982), or in models that are insensitive to the evolutionary process of design (Baer 1973). Common to all approaches, however, is the lack of distinction between the functional and substratum structures. Activity structures methodology, on the other hand distinguishes, between the goal oriented structures of behaviour, i.e. the functional structures, and their embodiments in a substratum structure, i.e. either applied hardware, or abstract (e.g. virtual machines). The coupling of these two types of structures was formalised by Kohout (1977). However, our concern here is different, that is the conceptual interpretation for the purpose of designing computer systems. The basic meta-principles of our design approach are given in the form of sixteen *basic assumptions* [2]

Here we should note that these ssumptions have been formulated after completing the implementation of our support tool and hence we believe they provide a practical advice for the success of any activity structures based computer system design and implementation. Other useful design definitions however are provided in section 1.2.

Assumption 1 (Essential Functionality) *Activity structures shell achieves its required goals by the cooperation of both the user and the machine environments as well as the cooperation of the functional structures within the machine environment.*

Assumption 2 (Functionality Uniqueness) *Each functional structure is characterised by a specific type of behaviour which is different from the others.*

Assumption 3 (Syntactical Decomposition) *Activity structures shell realisation syntactically decomposes into functional substructures.*

Figure 4.5 illustrates a typical functional structure decomposition.

Assumption 4 (Representing Functional Structures: The Static View) *Each functional structure is* statically *represented by a state determined system. This system is modelled by a* manager*. The manager's responsibilities and duties can be changed via changing the information deposited in the* resource descriptor *of the managed object.*

Figure 4.6 illustrates the statical components of a functional structure.

[2] Alternative concepts to basic assumption may be used, e.g. *postulate* in context of the general category of modality (c.f. Runes 1942) or *directive for definition* (c.f. Luschei 1962).

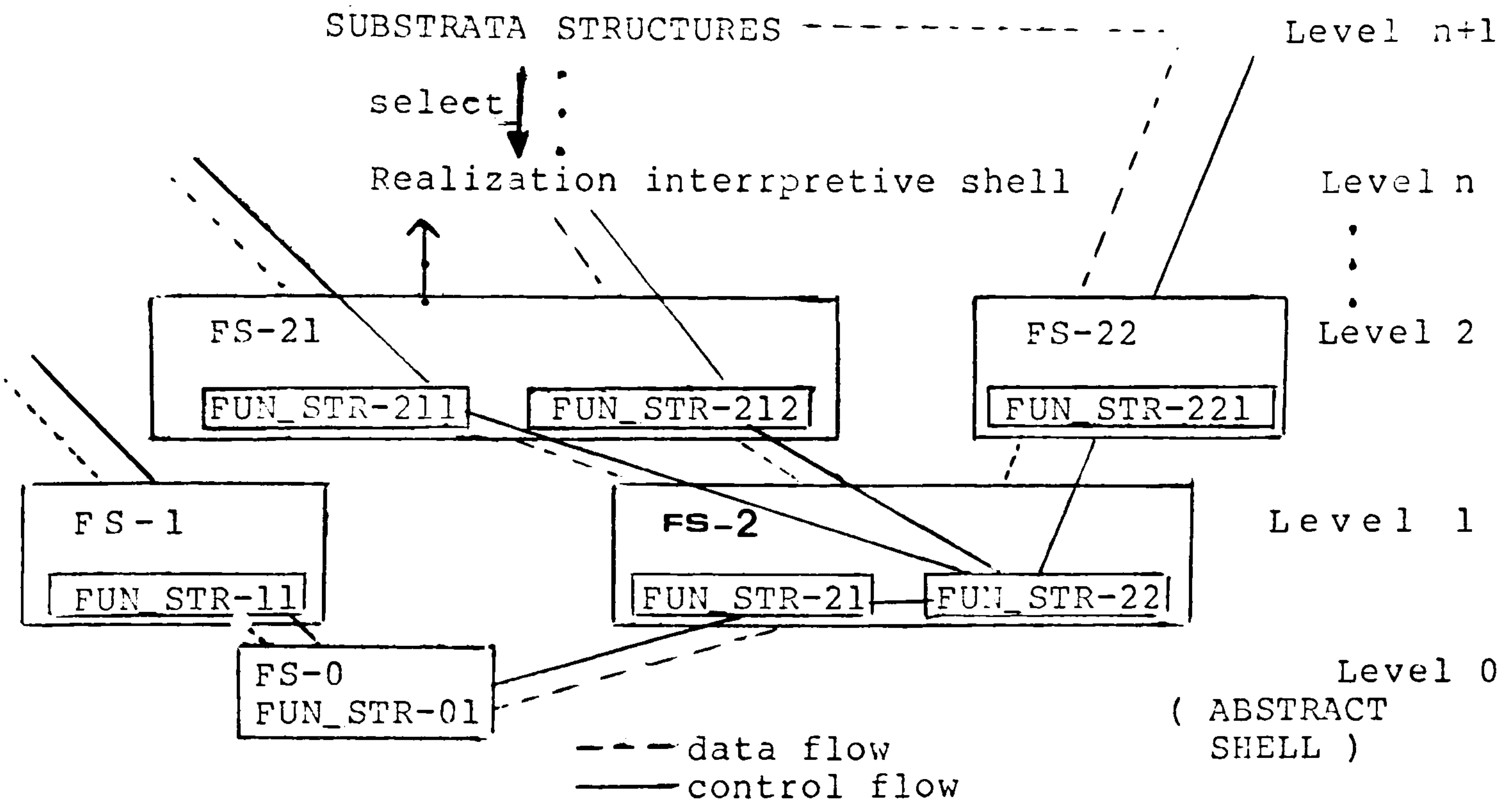


Figure 4.5: The Syntactic Decomposition of a functional structure.

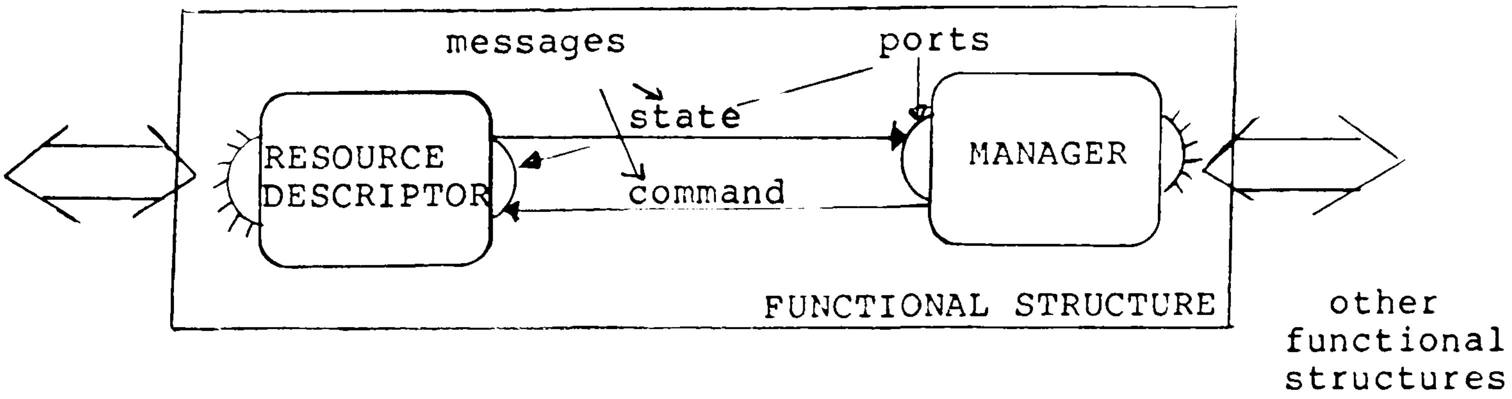


Figure 4.6: The Semantical Statical Description of a functional structure

Assumption 5 (Representing Functional Structures: The Dynamical View) *The change of each functional structure can be represented dynamically by a* process *which act as an active entity that is mainly issuing requests to the other processes and resources to accomplish its goal.*

The process is the result of executing the manager component. Managers perform several activities which may be executed independently of one another. Hence managers provide the user with a set of resource access operations (e.g. read, write, etc.), and encapsulate within it any scheduling policies for these operations. Managers are also responsible for synchronising the requesting processes of the shared resource descriptors by traping the processes requests and issuing the issuing the primitives that control the descriptors accesses. The functional sub-structures identify the required system managers, both local and the global activities. Resource descriptors help to perform operations requested by the managers.

Assumption 6 (Distribution Criterion) *The activity structures architecture is called distributed, if the functional structures, or their components communicate via a message passing technique.*

The main components utilised of this communication mechanism are messages and ports. Both components are resource descriptors, in which messages are used for inter- processes communication. Ports are recognised as resource descriptors that are independent of the processes which use them. Messages are placed in ports by a process with send access to the port. Messages are removed from ports by a process with receive access to the port.

Assumption 7 (Essential Substrata) *Two extensible primitive substrata structures are required to realise the activity structures shell. These are the* coroutines *and the* interpretive descriptors, *corresponding to the managers and the resource descriptors, respectively.*

According to Conway (1963),

> "a coroutine is an autonomous program which communicates with adjacent modules as if they were input and output procedures. The coroutine represent successive

passes, each of which transforms a stream of data, so that their execution can be interleaved in time according to the demand strategy."

Descriptors (Gaines 1974) form a structure which is defined by some common properties such as:

1. it specifies the operations that can be performed on their referenced resources,
2. it specifies the processes that are allowed to access their referenced resources, and
3. it specifies the level of protection, re-entrancy or sharing, etc.

An interpretive descriptor is an extensible data segment that refers to a resource. This segment can be interpreted at any level of implementation abstraction by changing its data contents (via parametric changes) or to a data type or register for example. Each interpretive descrptor is essentially composed of two fields (Mohamad and Cavouras 1984):

1. the unique name of the referenced resource (i.e. a pointer),
2. a control field that specifies which operations are permitted on the referenced resource through this descriptor. "Descriptors, also, are more general than the concept of capability" (Bishop and Barron 1981).

Other fields may be added and are specific to a particular implementation.

Assumption 8 (Concurrency Control) *Concurrency in any activity structures based shell is represented via the cooperation and synchronisation of coroutines.*

The way coroutines achieve concurrency and synchronisation can be illustrated via the following protocol describing the interaction of two coroutines in execution (this demonstrates how synchronisation and concurrency can be achieved): When coroutines P1 and P2 are connected (via ports) so that P1 sends items to P2, then P2 runs for a while until it encounters a read demand which means that it needs something from P1. The control is then transferred to P1 until it wants to write, whereupon the control is returned to P2

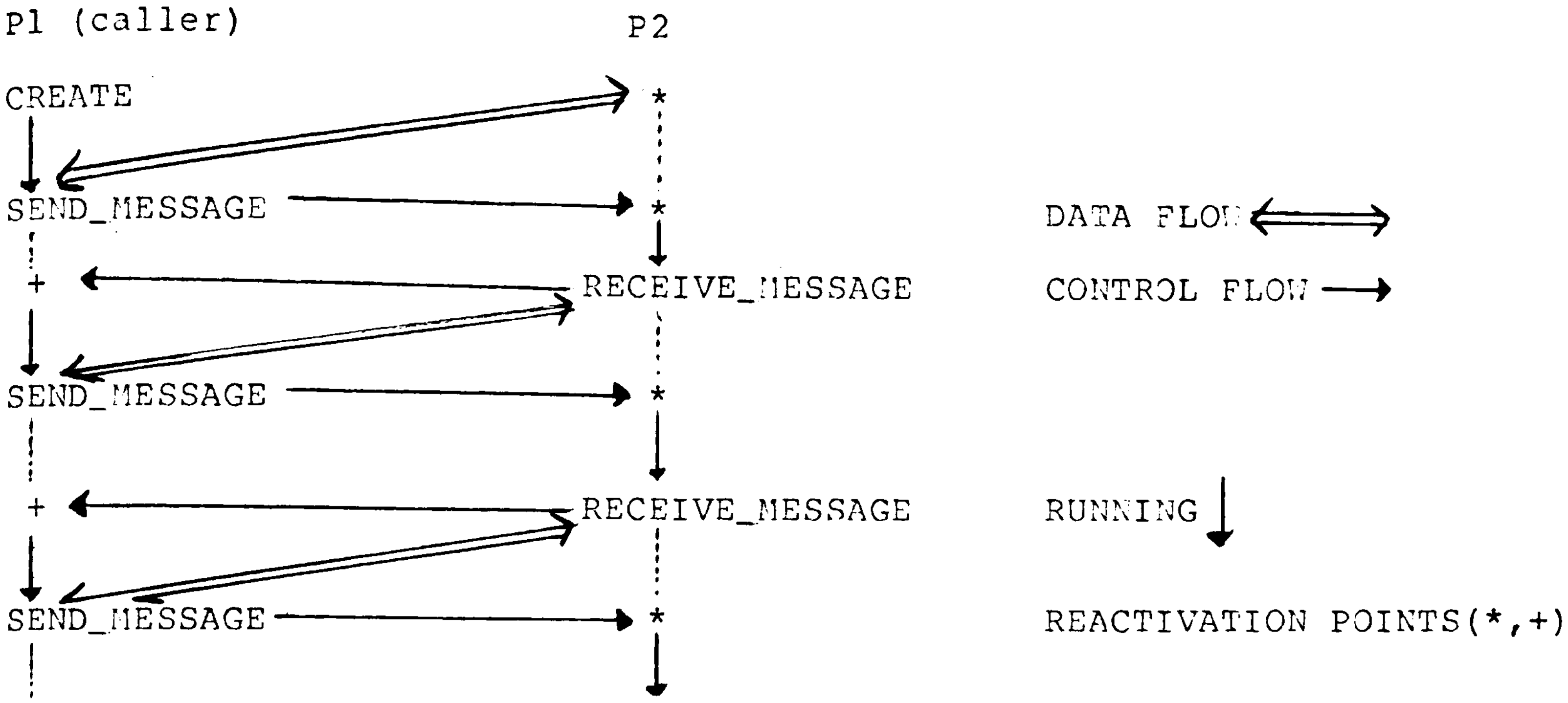


Figure 4.7: Example of Coroutines achieving Concurrency and Synchronisation.

at the point where it was left off (i.e the activation point). The Figure 4.7 illustrates the coroutines communication scheme.

In reality, coroutines cooperation is limited mainly by two resource constraints:

1. the number of messages produced by the sender cannot exceed the capacity of the message queue, and

2. the receiver cannot consume messages faster than they are produced by the sender.

These resource constraints are enforced by the implementation of a **synchronisation rule**. This rule states that if a sender attempts to place a message in a full message queue, it will be delayed until the receiver has taken another message from the message queue. Furthermore, if a receiver attempts to remove a message from an empty message queue, it will be delayed until the sender places another message in the message queue. The implementation of this rule has been achieved in our realisation by employing two types of semaphore; WAITING-FOR-ACTIVATION and ACTIVATED (see section 6.5.3).

Assumption 9 (Behaviour/Structure match) *The shell has maximal match in substrata if and only if the* semantic gap *between the functional structures and their corresponding substrata is minimal.*

Assumption 10 (Communication Styles) *The communication style of the shell in execution depends upon whether the time factor has been used.*

The communication is synchronous if the interconnection between the different processes is made through clocking devices otherwise it is asynchronous.

Assumption 11 (Computer Model) *A computer system model description represents an interconnection of cooperating processes, in which some of its processes can receive information from the user or the designer environments, and some processes can produce and pass information for the same environments. This communication model of a computer system model is intended to be used on uniprocessor hosts.*

In this model, processes running on the same processor can share the same address space. With a 'proper' programming language, a process can be affected by other processes only by communication, or by affecting descriptors which have been explicitly transmitted. This ensures privacy and data protection even in a shared address space. Processes running in the same address space can exchange arbitrarily complex descriptors or objects just by passing pointers. Processes running on different processors, however, communicate through restricted *"flat"* channels, e.g. character channels. In this case, complex objects have to be encoded to fit into flat channels, and decoded on the other side. The semantic is difference between exchange of objects in the same address space, where objects are shared, or in different address space spaces, where objects are copied. The basic communication is based on coroutines: both sender and receiver may have to wait until the other side is ready to exchange a message. The scheduling of most of the processes is non-preemptive: a running process will run until it explicitly gives up control by attempting to communicate; at that point other processes will get a chance to run. We assume a cooperative environment, where no process will try to take an advantage of the other processes, unless it has some reason to do so (e.g. a higher priority process may interrupt a lower priority process).

Our model differs from other communicating parallel processes models, such as Hoare's communicating sequential processes (CSP) model (Hoare 1978). In CSP a process issues either an X or XX command which can be verbalised as 'have-you-a' or 'here-is-a' respectively.

There is no input-output command of the form 'give-me-a' or 'take-a' which create a forced entry into another process; in contrast this criteria is provided in our model. Further, in the CSP model if there are several external requests on a process, the process itself decides which operation to is allowed to proceed. Thus the system activity is not capable of forcing a stop entry into the timer process. Even if we consider the incorperation of the monitor's concept (Hoare 1974), then once one process has gained access to the monitor no other process can; thus yet again, it is not possible for a higher priority process to gain access, once another lower priority process has gained the access. Therefore while using the models based on the monitors concept, we can not build a priority communicating processes. This explains why both CSP/K (Holt et al 1978) and Pascal-Plus (Welsh and Bustard (1979) have incorperated priority scheduling into their implementation of the monitor concept.

Assumption 12 (Measurement Probes) *Statistics can be collected via the insertion of certain performance probes at various places of the shell simulation.*

Assumption 13 (Simulation of a total shell) *The simulation of the activity structures based shell should include both the behavioural modelling of the machine and the user interaction environments.*

Assumption 14 (Stable Shell) *The activity structures based shell behaviour is stable if there exists a set of* admissible design data *(section 7.6) concerning both the user environment and the machine environment within which the* conversational environment *(section 4.6) can behave in a* self-regulating *manner.*

Assumption 15 (An Activity Structures based Computer System-ASCS) *An ASCS model can be produced by the total shell, and its interaction between the user environment and the machine environment is* maximally constrained.

Assumption 16 (Shell Soundness) *Any activity structures shell is said to possess a sound design if a compromise can be reached, between the intended conceptual model of the problem environment and the actual shell model of the designer environment. In an* *effective design the shell soundness can be reached by* tuning *the shell (i.e. finding an admissible design data, see section 7.6).*

Chapter 5

AN ABSTRACT SHELL FOR THE ACTIVITY STRUCTURES-BASED COMPUTER SYSTEM DESIGNS

5.1 Introduction

The assumptions 1-16 presented in section 4.7 represent a meta definition of the complete shell of the support tool. The complete shell consist, in fact of two separate simulation shells mutually coupled. Into the inner shell we embed the machine simulation environment, whereas into the outer shell we embed the user interaction environment.

The activity structures-based shell programs are then written to handle both the possible behaviours generated by the user stimuli, and then functional strategies and constraints of simulated general purpose computer systems.

More specifically we start with the identification of a well-defined, well-classified and general-purpose computer-oriented behavioural models (i.e. the functional structures). The grid (i.e. a specific resolution level of these behavioural models is of a rigorous but flexible nature, not only through their constructs but also by recognising the learning ability of both the user and the inner shell. In the complete shell itself, these behavioural models can be represented by the designer as a set of computer algorithms and resource descriptors whilst their requirements can be elicited from the constructor and processed in a participative or conversational way (which includes both the system user and the designer).

Our goal in this chapter is to introduce such behavioural models and demonstrate that these models, developed originally for a neurological knowledge-based system (c.f. Kohout

1976, 1978, 1981), are general and applicable to other computer systems design applications. The discussion of these behavioural models will be preceded by a section that illustrates the way we simulate the conversational environments. After describing the essential behavioural models (i.e. the essential functional structures), we introduce the probes that are needed for the monitoring and performance evaluation of the shell. Finally we discuss the choice of the shell implementation language.

5.2 Representing The Shell Conversational Environment

Oberquelle et. al. (1983) pointed out that the traditional computer system design methodologies do not include the communication behaviour between the user and the computer system in their design rules. In our design methodology, based on the activity structures, we include such behaviour, called the *communication behaviour*, as one of its essential design contributing factors. We characterise the communication behaviour as the behaviour resulting from the cooperation between the user interaction and the computer machine environments. We shall see later that it is essential to include *probabilistic* and *adaptive* characteristics *in the description of these environment.* The *empirical support* for the inclusion of cooperation comes from Kupka (1974) and Barber (1979) whereas *conceptual support*[1] is provided by Kohout (1976, 1978a). In section 5.4 we list the essential design features needed for simulating the activities of both the user interaction and machine environments.

In order to characterise statistically the interaction between humans and computer machines, we need to identify a 'prototype' model of human-computer dialogues. In other words, we need to identify the nature of the alternating sequences of user actions and the machine reactions. An effective statistical prototype model that can be used for this purpose has been defined by Kupka (1974) as a dialogue consisting of the user's 'local' model for isolated or randomised inputs, a corresponding 'local' model for isolated or randomised outputs of the computer machine, and a 'global' model combining both. This is unlike the traditional computer designs in which the dialog is subdivided, leading to the mode of

[1] Empirical support and conceptual support are used as technical notions corresponding to 'observational' and 'theoretical' evidence respectively. It is used in the same way as in 'GUHA's research (c.f. Hajek and Havranek 1978).

working where human and computer only co-act in parallel.

Barber, independently from Kupka, collected statistics on the users actions and the reactions of an interactive computer system (Barber 1979). He observed that although the user actions and the computer reactions seem to be of a rather random type (similar to Kupka's local models), there exists a definite statistical governing pattern of behaviour relating the user actions and the machine reactions. This has a resemblance to Kupka's global model. Barber measured the user actions and the computer reactions quantitatively, using two types of measures: the *user productivity* and the *job satisfaction*. These measures are somehow related to each other, as the statistics he collected show. In our opinion this is due to the adaptivity factor. Whenever the user productivity increases (e.g. due to higher user transactions), the machine tends to adapt to the user behaviour and increases its utilisation power. The penalty for this is the increased response time, resulting in a decrease of the job satisfaction. This is due to the *machine adaptivity* factor. Similarly, whenever the job satisfaction decreases, the poor computer response causes (statistical) reduction of the user productivity. It should be noted that job satisfaction and user productivity are the statistical notions that can not be reduced to non-statistical ones. *User* in this context means the *average user* that is a statistical measure expressing the mean value, or more generally, a statistical moment of the n^{th} order (for $n = 1, 2, \cdots, n$). This reflects the *user adaptivity* factor. Figure 5.1 presents an example of this relationship.

5.2.1 Generating User Activities:

In order to generate user activities for a computer system, one might try to replicate the 'shopping steps' of a user job as McDougall (1970) has done in his BASYS simulation model. But surely, *shopping steps*[2] represent randomised activities only, and cannot truly model the user events which form a part of the dynamics of the simulated computer system. The reasons for justification of this conclusion are quite obvious. Namely, the shopping steps do not capture that part of the dynamics that represents user adaptivity.

[2] These represent the system workload requirements that must be performed by the system. Steps in this context represent the different execution tasks required by each job during its execution time.

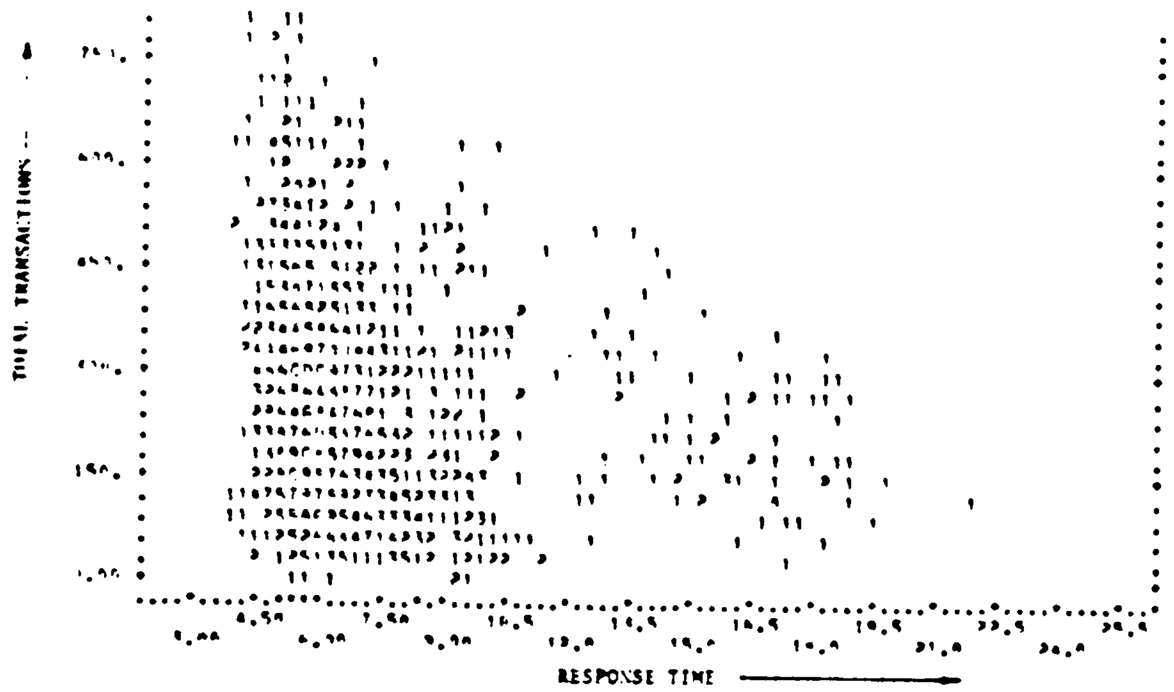


Figure 5.1: Barber's Model: User Productivity versus the job Satisfaction (Barber 1979, p. 29)

To generate rather more representative user activities, we need to generate them using a probabilistic distribution that fits a user particular adaptive mechanism. We call this type of user activities the *intention steps*[3]. The probabilistic generation of intention steps in our case depends on the type of the substrata the computer machine is supporting. Figure 5.2 shows a typical general-purpose computer substratum.

For such a type of substrata, the intention steps are generated in a random fashion and according to the following typical steps:

Intention step 1: A user job (batch or interactive) arrives randomly or according to a specified distribution. Upon arrival, the following job characteristics are determined either randomly or according to pre-determined distribution:

1. the total CPU time,
2. the average amount of central memory (CM) requested, and

[3] Intention steps in Kohout's Activity Structures scheme, are represented by Intention Structures, which are a particular kind of the user environment functional structures representing the participants intentions to act in certain way (Kohout 1976).

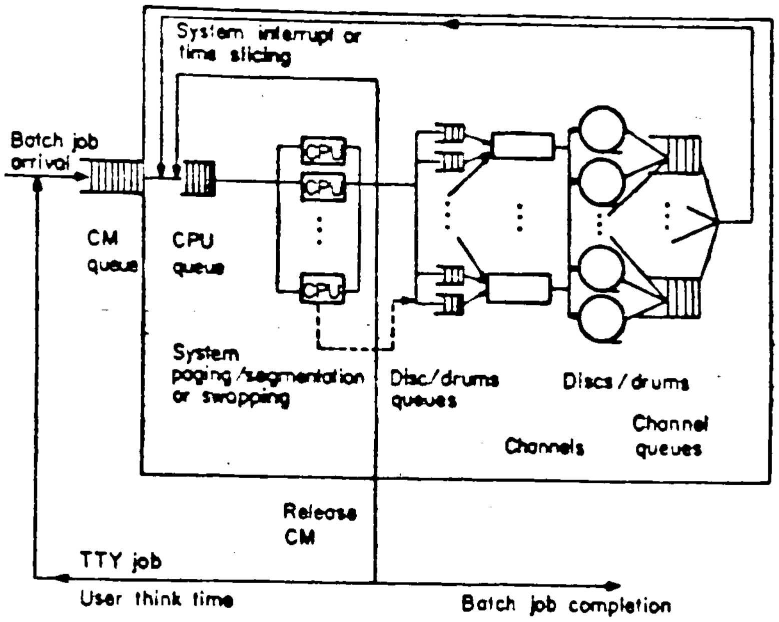


Figure 5.2: A General-Purpose Computer Machine Substrata

3. the number of I/O requests.

Intention step 2: The job makes a request for CM allocation. If the CM space requested is not available, the job enters the CM queue.

Intention step 3: After the job enters the CM, it immediately requests the CPU. If the CPU is free, it is assigned to the job and executes until some blocking conditions occur (i.e. a system interrupt, the time slice used up, the job is completed, or an I/O request is encountered). In the former two cases, the job releases the CPU, but is placed back into the CPU queue.

Intention step 4: When a job issues an I/O request, the CPU is released, and a specific disk is requested. Since the total CPU time and the number of disk requests for a job are predetermined, it is assumed for a fair utilisation of the non-sherable I/O devices, that jobs utilising these devices may be interrupted, and then they can continue at a latter time until their specified elapsed time finishes.

Intention step 5: In order for a job to access a designated disk, both the disk and the associated channel must be free. Otherwise, the job enters a disk queue. If the disk and the channel are both free, a *"disk seek"* time is generated. During the disk seek time, the disk is busy, whereas the channel is not.

Intention step 6: After completing the disk seek, a rotational delay time is generated. When this time expires, the channel is requested again, and if available, the data is transferred over the channel. The disk and the channel are both busy during the *"transfer time"* .

Intention step 7: When the data transfer is completed, the disk and the channel are both freed, and the job proceeds to request the CPU again.

Intention step 8: Upon completing all the CPU and I/O tasks for a given job, the CM allocated for that job is released. If the job is a batch job, it leaves the system; otherwise, the job is an interactive job, and has just completed a *"system response cycle"*, so a *"user think time"* is generated.

In our case, the generation of the *randomised part* of the various intention steps follows certain scheduling techniques. Our scheduling mechanism generates user intention steps according to certain parametrised distributions (mainly poisson type). In our case the average parameters (i.e. the distribution seeds) are supplied by the designer.

However, the *adaptivity part* is represented by a adaptive mechanism for each user intention step, which tries to optimise the 'best' region (between MAXSEED and MINSEED) within which the random activities can be normalised. The adaptivity mechanism uses three iterations, the 'Fix-MINSEED', 'Fix-MAXSEED' and 'Mid-MAXMINSEED' iterations. The first two iterations start by fixing one end point of the normalisation interval to the seed maximum or to the seed minimim respectively, iteratively adding/subtracting a *slit* (equal to the tenth of the difference between MAXSEED and MINSEED) until the other end is reached. For the third iteration, the normalisation interval starts by fixing the normalisation interval to a slit around the average seed, then the iteration starts by adding two slits, one for each direction, until the seed maximum and the seed minimum are reached. The adaptivity mechanism uses these iterations sequentially. In each iteration, it associates the change in the normalisation interval (i.e. state of size and location) with the system responses (i.e. the monitored resulted performance). The learning mechanism decides at the end of the three iterations the best normalisation interval according to the best resulting system response.

Such associations between the state of the normalisation interval of intentions can be represented by entries in a *table of connections* in which the complex optimisation task can start. Indeed, a table of connections requires a great deal of computer storage. For example, for R seeds and N values per seed, a parsimonious representation of the state requires the order of N^R storage cells. On the other hand, one needs only $N \times R$ cells to represent the status of each seed independently of the other seeds, and only R cells to represent the situation as a value of a linear polynomial. In reality, the psychological evidence indicates that humans seldom attend to more than a few environmental features at a time (Yntema and Mueser 1962), so a connection table of low dimensionality might be a reasonable representation. This is the representation we adopted. The routine responsible for generating the user intention in our implementation is called the *job scheduler* (see section 6.4.1.1).

5.2.2 Generating the computer machine activities:

The way we generate the computer machine (i.e. inner shell) actions or reactions is partly driven by the user actions and partly by itself. This means that the computer activities are driven by **external interrupts** (i.e. intentions of users) caused by events *"external"* to it, and by **internal interrupts** (primitives of the different management unites in the system) issued by its processes. In the actual implementation, these interrupts cause automatic entry to the interrupt service routines. The interrupt service routines, in turn, can cause further events and then *"return from interrupt"* to the interrupted process. For example, when a timer expires and its interrupt is serviced, the corresponding interrupt routine usually activate a *"scheduling routine"* (required for management and learning) and reprograms the timer to expire at the following interval.

This type of technique for generating machine activities not only aims at embedding a computer system model in a simulation of its environment, but it allows the overall system performance to be measured by direct experimentation. Our main routines responsible for scheduling the computer machine activities are:

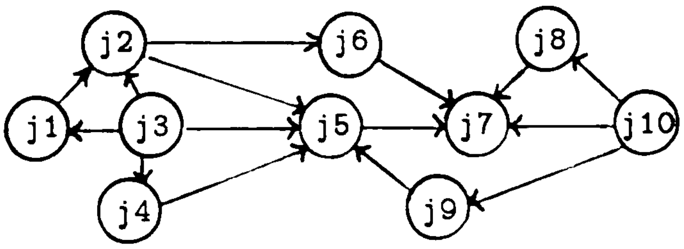
1. Processor demand scheduler,
2. Resources demand scheduler, and
3. Processes selection scheduler.

The details of these schedulers are given in the next chapter. The principles of generating the computer activities follow the *randomness* of the user actions intention steps, but the computer machine possesses different adaptivity mechanism. I would like to concentrate on the adaptivity mechanism that would introduce adaptivity in the machine environment. The advantage of using learning not only to cope with the changes imposed by the user actions but also by adaptivity of the inner shell, generally enhances the overall performance of computer systems. In our particular case, the conversational environment reaches its equilibrium or self-regulation.

Our *adaptivity* problem enhances performance in one specific task, the workload scheduling task with respect to the different system functionalities and constraints. A simplified example for this task is shown in Figures 5.3 and 5.4, in which scheduling enhances performance under one constraint, the protection, without breaking such constraint. The objective of this example is to complete all the jobs in as short time as possible by executing them in parallel, without violating the protection rules (i.e. achieving high-performance and protectibility goals). It is a difficult task faced by the management scientists, but it can be shown to be very similar to other optimisation tasks requiring a sequential set of decisions, e.g., the Traveling Salesman Problem. This task is very similar to the scheduling procedure used by Hsaio et al (1966) in a study, which suggested the performance gain adaptivity mechanism that we used. One can view performance optimisation tasks that require a sequence of decisions, as problems in finding jobs that are independent in their protection requirements and execute them in parallel. To aid our understanding, we draw our connection table as a directed graph in Figure 5.2.2. The nodes in the directed graph represent the jobs to be executed and the arrows are the context dependent protection requirements, between the jobs.

		INPUT NODES									
		j1	j2	j3	j4	j5	j6	j7	j8	j9	j10
O	j1	0	1	0	0	0	0	0	0	0	0
U	j2	0	0	0	0	1	1	0	0	0	0
T	j3	1	1	0	1	1	0	0	0	0	0
	j4	0	0	0	0	1	0	0	0	0	0
N	j5	0	0	0	0	0	0	1	0	0	0
O	j6	0	0	0	0	0	0	1	0	0	0
D	j7	0	0	0	0	0	0	0	0	0	0
E	j8	0	0	0	0	0	0	1	0	0	0
S	j9	0	0	0	0	1	0	0	0	0	0
	j10	0	0	0	0	0	0	1	1	1	0

(a) an example of a connection table



(b) the same example represented by a connection graph

Figure 5.3: An Example of a jobs connection settings.

For a given connection table setting, such as in Figure 5.3a, the adaptivity technique that will be used, is composed of the following steps:

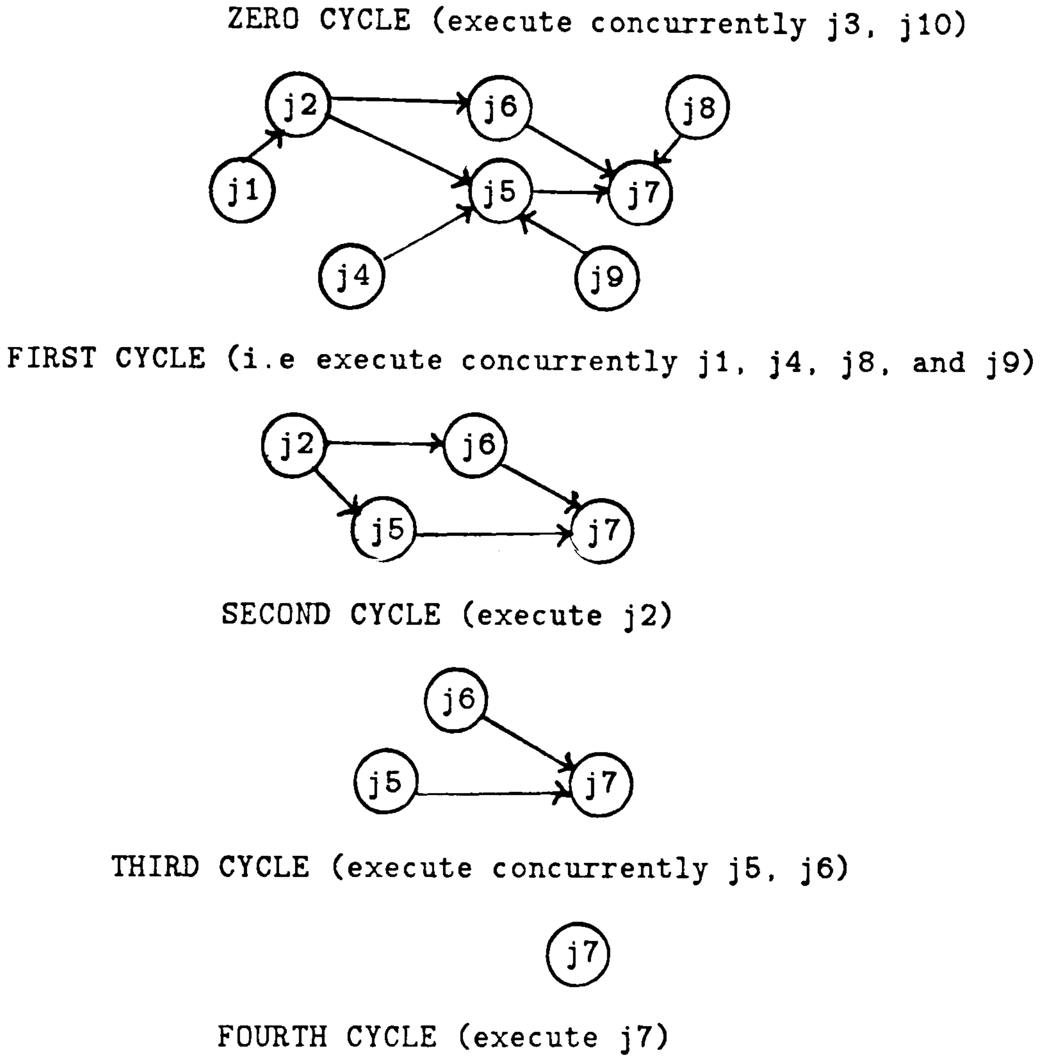
1. search for the job nodes with all arrows pointing out (OUT-NODES),

2. search for the isolated job nodes (ISO-NODES),

3. remove OUT-NODES and ISO-NODES and their immediately directed arrows from the connection table, and

4. repeat steps 2 and 3 on the new graph until empty graph situation is reached.

The results of performing this technique on the example given in Figure 5.2.2 is illustrated in Figure 5.2.2 below.



ZERO CYCLE (execute concurrently j3, j10)

FIRST CYCLE (i.e execute concurrently j1, j4, j8, and j9)

SECOND CYCLE (execute j2)

THIRD CYCLE (execute concurrently j5, j6)

FOURTH CYCLE (execute j7)

Figure 5.4: Performing The Adaptive-Technique Steps: an example.

This technique works well with simple connection tables that are of the directed graph type. With a generalisable connection table, such as the one used by us *descriptor-oriented architecture* (section 6.5.1) connecting the different jobs with their functional strategies, we need a more sophisticated adaptivity technique. The type of a generalisable technique that we adopted is a *"hill climbing"* procedure. The context in which this machine adaptivity technique is used, is discussed in section 5.3.2 (the inferential structures).

5.2.3 Towards Simulating the User and Machine environments:

For replicating the user and computer activities, *simulation* techniques are the most suitable methods of representation (c.f. Lindstrom 1981). We shall do the replication using an *activity-oriented* simulator. With simulation, however, we are bound to deal with several new design features. Indeed, simulation is the only method that can be used to generate user and computer activities and estimate the performance of new designs and new configurations before actually implementing them (which is the case of our inner shell). The new design tasks[4] that are required for the simulation are (c.f. Unger 1977):

1. *Describing the system descriptors and data structures and their attributes.* Descriptors and data structures are the element of the system model connected with, and influenced by, other elements of the system (e.g. processes). Here we may distiguish two types of descriptors and data structures: those needed for constructing the actual system model and those that are needed for the simulation process.

2. *Dealing with queues, sets or lists* Dealing with lists is essential because activities cannot be served at the moment they arrive. activities wait for service in queues: the service process of a queue gets out the first element of the queue and organises the tasks of this entity.

3. *Maintaining the simulation time* A 'discrete event' simulator maintains a simulation clock (i.e counter) or possibly several clocks, advanced after each change caused by the systems activities and interrupts. The change in the simulation clock is advanced by a variable amount corresponding to the real time that must elapse before the next change takes place.

4. *Describing and scheduling events or activities:*According to a specific scheduling policy the next event is chosen to be served. Scheduling policies varies according to the tasks required, for example the jobs arrivals are scheduled according to a Poisson random

[4] See Kohout (1978a, 1978) for the definition of a task as a protected activity directed towards a specific aim.

policy, whereas the scheduling of the eligible processes that are need to be executed concurrently is decided by the inferential structures.

5. *Collecting the statistics generated by the simulator*: The simulator maintains several monitors that record the various activities and timings produced by several system elements.

In an activity-oriented simulator the occurrence of each computer interrupt is made by a user activity. The sequence of activities can be established by generating a list of future user activities (randomly and according to a learning mechanism). This can be stored in the **activity list or the dynamic list of user intentions**. Our tool can now be driven by a simulation control program which uses the entry at the head of the activity list. Each activity list entry specifies activity time, an activity identifier, the associated process (or whether the interrupt is external) and other information associated with this event (see Figure 5.5).

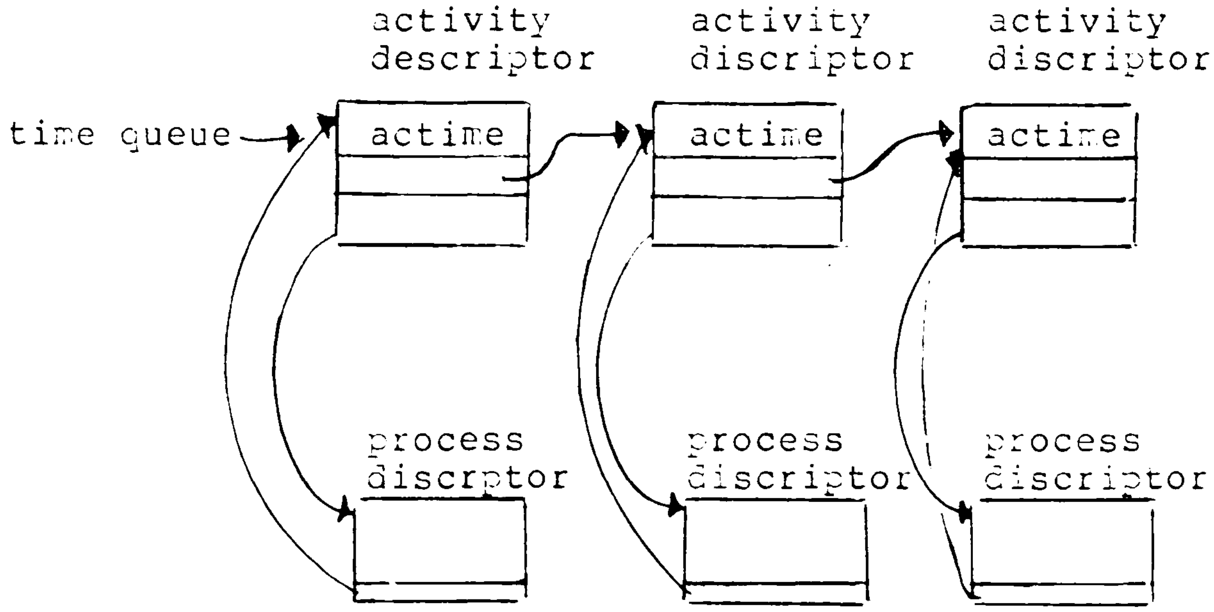


Figure 5.5: Activity descriptors and process descriptors.

The current simulation time is advanced to the activity time specified (which represents the absolute time the interrupt occurres); the other fields are saved and the entry at the head of the activity list can now be deleted. For example, assume that we have two processes A and B with A currently executing (see figure 5.6). A is then at the head of the activity list (or time queue) and B follows. In the past, process B has executed and its last action was to activate A at time 100 and put itself into sleep until time 200. Its execution has stopped after the **block** (200). A is now executing and has come to the block (300). Current time is 100 (always the time of the head element in the time queue). The routine block is now called, and it takes the return address of the calling routine (A in this case) and saves it in the process descriptor for A. It removes the top activity descriptor and puts a new one at time 400 linked to A. Then the execution resumes at the return address in the process descriptor of the head element of the time queue (now B) which is the point after the block (200) in B's code.

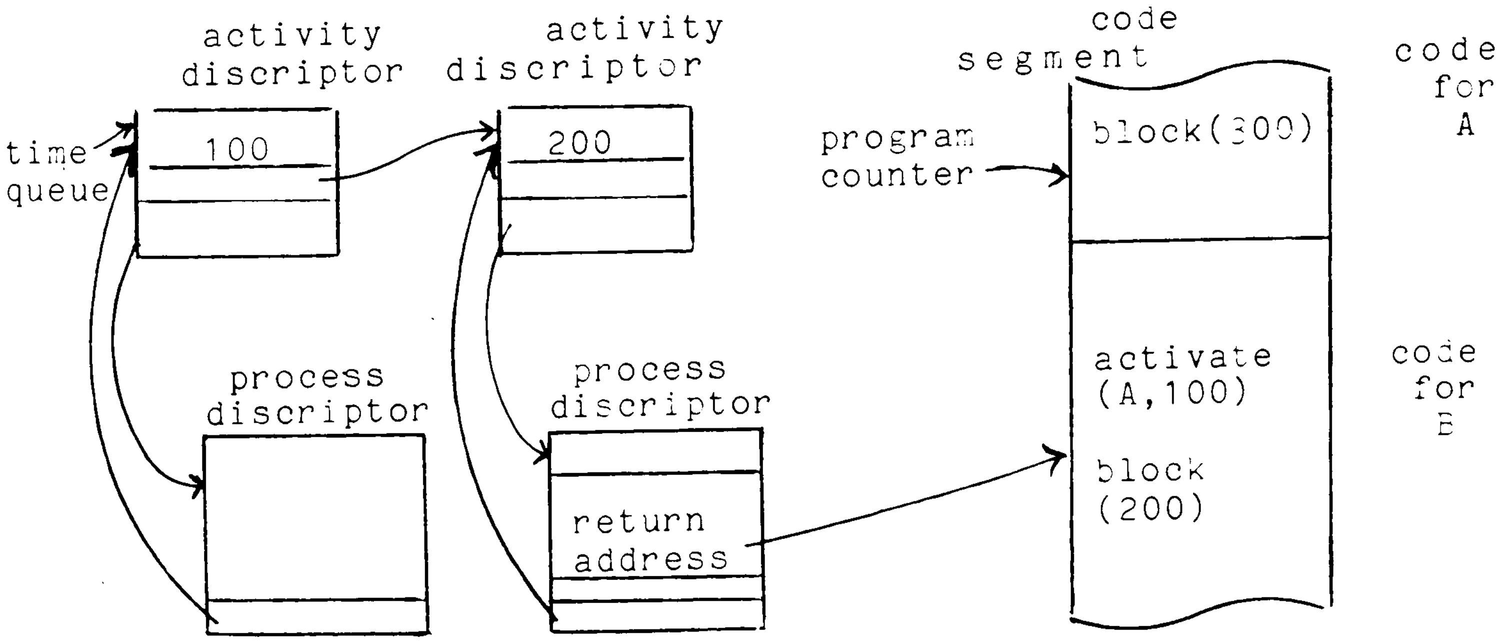


Figure 5.6: Activity list servicing example.

More detailed description of these features will appear in the next chapter.

5.3 The Functional Structures of The Inner Shell

This section provides fairly rigorous, although still heuristic, ways of identifying the functional structures or the behavioural models of activity structures based inner shell in such a way as to facilitate making relationships between these structures and the computer shell.

There can be no doubt that the function idea provides a conceptual viewpoint of computer system inner shell operations. But this idea raises another question that needs to be answered. Namely, at what phase this conceptual view needs to be modelled ? As pointed out in chapter 4, the different phases of conceptualisation of the model are linked only through the designer activities (i.e. the designer experience) and otherwise they are independent. These design activities are reflected in the designer experience hence, from a proper modelling point of view we needs to start from the top of the conceptualisation hierarchy, the Goal Phase.

We can also say that a function identifies a component part of the total set of the system operations. This leads to the question of functional decomposition which are collected together to form the functional structures (c.f. section 4.6.1, 4.6.2) and breaking system functionalities into subfunctions. We believe that the fine details of the functional decomposition are highly related to the details of system realisation. This functional structures decomposition (Kohout 1982) has been used successfully to construct several medical diagnosis systems including expert systems, a DSS system, information retrieval systems (Kohout et al 1984, 1985, 1986). In this section we are demonstrating that these functional structures can be ported for the use of constructing general purpose computer systems shells. This chapter, however, concentates on the design abstraction of the complete shell and implementation details will be left to the next chapter.

It should be stressed that the shell model is the 'actual shell conceptual model' of computer systems and not the 'intended shell conceptual model' of the problem environment. The *actual shell conceptual model* is the collection of the design facts and tasks that capture the essential functionality (c.f. assumption 1, section 4.7) of the computer system and the *intended shell conceptual model* is the model of the computer system that should be activated according to specified design requirements. The implementation of the actual shell conceptual model is presented in chapter 6, whereas the dialog between the intended shell conceptual model and the actual model is left to chapter 7.

The essential functional structures into which the decomposed subfunctions are collected are listed below:

1. *The Design Modules* :
   (a) the knowledge representation structures,
   (b) the inferential structures,
   (c) the control structures,
2. *The Constraint Modules* :
   (a) the protection structures,
   (b) the communication structures, and
   (c) the interpretive structures.

In the sections that follow we describe each of these in an abstract conceptual way. The details of their realisation are left to chapter 6.

5.3.1 The knowledge representation structures:

The knowledge structures represent a knowledge base which stores specific knowledge of the general purpose computer design, which can be facts, hypothetical assumptions, or heuristics. Originally Kohout et al (1986a, 1976) represented the knowledge structures, (for example within the CLINAID system) by using the *semantics descriptors*. Similarly, in my representation, all knowledge is partitioned into discrete structures (*descriptors*) having individual links (*ports*). This structure I shall call *Interpretive descriptors*. Interpretive descriptors can be used to represent broad concepts, classes of objects, or individual instances or components of objects (e.g. information on devices, information on the protection required to be achieved on the access of these devices, etc.). Interpretive descriptors are realised using the concept of data-type (c.f. Mohamad 1982) as described in chapter 6. They are joined together by an appropriate *descriptor meta-structure* (c.f. Kohout et al 1986) that provides for the transmission of common properties among the descriptors. In our case, the descriptor meta-structure is represented by the communication functional structures discussed in section 5.3.5.

Here we define the *descriptor-oriented architecture* as an object-oriented, port-based architecture which manages ports, processes, a descriptor directory, and the delivery of descriptors via the ports. Descriptors are the instances of abstract data types, and ports (which are channels for communication between processes) are themselves descriptors. Processes may request the creation of the ports and then execute operations which send and recieve descriptors of the ports. Processes may also use the ports to execute operations on remote objects by sending requests themselves as descriptors. Sending and receiving descriptors of ports leads to the creation, blocking, and starting of processes, and is the only mechanism controlling processes after the system is initialised. In other words the descriptor-oriented architecture is a system which manages the manipulation of the knowledge base of the complete shell (Mohamad 1982). This architecture essentially contains the directory of all the shell descriptors as well as their connections (i.e. their meta-structure).

The other main characteristic of the descriptor-oriented architecture of chapter 6 is that it incorporates both, the *addressing* and the *tagging* features. These are a manipulated set of mechanisms for management of the stored information. For a fuller understanding of further sections it may be useful to give a brief overview of this mechanism here, in anticipation of details in chapter 6.

The basic management mechanisms of our descriptor-oriented architecture consists essentially of a central processing unit (Figure 5.7) generating addresses in a segmented virtual memory space. The virtual space is composed of 2^e segments: a virtual address, as generated by the CPU, is composed of an triple (IDs, AR, OF), where IDs specifies two unique names one for the data segment and another for the associated port; AR represents the access rights; OF identifies addresses of two specific storage units associated with this particular descriptor, one for *relevant port*[5] and the another for a data segment.

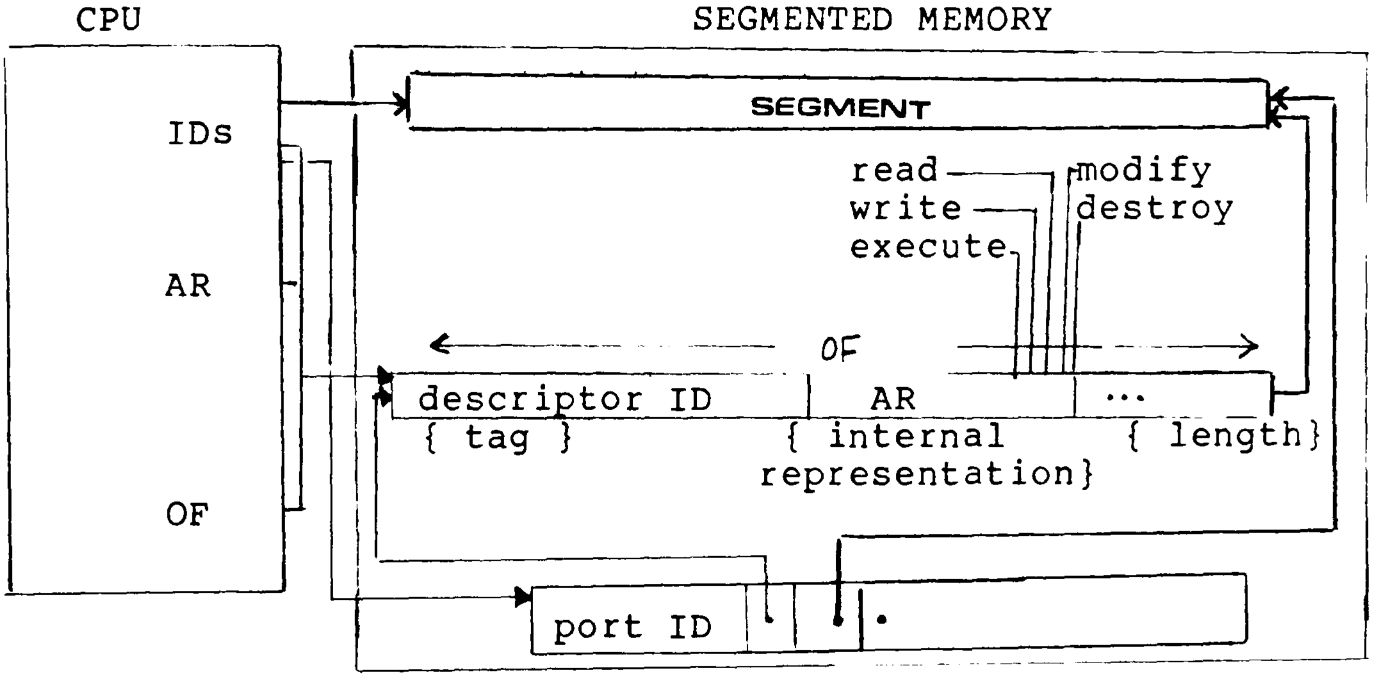


Figure 5.7: Featuring the mechanism for generating our descritor-oriented architecture

Each generated descriptor describes a single object of a specific type. More precisely, an object is represented by a single descriptor segment, if it belongs to a machine or represents one predefined types of the knowledge structure. On the other hand, a user-type object consists of a collection of descriptor segments (which, in their turn, contain other objects of a machine and/or predefined types). Each descriptor segment is partitioned into three portions, called the tag, the internal representation and the length (see Figure 5.7).

This representation of descriptors, again given at a greater detail in chapter 6, is quite general. It was presented here mainly to illustrate the essential idea of descriptors. However, our descriptor-oriented architecture contains several types of descriptors, such as the memory descriptors, the segment table descriptors, the process descriptors, the device descriptors. Their details will also be left to the next chapter.

[5] Relevance of a port is a technical phrase referring to the ownership notion of the port (c.f. Stemple et al 1982).

5.3.2 The inferential structures:

A computer system can be imagined as an information system which receives and manipulates user transactions, and retrieves information for these transactions in the response to the user query. The inferential structures represent a set of certain search strategies. Our inferential strategies are defined as strategies consisting of two mutually dependent search mechanisms; a *memory based search mechanism* and a *processor based search mechanism.* For the earlier examples of such type of inferential structures in other application contexts see Anderson, Kohout, et al (1985) and Kohout (1983).

User demand is defined in our simulation as the measure provide of the system workload. This measure is represented by a function producing one output index: the number of concurrent user jobs. The input parameters to this function are the user intention parameters (section 6.2) which include the average number tasks per user, the user productivity (average system replies/total tasks - faulty intentions), average user think time, etc.

The **memory search mechanism** represents an iterative process whereby a set of user-judged relevant working set descriptors (pages or segments) at any point in the search is used to refine and improve on the remainder of the user's search. This inferential strategy improves the overall system performance in two directions; it minimises the search time and it prevents the propagation of errors (faulty accesses). Figure 5.8 illustrates the general idea behind this inferential mechanism.

Machine effectiveness is said to reach its maximum when certain generalised system performance indices (e.g. average response time, average system throughput) equal or exceeded their specified thresholds (see section 7.5).

On the other hand, the **processor based search mechanism** represents the process of adjusting periodically the number of eligible processes allowed to enter the main memory (i.e. eligible for execution), so that on average the target number of working set calculated in advance is kept under control (this is a demand and anticiptory policy, see section 6.5.4) and hence the processor utilisation is kept effective (i.e. avoiding thrashing). The way the processor based inferential mechanism adjusts the number of eligible processes allowed to

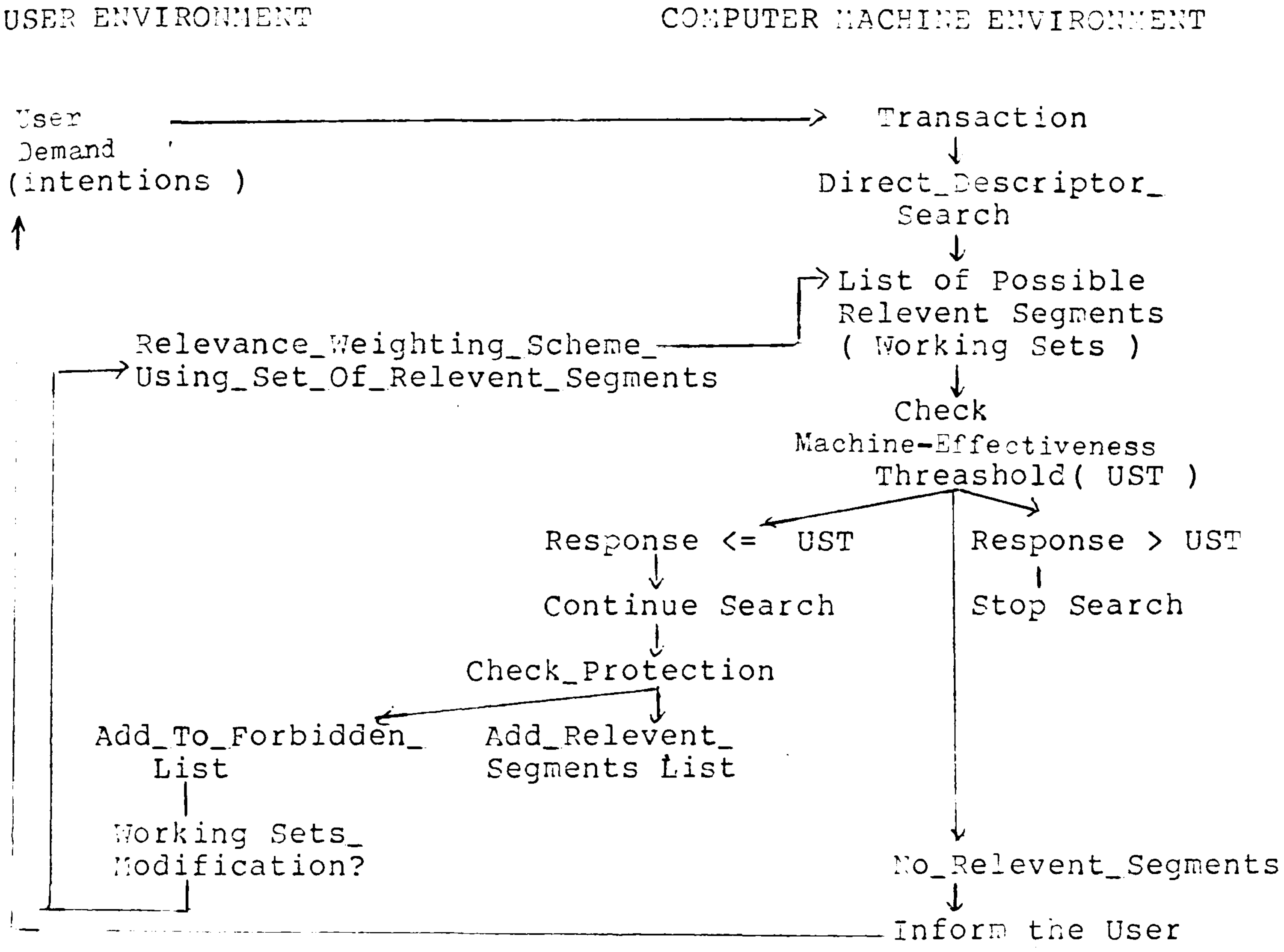

USER ENVIRONMENT
COMPUTER MACHINE ENVIRONMENT
User Demand (intentions)
Transaction
Direct_Descriptor_ Search
List of Possible Relevent Segments (Working Sets)
Relevance_Weighting_Scheme_ Using_Set_Of_Relevent_Segments
Check Machine-Effectiveness Threashold(UST)
Response <= UST
Response > UST
Continue Search
Stop Search
Check_Protection
Add_To_Forbidden_ List
Add_Relevent_ Segments List
Working Sets_ Modification?
No_Relevent_Segments
Inform the User

Figure 5.8: The memory based inferential mechanism.

enter the main memory is based on a *balancing mechanism* between the amount of user loss and the system loss. The *user loss* function represents the intended user job strategy of assigning priorities (see Figure 5.9). This strategy takes into account the user required time limits as well as the urgency of finishing the job within each time limit (using different function slopes) and assigns loss quantities to each case. The type of user loss function we use is given below:

$$\begin{array}{c}\text{Intended}\\ \text{Job Priority}\end{array} = \begin{cases} R \times mh & 0 < R \leq th \\ R \times minf + th(mh - minf) & th < R \leq tinf \\ tinf \times minf + tinf(mh - minf) & R > tinf \end{cases}$$

The parameters R, th, tinf are all expressed in time units and mh along with minf represent the user job urgency (line slopes). The intended job priority, hence, represents linearly-ordered set of functions (the change of slope occurs after each user job time deadline) of user loss occurred to the user process. The shape of this function is given in Figure 5.9.

This type of function was originally used by Denning (1979) for processor scheduling (resource balancing strategy), and employed by the author of this thesis for identifying the average user intended loss that occurred through the user jobs execution time. Note that with the increasing delay the resulting user loss becomes higher.

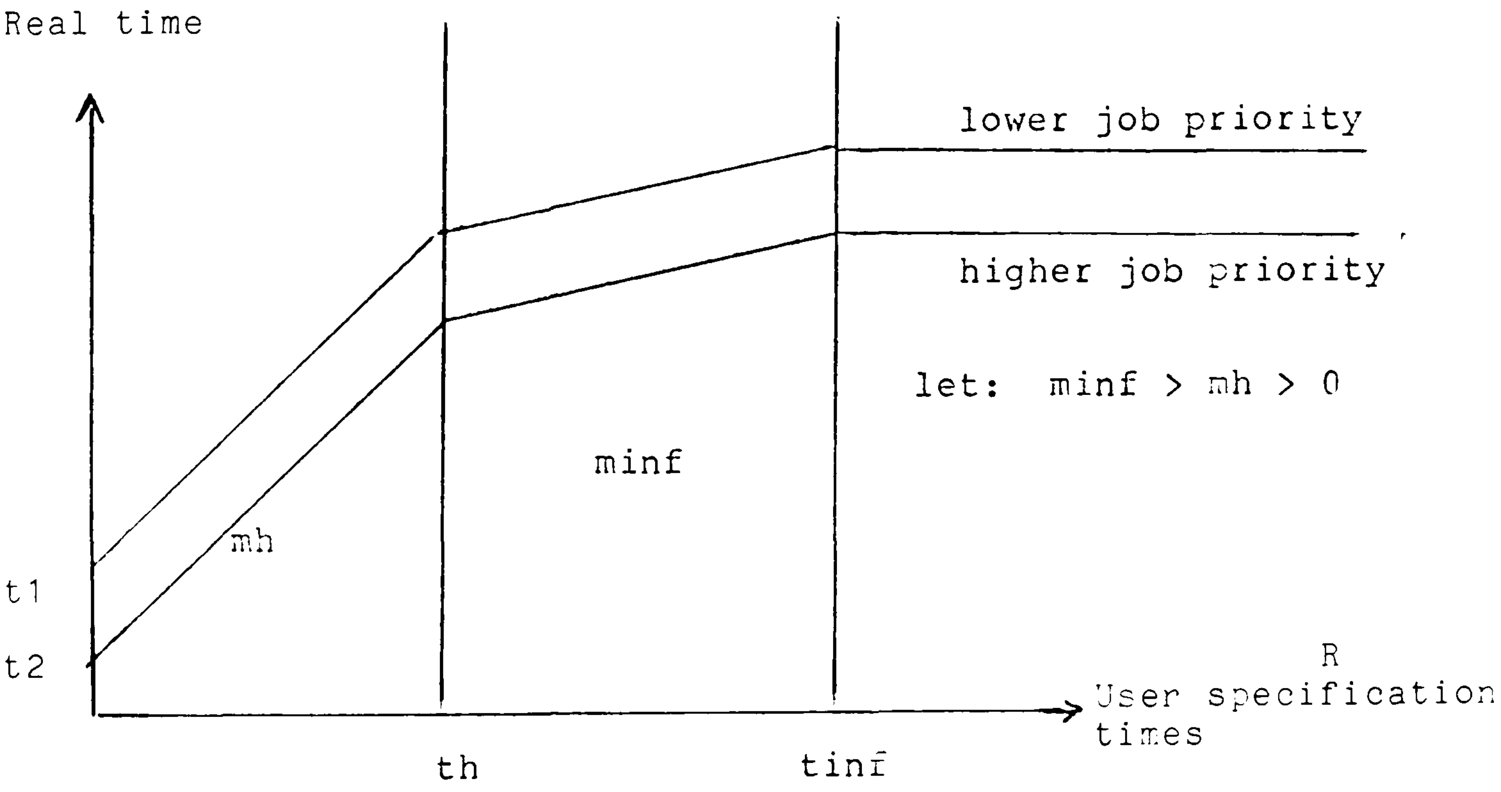


Figure 5.9: The shape of the user loss function (the user intention service policy)

The *system loss function* measures the amount of the processor work pw (amount of service the process received + the processor utilisation). Then, the balancing mechanism attempts to select the eligible processes for execution according for their highest user loss and the lowest system loss, provided that the processor utilisation is lower than an acceptable threshold (i.e. the amount that does not cause *thrashing* c.f. Denning 1969). There is no selection performed for eligible processes, if the processor utilisation exceededs its acceptable threshold. When an eligible process is selected by the balance mechanism, its priority is assigned by using the following function:

$$\begin{array}{l} \text{Intended} \\ \text{Process Priority} \end{array} = \begin{cases} R \times mh + pw & 0 < R \leq th \\ R \times minf + th(mh - minf) + pw & th < R \leq tinf \\ tinf \times minf + tinf(mh - minf) + pw & R > tinf \end{cases}$$

5.3.3 The control structures:

Our definition of the computer system control structures goes beyond the traditional definition of control as manifested in the conventional operating systems literature. Our extension includes the flexibility criterion required for distributed control, according to nature of the distribution of the different functional structures. The basic scheme we adopted for this purpose, is a development of the *communicating automata scheme* of Kohout (1976, Gaines and Kohout 1975). Our scheme provides a central control unit for the management of the distributed communicating modules. We call our control scheme the *kernelised communicating distributed modules*. The main advantages of this scheme are

1. it enables modules (i.e. a component of the functional structures) to be loosely connected, in this case the only change needed is to specify the communication paths required by the user or the designer,

2. t enables us to execute these modules concurrently, and

3. odules have simple connections which allows us to perform system development (e.g. by adopting the vertical modules migration strategy (Stockenberg and van Dam 1978)).

However, the communicating modules are represented by our implementation as a set of managers (coroutines). Figure 5.10 illustrates our abstract view of controlling a general purpose computer system via the use of kernelised communicating distributed modules scheme.

Indeed, we find that the idea of classifying the communicating modules into different subclasses is quite important for identifying the level of control under which the shell operates. These subclasses are ordered into the following hierarchy:

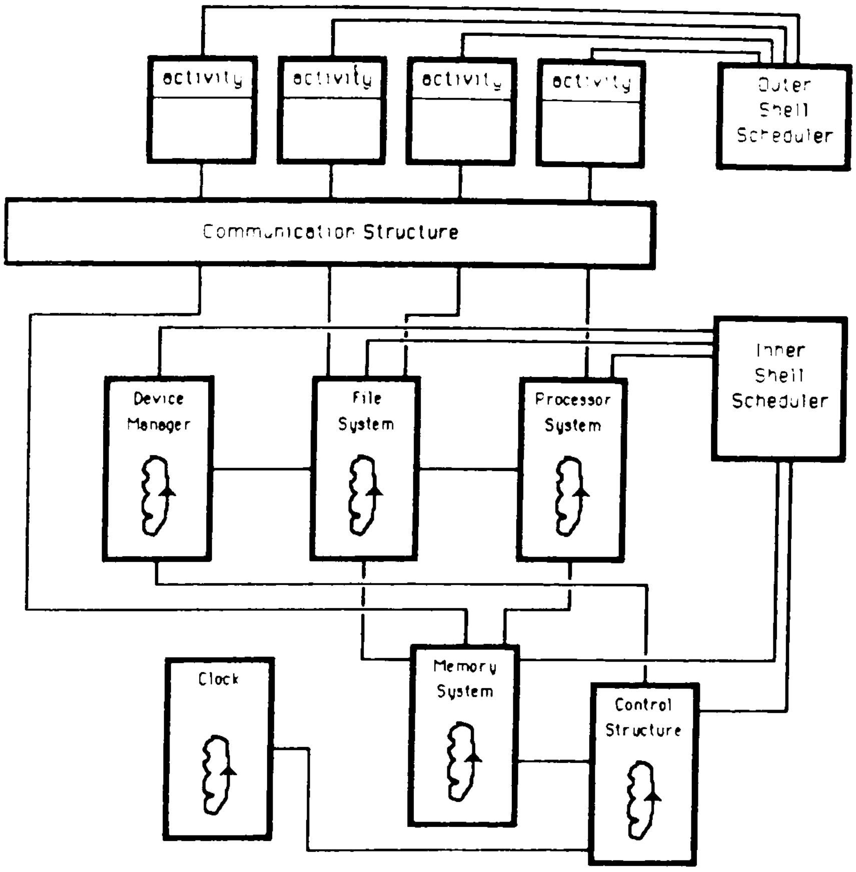


Figure 5.10: Kernelised communication distributed modules of a general-purpose computer system.

1. *The Application Control Module.* Modules in this class control the information generated by user processes. The users generate activities without the regard for the potential interference of the other concurrently executing user jobs and hence the application control modules are needed for controlling these activities. In our shell simulation, this class represents a reentrant coroutine program which independently models the execution control of all user processes. This is an activity-oriented simulator with its own simulation control program, activity lists and activities. The design of this coroutine program is influenced by the design of the user process run time support system (run time monitor) and hence the actual system itself.

2. *The Services Control Module.* Modules in this class deal with control services that depend on neither specific hardware nor specific user application programs. In our shell simulation, this class represents a set of coroutines one for each machine environment process in the complete shell. Among such modules are the inferential structures, the

communication structures and the control structures (see sections 6.5.4, 6.5.3, 6.5.5).

3. *The Hardware Services Control Module.* Modules in this class present the control for the physical hardware being used, including its communication interfaces. From applications and control services, they screen, to the extent possible, the errors (including faulty accesses), limitations, and idiosyncrasies of less-than-perfect hardware. Operating systems usually include most of the services in this category. In our shell simulation, these modules represent *uninterruptable kernel* interface which is driven by activities that have one-to-one correspondence with the real system activities (i.e. the external and internal interrupts of the system needed to be simulated). This interface includes the (activity-oriented) simulation control module, the activity routines each of which provides an effective replication of the corresponding real system interrupt routine and a **dispatcher**. In brief, this interface models the nucleus of the real system supervisor and the actual physical hardware and, therefore, its design is influenced by the absence of the latter from the shell simulation (i.e. hardware details can be changed by changing certain parameters in our simulation).

The design of an application module is essentially unaffected by whether its requests are intended for sequential or concurrent execution. It can be interpreted as one or more hierarchies of modules executing from a single virtual memory. A call from an application sub-module to another application sub-module can be interpreted conventionally, as if the calls have resulted in an immediate transfer of values and control.

In contrast, the control services modules view such calls as requests for the transmission of messages within a *"network of input/output communicating modules"* .

Modules communication consists of requests from higher modules (nodes) to lower modules (e.g., from M12 to M22). As shown above in Figure 5.11, the communication modules consist mainly of two kinds of nodes (i.e. excluding the kernel module). *External nodes* correspond to the external sources such as user processes, terminals, etc; *non-external* communication modules are design objects, each consisting of a set of procedures and data objects (e.g. the memory coroutine).

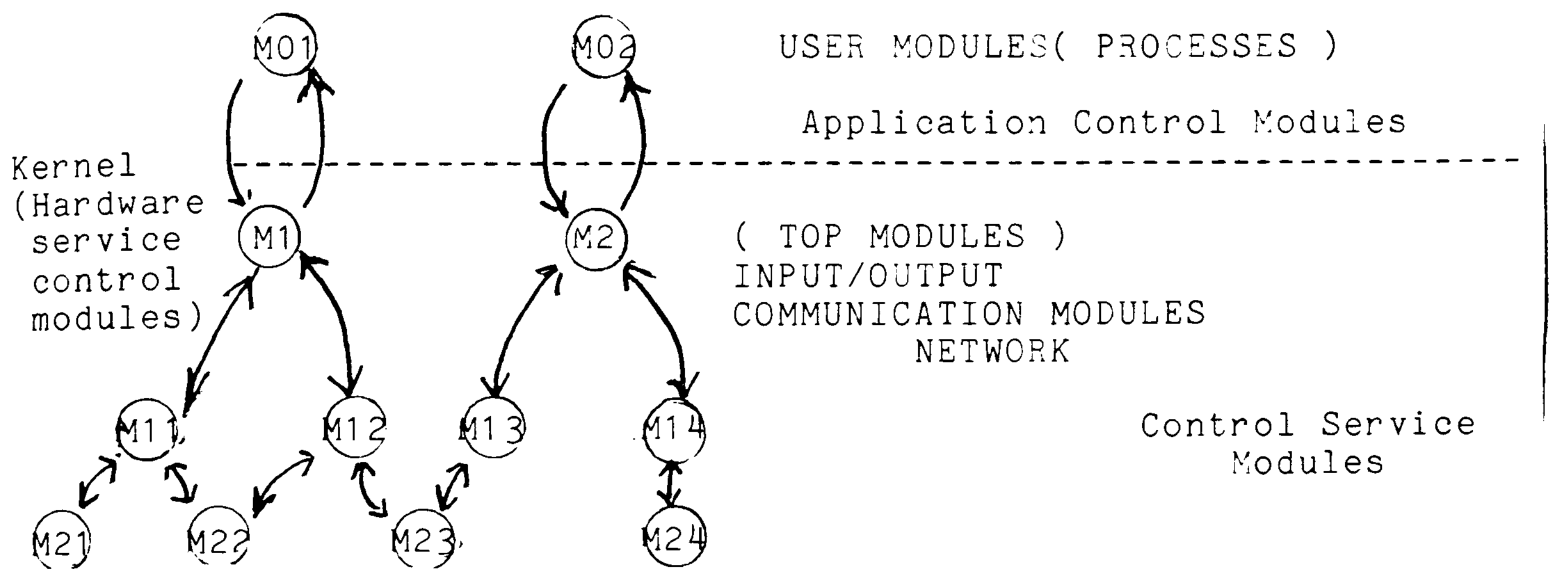


Figure 5.11: An abstract view of a communication modules network.

However, the module data flow, shown above in Figure 5.12 can be characterised as follows:

1. represents the delivery of a request message to a module,
2. the module uses as its arguments the contents of the request, as well as the resource object contained in the module,
3. during the execution of that module, change is made to the resource state and local variables, and
4. at module execution completion, a reply message is returned to the requester. The (operational) data object in a communication module, called a resource, survives the execution of successive processes. Its current value is called resource state of the communication module.

Aside from local variables, which it declares, a communication module can directly access only its own module's resource. All other resources, even those in the modules it calls, are hidden and should be accesses indirectly via the kernel. In other words, a calling

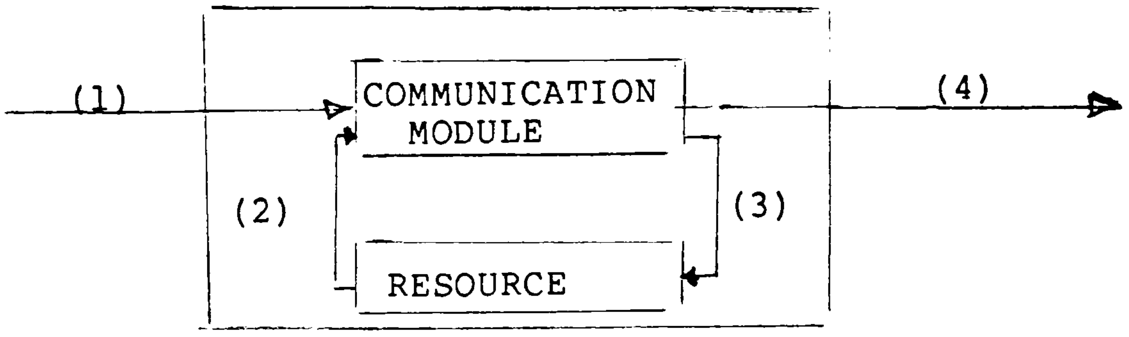


Figure 5.12: The internal structure of a communication module.

module may know that a subordinate module contain certain values of interest, but it can neither access the values directly, nor would it know that the values are kept, say, in an array.

Messages, which correspond to passed parameters, are of two kinds: *requests* and *replies*. Requests originating at user nodes or communication modules are directed to named subordinate modules. The replies, consisting of the results of the call are returned to the anonymous requester. The computation that results from the arrival of a request at a module is called an *operational task*. Thus, in Figure 5.11, when the module of M11 is initiated, a task is defined. If that module calls the module of M21, a second task is defined. Each task completes the computation required by the specification of its associated module. Therefore, the first task includes the execution of only the module of M21. A *process* is a special kind of task that consists of the computation that results when a request sent by an external user node arrives at a top (or a father) module. Finally, a set of processes resulting from related requests from an external user node is called *transaction*.

5.3.4 The protection structures :

A great deal of work has been done in recent years on inter-process communication between concurrent processes (Chandy and Misra 1979, Hoare 1974, Lamport 1978, Lelann 1977). Some synchronisation and protection algorithms have been implemented in which the logic is distributed among the processes. This distribution of logic raises a problem similar to that encountered in uniprocessing systems using semaphores: when a cooperation mechanism is distributed among its users, misuse of the mechanism by an individual process can affect the operation of the other processes (c.f. Rushby and Randell 1983). Distributed algorithms exist, that carry out synchronisation correctly (Peacock et al 1979), *but only in the case that they are used correctly.* Their correct use crucially depends on the correct understanding of the algorithm, and on a voluntary cooperation between the participating processes at the run-time. For a synchronisation algorithm to be *a robust one,* it must also be a *protection* algorithm in the following sense: it must continue to enforce the user-defined cooperation (synchronisation) on its constituent processes, even if it is misused by some of these processes.

This problem was solved in the uniprocessing systems by the introduction of monitors (Hoare 1978), and path expression (Andler 1979). Synchronisation between users of a resource is enforced by code in the resource itself. Since the synchronisation code is inside a single process (the resource), it can be guaranteed that the misuse by one process cannot affect other user processes. On the other hand in the case of distributed resources we cannot embed synchronisation code in any single process. If we do this the process becomes a *non-distributed controller* for the whole subsystem. Synchronisation code for distributed resources has to be distributed, yet designers usually want to verify their systems, and they want to verify them as absolute/complete systems (c.f. Williams 1983, Feiertag and Neumann 1979, McCauley and Drongowski 1979), not piecemeal as individual processes. This implies that, even if they are implemented in a distributed fashion, algorithms should be specified centrally (c.f. Heinrich and Kaufman 1976).

From the above discussion, it would follow that the *kernelised approach to distribution protection of Kohout* (i.e. his CLINAID whiteboard as in Kohout et al (1985)) is well justified to be used for the design of protected distributed computer systems.

Our protection structures are used not only for distributed protection (static protection), but can be also used for two other purposes:

1. to enforce protection policy for the control of multiple accesses of system resources that are shareable. The reason behind selecting such a policy cames from our intention to improve the overall performance of the modelled computer systems. This decision has been taken by the author, since it has been proven that the single access protection policy manifests an exponential complexity when used to implement multiple access protection (needed in a concurrent system such as ours), whereas multiple access protection policies cope with the accesses of the concurrent environment in the polynomial time (c.f. Antonelli and Iazeolla 1983)

2. to enforce a protection policy for the control of protection dynamics. Since each resource has a designated owner process, each resource owner may allow some partners (processes) to share the access to that resource (e.g.using the *pass* primitive). The owner may also specify a keyword (in the owner case *permit*) and any process that can produce this keyword is termed the keyholder and may access the shared resource. The mechanism for enforcing this protection dynamics has been described by Kohout (1976) and is used by the author to enforce the protection dynamics of the processes sharing memory segments (see section 6.5.6). Here we should note that pass and permit protection dynamics has been implemented by the auther for the first time since it has been developed by Kohout in 1976. Indeed there are several models that enforces the protection dynamics criterion, such as the take-grant model (Snyder 1981), the authorisation model (Fagin 1978, Griffiths and Wade 1976) and the dynamic authorisation model (Kambayashi 1981). All these models operate by assuming the criteria of the process independency rather than cooperation (see section 4.7. for the notes on the CSP model which enforced the process independency criterion). We beleive by using the pass and permit model the protection can be enforced on the cooperating processes sharing a common knowledge base (see section 6.5.6).

Basically, the protection structures utilise certain protection descriptors that generally belong to the knowledge representation structures.These descriptors act as tickets to access their associated objects. Each descriptor may have the following representation:

OBJECT ID PORT	ID	Access rights	Base/limit

This representation is an idealised description, and does not correspond exactly to the implementation of descriptors on any existing systems. However, we distinguish two types of protection descriptors: the user intention descriptor (i.e. user activities or transactions) and the system objects permission descriptor (which stores the permission access rights allowed by the system to each object or resource). By means of these two types of protection descriptors, the protection mechanism enforces protection by allowing intentions that match the permissions (with dynamic protection mechanism the transform of accesses is done first) ; if this is not the case, the **faulty intention** interrupts arise (see Figure 5.13).

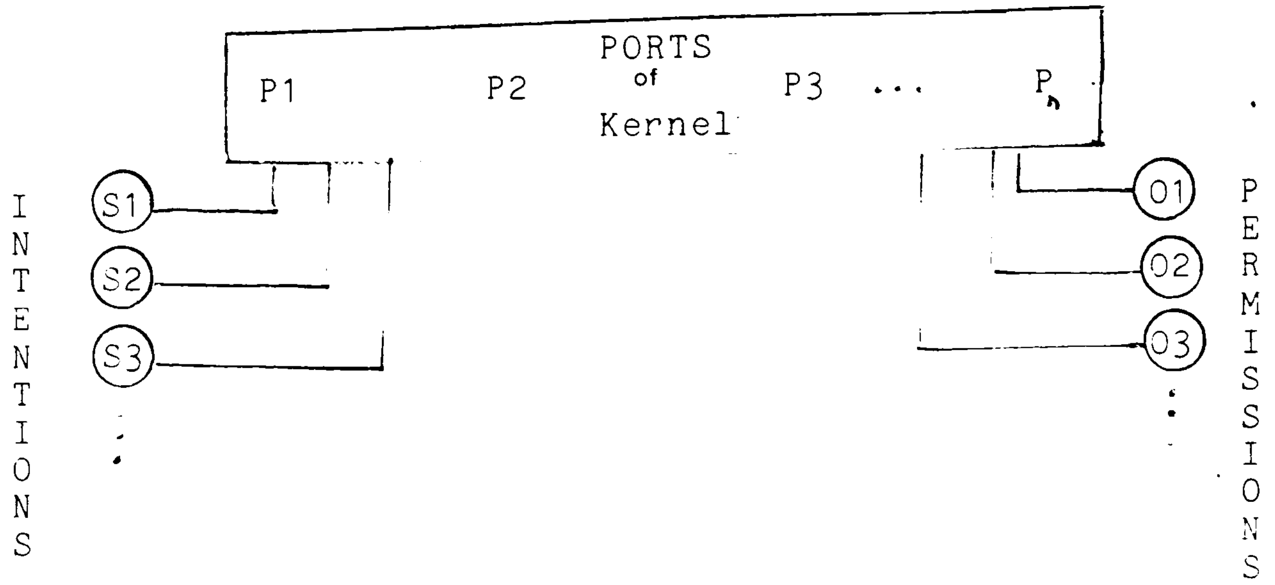


Figure 5.13: The Basic Components of our Protection System

Ports are used in our protection structures to provide the means for protection distribution. Conventionally, ports have been utilised as communication channels between communicating processes (Stemple et al 1983). We extended the use of the ports for achieving protection distribution. For this purpose, a port is viewed as an abstract data type and protection is achieved by restricting the operations that are available to processes that manipulate ports. Further, we equate ownership of a port with possession of a descriptor to request operation on an object. This type of protection is referred to as **port-oriented** protection (see Figure 5.14).

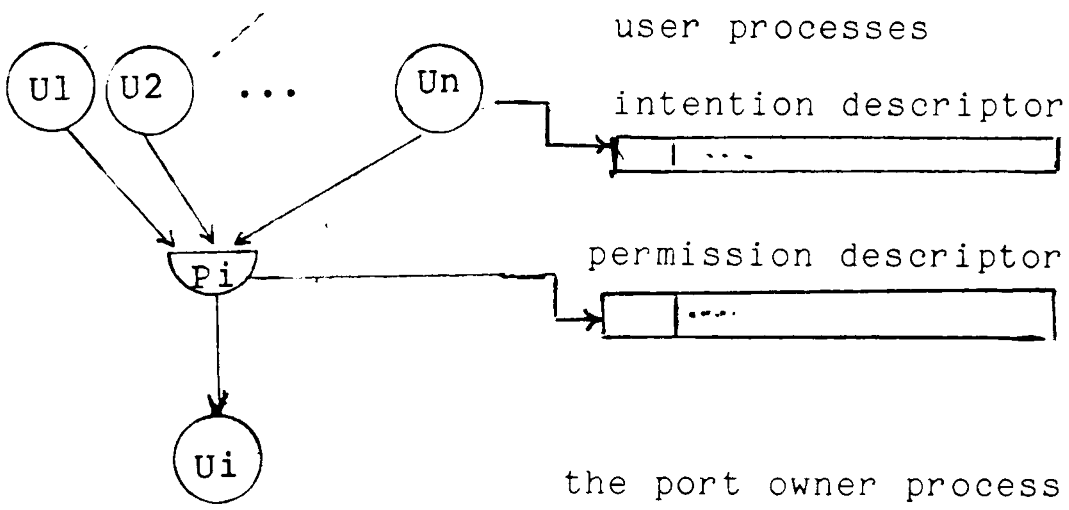


Figure 5.14: The Port-Oriented Protection System.

A port constitutes a communication path between sets of communicating, cooperating processes. In processes that communicate, a port can mask the identity of the processes involved in this communication. Messages are placed in ports by a process with the *"send"* access to the port. Messages are removed from the ports by a process with receive access to the port.

5.3.5 The communication structures :

The communication structures are viewed by the author as centralised media for communication and control among the various shell processes. The construction of the communication structures is based on the concept of *communication partners* (participants) for processes (outlined by Kohout 1976): Those participants which have the same father are considered as communication partners. Subprocesses, subsubprocesses etc (i.e. parts of processes) may communicate only with the communication partners of the processes to which they belong. On the other hand the internal structure of a process has to be invisible to its communicating partners in order to keep changes in the internal structure of the communication partners independent of communication. Figure 5.15 shows an example. The communication of process 1 are process 2 and process 3, whereas those of subsubprocess 2.1.2 are are subsubprocess 2.1.1, subprocess 2.2, process 1 and process 3.

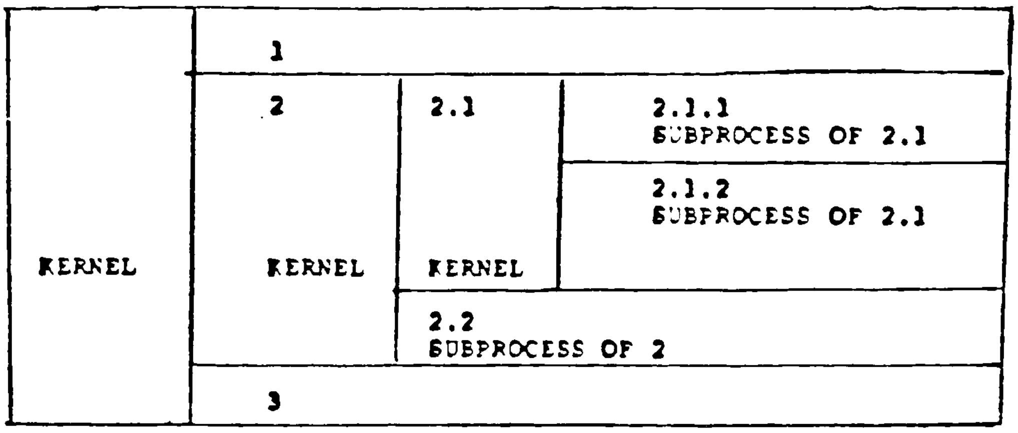


Figure 5.15: Interprocess communication mechanism– Communication Parteners.

Basically, our processes communication mechanism represents a message passing system in which each order is acknowledged by a response. Cooperating processes in our system communicates by sending messages to each other. Messages are transmitted from one process to another by means of message buffers, selected from a common pool within the kernel. The communication structures (kernel) administer a message queue for each process (see Figure 5.16). The rules of our communication mechanism are:

1. any order may only be sent by specifying the receiver port identifier,
2. a response may only be sent in return to an order which has been received,
3. a response may only be received, which refers to an order which has previously been sent, and
4. an order may be received without restrictions.

In general, we view the computer system as a pool of processes. These processes are confined to individual environments so that they are unable to communicate directly or

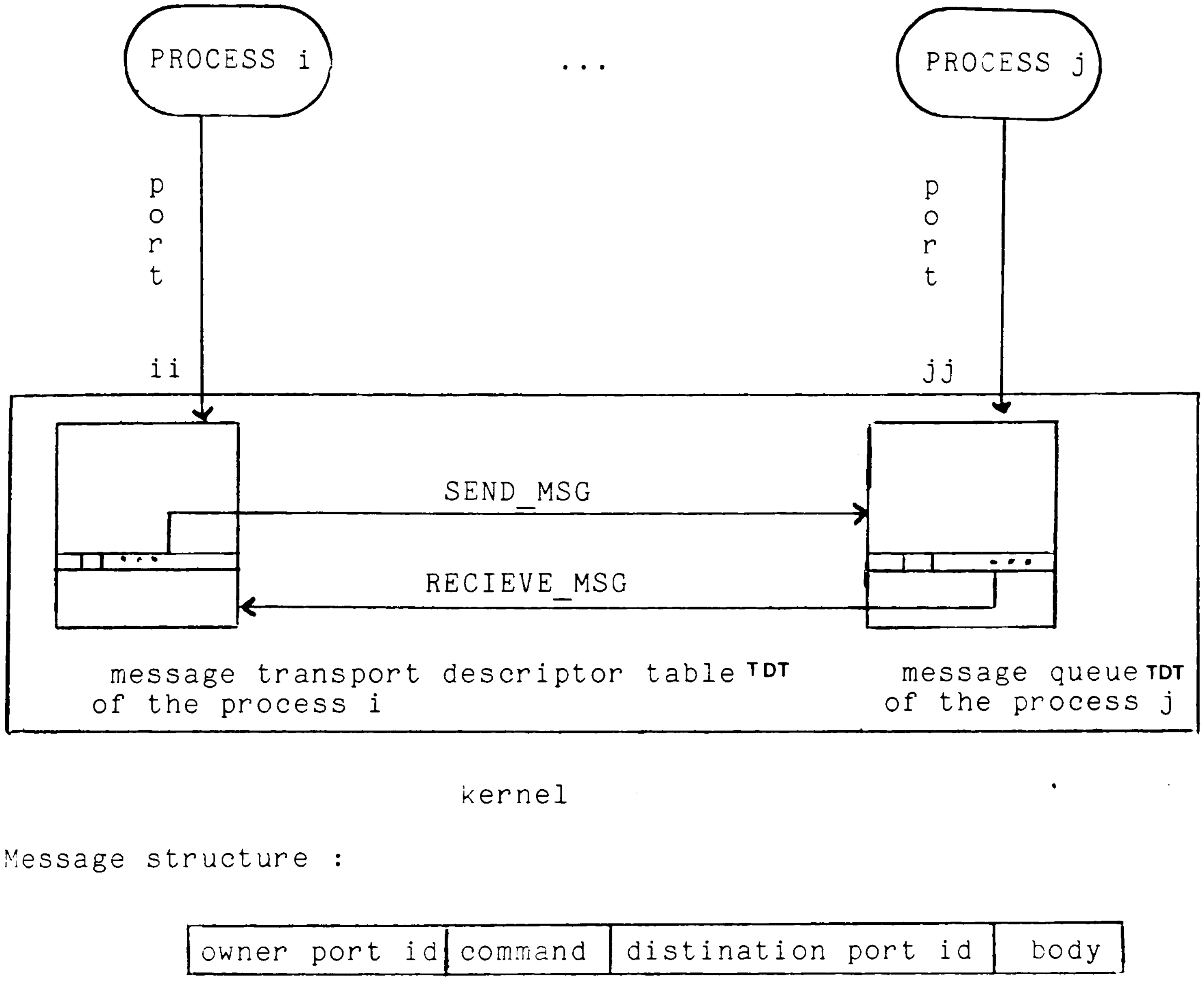


Figure 5.16: An abstract view of our communication data structures.

indirectly with any other processes, except via the kernel process. Processes communication is mediated by the kernel process which manipulates the messages that are picked-up and directed by the ports. Each message consists of a body (text of the message), and of the access transportation primitives (i.e. send or receive commands), a sender process message transport descriptor port, and receiver message transport descriptor port. The objective of the communication mechanism is to transport the sender message transport descriptor to the receiver message transport descriptor. For each process, all the messages (possibly none) to be mailed during some time period are placed inside a message transport descriptor table (or a bundle) which is addressed to the kernel process. The kernel process receives bundles, sorts messages, checks the protection status then requeues them into a single bundle (possibly empty) for each process.

The details of the information outlined in this section are presented in chapter 6, section 6.5.3. It should be noted that synchronisation of the executing processes is a byproduct of the communication structures activities (see assumption 8 of chapter 4).

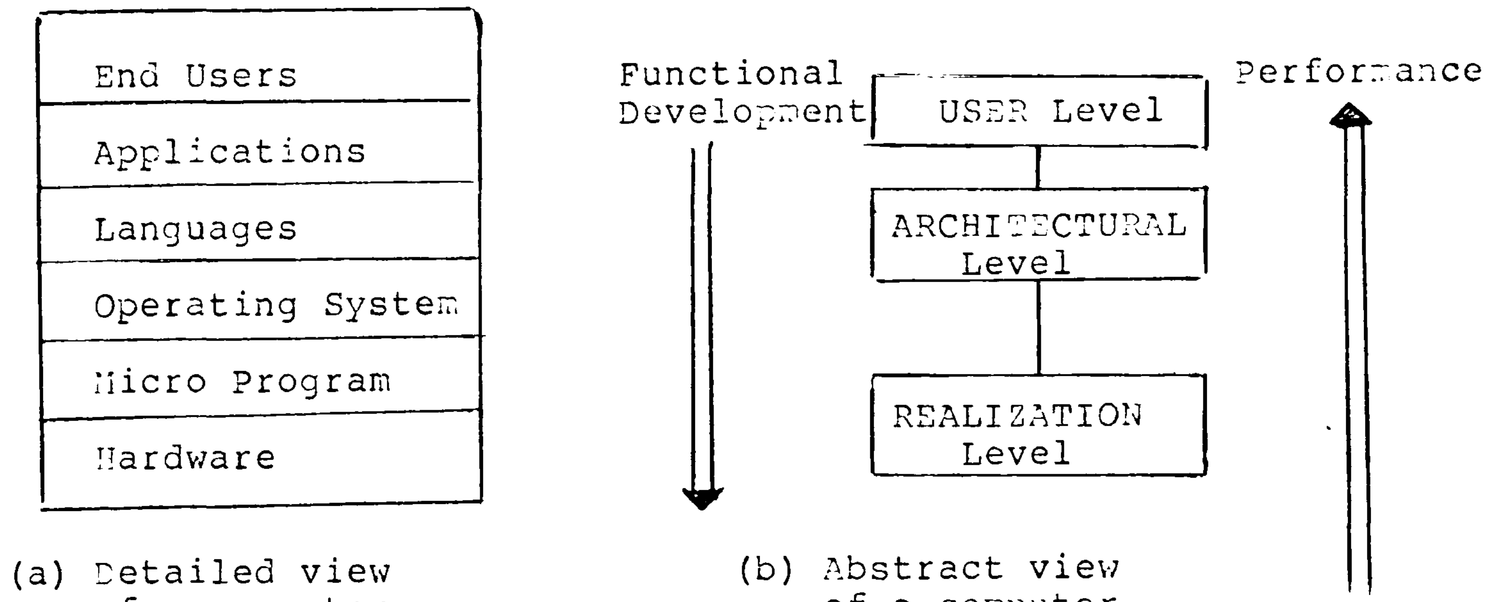
5.3.6 The interpretive structures :

Interpretive structures are concerned with the effective representations/interpretations of the simulated computer structures. The most important structure to be simulated is the process structure which consists of the data descriptor and the algorithms that carry out the instructions (c.f. assumption 5, section 4.7). This process structure, carries out the tasks required by the different functional structures and therefore it has to be efficiently represented. This requires to pay the attention to the following three design issues:

1. *The realisation level of the algorithm.* As illustrated in Figure 5.17, the implementation of algorithms on a level closer to the bottom layers yields higher performance at the price of lesser flexibility or transparency (in terms of possible changes of the module code). In general, implementation on a lower level is chosen when high performance is required. Vertical migration method may be used to transfer higher level modules to lower level modules in order to gain performance, but this method proves to be ad hoc and produces ill-structured system design (Stockenberg and van Dam 1978).

   However, many performance decisions at one level can only be made if the knowledge about the performance of the lower level system components is available. For the level we implemented, the complete shell algorithm is near the centre of the hierarchy (i.e. using a high level language) in order to gain a moderate functional flexibility and reasonable performance (in a way similar to the design of the Burroughs B6700 or the SWARD systems, c.f. Wegner 1971). The details of implementing in a high-level language, the shell algorithms of our tool are presented in the next chapter.

2. *The representation of the information structures.* Performance, at any design level, depends upon the structure of the computation process (i.e. size, etc) being performed,



(a) Detailed view of a computer hierarchy

(b) Abstract view of a computer hierarchy

Figure 5.17: Hierarchical layers of a computer system.

the characteristics of the information involved in the computation, and the way the different mechanisms treat the 'addressing'[6] of the structures in the computation. Changes in the way a computation is performed, or in the properties of the information being processed, have the impact on the performance of the system at a higher design level. Indeed, flexible interpretive structures should be used to tune the performance. We find that flexible data types such as *"descriptors"* are quite important to be used as our interpretive structure, since they can describe properties of any complex object altered during tuning these structures (Mohamad and Cavouras 1984). The next chapter presents our descriptors implementation.

3. *The representation of possibilistic changes.* In order to achieve a flexible representation of both, the algorithmic and the data structures, the change of representation must

[6] e.g. a memory addressing scheme, c.f. Fabry 1974.

be achieved in an effective and easy way. These changes are necessary to study the effect of the different design factors upon the performance, as well as to arrive at the admissible design factors that can let a particular system design to be optimal. In our shell design we adopted a *parameterisation technique* (c.f. Hughes 1981) which aims at binding the key changes to corresponding parameters. These changes can be elicited from the designer in a friendly way. We identified three groups of parameters:

(a) the user intention parameters for the outer shell,

(b) the complete shell initialisation parameters, and

(c) the computer inner shell design parameters.

The details of these parameters depend upon the way the implementation is carried out and hence it will be left to the next chapter.

5.4 Performance Probes of The Shell

Performance monitoring is an important process that provides the essential information about the shell status under the different workload and shell settings or representations. From the monitored information, the designer can tune or optimise the given computer design. The monitoring process uses special statements called the software probes. Software probes (i.e. monitoring code statements) can be used to collect the essential statistics. Here we list and define the essential performance probes that we used. These are:

1. The *user-oriented performance parameters probes* include the following:

   (a) *User productivity:* This measure provides the picture about the load of the system. There is a vast number of parameters contributing to the productivity of the computer system user within the outer shell. These parameters are called the user intention parameters (see section 6.2) and consist of parameters such as the average user think time, the job arrival speed, the average number of user tasks per a job, etc. The representative performance probe that we used to measure the average user productivity within the outer shell is a probe that measure the demand of the users (i.e. the number of competing concurrent user jobs) within the outer shell.

2. The *machine oriented performance probes* include the following[7]:

   (a) *Average Response Time:* This is the time required to respond to a user command issued by a terminal. The response time includes the overhead time, the request productive time, the time spent accessing the disc files and so forth. It is a complex function, which depends upon the number of active users in the system as well as on the actual design of the system itself. It thus follows that the response is rather meaningless measure unless it specified under which conditions it was measured. Consequently, average response time is more meaningful since it is based upon statistics gathered for a specified period of time.

   (b) *Turnaround time:* This measure reports the average time a batch job requires to pass through the system. It can be calculated from the average time spent on all the system resources.

   (c) *System throughput:* This is the average number of processes or jobs processed by the system per unit of the total elapsed time. The throughput of a system can determine the amount of the work that can be done per unit time.

   (d) *Effective degree of Multiprogramming or Multiprocessing:* This measure represents the average number of user processes that can communicate while being resident in the central memory.

   (e) *Devices utilisation:* This measure represents the time for which the devices are busy or idle. It is expressed as the ratio of the use or idle time to the total elapsed time.

   (f) *Devices queueing indices:* This set of measures indicates the efficiency of the services at each particular device. These measures are:

      - device queue size,

[7] Any of these measures may be used as the machine environment effectiveness measure.

- device queue mean length (i.e. the mean number of items waiting for the service for a period of time),
- maximum waiting time of the items in the queue,
- mean waiting time for the device.

(g) *Central processor overhead time:* This is the time spent by the central processor in performing the system functions. Overhead time can be a useful measure for the designer when he attempts to assess the cost of his solutions of a particular design problem.

(h) *Devices productive time:* This is the time for which the devices were used in processing user processes.

5.5 The Implementation Language of The Shell

The programming language is the media for selecting the substrata required for implementing the functional structures as well as the other simulation structures of the shell. In order to search for a programming language that is capable of describing the strategies, techniques and structures required by the activity structures shell design, and can achieve the maximum possible flexibility we should look for a language that does not restrict the programmer to only one **paradigm**. We have to search for multiparadigm languages (c.f. Hailpern 1986). A programming paradigm is a way of approaching a programming problem (i.e. a way of restricting the solution set). By analogy, structured programming restrains the programmer from using all the unstructured constructs available in a conventional language. That is, any acceptable paradigm allows the programmer to use only restricted set of concepts. Because our shell is a possibilistic design tool, we need a multiparadigm language in order to represent whole families of concepts and structures.

Conventional computer systems software (e.g. operating systems) is not adequate multiparadigm system because of the strict separation of the different paradigms and the static nature of their linkage. Ideally, a multiparadigm software system should allow language constructs from different paradigms to coexist within one program or module. Each paradigm of such a software system should be able to refer to, and depend upon, services provided by the other paradigms.

We conducted a study to determine what would be the most suitable multiparadigm programming language to implement or simulate an activity structures based designs (Mohamad 1982). The results of this study let us to select the programming language C for the following reasons:

1. It supports several programming paradigms that is
    - imperative programming paradigm (it supports sequential, block-structured commands with static scoping of variables)
    - object-oriented programming paradigm (it can group data into objects or data types where each object can possess a set of operations programmed to manipulate it).
    - parallel programming paradigm (using the scheme of coroutines, the specification of multiple processes can be done in the context of a single processor or distributed collection of processors).
    - real-time programming paradigm (incorporating a simulated clock or using the host computer clock, our simulation in C can specify all the constraints required to control the physical devices of a computer system).
2. The C programming language possess the following characteristics (Kernighan and Ritchie 1978, Bailes 1985, Deridder 1986):
    (a) it is a block structured language,
    (b) it supports information hiding,
    (c) it possesses powerful constructs (e.g. array of functions and procedures),
    (d) it supports dynamic memory management and allocation,
    (e) its source is portable,
    (f) supports coroutines,

(g) it has powerful preprocessor and macro substitution, and

(h) it is closer to the machine architecture than other "higher" level languages (such as Pascal, ADA, Modula)

3. Finally, another factor that leads us to select the C language is that the language is available on our host computer system.

However, the only disadvantage encountered with the C programming langauge is that it is not a strongly typed language. This is not a limitation of the language since it allows type definitions and type checking. Types in C are built up out of the basic types with the type operators as in Algol-68 or Pascal. The exception is that procedure declarations need to give only the result type, and not that of the arguments. To several researchers C types syntax and semantics is irreguler and messy (c.f. Anderson 1980).

Chapter 6

THE IMPLEMENTATION DETAILS OF THE SIMULATION OF AN ACTIVITY STRUCTURES BASED COMPUTER SYSTEMS POSSIBILISTIC GENERATOR

6.1 An Overview

The primary goal of this chapter is to describe the construction and the use of a possiblistic generator for generating activity structures based computer system architectures. We believe that the construction of such a possiblistic generator will lend credence to the claim that the methodology of activity structures is useful in constructing effective computer systems.

In chapter 5, the abstract way that initiate the construction of a required possiblistic generator was described. In this chapter, we concentrate upon the designer activities. These are:

1. **selecting the main design features required by the problem environment** (This consists of identifying the macro elements of the possibilistic generator),

2. **refining these features to produce the intrinsic design details** (i.e. identifying the micro elements of the possibilistic generator), and

3. **representing and describing the refined details in a suitable and flexible form of implementation** (This representation should take into account both, parametric and non parametric changes that are likely to be performed by the designer

during his/her exploration step).

There is the fourth step, the goal directed activity of the designer to perform changes to force the generated system to act in a way interesting to the designer. This is called the exploration step. This however will not be discussed here but it will be left to the next chapter.

6.2 The Designer First Activity: Selecting Components Of A Possiblistic Generator

The first design step is to elicit the design requirements from the user construction environment. For this purpose, we implemented a friendly program that is called the PREPROCESSOR and which is used to collect these requirements. The questionnaires in this program have been designed to include the intrinsic features of several general purpose computer systems.

The systems selected to be the candidates for testing my generator represent the most important computer systems which appeared in the current state of the art. These systems are claimed to possess the following interesting functional features:

1. protection enforcement,
2. communication mechanisms (e.g. message-passing),
3. sophisticated control structures,
4. inferential capabilities, and
5. effective representation and knowledge base structures.

After the survey of the literature I selected the following systems (Mohamad 1982, 1982a, Mohamad et. al. 1984):

- NUKE system (Crowley 1981),
- THOTH system (Cheriton 1979),

- GUTENBERG system (Stemple, Vinter and Ramamritham 1982),
- HYDRA system (Wulf et. al. 1974),
- KSOS system (McCauley and Drongowski 1979), and
- CAP system (Wilkes and Needham 1979).

These systems form a family that I shall call a *class of highly constrained existing computer systems*. What all these above listed systems have in common, is their basic construction unit-**process**.

The process can be described as an active functional entity (i.e. a management unit within the functional substructure) using the activity structures terminology (see postulate number 5).

An important factor that contributed to the selection of the above systems, is the fact that they have, in general, the same process organisation as the activity structures' process: the **message-passing** and the **hierarchy architecture**. Message-passing activity between system processes is achieved by means of output-to-input connections which utilise messages and ports. The hierarchical architecture of the system processes provides a mechanism for stepwise refinment that is required for the implementation of the functional substructures.

The processes that are found to be shared by the above mentioned class of highly constrained systems are the following:

1. the memory process,
2. the processor(s) process,
3. the kernel process mainly utlised for communication,
4. the devices control process,
5. the protection process,
6. the job scheduling process, and
7. the file system process.

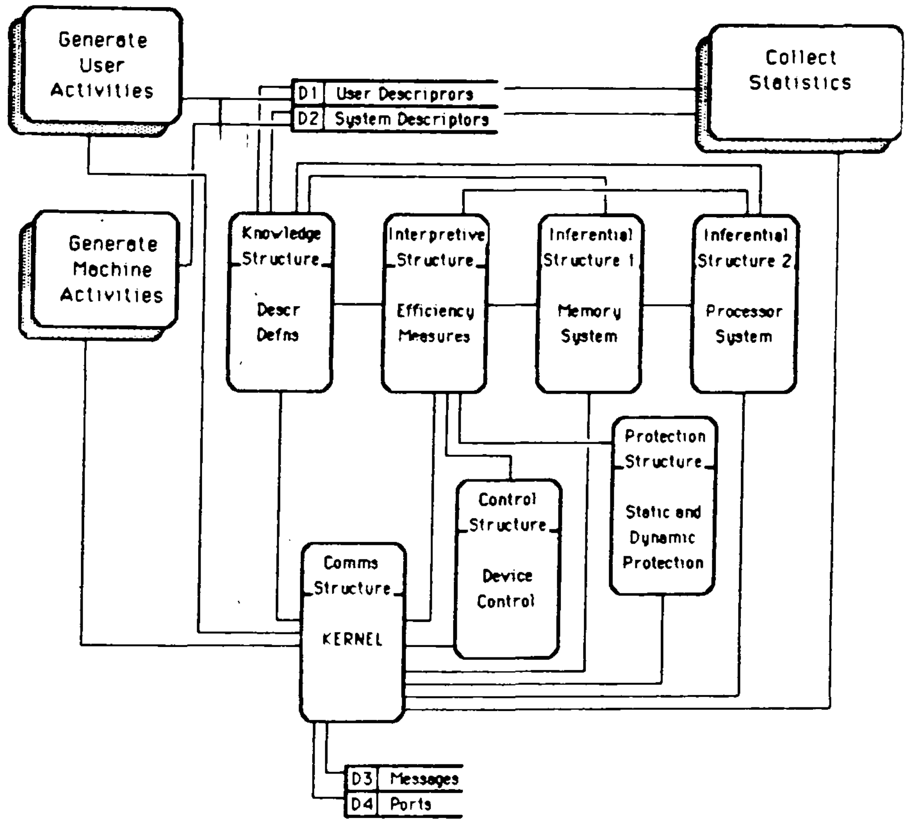


Figure 6.1: Fitting the processes of the class of highly constrained systems to the different functional structures of our generator.

The processes 1 to 7 above capture the activities of our selected class of highly constrained systems. In order to test the generator and study the behaviour of this class of systems using the activity structures based architecture, we have to put these into a one-to-one correspondence with the essential functional structures of the generator. The correspondence is depicted in Figure 6.1. The correspondence process may be controversial, but it is of vital importance for the initial *verification/testing* of the possibilistic generator behaviour (specially if the performance statistics is available for one of the existing class members). Also it is important for demonstrating that our possibilistic generator is capable of capturing the essential design features of any current highly constrained computer system design.

Figure 6.1 shows the interconnection of the functional structures of the generator with their overlay by the activities of the above defined class of highly constrained architectures. The actual realisation details of these processes are presented in sections 6.4, 6.5 of this

	Type of Question	*Explanation*
1.	Job type ?	interactive or general
2.	Job Arrival Speed ?	mean-interarrival time(sec)
3.	Average Processor Time ?	measured in microseconds
4.	Average Memory Space ?	measured in bytes
5.	Average No. of Job Tasks ?	(1,2,···,20)
6.	Mean Size of Backing Store space ?	measured in bytes
7.	Average No. of the Job's Backing Store files ?	(1,2,···,10)
8.	Mean No. of Backing Store Input Records ?	(1,2,···,1000)
9.	Mean No. of Backing Store Output Records ?	(1,2,···,1000)
10.	Average User Think Time ? (only for interactive type)	measured in seconds
11.	Average Indicies of Job Intention Service Policy (mh, minf, th , tinf) ?	see section 5.3.2
12.	Average Faulty Intended Accesses	(0,1)
13.	Mean No. of User Productivity	measured in transactions
14.	Average Satisfactory Response Time	measured in microseconds
15.	Average Satisfactory Turnaround Time	measured in microseconds

Table 6.1: User Environment Intention Parameters.

chapter.

The elicitation questionnaire must contain a list of queries that capture the *average user environment intentions*. The generator uses these intentions as seeds to generate random or variable workload within normalised intervals (the user learning mechanism (section) tries to obtain the optimal intervals) in which it can replicate a realistic computer system workload. Table 6.1 illustrates the type of queries adopted by our PREPROCESSOR program.

In our implementation, these parameters initiate the generation of the user intention steps. The process of generating these intention steps starts from the user environment (via two routines the PARTICIPANT-ONE and PARTICIPANT-TWO (see section 1.2)) which provide the average user demand expressed in the average number of concurrent jobs required by the user environment. The detail descriptions of each job requirement are assigned by the JOB-SCHEDULER routine which takes the intention parameters (i.e. as input seeds for random generators) and assigns to each job descriptor its random intention steps. The distribution used is the Poisson distribution. The reason behind selecting this type of distribution is that it replicates the computer system workload main characteristics e.g. the jobs arrivals (c.f. Steel and Torrie 1980). Here we should note that the PARTICIPANT-ONE and PARTICIPANT-TWO routines do not only generate the average user demand but also have the capability to inspect the performance status of the system and take corrective actions for adjusting the average number of concurrent jobs. The aim of these corrective actions is to force the system to show interesting behaviours. There is a secondary corrective mechanism which is controlled by the JOB-SCHEDULER. This mechanism corrects the amount of load assigned to each intention step. It operats by adjusting the normalisation interval size used around each distribution seed. This mechanism stores only the previous interval size used for each distribution function, since the best previous estimate and the new estimates will be compared to it. If the new estimate is better, the comparison is positive; if it is worse, the comparison is negative. These two corrective mechanisms help the user environment to adopt to the machine environment reaction activities.

6.3 The Designer Second Activity: Decomposing The Possibilistic Generator

In this section we are concerned with partitioning the functional activities of the upper abstraction phase (i.e. the Goal Phase), since we believe each optional pass from one phase to the other must be based on some perceived concepts of the previous phases. Only after the designer becomes increasingly familiar with a target system produced from one phase, she/he can use the optional path to the second phase and produce a more sophisticated target system better the one derived exclusively by the upper phase. However, the partitioning of the functional activities within the upper abstraction phase involves the six essential functional structures of Figure 6.1.

Moreover, the design partitioning step should prepare also for the next design step of exploring the target system. For this purpose, arrangement should be made at this stage, for any expected changes. In our generator we allow for possible changes within the algorithms of the functional structures and within the resource descriptors of the functional structures (i.e. changes within the computer machine environment). The algorithmic changes are performed by by means of what is called non- parametric alterations (see section 7.6.1). Each non-parametric change performed by the designer on the possibilistic generator, will produce a different version called a *possibilistic constellation* (see section 6.3.2). The changes on the resource descriptors are performed using parametric changes (see section 6.3.1). The parametric changes that are made on a possibilistic constellation will produce *possibilistic general computer system family* (see section 6.3.2).

These changes are elicited from the designer using the PREPROCESSOR program. The data collected by the PREPROCESSOR for this purpose is stored in a specific data file (SETTING-REDUCTION-DATA) that will be used by possibilistic generator for performing any possible alteration. The changes we performed upon the possibilistic generator of a general purpose computer system are described in the sections 6.3.1 and 6.3.2.

6.3.1 Producing Possibilistic Constellation by the Non-Parametric Changes

This type of change adds or removes certain functional substructures (i.e. *major changes within the computer machine environment*) that exist in the main possiblistic generator. In principle, many changes of this kind can be done within the design framework of the activity structures. However, we implemented only a limited, but, important subset of these functional changes. We can do the following in our simulation of the possibilistic generator:

-

6.3.1.1 adding/removing dynamic protection activities:

This change is made by answering the relevant questions asked by the PREPROCESSOR[1].

[1] The choice of dynamic protection, forces the protection descriptors to be of variable size, and consequently the memory must manage segmentation. In choice of static protection the memory may manage paging. This is signalled by the PREPROCESSOR automatically when the designer selects the type of

The protection activities that we selected form a part of the protection functional structures which enforce the protection of memory descriptors or segments/pages against unauthorised accessing processes (Mohamad 1982). The dynamic protection is provided in two wasy. Firstly, by passing as well as accepting access rights between the user processes. Secondly, by mediating between the access intentions and permissions of the processes. If an intention matches a permission, access is granted, otherwise the access is denied. This mechanism has been developed from a mechanism proposed earlier by Kohout (1978).

When the dynamic protection is required, the PREPROCESSOR program inquires about the following:

Q1 do you require to let the older (in time) processes to have lesser protection priority ? Since the older processes may gain permission to access a vast number of resources (sharing them with others), this will cause the performance to degrade. If we limit the effect of the older processes, then we expect performance to be enhanced.

Q2 if the answer is yes to the above question, then the protection policy enforced by the mechanism is called **limited sharing protection policy** (i.e. pass access rights are granted only from lower (older) (or the same level) to upper (new)(or the same level). The permit access rights are granted by that level only from the upper (or the same) priority level). Otherwise, if the answer is negative, then a **maximum sharing protection policy** is assumed.

These answers collected by the PREPROCESSOR are stored in a special data file called SETTING-REDUCTION-DATA which in turn will be read by a special routine within the possibilistic generator called the POSSIBILISTIC-SIMULATOR-LOADING routine. This routine also performs the following tasks of system intialisation:

- initialise the main lists pointers(e.g. process descriptor table),
- allocate all the available memory to a free list, and
- intialise the processes status (to executing).

memory protection he/she requires.

6.3.1.2 adding/removing users background blackboard:

Background users blackboard represent a mechanism by means of which the users can deposit their noninteractive tasks. The tasks are then performed in parallel or, at a later time, while the users are involved in another interactive task (on a foreground blackboard). This type of activities used in the Unix operating system where the user can operate his/her jobs on foreground/background queues (Dunsmuir and Davies 1985). These activities are managed by foreground/background blackboards (Dietterich and Buchanan 1983). By adding a background blackboard, an interactive system can be transformed to become a system of a general type (as in adding spoolers to a multiaccess computer system). This change is elicited from the designer by the PREPROCESSOR program and read by a special routine within the possibilistic simulator (POSSIBLISTIC-SIMULATOR-LOADING). If a background blackboard is required then the designer should specify the capacity of each background blackboard (measured by their capacity to hold jobs).

6.3.2 Producing Possibilistic Family by the Parametric Changes

In this section, the designer initiates the possibilistic constellation resource descriptors parameters to the required design setting (i.e. *performing minor changes within the computer machine environment*). The changes mainly specify the substrata capabilities (software and hardware) required to realise the activity structures based computer system. By setting these changes the possibilistic constellation is restricted to a *possibilistic family* of computer system. These changes are elicited by the PREPROCESSOR program and the data collected are deposited in a specific file called the

POSSIBILISTIC-GENERATOR-SŴHW-SETTING

The designer provides the relevant setting data based upon his/her experience or on the installation data collected from the manufacturer or some design manuals and texts (c.f. Shaw 1974, London 1973, Yourdon 1972). The relevant quetionnaires of the PREPROCESSOR are designed to reflect the software and hardware capabilities of the class of highly constrained systems.

The following are the questionnaires that are needed to produce the software and hardware characteristics supporting the class of highly constrained systems. For other, different hardware or classes of systems the questionnaires have to be approprrately redefined.

Q1: PROVIDE THE SOFTWARE SUBSTRATA CHARACTERISTICS OF THE
REQUIRED TARGET COMPUTER MACHINE ENVIRONMENT ?

Switching Time from One Process to Another in msecs =
Process Invocation Time in msecs =
Average Primitive Call Time in msecs =

Time to Service The Kernel Routine Which Deal With
Timer Interrupt =
Job Arrival =
Completion Interrupt =
Access Faults =
Abort =
Halt =
Send Message =
Receive Message =
Time Required to Generate An Activity (i.e Receive Event) =
Delete Port =
Create Port =
Changing The Memory Working Set =
Starting Process =
Stoping Process =
Initiating I/O =
Creating Protection Descriptor =
Destroying Protection Descriptor =
Modifying Access Rights (using Pass/Permit) =

Q2: PROVIDE THE HARDWARE SUBSTRATA CHARACTERISTICS ?

STATE THE PROCESSOR CHARACTERISTICS ?
Processor Speed (time to move on byte into the main memory)
STATE THE CENTRAL MEMORY CHARACTERISTICS ?
Average memory size for non resident processes (in bytes) =
Segment/page size (in bytes) =
STATE THE BACKING STORE DEVICES CHARACTERISTICS ?
The Disc characteristics ?
Disc transfer time (in msecs/byte) =
Disc positioning time (in msecs) =
Disc latency time (in msecs) =
Disc record size (in bytes) =
The Drum characteristics ?
Drum transfer time (in msecs/byte) =
Drum positioning time (in msecs) =
Drum latency time (in msecs) =
Drum record size (in bytes) =
STATE THE DEVICES REQUIRED TO SUPPORT THE FOREGROUND BLACKBOARD?

The media that is required to let the users to specify their required computational tasks on the foreground blakboard[2] is the 'terminal' device. For this purpose we assumed a general purpose DEC system terminal characteristics (300 band (i.e. 300 bit per second) serially transmitted) (Watson 1970). However, the designer must specify how many foreground user blackboards he/she requires ?

```
STATE TARGET SYSTEM PERIPHERALS NEEDED TO SUPPORT THE BACKGROUND
BLACKBOARD ?
```

If background blackboards are also required as initiated by the SETTING-REDUCTION-DATA then the relevant general computer system devices capabilities must be provided. The relevant devices are mainly those that support a batch computer system which include the line printer and possibly the card reader. Hence the parameters needed to be specified are:

```
  STATE LINE PRINTER CHARACTERISTICS         ?

  Line printer transfer time (in mescs/byte) =
  Line printer positioning time (in msecs)   =
  Line printer latency time (in msecs)       =
  Line printer record size (in bytes)        =

  STATE CARD READER CHARACTERISTICS          ?

  Card reader transfer time (in msecs/byte)  =
  Card reader positioning time (in msecs)    =
  Card reader latency time (in msecs)        =
  Card reader record size (in bytes)         =
```

6.4 The Designer Third Activity: Representing The Possibilistic Family

This is an important designer activity which realises the desired general computer system family. For this purpose the designer identifies the possibilistic (i.e. extensible) substrata structures which are used here to implement the manager algorithms and their resource descriptors of the possibilistic family. For a realistic implementation the designer should represent the design structures of the conversational environment.

[2] The concept of Blackboard is used here to refer to the media where the user job can be submitted. Blacboards are used differently in AI (c.f. Craig 1986).

6.4.1 The Generation of the Conversational Environment:

In order to replicate the conversational environments, we divided the simulation process of the possibilistic family into two-level simulation programs. The*outer level* program generates the user interaction environment (generating and scheduling the user activities) and provides the media for the declaration of the required information structures and their primitive operations. We refer to the outer level program as the JOB-SCHEDULER (note that the functions of our job scheduler is quite different from those used in the conventional operating systems. We selected this name (and possibly some other terminologies in this thesis), however, in order to be consistent with the current state of art computer system design terminology.).

The*inner program*, on the other hand, replicates the essential machine environment functional structures and is driven by activities that are**external** and**internal** to the possibilistic family. The External activities (interrupts) are caused by the hardware devices, whereas the internal activities (traps) are issued by the processes concurrently executing inside the system. All inner simulation processes are modelled using reentrant coroutines which possess their own activity lists, control mechanism and activity routines.

In order to obtain the activities forming the two, inner and outer environments, we need to create the essential scheduling mechanisms which act as their generators. These scheduling mechanisms take into account the criterion of randomness as well as the learning capability of the environment. The mechanism for generating the user interaction environment (the outer simulation) hence consists of a job scheduler. The mechanisms for generating the machine environment (the inner simulation) consists of the following:

1. the processor(s) demand scheduler,
2. resources demand scheduler, and
3. processes selection scheduler.

the two environments. In the following subsections, we briefly describe these scheduling components.

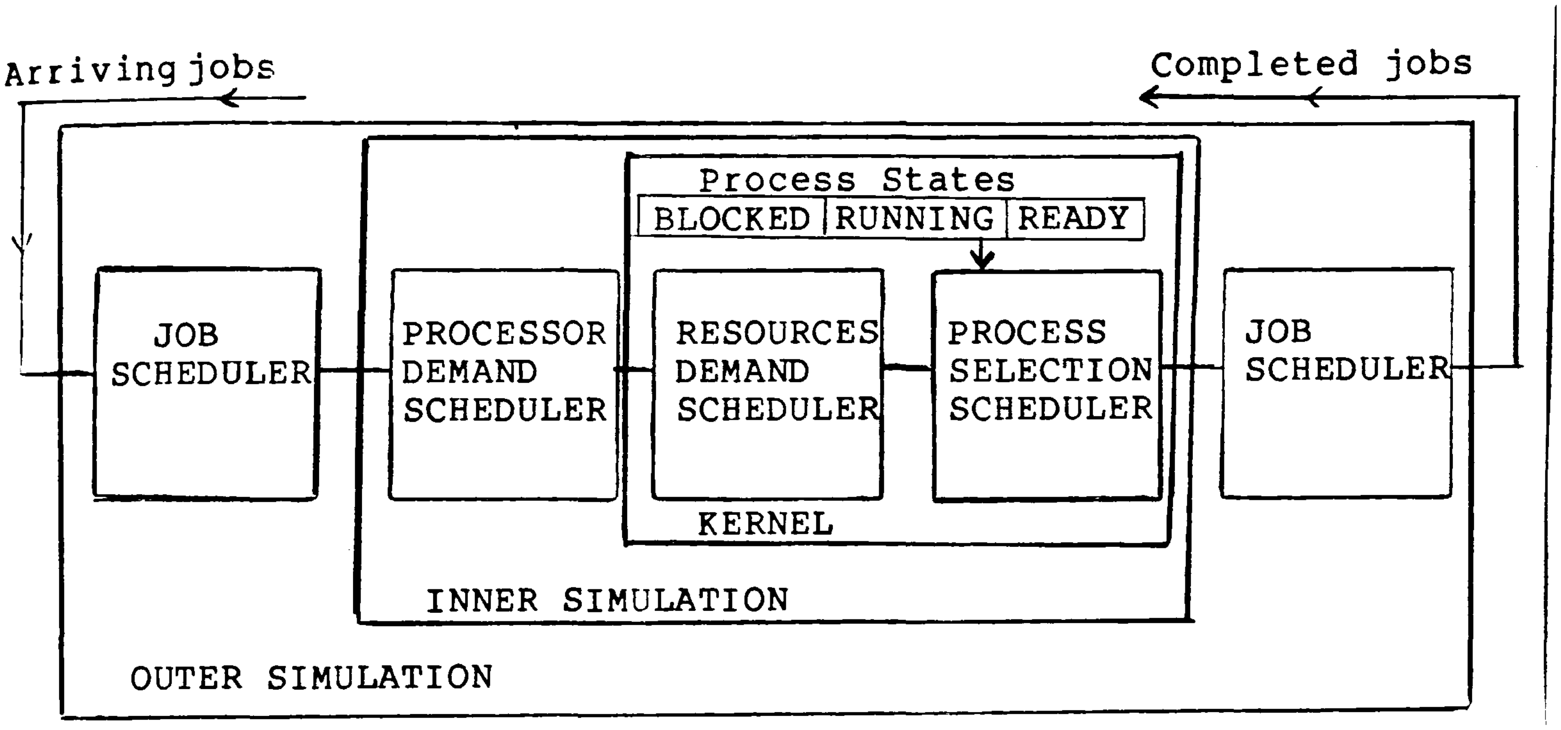


Figure 6.2: Schematic view of the scheduling components of

6.4.1.1 Generating the user environment activities: The Job Scheduler

The job scheduler can be viewed as a macroscheduler whose basic functions are:

1. to create user jobs descriptors (called JOBMIX);
2. to assign to each job descriptor the intention steps according to predefined distributions;
3. to order the JOBMIX according to their intended priorities as defined by the intention service policy function (section 5.3.2);
4. to administrate the learning of the user environment.

The job scheduler can be viewed as an overall supervisor which assigns resources to jobs according to the intention steps. Thus, it is entered when the user environment or machine environment changes. The sequence of actions that cause such changes are as follows:

1. a job arrives (from the background user blackboard) at the system or new user attempts to log-in (within the foreground user blackboard),

2. a job leaves the system or a user logs-out,

3. the output spooler completes the printing of a background user job,

4. periodically at a fixed time interval (the time slice (see section 5.2.2)) to assess the user learning trend (see section 7.2).

The algorithm defining the JOB-SCHEDULER tasks in our implementation is given below[3]:

```
JOB-SCHEDULER()
{
CHECK-POINT :
    READ-USER-INTENTION-SEEDS
    CREATE-FOREGROUND-JOB-MIX
    CREATE-BACKGROUND-JOB-MIX
    INITIALISE
    CALL PRODUCE-STAT
    FOR (EACH SEED) DO {
        RADMSEED = POISSON (SEED)
        GAUGE    = RESPONSE-TIME + TURNAROUND-TIME
        IF GAUGE <= THRESHOLD THEN {
            FIXMINSEED    = TRUE
            FIXMAXSEED    = FALSE
            MIDMINMAXSEED = FALSE
          /* FIX-MINSEED LEARNING */
            TEMP          = MINSEED
            IF (TEMP < MAXSEED) THEN {
                NORMSEED  = NORMALIZE (RANDSEED) /* BETWEEN TEMP AND MAXSEED */
                IF INITIAL THEN
                    JOB-DESCRIPTOR-SEED-FIELD = NORMSEED
                IF NOT INITIAL THEN
                  . IF GAUGE >= PREVGAUGE THEN/* NO IMPROVEMENT */
                        JOB-DESCRIPTOR-FIELD = PREVNORMSEED
                        ELSE/* IMPROVED PERFORMANCE */
                            JOB-DESCRIPTOR-FIELD = NORMSEED
               INCREMENT (TEMP)
               PREVNORMSEED = NORMSEED
               PREVGAUGE    = GAGE
           }
           ELSE
               STORE-BEST-SEED-FOR-BEST-GAUGE
           FIXMINSEED = FALSE
           FIXMAXSEED = TRUE
           PERFORM FIXMAXSEED LEARNING/* COMPARE TO BEST GAUGE */
           BEST-GAUGE    = NEW-BEST-GAUGE/* IF ANY */
           FIXMAXSEED    = FALSE
           MIDMINMAXSEED = TRUE
           PERFORM MIDMINMAXSEED LEARNING/* COMPARE TO BEST GAUGE */
           READ-INTERACTIVE-JOBMIX;/* AS IF GENERATED FROM THE USER
```

[3] The algorithms in this thesis are written is a pseudo-C language which was used originally by Professor Stephen Kaisler, the leading computer design consultant of George Washington University (Kaisler 1983).

```
                          TERMINALS OR THE FOREGROUND BLACKBOARD */
           READ-BATCH-JOBMIX;/* FROM THE INPUT-SPOOLER OF THE USER
                                          BACKGROUND BLACKBOARD */
           IF NUMBER-OF-JOBS <= (NUMBER-OF-TERMINALS
            OR INPUT-SPOOLER-CAPACITY) THEN
               CALL SCHEDULING-POINT/*FOR ACCEPTING MORE INTERACTIVE JOBS*/
           ELSE
               CALL NEXT-BATCH-JOB/* TO INVOKE THE NEXT BATCH JOB */
SCHEDULING-POINT :

           ISSUE-CALL-MESSAGE-TO-ALL-PROCESSES (PORTS);/*INCLUDING
                            UNDEFINED PORT FOR ANY USER PROCESS */
           IF MESSAGE-IS-RECEIVED-FROM-TERMINAL-MANAGER THEN
               CALL ENTER-INTERACTIVE-JOB
           CALL CREATE-ROOT-PROCESS
           CALL LOAD-ROOT-PROCESS
           IF MESSAGE-IS-RECEIVED-FROM-A-USER-PROCESS THEN
               CALL DELETE-ROOT-PROCESS
           IF MESSAGE-IS-RECEIVED-FROM-OUTPUT-SPOOLER THEN
               CALL DELETE-JOB
           IF MESSAGE-IS-RECEIVED-FROM-INPUT-SPOOLER THEN
               CALL DELETE-NEXT-BATCH-JOB-FOR-ACTIVATION
           CALL CREATE-ROOT-PROCESS
           CALL LOAD-ROOT-PROCESS
           IF MESSAGE-IS-RECEIVED-FROM-THE-STATISTICS-MONITOR THEN
               CALL COMPARE-WITH-PREVIOUS-PERFORMANCE
           IF THE COMPARISON IS NEGATIVE THEN
               CALL ADJUST-SEED-NORMALISATION-INTERVAL/* THIS IS
                     A PROCEDURE REPRESENTING THE USER LEARNING
                     CAPABILITIES WHICH EFFECT THE SPEED OF
                     ARRIVAL OF THE INTERACTIVE JOBS */
           }
```

6.4.1.2 Generating the computer machine environment activities:

There are in our simulator three schedulers that generate the computer machine environment activities. The schedulers describe the randomised machine activities and do not address the machine learning activities. These are done in a different part of the machine environment, the inferential structures. The three schedulers are described below:

6.4.1.3 DESCRIPTION OF THE PROCESSOR DEMAND SCHEDULER

This scheduler is the first part of the inner simulation responsible for generating the activities of the machine environment. It performs the following functions:

1. Orders by learning (according to the priotrities supplied by the inferential structures) the processes in the processor dispatching queue (i.e. move to the front the less

restricted processes).

2. Calculates the resource allowance of the processes.

3. Reports to the job scheduler.

4. Processes the events collected by the resources demand scheduler.

The processor demand scheduler is entered whenever a scheduling activity occurrs. The primary scheduling activities are:

1. creating or deleting a process.

2. reordering the dispatching queue at fixed intervals.

The creation and deletion of processes are requested by several parts of the possibilistic generator (including calls from the job scheduler, calls from the communication structure, etc.). Since the process creation and deletion contributes largely to the activities of the machine environment, we briefly illustrate our implementation of the way the processes are created or deleted within the possibilistic family.

The procedure responsible for creating and deleting processes is called PROCESS-GENERATOR.

The PROCESS-GENERATOR assumes the tree structure for the processes organisation, since it gives the most natural way of representation (Watson 1970). It is invoked whenever a parent process in the tree structure decides to create or delete a son. When asked to create a process, the PROCESS-GENERATOR has to read the process's program file from backing store. The program file contains information about the modules which make up the process such as memory descriptor length and locations, virtual devices numbers, etc. In a batch system, information can be also extracted from the Job Control Cards. The PROCESS-GENERATOR uses the jobs intention steps to set up the entries for the system processes table. From this table the PROCESS-GENERATOR creates the sons for the processes.

In a virtual memory system employing the working set concept to impose load control (see section 6.5.4) the PROCESS-GENERATOR has the additional responsibility of estimating the process's initial working set size, since an initial estimate of zero size can lead to a very large eligible set and the memory system(manager) queue can overflow (thrashing). The following algorithm defines the PROCESS-GENERATOR duties:

```
PROCESS-GENERATOR()
{
ESTIMATE :
    ESTIMATE-INITIAL-WORKING-SET-OF-THE-PROCESS/* USING A FRACTION
              OF THE PROCESS SIZE AND A FRACTION OF MEMORY SIZE */
    CALL GENERATE
GENERATE :
    CALL CALL-MESSAGE-TO-ALL-PROCESSES ( PORTS )/* INCLUDING
                          UNDIFINED PORT FOR ANY USER PROCESS */
    SWITCH COMMAND-OF-THE-MESSAGE-RETURNED OF {
     1 : CALL CREATE/* A PROCESS REQUIRE A SON TO BE CREATED */
     2 : CALL FAIL/* UNSUCCESSFUL LOADING THEN RETURN TO MAIN-ENTRY*/
     3 : CALL DELETE/* A PROCESS REQUIRE A SON TO BE DELETED THEN
                              RETURN TO GENERATE */
    }
}
```

We now provide the abstract algorithm used for the implementation of our processor demand scheduler:

```
PROCESSOR-DEMAND-SCHEDULER ()
{
   INITIAL-ENTRY { this entry set up several important variables,
                   such as free list pointer, processor
                   dispatcher list, etc. }
    MAIN-ENTRY{ In this entry the following is performed:

               (1) provide each new job with the important
                    processes port identifiers (e.g.  job
                    scheduler, memory manager) by issuing
                    RECEIVE-ACTIVITY primitive
               (2) if ACTIVATION is received then
                     call PROCESSES-SCHEDULING-MANAGER/* ORDER
                            THE DISPATCHING LIST */
                   else
                     switch message-command of
                      CREATE-PROCESS-DESCRIPTOR when new process
                                              is created
                      LOAD-CREATED-PROCESS and if failed issue
                                           LOAD-FAILED primitive
                      REPORT-ON-PROCESS-QUEUE in order to admit new
                                           processes
                      DELETE-PROCESS-DESCRIPTOR
```

6.4.1.4 The Description Of The Resources Demand Scheduler

This scheduler forms the second part of the inner simulation which contributes to the activities of the machine environment. Within our implementation this scheduler is considered also as a part of the communication structures. The functions of this scheduler are:

1. to allocate resources to processes as soon as they become available. Scheduling decisions taken at this scheduler determine the rate at which the system is able to respond to real time activity.
2. to simulate a virtual machine for each process and implement a set of primitives which enable concurrent processes to achieve mutual execution, synchronisation and communication with one another.

Since this scheduler is invoked whenever an interrupt (internal or external) occurrs, it function should be confined to the examination/modification of the states of processes and the collection of measurements by the processor demand scheduler as well as the job scheduler.

6.4.1.5 The Description Of The Process Selection Scheduler

This scheduler forms the third part of the inner simulation and contributes to the activities of the machine environment. This scheduler is also a part of the communication structures of the generator. Within the area of operating systems this scheduler is called often the dispatcher. It is invoked after the handling of an interrupt has been completed, in order to allocate the central processor various processes that demand it. Its function is limited to choosing the next process to be executed from the processor queue (the queue of eligible processes). In the following we present an abstract view of this scheduler:

```
PROCESS SELECTION SCHEDULER ()
{
   PROCESS = PROCESSOR-QUEUE-HEAD
   IF PROCESS is not empty THEN
      { /* here the processor is not idle */
      IF PROCESS is not the current-process THEN
         { /* perform context switching */
         RESTORE the context of the PROCESS
         current-process = PROCESS
```

```
        }
      ENTER-ANOTHER-PROCESS
      }
    CALCULATES-THE-IDLE-TIME-OF-THE-PROCESSOR
    IDLE-LOOP-WHICH-TERMINATE-BY-THE-ARRIVAL-OF-NEW-PROCESS
}
```

6.5 The Implementation Details of The Functional Structures

In this section we are introducing the implementation details of the essential functional structures, whereas in chapter 5 we introduced the abstract details of their functionalities only.

6.5.1 The Implementation of Knowledge Representation Structures:

The implementation of these structures represents a programming segment of code (added to the possibilistic family using #include macro), which includes the major information or knowledge structures declarations along with their manipulation routines (primitive issuing routines). The knowledge structures (INFORMATION-STRUCTURE) is a programming segment that includes the following three parts:

1. INFORMATION-STR-GENERAL DECLARATION part,
2. INFORMATION-STR-PRIMITIVE-ROUTINES part, and
3. INFORMATION-STR-SIMULATION-ROUTINES part.

In the first part, the essential data types are defined, such as **queue-structure**, **message-structure**, **activity-list**, **interrupt-list**, **port-descriptor**, **job-descriptor**, **job-descriptor-list**, **process-descriptor**, **process-descriptor-list**, and **segment-permission-descriptor**.

Figure 6.3 illustrates the essential descriptors used in our implementation to construct the possibilistic generator of computer systems.

These data structures generated by the data types are represented by certain linked lists and records. We detail here the structure of two of the most essential records. Those which cąpture the intentional notion of the user interaction environment via the job descriptor as well as the permitted capabilities of the machine environment via the process descriptor.

THE JOB DESCRIPTOR This is a record consisting of the following fields:

1. job identifier: system generated, a random number of 32-bit fix point number,
2. think time, this is expressed in miliseconds and is only used for interactive users,

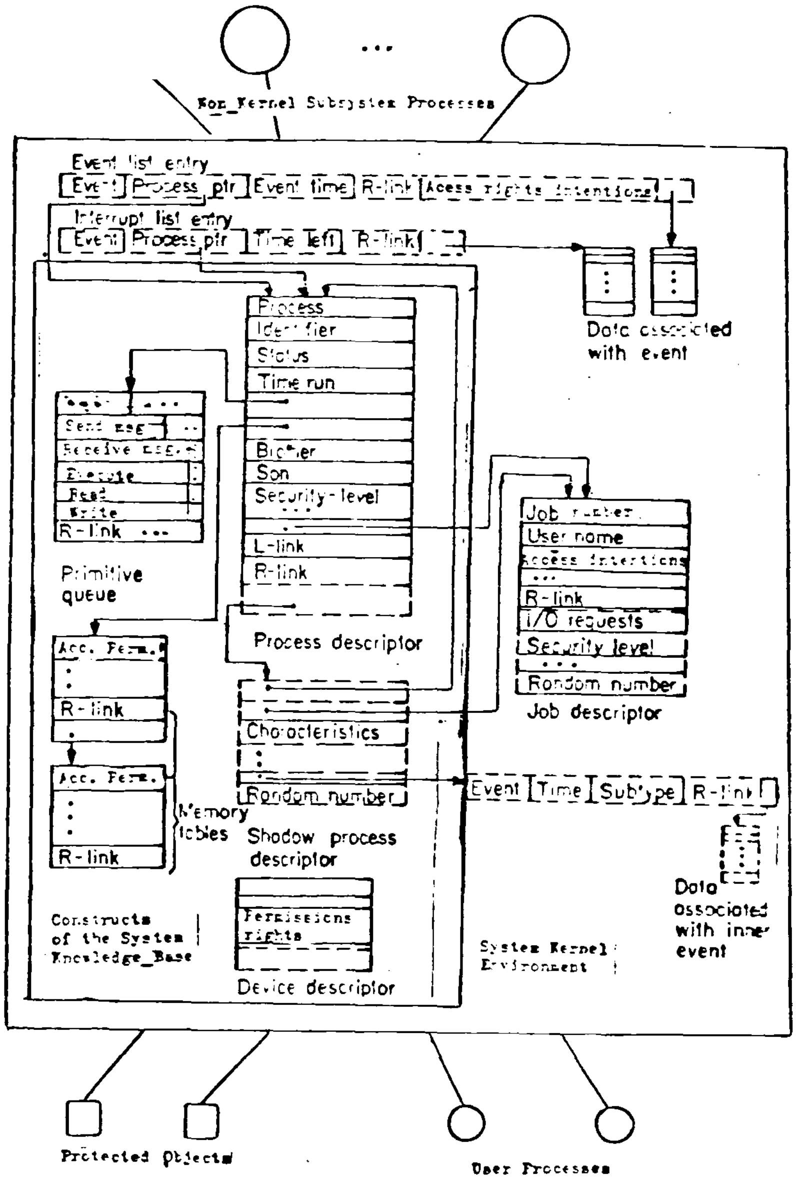


Figure 6.3: The essential descriptors of the possibilistic generator

3. forward job table pointer,
4. processor time, this represents the user estimated time of a batch (background) job , or the maximum allowed execution time for the interactive (foreground) job,
5. arrival time, specifies the time the job had arrived,
6. Job intention priority, a fixed point number extracted from the intention service function (mh,th,minf,tinf),
7. number of job tasks, it is assumed that one process will be created to carry out each task,
8. job type, 1 is batch and 2 for interactive,
9. job productivity mean number, number of input records or the number of multiaccess interactions,
10. number of backing stores records required,

11. number of output records,

12. number of user specified disc files,

13. the maximum central memory space required by the job, this is a 32-bit fixed number,

14. faulty access rate,

15. a boolean field determining whether the user allow to pass his access rights or not,

16. a boolean field determining whether the user permits other users copy their access rights,

17. pointer to job interpreter process,

18. accounting field, a floating point variable,

19. job status, this field take the following values,

    1 being read into input well for the background jobs,

    2 newly arrived and waiting to be activated,

    3 activated,

    4 failing to be loaded in the system (no space only if other job(s) leaves the system)

20. the mean satisfactory response of time, a number measured in microseconds (only for interactive jobs), and

21. the mean satisfactory turnaround time, a number measured in microseconds.

THE PROCESS DESCRIPTOR This is a records consisting of the following fields:

1. A process identifier,

2. The segment/page descriptor table entry number for the process's segment/page zero,

3. A pointer to the corresponding entry of the *shadow process table* which contains the characteristic of the process,

4. A save area which holds the current values of registers when the process is interrupted during its execution. In our implementation this consists of the process' entry point,
5. A one bit marker specifying whether the process has been interrupted at least once during execution or not
6. The entry of the segment/page the process is accessing
7. The set of the process's port descriptors
8. A message awaiting mask which indicates the ports through which messages will be accepted
9. Message queue head buffer pointer i.e. a pointer to the first message waiting to be received by this process. If there are no messages readable by this process, the value of the pointer is NULL.
10. Message queue tail pointer similar to the above
11. A state variable which specifies whether the process is ready to run or the reason for which it is blocked as follows:

    0 ready to run

    1 blocked for control access violation

    2 waiting to be killed (removed)

    3 waiting for message

    4 waiting for activity

    5 waiting for creation

    6 waiting for terminal interaction

    7 waiting for son to die

    A blocked (dead) process is not considered runnable by the dispatcher,
12. A state variable specifying whether the process can be started or the reason for which it is stopped as follows:

    0 not aborted (can be started)

1 aborted because its working set does not fit in the store provided

2 aborted while one of its segments/pages is removed from the store

A process is not considered runnable (executing) by the dispatcher if it is aborted.

13. Accumulated run (execution) time
14. Accumulated ready time
15. Accumulated blocked (dead) time
16. Accumulated unblocked (undead) time
17. CPU time used by the process until the start of its most recent interaction
18. A bit indicating whether any of the process's segments/pages are in store or not
19. A bit specifying whether the process has been activated or not
20. Process's working set size i.e. the number of bytes that its working set occupies
21. Process's critical time
22. Time the process has changed state
23. The eldest (alive) son identifier
24. The process's elder (alive) brother identifier
25. Backward pointer to the previous (in priority) process in the dispatcher list (processor queue)
26. Forward pointer to the next process descriptor in priority order, i.e. the forward dispatcher link
27. Identifier of the parent process
28. Accumulated resources used by the process
29. The start time of the "last" (most recent) interaction
30. Time the process had run until the previous processor demand scheduling time
31. Process's type
32. The accumulated number of messages received from each system process

In the second part of the INFORMATION-STRUCTURE, we define the following primitive issuing routines that operate upon the data structures defined earlier at the beginning of section 6.5.1. Among those routines, are the following:

- PASS-MSG(),
- CALL-MSG(),
- PORT-CREATE(),
- FIND-INTENDED-E-PROCESSES(),
- CALL-ACTIVITY(),
- PORT-DELETE(),
- PROCESS-STARTING(),
- PROCESS-STOPPING(), and
- SET-CHANNEL().

In the third part of the INFORMATION-STRUCTURE, we defined those routines that perform the primitive operations upon the data structures that belong specifically to the simulation. Among these routines are the following:

- INFORM-ERROR() which reports any error signal in the simulation process,
- PRINT-STATISTICS(),
- ADD-ACTIVITY-IN-LIST (),
- STORE-STATISTICS(),
- ADD-IN-LIST(),
- PRODUCE-STATISTICS() which possess the following routines
    - GENERATE-TIME(),
    - GENERATE-RANDOM-NUMBER(), and
    - GENERATE-DISTRIBUTION().

6.5.2 The Interpretive Structures: An Implementation Outline

This section is concerned with the effective representation of the following structures:

1. algorithms produced by the functional structures and
2. descriptors which record the design information requested by

the designer as well as any changes occurrring within the machine environment. The issue of structure representation is splited into static and dynamic representation. In section 5.4 we described some standard metrics by which we

measure the effectivity of the overall design. With the the use of interpretive structures, we are concerend with some additional measures that provide the design with flexibility, efficiency, and transparency. For this purpose we implemented the following facilities:

1. THE STRUCTURES FLEXIBILITY FACILITY: Using our highly parametrised possibilistic generator, we can change (using parameters) some characteristics of any descriptor declared within the INFORMATION-STRUCTURE or used for the simulator tables. This can be changed by using the required parameters within the POSSIBLISTIC-SIMULATOR-LOADING routine. By changing the descriptor size, for example, we can achieve the minimisation of the semantic gap between the generator and the host computer and optimise the best average segment descriptor size that can be used for our generator memory system.
2. EASE OF CHANGING THE SOFTWARE AND HARDWARE CAPABILITIES: Using our highly parameterised generator, we are able to change (using parameters) the software and hardware capabilities. The software structures in our tool represent the generator functional structures [4]. The parameters, specially the hardware paramerters, were extracted from the common characteristics used by the class of highly constrained systems (see section 7.5). However, in order to change these parameters, the

[4] software changes include interrupt timings of some functional structures (e.g. the communication structure), whereas the hardware structures are simulated by replicating their interrupt style and timing as well as their characteristics (e.g. device characteristics are stored in device descriptors)

designer needs only to execute the PREPROCESSOR program and to answer the relevant questionnaires. The required changes are then collected from the PREPROCESSOR pool of data by a special routine called POSSIBLISTIC-SIMULATOR-SWHW-SETTING. This routine assigns the questionnaire replies into a HW/SW setting file for defining the type and the characteristics of the software and hardware required. Afterwards, the support tool can be executed, in which this HW/SW setting file is used to initiate its required computer system model. Using this type of change, we can study the effects of different hardware and software settings as well as to find which of these settings are able to enforce the self-regulation criterion (see section 7.2).

3. SELECTING EFFECTIVE MECHANISMS FOR THE GENERATOR DYNAMICS: In our implementation, we achieved an effective representation of the generator dynamics by using two mechanisms. The first manages concurrency and synchronisation via using the notion of coroutines (see section 5.3.5). The second mechanism uses effective descriptors addressing. Here we would like to illustrate the function of the second mechanism, since it has been mentioned for the first time in this chapter. First of all let us list some traditional addressing policies ways and outline their implementation problems (see Table 6.2):

   Observing the problems associated with mapping tables should lead us to inquire whether it might be possible to avoid the need for a mapping table altogether. Our answer is that it can be done. In fact, the function of a mapping table is to establish a correspondence between the descriptors identifiers with locations. It can be avoided if the location of a descriptor is in fact its unique identifier. This suggests that there are at least two ways in which an identifier can be unique:

   (a) in time: as when the identifier is assigned from the current value of a clock or a counter, and

   (b) in space: as when the identifier refers to the current location of an object in a virtual memory with a linear address space.

	Maping Strategy	*Practical Problems Associated*
1.	add Base/limit field to the descriptor (e.g. Chicago MNC)	large overhead in updating descriptors (Fabry 1974)
2.	add descriptor mapping table– to overcome overhead	
	a. hashed mapping table (e.g HYDRA,System 250)	this sort of tables is feasible where the size of descriptors = memory segment size (ENGLAND 1974) and not for smaller or larger as in our case
	b. hierarchical mapping tables (e.g. CAP)	unsatisfactory for sharing descriptors; when a descriptor belongs to a process in a subtree it cannot be shared with a process in another subtree (Herbert 1978)

Table 6.2: Conventional Descriptor Mapping Techniques.

The second strategy has been used, because as the processes exhibit locality of reference, most descriptors should be located close to the objects they reference. We believe that this mapping strategy enhances the performance of the produced computer system design (Mohamad and Cavouras 1984). This strategy will be administered by the INFERENCE-STRUCTURE of our possibilistic family.

4. FLEXIBLE PROCESSES CONNECTIVITY: By this we mean that the allowable co-operation paths between processes during the execution of the poosibilistic generator can be initiated or changed by using the *port map*. The port map can be changed by altering the parameters within the POSSIBILISTIC-SIMULATOR-LOADING routine. Table 6.4 illustrates the port map adopted for our implementation for the notable processes that are generated by our possibilistic generator.

Process Name	*ID*	*Port Connections*							
Any other process	-1								
IS-Memory(Main Core)	0	12	8	4	1	5	-1	-1	-1
IS-Processor Process	1	12	0	8	-1	-1	-1	-1	-1
CS-Operator Console	2	12	2	-1	-1	-1	-1	-1	-1
CS-Terminal manager	3	12	3	-1	-1	-1	-1	-1	-1
IS-Memory (Drum)	4	12	4	-1	-1	-1	-1	-1	-1
IS-Memory (Disc)	5	12	5	-1	-1	-1	-1	-1	-1
CS-Line printer	6	12	6	-1	-1	-1	-1	-1	-1
CS-card reader	7	12	7	-1	-1	-1	-1	-1	-1
Process generator (Machine activities)	8	12	1	0	5	-1	-1	-1	-1
CS-File system	9	12	5	-1	-1	-1	-1	-1	-1
CS-Output Background Blackboard	10	12	9	6	-1	-1	-1	-1	-1
CS-Input Background Blackboard	11	12	9	-1	-1	-1	-1	-1	-1
Job scheduler (user activities)	12	12	8	1	3	10	11	-1	-1

Table 6.3: The Port Map, Where IS : Inferential Structure CS : Control Structures

6.5.3 The Communication Structures: The possibilistic generator kernel

The kernel of the possibilistic generator consists of communication structures together with scheduling mechanisms for generating the activities of the machine environment. The purpose of the kernel is to provide an environment in which the processes can exist; this implies handling interrupts, switching processor(s) between processes and implementing mechanisms for interprocess communication and synchronisation. The kernel is automatically entered in any of the following circumstances:

- an interrupt occurrs,
- a process issues a primitive to execute some function or requesting a kernel instruction (i.e. privileged for protection purposes).

Hence, the kernel consists of the three parts:

1. the *intialisation process*,

2. the *interrupt handler* (initial handling of all interrupt),

3. routines which implement the inter-process *communication and other primitives* (these routines are invoked via primitive calls in the process concerned), and

4. a *scheduler which switches processor(s) between processes* (some times it is called selecting process scheduler or the dispatcher).

The *initialisation process* performs the following tasks:

```
PICK AN ACTIVITY FROM THE ACTIVITY LIST
IF ACTIVITY-TIME >= OBSERVATION PERIOD THEN
   CALL PRINT-STATISTICS
 IF ACTIVITY-TIME >= SIMULATION-TIME THEN
    TERMINATE SIMULATION
 ELSE
  ASSIGN THE ACTIVITY CHARACTERISTICS TO GLOBAL VARIABLES
  ADVANCE THE SIMULATION TIME TO THE SPECIFIED ACTIVITY TIME
  DELETE THE ACTIVITY THAT IS READ FROM THE ACTIVITY LIST
  IF CENTRAL PROCESSOR IS NOT BUSY THEN
     UPDATE THE CENTRAL PROCESSOR IDEAL TIME
  IF THE CENTRAL PROCESSOR IS BUSY
     AND THE ACTIVITY TYPE IS AN EXTERNAL INTERRUPT THEN
     SUSPEND THE EXECUTING PROCESS
     IF THE SUSPENDED PROCESS IS A SYSTEM PROCESS THEN
        INSERT THE ACTIVITY IN THE INTERRUPT LIST
     ELSE
        INSERT THE PROCESS IN THE (USER) ACTIVITY LIST
  CALL U-EXECUTE-T
  TRANSFER CONTROL TO THE REQUIRED KERNAL ROUTINE
            SPECIFIED BY THE PICKED ACTIVITY
```

The *interrupt handler* consists of interrupt service routines which are responsible for responding to signals both from the outside world (external interrupts) and the unusual conditions from the processes in the system itself. The functions of these routines are to determine the sources of the interrupt, save the required information, and service the interrupt. These functions are clearly dependent on the hardware structures and the type of interrupt involved.

The occurrence of an interrupt (external or internal) will often alter the status of some process (e.g. from dead to ready). One of the consequences of the status change is that the process executing before the interrupt has occurrred, may not be the most suitable to run afterwards. The question of, when to switch the processor between the processes, and which process to pick up, is the task of the dispatcher. Hence, the control flow is always transferred after serving any interrupt routine to the dispatcher.

However, the *interrupt routines* are the primitive routines designed to respond to the external and internal interrupts. The external interrupts routines include:

1. INTERRUPT-TIMMER(it schedules the next timer activity and invokes the PROCESSOR-SYSTEM),
2. JOB-ARRIVING-INTERRUPT (this is invoked by the TERMINAL-ROUTINE when new user had loged in or by the JOB-SCHEDULER when a batch job is activated), and
3. RETURN-FROM-INTERRUPT (to report completing the interrupt (0 for successful, 1 for unsuccessful completion).

The task of the internal interrupt routines is to service the traps. When a process at any time during its execution causes fault, aborts (because of protection rules violation, overflow, etc.) or if causes issuing a privilege instruction such as PROTECTION-VIOLATION-RULE it calls the interrupt routine. All internal interrupt routines involve blocking of the process that caused the interrupt. Thus all the internal routines return control to the dispatcher except MISSING-MEMORY-SEGMENT (see below). The internal interrupt routines consist of:

1. CHECK-SEGMENT (to check any segment/page existence in the main memory if it is control is transferred to the calling process, otherwise to call MISSING-MEMORY-SEGMENT),
2. MISSING-MEMORY-SEGMENT (is invoked when a missing segment has occurred, to send a message to the MEMORY-SYSTEM and to block the calling process),
3. PROTECTION-VIOLATION-RULE (it is invoked by the PROTECTION-SYSTEM to terminate a process, hence issuing DEATH-SENTENCE to that process and to all the processes that are the process' descendants),
4. UPDATE-DEAD-TIME (updates the accumulated time the process spends in the dead state),

5. U-EXECUTE-T and U-READY-T (updates the times the process in the executing and the ready states), and

6. U-RESUME-T (updates the times the process spent in ready state).

In the second part of the kernel are the communication routines. These include the following routines:

PASS-MSG This communication routine is invoked by any process to send a message to any other process. This routine performs the following operations:

```
PICK THE MESSAGE THAT IS REQUIRED TO BE SENT
USE THE INVOKING PROCESS ID IN THE MSG
CHECK IF THE INVOKING PROCESS STILL EXIST
IF IT DOES NOT EXIST THEN
    A REENTRY IS MADE TO THE INVOKING PROCESS
    ADVANCE SIMULATION TIME BY THE REENTRY TIME
IF IT EXIST THEN
    CALL U-EXECUTE-T /* UPDATE EXECUTION TIME OF THE INVOKING PROCESS */
    ADVANCE SIMULATION TIME BY THE SEND TIME
    CALL MOVE-MESSAGE /* TO SEND THE MESSAGE */
```

MOVE-MESSAGE This procedure is used to carry a message to the sender. It has one parameter, the receiver port number. It performs the following operations:

```
A MESSAGE DESCRIPTOR IS ALLOCATED
ASSIGN WITH THE SENDER'S IDENTITY + MESSAGE CONTENTS
UPDATE THE CURRENT AND MAXIMUM MESSAGE BUFFERS
IF RECEIVER IS NOT A USER PROCESS THEN
   CALL UPDATE-STATISTICS/* UPDATE THE RECEIVER PROCESS
              MESSAGE QUEUE STATISTICS */
ELSE/* THE RECEIVER IS THE TERMINAL MANAGER PROCESS */
   IF THE SENDER IS NOT THE KERNEL PROCESS THEN
     END OF INTERACTION
     CALL UPDATE-STATISTICS
   IF THE SENDER IS THE TERMINAL MANAGER THEN
     A NEW TRANSACTION TAKES PLACE
     ASSIGN THE CURRENT SIMULATION TIME TO THE PROCESS
           START INTERACTION TIME
     ASSIGN THE PROCESS STATUS TO READY
RETURN FROM MCVE-MESSAGE
```

CALL-ACTIVITY This routine is used to call an activity. The calling process issues a message (the message postmark is changed to activation) by calling CALL-MSG routine. If it receives the reply back, the activation postmark is cleared and the simulation time is advanced by the time difference defined as the time needed to receive an activity minus the time needed to receive a message. This difference is called activation time).

CALL-MSG This procedure is issued by the processes requiring a reply from other processes. It can, however, accept or reject any of these messages. The routine performs the following:

```
PICK A MESSAGE FROM THE PROCESS MESSAGES QUEUE
EXAMINE THE MESSAGE BUFFER
IF MESSAGE IS ACCEPTED THEN
   DECREASE THE MESSAGE QUEUE ENTRIES IS DECREASED BY ONE
   IF THE CALLING PROCESS IS A NON USER ONE THEN
      CALL PRODUCE-STATISTICS
IF A MESSAGE IS NOT ACCEPTED THEN
   CALL U-READY-T
   ASSIGN TO MESSAGE POSTMARK THE WAITING STATE
```

PROCESS-STARTING This routine clears the aborted state of the process in the specified port. The U-EXECUTE-T routine is to update the executing process' executing and ready times. Then the control flow is transferred to the dispatcher.

PROCESS-STOPPING This routine sets the status of the executing process to the ABORT state. The caller process is reentered.

PORT-DELETE The specified port receives the status "undefined"(-1) and the caller process is reentered.

PORT-CREATE specified port is assigned the process identifier contained in the message and the caller is reentered.

FIND-INTENDED-E-PROCESSES Assigns the intended number of processes that execute at the same time, and then reentered the caller process.

SPLY-TIME This retunes the system (simulation) time.

SET-CHANNEL This routine initiates a given channel program. It is invoked by the CONTROL-STRUCTURE to start input or output operations. The messages used to set the channel include the device number, the data block contents, and size in bytes. Then the current simulation time is advanced by the time required to execute the SET-CHANNEL routine. If the channel is set to a device that is not a terminal device, then:

```
THE DEVICE STATUS IS CHANGED TO BUSY
UPDATE THE DEVICE IDEAL TIME
SCHEDULE WHEN THE USE OF DEVICE WILL BE FINISHED/* IT IS
     SIMPLY = CURRENT SIMULATION TIME + DEVICE OPERATION
     TIME (DEVICE POSITIONING TIME+ LATANCY TIME +
     TRANSFER TIME */
```

If the device is a terminal, an exponential sample is derived from the user think time and it is added to the current simulation time to schedule the next terminal interrupt. At the end, the CONTROL-STRUCTURE (or the DEVICE-SYSTEM) is reinvoked.

WAKE-PROCESS This routine receives the port number of the process that is needed to be activated. A message with an activation marker is sent to the process needed to be a wakened. If that process is not activated, its status is changed to ready state. If it was activated before, a message informing that it has already been activated is returned.

The fourth part of the COMMUNICATION-STRUCTURE is the dispatcher. It is concerned with picking up the ready and aborted processes for execution. The operations performed by this part are described in the following:

```
THE DISPATCHER() {
    PICK THE FIRST PROCESS AT THE DISPATCHER QUEUE /* i.e. HEAD-PROC */
    INITIALISE PROCESS-COUNTER (PROC-COUNT) TO ZERO
    WHILE HEAD-PROC IS NOT EMPTY
      AND PROC-COUNT <= MAXIMUM PROCESSES ALLOWED FOR PARALLEL
      EXECUTION DO
          IF THE PICKED PROCESS IS READY THEN {
              IF THE HEAD-PROC <> THE JUST EXECUTING PROCESS THEN {
                  ADVANCE SIMULATION TIME BY CONTEXT SWITCHING TIME
                  ADVANCE KERNEL OVERHEAD TIME BY CONTEXT SWITCHING TIME
                  JUST EXECUTING PROCESS := HEAD-PROC
              }
          CALL U-RESUME-T
          ADVANCE SIMULATION TIME BY THE INVOKING PROCESS TIME
          IF HEAD-PROC IS A USER PROCESS AND ALREADY BEEN INVOKED THEN {
              MARK IT ACCESSING SEGMENT
              INCLUDE THIS SECTION IN THE WORKING SET
              IF THIS SEGMENT IS NOT IN MAIN MEMORY THEN
                  CALL MISSING-MEMORY-SEGMENT
          }
     /* THE SEGMENT IN MAIN MEMORY OR IS UNDEFINED */
        IF THE PROCESS HAS NOT BEEN ABORTED THEN
            INVOKE ITS COROUTINE
        ELSE
            RESTORE IT
        IF THE HEAD-PROC IS NOT A USER PROCESS THEN {
            PUT HEAD-PROC STATUS TO READY
            SAVE DESCRIPTOR OF HEAD-PROC FROM THE INTERRUPT LIST
            UPDATE THE INTERRUPT LIST STATISTICS
            DELETE THE ENTRY (DESCRIPTOR) FROM INTERRUPT LIST
        }
        ELSE IF IT IS A USER PROCESS {
            SAVE THE DESCRIPTOR OF THE USER ACTIVITY LIST
            IF THE DESCRIPTOR REFER TO SEGMENT FAULT THEN
                ASSIGN THE ACTIVITY DESCRIPTOR TO THE INTERRUPT LIST
            IF THE PROCESS IS ABORTED THEN
                CHANGE ITS STATUS TO READY
            UPDATE THE USER ACTIVITY LIST
            DELETE THE ACTIVITY DESCRIPTOR FROM THE ACTIVITY LIST
        }
        INSERT A NEW ACTIVITY DESCRIPTOR PROCESS
    }
    PROC-COUNT := PROC-COUNT + 1
    HEAD-PROC := THE NEW PROCESS IN THE DISPATCHER LIST
  }
  ASSIGN THE PROCESSOR STATUS TO NOT BUSY
  RETURN TO THE KERNEL INITIALISATION PROCESS
}
```

6.5.4 The Inferential Structures:

This section is concerned with the automatic adjustment of the number of eligible processes allowed to execute at the same time and with the maximisation of the total amount of users productivity (or total load). The inferential structures are responsible for preparing the eligible set which consists of the processes residing in the main memory. The inferential structures select processes from the inactive set in the activity list with the ready status, (i.e. from the set of jobs residing in the system queues or possibly in the secondary storage, ready to be executed (see Figure 6.4).

The contents of the eligible set influences the performance of the computer system. Both user-oriented and system-oriented performance measures are affected by the inferential structures. Indeed, user-oriented and system-oriented objectives may be in conflict. A major problem in the design of the inferential structures is the resolution of this conflict to force the system to survive.

Associated with each user job is an intention policy service function (i.e. a user-oriented measure) which is a function of the total system time (see Figure 5.9 of section 5.3.2).

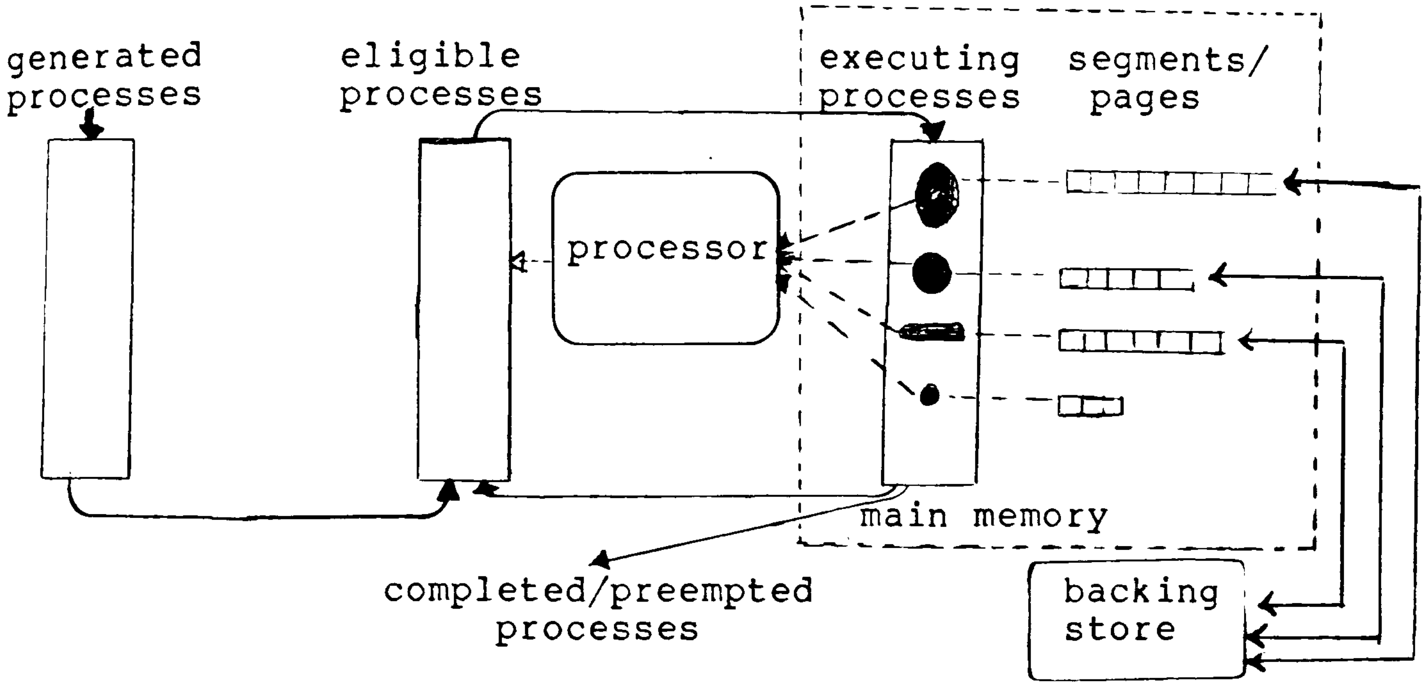


Figure 6.4: Processes Flow in the Two Inferential Components.

The intention policy function represents the user losses for having to wait for the systems response. This user loss function varies with the system time and according to certain specified deadlines.

On the other hand, the system losses, like the user service policy function, can be expressed by adifferent function that consists of two parts. The first part, as shown in Figure 6.5, is a rather sharp loss function, whereas the second reflects a somewhat smaller loss rate. This system loss function represents a measure of the generator's processor utilisation. A system loss function that assesses the processor utilisation loss only is chosen because the processor system represents one of the significant devices the management of which is largely affects the overall system performance[5]. The reason for the choice of the function of such a shap is that when the processor is partially allocated to the system competing processes, deadlocks conditions may develop (cf. Coffman et al 1971) or processor

[5] The processor device refers to the virtual CPU of mutliprocessing computer system consisting of a uni-processor device.

overhead time may rise sharply. The latter occurs when the admitance of the competing processes aquiring the processor is not under control causing the thrashing condition to be reached (c.f. Peterson and Silberschatz 1983).

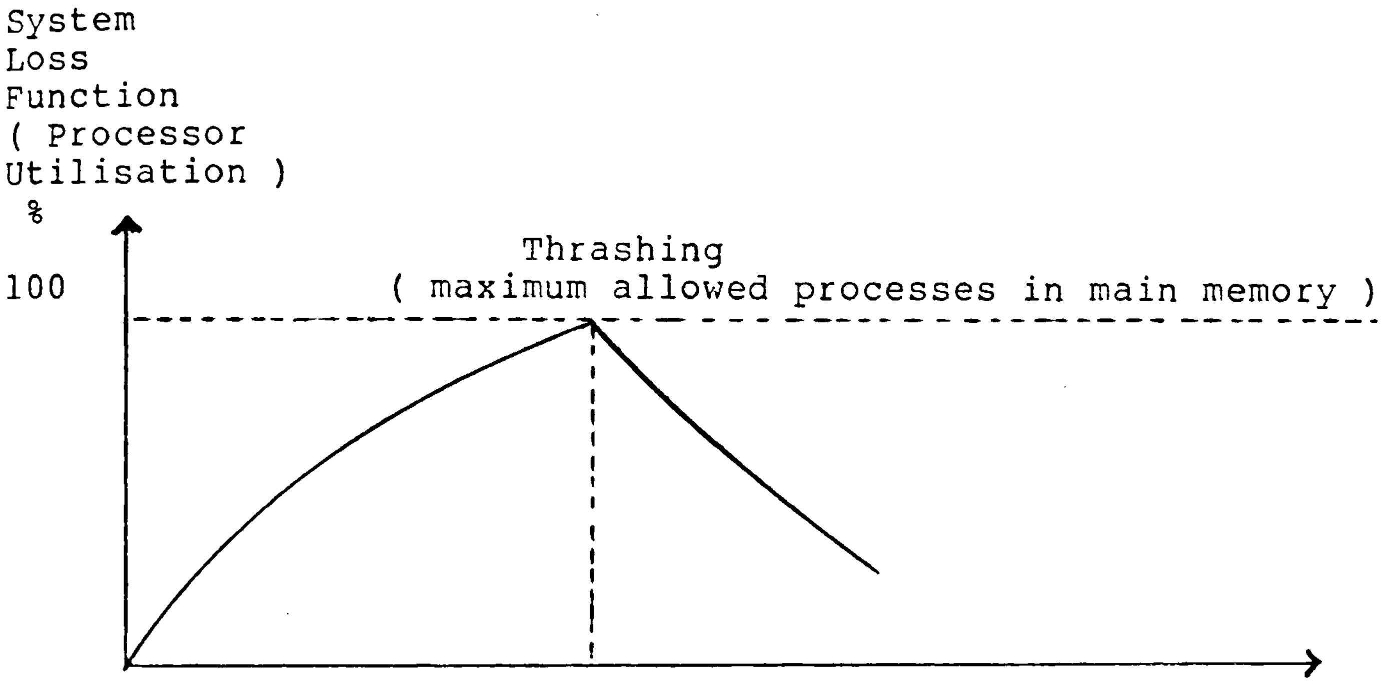


Figure 6.5: The System Loss Function.

The inferential structures, in our implementation represent a group of algorithms that balance the user loss (or user demand as in section 7.6) with the system loss (or machine environment effectiveness as in section 7.6). The inferential structures comprise two algorithms: a memory system and a processor system algorithms.

The first algorithm (i.e. the one concerning the memory system) makes a periodic assessment about the number of segments/pages to be held in the main memory before it is requested by the processes utilising the central processor. If this algorithm is not present, the executing processes will be continually interrupted by the segment/page faults which contribute towards thrashing. Keeping the number of the segments/pages to the necessary minimum is known to us as the**working set** (Note that is slightly different from the original definition of Denning 1968). In order to avoid thrashing, the pages/segments number demanded by the eligible processes that are striving to be in the main memory must be no greater than the maximum number of working set size that can be held in the main memory. Adjusting the demand according to the memory size available, requires the adjustment/ordering of eligible processes, which is the responsibility of the second algorithm.

The second algorithm (i.e. the one concerning the processor system) adjusts periodically the number of eligible processes so that the number of the processes actually entering the main memory is, on average, equal to the target number calculated in advance (i.e. to match the processes demand of pages/segments according to the availability of memory size). The design of these two algorithms is discussed in the following two sections.

6.5.4.1 The First Inferential Structure: The Memory System

In order to construct realistic simulation of a computer system memory management, we implement the following basic functions[6]:

1. Recording function
2. Fetch function
3. Placement function, and
4. Replacement function.

We describe these functions in the sequel.

Recording Function *keeps track of the location of the pages of each process in the memory system.* Note that the way we record this information is quite important from the performance point of view, as indicated by the INTERPRETIVE STRUCTURES (section 6.5.2).

In our simulation, we record location information in two structures:

1. the *page location stack* which stores the main protection descriptors (sometime called capabilities (c.f. Wilkes and Needham 1979)) and
2. the *main memory tables* (free and occuppied) for storing the actual page contains.

[6] Segmentation is used when the dynamic protection is incorporated within the memory system; otherwise paging is used.

Page Location Stack (PLS) is the central recording medium that indicates the actual location of pages and stores the information on them. We choose a stack structure for the following reasons:

1. Since each process exhibits locality of reference then the stack structure provides the right structure for keeping close to the top of the stack, the pages that are referd to by the requesting process;
2. Since each entry of the stack has a unique location, hence this location can be used as the unique identifier of the address of the page thus minimising the page size (it is known that minimising page size is an important factor affecting performance (Lindsay 1973)) (see Figure 6.6).

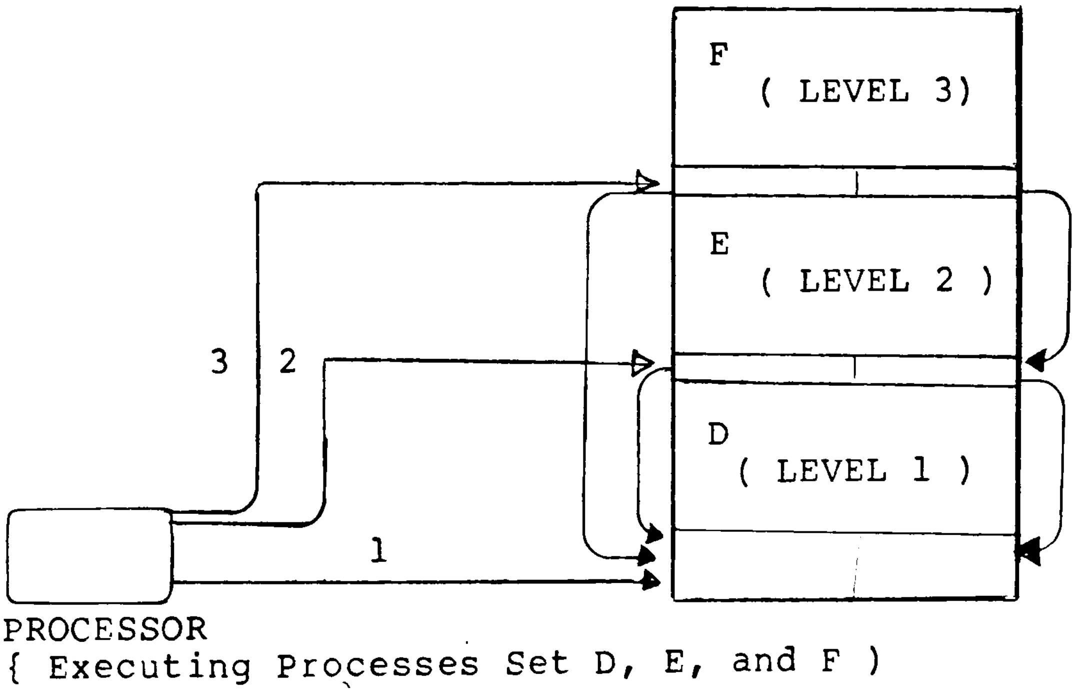


Figure 6.6: Our PLS stack structure

Each entry to the PLS contains information about the pages contents and the requirement of its protection. This information is represented by the following fields:

- the page length;
- the identifier of the process to which the page belongs to;
- a bit specifying whether the page is a member of working set, or not;
- a bit specifying whether the page is present in the main memory or not;
- a bit specifying whether the page is on drum or disc (these are the backing store types considered in our simulation),
- the page (permission) access rights. This is a two-bit field (00 for execute, 01 for write, 10 for read, and 11 read and write),
- a bit specifying whether the page has been referenced or not,
- static link of the next page in the stack, and
- dynamic link to the next process's page (-1 if undefined).

Main Memory Tables Used for keeping the actual contents of the pages. In our simulation, we used two tables: one for the allocated areas for pages (*occupied table*) and the other for the unallocated (*free table*). Each of these tables is variable in length. The entries of these tables contain the following fields:

Free Table area address, area size, and forward pointer.

Occupied Table location of the corresponding PLS entry, the owning process identifier or (-1) if it is shared, and forward pointer.

Fetch Function determining when and what kind of information is to be moved from the backing store into the main memory. There are in principle two policies for fetching nonresident pages to be moved into the main memory (c.f. Lister 1975): the **anticipatory policy** and the**demand policy**. The first policy relies on the predictions of the future behaviour trajectory of the process. The second policy generates a fetch request when a page fault has occurred. In our simulation, we implemented a hybrid policy which combine together the anticipatory and demand policies. When the first page fault occurrs, a fetch request is issued (according to the demand policy) not only to that particular page but also to all the relevant pages that were previously accessed by the same process (in this case according to the anticipatory policy).

Placement Policy determining the specific locations into which the information is to be placed in the main memory, and updating the allocation information. The placement policy tries to find an area in the main memory big enough to accommodate the incoming page. There are numerous placement policies. Among these policies are the best fit, worst fit, first fit, cyclic first fit, and buddy placement (Lister 1975, Madnick and Donovan 1974, Peterson and Silberschatz 1983). In our simulation, we have implemented a *cyclic first fit*. The reason behind selecting this policy is that the demand for pages from the executing processes was calculated in advance by the processor system to fit the available memory size at the time when these processes had been eligible for execution. The free table is implemented as a circular list with a start pointer. Each search begins with a designated hole and advances the pointer cyclically by one element until the placement has succeeded Thus this approach tends to reduce the overhead, since it deposits the fragments over the entire table. Only when the free table becomes too fragmented, the placement policy calls a *shuffling routine* which compacts all the small holes into a one big free hole.

Replacement Function determining which information is to be removed from the main memory to the backing store, in order to make the room for the information being moved in. Information can also be replaced when its utilising process is deleted. The allocation information must also be updated. There is a vast number of replacement policies cited in the current literature, such as the least recently used, oldest resident, the least frequently used, the reference bit policy, and the second chance policy (Kaisler 1983, Watson 1970, Lorin and Detail 1981, Joseph et al 1984). All of these policies have certain justification their use in the case when only a memory system is concerned. However, in our simulation the memory system replacement policy is linked with the inferential strategies of the processor system, and for this reason the above named policies can not be used. In our case, the replacement of a page from the occupied table, takes place according to a list of rules. These rules are as follows:

1. select either pages that belong to an inactive process (e.g. waiting to be killed), or
2. pages that belong to a process which is aborted waiting for terminal, or
3. pages that belong to a process with lower priority than the one which faulted, or
4. a page that does not belong to the faulting process's most recent working set.

To conclude, we describe the essentials of the memory system discussed in the present section.

```
MEMORY-SYSTEM WITH-STATIC-PROTECTION() {
INITIALISE:
  INITIALISE THE FREE TABLE
  INITIALISE THE OCCUPIED TABLE
PERFORM-MEMORY-FUNCTIONS:
  CALL CALL-MSG
  IF MESSAGE IS RECEIVED THEN
     SAVE THE REQUESTING PROCESS IDENTIFIER
     CHECK THE MESSAGE COMMAND
  SWITCH MESSAGE COMMAND OF
  ALLOCATE-PAGES:
    CREATE NEW PAGE     /* REQUEST BY THE PROCESS-GENERATOR */
    IF NO SPACE IN PLS THEN
       CALL PASS-MSG    /* OF FAILURE TO THE PROCESS-GENERATOR */
    ELSE
       THE NEW PAGE ADDRESS := EMPTY STACK POSITION
       THE NEW PAGE LENGTH := EMPTY STACK HOLE LENGTH
       ASSIGN THE OWNER PROCESS IDENTIFIER TO THE NEW PAGE
       UPDATE FORWARD TABLE
       UPDATE FREE TABLE
       CALL PASS-MSG/* OF SUCCESS TO THE PROCESS-GENERATOR */
   DELETE-PAGE:
     DELETE A PROCESS'S PAGES/* REQUEST FROM PROCESSOR-SYSTEM */
     LOOP
     CHECK THE PLS TABLE/* FROM THE PROCESS ID WE FIND
                             PAGES IDENTIFIER */
     IF THE PAGE IS FOUND IN PLS THEN
        CALL DROP-PAGE
        ASSIGN THIS PAGE STATUS TO EMPTY IN PLS
     END-LOOP
     CALL PASS-MSG/* TO THE PROCESSOR-SYSTEM */
   BRING-PAGE:
     /* WHEN A PROCESS PERFORMS A PAGE FAULT */
     LOOP (ON-OFF)
      IF PAGE LENGTH <= AVAILABLE MEMORY THEN
      {
       CALL PLACE-PAGE
       IF PLACEMENT IS SUCCESSFUL THEN
       {
         PAGE ADDRESS:= STACK POSITION
          IF SELECTED FREE AREA LENGTH = PAGE LENGTH THEN
             SEG/PAG ADDRESS+ STACK POSITION + PAGE LENGTH
             UPDATE FREE TABLE
          APPEND NEW ENTRY TO OCCUPIED TABLE
          CALL PASS-MSG TO BACKING STORES TO BRING THIS PAGE
          MEMORY:= AVAILABLE MEMORY - PAGE LENGTH
          PRESENT-IN-MEMORY-SWITCH := TRUE
          MAKE THE STATUS OF ABORTED PROCESS:= READY
          ADD TO THE MISSING COUNT OF PAGE OWNED BY THAT PROCESS
          GOTO PERFORM-MEMORY-FUNCTIONS
      }
      }
     ELSE
     {
      CALL REPLACE-SEGMENT/PAGE
      IF NO PAGE CAN BE REPLACED THEN
         CHANGE THE OWING PROCESS STATUS TO ABORTED
         UPDATE THE OWING PROCESS EXECUTION TIMING
         GOTO PERFORM-MEMORY-FUNCTIONS
      IF THE PROCESS IS NOT WAITING TO BE KILLED THEN
         IF THE PAGE IS NOT THE DRUM THEN
            CHANGE THE STATUS OF THE PROCESS TO BE ABORTED
            CALL PASS-MSG TO DRUM-MANAGER
            WAIT FOR REPLY FROM THE DRUM-MANAGER
            UPDATE TABLES
            GOTO PERFORM-MEMORY-FUNCTIONS
 }
  STATIC-PROTECTION:
  IF PAGE-PERMISSION-RIGHTS MATCHES THE-INTENTIONS-OF-
     -THE-ACCESSING-PROCESS THEN
     CALL PROTECTION-VIOLATION-RULE
  GOTO PERFORM-MEMORY-FUNCTIONS
}
```

6.5.4.2 The Second Inferential Structure: The processor system:

In the previous section we described the first inferential structure, the memory system. The present section deals with the second inferential structure – the processor system. It is not a conventional processor system, because it is not only concerned with a straightforward allocation of processor(s) among the set of ready (eligible) processes but performs also other functions.

Our processor system extends the traditional functions as:

- to ensure an acceptable level of resources utilisation,
- to provide fast response time to multiaccess jobs,
- to provide high throughput for batch processing jobs,
- to incure low overheads, and
- to reorganise any priority setting which may be passed by the job scheduler.
- to cooperate with the memory system in matching the processes demand to the available resources.

Our processor system tries to compromise between the above mentioned objectives, by ensuring an acceptable degree of multiprocessing. The main tasks of our processor system are:

1. the creation and deletion of processes,
2. the preparation of the dispatcher list from the list of ready processes, and
3. determining the subset of the processes which are eligible for dispatching by calculating the pages/segments demand of the processes and chosing those which fit the available size of memory.

The main activities that force the processor system to be invoked are:

1. a process is created,
2. a process is deleted, and
3. a scheduling timer interrupts. The third activity is generated by the processor system, in order to have time to re-organise the dispatcher list and/or by the other processes suffering from a deadlock situation (i.e. *avoiding deadlocks*).

The scheduling policy selected for the processor system is important for determining the way the next process is to be picked up from the dispatcher list, as well as for determining the set of the eligible processes. We organise the dispatcher list by ordering its items according to their user loss function value and by the amount of service they receive.

There exist a vast number of scheduling policies that can be used to schedule the way the processor system select the eligible set number. Table 6.4 lists the notable policies used for processor scheduling that are described in the current literature.

	Policy Name	*Realised On*	*Reference*
1	I/O-CPU BALANCE	TITAN	Wilson 1971
2	Round Robin	HIS 6000	Watson 1970
3	Deadline Scheduling	UNIVAC EXEC VIII	Lorin 1972
4	POLICY-DRIVEN	NOT IMPLEMENTED	Bernstein et al 1971

Table 6.4: Different Processor System Scheduling Policies.

These policies, with the examination of their advantages and drawbacks have been the subject of much research (Coffman and Kleinrock 1968, Denning 1969, Mullery and Driscoll 1970, Kleinrock 1970, Keller 1975). I have developed a slightly different policy that will be described in the sequel. That is again for the same reason as in the case of the memory system, mainly because we have to deal again with the two inferential systems. Our policy can be classified as being of the policy-driven type. It simply starts by allocating the processes priorities according to their user loss functions and the amount of service received by these processes within the dispatcher list. It uses the deadlines specified by the user loss function (see section 5.3.2).

In our simulation, the processor system is entered every t time, during that time our policy driven function to calculate the working set sizes of the processes within the dispatcher list. When the generator just starts operation, it starts with initial working set size preset to a particular value. The current working size is then calculated according to the following formula:

```
ws size = C * virtual store used in the last t units
          + (1 - C) * previous working size
where C is a damped historic count defined as the run time used
in the last interval/total run time used
```

Our processor system is also entered every k time in order to

1. calculate the priority figures of those processes (in the dispatcher list which have recieved some service,
2. record the service time corresponding to the processes's job
3. order the dispatcher list according to the calculated priorities
4. to establish the number of processes that become eligible for execution.

This is done by scanning the dispatcher list to determine how many processes's working sets can fit in the store that is available.

This number determines how many pages of the eligible processes characterised by their working sets that can be placed in the physical store are available. Deadlock can occurr if the highest priority process in the dispatcher list has a working set size which is greater than the available size, but our algorithm prevents this to happen.

In order to reduce the overheads, we choose the time interval t to be equal to the time interval k and call this *time slice.* It is determined at the system loading time. The process priority function is further affected by the process type, as shown by the formula bellow:

$$\text{Process Priority Function} := \text{PPF} + \text{processtype} \times \text{time slice}$$

The process type is determined as follows:

TYPE	*PROCESS*
0	for lowest priority batch process
⋮	⋮
n	for the highest priority batch process
n+1	for interactive process
n+2	for system process

By reordering the dispatcher list according to their paging demand, the availability of the store and the process type, we can obtain the list of eligible processes.

Now we turn our attention to the implementation details that are used in our simulation for constructing the (virtual) processor. The following is a brief description of the processor system algorithm:

```
PROCESSOR-SYSTEM()
{
START-UP:
CALL INITIALISE POLICY-FUNCTIONS/* TO ALLOCATE PROCESS TYPE
          NUMBERS ACCORDING TO THE PROCESS LEVEL */
INITIALISE THE PROCESSOR PRIORITY FOR EACH PROCESS TO ZERO
          /* BY SCANNING THE DISPATCHER LIST */
INITIALISE SEVERAL VALUES FROM THE SYSTEM LOADING STRUCTURE
INITIALISE THE ELIGIBLE SET NUMBER TO NO. OF SYSTEM PROCESSES
GOTO PROCESSING-UNIT
PROCESSING-UNIT:
CALL CALL-ACTIVITY
IF NO ACTIVATION AND A MESSAGE IS RECEIVED THEN
   SWITCH MESSAGE COMMAND  OF
   {
   CREATE:
   /* THIS COMMAND BELONG TO A MESSAGE RECEIVED FROM THE
      PROCESS GENERATOR REQUESTING A DESCRIPTOR TO BE RESERVED
      FOR A PROCESS TO BE CREATED*/
   IF THE FREE LIST IS EMPTY THEN/* NO DESCRIPTORS AVAILABLE*/
       CALL PASS-MSG TO THE PROCESS-GENERATOR
   ELSE
       MARK THE FREE LIST AND THE PLS
       UPDATE THE FREE LIST AND PLS ENTRIES
       CALL PASS-MSG TO THE PROCESS-GENERATOR
   DELETE:
   /* THIS COMMAND BELONG TO THE PROCESS-GENERATOR REQUESTING
      A DELETION OF A PARTICULAR PROCESS */
   CALL DELETE-PROCESS
   CALL PASS-MSG TO MEMORY SYSTEM TO DELETE ITS SEGS/PAGS
   WAITING FOR REPLY
   UPDATE THE FATHERS AND BROTHERS POINTERS OF THIS PROCESS
   CALL PASS-MSG TO THE PROCESS-GENERATOR
   LOADED:
   /* THIS COMMAND BELONG TO THE PROCESS-GENERATOR AFTER A
       PROCESS BEING CREATED */
   ASSIGN THE PROCESS VALUES TO ITS RESERVED PLS ENTRY
   INITIALISE CERTAIN VALUES
   GOTO PROCESSING-UNIT
   LOAD-FAILED:
  ./* THIS COMMAND BELONG TO A MESSAGE RECEIVED FROM THE
      PROCESS-GENERATOR */
   UPDATE THE FREE LIST AND THE PLS ENTRIES
   GOTO PROCESSING-UNIT
   }
ELSE/* IF ACTIVATION IS RECEIVED */
  MANAGER:
  SAVE CURRENT TIME
  LOOP
  SCAN THE DISPATCHER LIST
IF PROCESS IS ABORTED BECAUSE IT IS WAITING FOR WS THEN
   CHANGE ITS STATUS TO READY
IF PROCESS IS ABORTED BECAUSE IT VIOLATE PROTECTION RULE THEN
   CHANGE ITS STATUS TO BLOCKED
IF PROCESS RECEIVED SOME SERVICE THEN
   CALL RESOURCE-SINCE-TIME
   FIND ITS NEW WORKING SET
   CALL PROCESSOR PRIORITY/*TO ASSIGN THE PRIORITY TO THAT
   PROCESS ACCORDING TO THEIR WORKING SET AND THE STORE SIZE
   AVAILABILITY*/
END-LOOP
SORT THE DISPATCHER LIST ACCORDING TO THE PRIORITY NUMBERS
SCHEDULE THE NEXT SCANNING INTERVAL
UPDATE THE NUMBER OF ELIGIBLE PROCESSES
/* BY CALLING FIND-INTENDED-E-PROCESSES */
CALL REPORT-TO-JOB-SCHEDULER/* TO CONTROL THE NUMBER OF
      USERS ADMITTED TO THE SYSTEM */
}
```

6.5.5 The Control Structures: The I/O control

The main purpose of our control structures is to derive and coordinate the operations of the various management units of the essential functional structures. The control structures are needed here to coordinate the I/O activities among the different system devices (management units of control). Controlling the I/O devices in our simulator can not be achieved simply using a communication mechanism as it was done in the COMMUNICATION-STRUCTURE. The initial design specification of our activity-structures based computer system requires a partially controlled variation in the I/O devices managements because the devices should be changeable. The design of our possiblistic generator must allow some features of I/O control to vary. The variations may occurr in the I/O device characteristics, or in the way batch jobs that are controlled by the user background blackboard. Also the control of interactive I/O operations which are controlled by the user foreground blackboard might vary. The I/O batch control devices can be removed when required.

. Further responsibilities of our I/O control structures consists of ensuring

- I/O devices independence,
- a uniform treatment of I/O devices, and
- high I/O devices utilisation (instituted by adopting the policies that match the nonuniform requests of processes with the uniform speed of those I/O devices for which it is

required).

The I/O control structures must cope with a wide variety of I/O devices and are therefore quite complicated at the detailed levels of their design. There are basically three main types of I/O devices: *dedicated*, *shared*, and *virtual*. Dedicated devices are those which are more effectively assigned to one process for a given time period, even though the process may not be able to utilise them continuously. In this category are the line printers, card readers, tapes, etc. Dedicated devices require advanced reservation for their use by specifically reserving them for a given duration. Shared devices can be allocated among different processes at a much faster rate when the sharing is indivisible at the process level. While allowing the access to only one process at a time, the devices can rapidly complete their service for individual processes and be quickly switched to the service requests of other processes. In this category are such on-line auxiliary storage units as drums and discs. Here, no special reservation or allocation is required.

Since dedicated devices are responsible for the deadlocks, we avoid this problem in our simulation by simulating the dedicated devices by the shared ones. In particular, this decision is made for those I/O devices that need to be added or removed from the system. The resultant devices are called virtual. The control unit that is responsible for the management of the dedicated and the shared devices is called permanent I/O control, whereas the management unit that is responsible for the virtual devices is called removable I/O control.

The main design point that needs to be highlighted here is about the way these two control units should be constructed. This can be achieved by associating the device characteristics with the devices themselves rather than with the processes which handle them. This will let the resulting control unit to be a general- purpose one. The way in which this design objective is achieved is to place the device-dependent characteristics in tables which have been designated to be used by a general-purpose routines, which are device-independent.

In our simulation, these tables are represented by a list of descriptors. Each device has an associated device descriptor which contains the relevant information about this device. The descriptor information includes the physical device number, the device name, the device status (busy or free), and the channel/control unit number to which the device is attached. The necessary isolation of device characteristics can be achieved by including additional information in the device descriptor and by using the descriptor as a source of information for the associated device control unit. The additional information that can be added to the device descriptor includes the following:

- instructions which operate the devices
- pointers to character translation tables
- an indication whether the device is in a character or word mode
- an indication whether the device requires a buffer
- the buffer size, if required
- whether the device is random or sequentially accessed,
- the address of the driving control unit.

In our simulation, all the device descriptors are kept in one table called the I/O Device Control Table (IODCT). The physical device number is expressed by the descriptor position in the IODCT table.

6.5.5.1 Permanent I/O control

There are three main sub-units that contribute to the permanent I/O control:

1. the device handlers,
2. the terminal system management unit, and
3. the file system.

The permanency of an I/O control unit means that the communication ports of this unit always exist (i.e. must be defined for all other system processes). The reason behind subdividing the permanent I/O control is that in our design each I/O device has a separate device handler process associated with it. If several of these processes operate in a similar way, they can make use of sharable programs and any differences in behaviour can be derived from the information in the corresponding descriptor. Thus, we have concentrated our separation on the *terminal system unit* and the*file system unit* because they are sufficiently different from the other I/O permanent devices to merit a special consideration.

1. *The I/O device handlers functions:* A device handler is a process which is responsible for servicing the requests on its associated device and for notifying the orginating process when the service has been completed. Each device handler operates in a continuous cycle during which it:

   (a) receives a message which includes the following:

      - operation type (read, write, advance, rewind)
      - main memory address
      - backing store address
      - length of transfer (in blocks)

   (b) translates the message to appropriate commands for a channel (these can be extracted from the device characteristics held in the device descriptor) and constructs the channel program,

   (c) sets up the port to the device if necessary,

   (d) initiates the I/O operation

   (e) waits for the operation to be completed,

   (f) handles error conditions,

   (g) notifies the originating process, and

   (h) connects the port to the device, if necessary.

   Here we briefly describe the implementation algorithm we used:

```
I/O-DEVICE-HANDLERS() {
PERFORM-I/O:
  CALL CALL-ACTIVITY/* PROVIDING THE PORT TO THE KERNEL
                      OR ANY REQUESTING PROCESS */
  IF WAKE-PROCESS IS RECEIVED AS RESPONSE THEN
     CLEAR OR REST THE APPROPRIATE DEVICE
  IF THE MESSAGE RECEIVED SPECIFIES BLOCK/RECORD TRANSFER THEN
     CHECK MESSAGE-VALIDITY
     IF IT IS VALID THEN
     ASSIGN PHYSICAL DEVICE NUMBER
     CALL SET-CHANNEL/* TO START DEVICE OPERATION */
```

```
   CALL CALL-MSG/* REQUESTING REPLY FROM THE DEVICE
              WHEN FINISHED */
   IF A MESSAGE ANSWER IS RECEIVED THEN
     CHECK MESSAGE/* WHETHER OPERATION WAS SUCCESSFUL
                OR NOT */
     CALL PASS-MSG/* SEND REPLY TO THE REQUESTING PROCESS*/
     IF THE DEVICE WAS A SHEARABLE ONE THEN
       CALL PORT-DELETE
  GOTO PERFORM-I/O
}
```

2. *Terminal system management functions* The terminal system manages the user terminals (i.e. typewriter like excluding the operator's console) which are attached to a special channel called the multiplexor. The terminal system must be prepared to be engaged in more than one conversation at a time. It is unlike the other device handlers, in which the terminal system cannot be restricted to accepting one message at a time. The consequence of having a single message at a time causes the other process requests (including those from the terminal users) to wait for a long time. Hence, as soon as the terminal system has started a lengthy action by sending a message to some other process it must be able to receive further messages or to start other actions.

   There are two types of message ports attached to the terminal system. The control port is primarily used to inform the job scheduler when a terminal dials up. The communication port is used to transfer lines of data to or from a terminal. Following any log-in, the terminal system asks the job scheduler to create a command interpreter process to communicate with the terminal. Once the user has successfully been connected, the processes created for that user are communicating with his terminal directly.

   There are two main approaches to the terminal communication: transmitting a single character at a time and transmitting a block of characters at a time. The former approach would seem to give the user greater flexibility than the later, but it requires more system time for character handling. For the system which has a small number of terminals attached, the overhead involved in handling the terminal I/O is not excessive. For systems using a large number of terminals however, this approach is too inefficient. Thus, in our simulation, it was assumed that a block of characters is transmitted at a time. At the input, this is usually one line at a time. It is also assumed that the blocks of characters are transferred directly to/from main memory through the multiplexor channel (which has it own buffer).

Associated with each terminal is a terminal number. The terminal number, like the device number, is the number used by the system for indexing files associated with terminal I/O and for accessing the correct buffer. This number is assigned when the terminal makes the first contact with the system. Similarly, each terminal has a descriptor associated with it which contains various types of status information used for control purposes. These include the following:

- indications of whether the terminal is linked to another one,
- whether another terminal is linked to this one and the number of the linked terminal,
- the break character set,
- the type of character code conversion required, etc.

Here we briefly describe our implementation algorithm:

```
TERMINAL-SYSTEM()
{
START-UP:
   NUMBER-OF-USER-REJECTED := 0
   SET THE DEVICE NUMBER GIVEN TO EACH TERMINAL/* BY USING
                      ITS CORRESPONDING DESCRIPTOR */
   GOTO MANAGEMENT-UNIT
MANAGEMENT-UNIT:

   CALL CALL-MSG/* INCLUDING THE PORTS TO JOB SCHEDULER,
                KERNEL, AND ANY REQUESTING PROCESS */
   IF MESSAGE IS RECEIVED AND IT SOURCE IS THE KERNEL
       FOLLOWING THE USE OF SPECIAL CONTACT CHARACTERS THEN
      CALL PASS-MSG/* TO JOB-SCHEDULER CHECK WHETHER THE
               USER WHO MADE CONTACT CAN LOG-IN OR NOT */
      GOTO MANAGEMENT-UNIT
   IF MESSAGE IS RECEIVED AND THE SOURCE WAS THE KERNEL
       FOLLOWING AN I/O INTERRUPT THEN
      CALL PORT-CREATE/* TO THE PROCESS ASSOCIATED WITH THAT
                         TERMINAL */
      CALL PASS-MSG/* TO THAT PROCESS */
      GOTO MANAGEMENT-UNIT
   IF MESSAGE SOURCE WAS THE JOB SCHEDULER IN RESPONSE
       TO USER BEING REFUSED ENTRY TO THE SYSTEM THEN
```

```
        ADD ONE TO THE NUMBER-OF-USERS-REJECTED
        GOTO MANAGEMENT-UNIT
     IF MESSAGE SOURCE WAS A REQUESTING PROCESS/* END OF
         INTERACTION */
        SET UP A MESSAGE TO START THE SPECIFIED TRANSFER
        CALL SET-CHANNEL
        CALL PORT-DELETE
        GOTO MANAGEMENT UNIT
  }
```

3. *The file system functions:* Conventionally, a file system deals with the physical features of files, namely with:

   - physical organisation and access methods,
   - basic file system,
   - access control verification,
   - backing store management, and
   - symbolic file system.

These functions have been described in several places in the literature and the reader is referd to one of such references (e.g. Watson 1970). In our simulation, our filing system is modelled in a quite simple fashion. Its function is to accept the messages from other processes and to send these messages to the disc manager to satisfy the processes' requests. Thus, the reason for introducing our file system was to ensure the right amount of the message I/O traffic within our possibilistic generator (i.e. in comparison to the class of highly constrained system). Note that the requests from the user processes could include the opening and closing of files.

Here we briefly describe the implementation algorithm used in our simulation of a file system:

```
FILE-SYSTEM()
{
START-UP:
  CALCULATE THE NUMBER OF DISC TRANSFERS REQUIRED TO MOVE
     A PHYSICAL BLOCK
  GOTO FILE-I/O-UNIT
FILE-I/O-UNITE:
  CALL CALL-MSG/* PROVIDING THE PORT NUMBER FOR THE DISC
       MANAGER AND ANY REQUESTING PROCESS INCLUDING UNDEFINED */
  IF A REPLY MESSAGE IS RECEIVED FROM A REQUESTING PROCESS THEN
     LOOP (NUMBER OF TRANSFERS REQUIRED TO MOVE A PHYSICAL BLOCK)
       CALL PASS-MSG/* TO THE DISC MANAGER */
       CALL CALL-MSG/* WAITING FOR REPLY */
     END-LOOP
  CALL PASS-MSG/* TO THE REQUESTING PROCESS INFORMING THE REQUEST
                IS SATISFIED */
  CALL PORT-DELETE
  GOTO FILE-I/O-UNIT
}
```

6.5.5.2 Removable I/O control

The operation of the removable devices consists of alternations of messages sent and received to/from the devices they deal with. In our simulation model, the only removable I/O control devices are the background blackboards (or the spoolers). By removing these devices our possibilistic generator is left with the foreground user blackboard. The removal of these devices is signaled during the initial loading time. The way in which the requests for removal are collected by the generator via the PREPROCESSOR is determined according to the designer requirements.

When the spoolers are attached to the simulation model, their port numbers are made known to the job scheduler. Here we assume that each spooler possess a pool area which is capable of holding an input deck or a printout of a suitably large size and that over a period of time the speed of the devices is adequate to handle all the generated requests. When a job opens a stream it is allocated a file (spool area) on the disc and all the input or output in the stream is directed to this file. When the stream is closed, the spool area is available for the use by another job.

If the I/O performed by the spoolers is (unblocked) undead and unbuffered then, although we can produce multiple virtual devices and multiple copies of the same output without rerunning a job, spooling can cause drop in their speed, compared to the speed of the actual physical devices it simulates. The efficiency of the management of the speed of access and space of buffers of our spoolers if effected by the following factors:

1. by using virtual instead of dedicated devices (effect on speed)
2. minimising the gaps between blocks in the buffers (effect on space)

3. the use of multi-buffering (effect both the space and the speed).

However, it is necessary to keep track of the input/output spool areas which are occupied or available. Generally, this can be done using the spoolers descriptors, the job scheduler descriptors, and the file system descriptors. In our simulation however this is done by the file system descriptors.

Here we briefly describe the implementation algorithms of two removable spoolers: the *input spooler* and the *output spooler.*

```
INPUT-SPOOLER() {
START-UP:
   CALCULATE THE NUMBER OF LOGICAL RECORDS CAN BE HELD IN THE
          PHYSICAL BLOCK
   GOTO PERFORM-INPUT-SPOOLING
PERFORM-INPUT-SPOOLING:
   CALL CALL-MSG/* SPECIFYING ITS PORTS TO THE JOB SCHEDULER,
                   FILE SYSTEM AND THE CARD READER */
   IF A MESSAGE IS RECEIVED FROM THE JOB SCHEDULER TO
         READ A JOB THEN
      LOOP (ACCORDING TO THE NUMBER OF INPUT RECORDS)
        CALL PASS-MSG/* TO THE CARD READER TO FILL THE FIRST
                          BUFFER */
        CALL CALL-MSG/* WAITING FOR REPLY */
      END-LOOP
      IF THE BUFFER HAS BEEN FILLED THEN
         CALL PASS-MSG/* TO FILE MANAGER TO WRITE THE BLOCK ON
                          DISC */
         LOOP (UNTIL NO MORE CARDS TO BE READ)
           CALL PASS-MSG TO THE CARD READER
         ' CALL CALL-MSG TO WAIT FOR REPLY
         END-LOOP
         IF THE OTHER BUFFERS HAS BEEN FILLED THEN
            CALL PASS-MSG TO THE FILE SYSTEM TO EMPTY IT
            CALL CALL-MSG TO AWAIT MESSAGE FROM THE FILE SYSTEM
         CALL PASS-MSG TO THE JOB SCHEDULER INFORMING THAT
                       THE JOB ALREADY IN THE SPOOL AREA
   GOTO PERFORM-INPUT-SPOOLING
}

OUTPUT-SPOOLER() {
START-UP:
  CALCULATE THE NUMBER OF LOGICAL OUTPUT RECORDS
  GOTO PERFORM-OUTPUT-SPOOLING
PERFORM-OUTPUT-SPOOLING:
  CALL CALL-MSG/* SPECIFYING ITS PORTS TO THE JOB SCHEDULER,
                  FILE SYSTEM, AND THE LINE PRINTER */
  IF A MEAAGE IS RECEIVED FROM THE JOB SCHEDULER THEN
     CALL PASS-MSG TO THE FILE SYSTEM TO READ THE FIRST BLOCK
            OF RECORDS
     LOOP (UNTIL NO MORE RECORDS LEFT)
       CALL CALL-MSG PROVIDING THE PORT TO THE FILE SYSTEM
       IF A REPLY MESSAGE IS RECEIVED THEN
          IF THERE ARE ANY MORE DISC RECORDS TO BE TRANSFERRED
                 THEN
              CALL PASS-MSG TO THE FILE SYSTEM TO READ
                    THE NEXT BLOCK AND FILL THE OTHER BUFFER
              IF THE FILE SYSTEM IS BUSY READING THEN
                  CALL PASS-MSG TO THE LINE PRINTER/* TO
                         PRINT THE FIRST BUFFER */
                  CALL CALL-MSG TO WAIT FOR REPLY
      END-LOOP
      CALL PASSMSG TO THE JOB SCHEDULER TO INFORM THAT THE JOB
            HAS BEEN PRINTED
  GOTO PERFORM-OUTPUT-SPOOLING
}
```

6.5.6 The Protection Structures

The protection structure manages the control of access to the resources. More specifically, it is concerned with two kinds of entities:

1. the *resources* available in the system determine the *permissions*, specifying the *allowable operations* on *objects* (i.e. fields, segments, processes, etc.)
2. the *participant requirements*, the *intentions* of *subjects*, (user jobs, processes) determining the operations to be requested to be performed on objects.

Describing the protection structure of a computer system means describing the protection mechanism of that system which acts as an agent (or mediator) checking the legality of every reference by a participant to a resource. This legality (i.e. the rights to be performed actions) is codified in the protection descriptions.

The protection mechanism suitable for protection enforcement in activity structures-based designs, must support the remote accesses of resources at the different distributed, structures. Such a protection mechanism has been developed by the author, and it is based on the notion of *port*. Ports provide the means for manipulating resources, while the protection descriptors (i.e. capabilities) independently provide protection description. In our approach, the ability to manipulate a resource using any of a given set of operations is equated with the ability to send a request through a port connected to that resource manager. Thus, possession of a port implies the right to perform a particular set of operations on the resource associated with that port. The communication structure facilities are thereby extended to provide the protection mechanism for all the resources in the system.

Processes are permitted to create, and thereby own, ports in order to perform a specific set of operations on a set of specified resources. This permission is given by the underlying resource manager. In order to perform a specific operation on a resource, a process sends a message to that manager of that resource through the port created for this purpose. On the completion of the requested operation, the result of the operation is sent back to the requesting process. A process (including a user process) is permitted to request a certain operation on a resource only via a particular specified port. Any request is granted if and only if it has the intentions that match the permission rights allowed for that particular resource.

In our simulation, we should note that the permission operations are implicitly associated with the ports manipulation for each system management unit and considered to be of a *static type*. However, the protection system can be extended to include the means for enforcing the ***dynamic protection***. By dynamic protection we mean the structures in which either the intentions or permissions may change after their initial assignment. Such a dynamic protection mechanism was originally proposed by Kohout (1976); it can be incorporated within our protection structure to add to it the criteria for handling of decentralised dynamics (according to the distribution nature of the functional structures).

The criteria for decentralised dynamics of protection, are enforced on the memory system only, in our implementation. It, of course, can be extended to the file system and other systems. Using our possiblistic generator, the designer can decide at the system loading time whether some dynamic protection is required and/or whether he want limited sharing or maximum sharing protection policy. For these purposes new facilities must be added to the memory system of static protection in order to enforce those operations that are required for the dynamic protection. These are:

- adding/deleting permission rights,
- passing permission rights,
- permitting new permission rights, and
- enforcing either maximum sharing policy or limited sharing policy.

Since permission rights were included in the memory page descriptor, then adding, deleting and transferring these rights lets the size of the page to be variable. This means that with dynamic protection mechanism, the memory system should manage segmentation as well.

Here we briefly describe the new memory inferential system with the enforcement of the decentralised dynamics protection criteria:

```
MEMORY-SYSTEM-INCLUDING-DYNAMIC-PROTECTION ()
{
INITIALISE:
  INITIALISE THE FREE TABLE
  INITIALISE THE OCCUPIED TABLE
  INITIALISE PROCESS-AGE-THREASHOLD
PERFORM-MEMORY-FUNCTIONS:
  CALL CALL-MSG
  IF MESSAGE IS RECEIVED THEN
     SAVE THE REQUESTING PROCESS IDENTIFIER
     CHECK THE MESSAGE COMMAND
  SWITCH MESSAGE COMMAND OF
  ALLOCATE-SEGMENTS:
    CREATE NEW SEGMENT /* REQUEST BY THE PROCESS-GENERATOR */
    IF NO SPACE IN PLS THEN
       CALL PASS-MSG/* OF FAILURE TO THE PROCESS-GENERATOR */
    ELSE
       THE NEW SEGMENT ADDRESS := EMPTY STACK POSITION
       THE NEW SEGMENT LENGTH := EMPTY STACK HOLE LENGTH
       ASSIGN THE OWNER PROCESS IDENTIFIER TO THE NEW SEGMENT
       UPDATE FORWARD TABLE
       UPDATE FREE TABLE
       CALL PASS-MSG/* OF SUCCESS TO THE PROCESS-GENERATOR */
   DELETE-SEGMENT:
     DELETE A PROCESS'S SEGMENTS/PAGES/* REQUEST FROM PROCESSOR-
                                         SYSTEM */
     LOOP
     CHECK THE PLS TABLE/* FROM THE PROCESS ID WE FIND
                             SEGMENTS IDENTIFIER */
     IF THE SEGMENT IS FOUND IN PLS THEN
        CALL DROP-SEGMENT
        ASSIGN THIS SEGMENT STATUS TO EMPTY IN PLS
     END-LOOP
     CALL PASS-MSG/* TO THE PROCESSOR-SYSTEM */
   BRING-SEGMENT:
     /* WHEN A PROCESS PERFORMS A SEGMENT FAULT */
     LOOP (ON-OFF)
      IF SEGMENT LENGTH <= AVAILABLE MEMORY THEN
      {
       CALL PLACE-SEGMENT
       IF PLACEMENT IS SUCCESSFUL THEN
       {
         SEGMENT ADDRESS:= STACK POSITION
          IF SELECTED FREE AREA LENGTH = SEGMENT LENGTH THEN
             SEGMENT ADDRESS+ STACK POSITION + SEGMENT LENGTH
             UPDATE FREE TABLE
          APPEND NEW ENTRY TO OCCUPIED TABLE
          CALL PASS-MSG TO BACKING STORES TO BRING THIS SEGMENT
          MEMORY:= AVAILABLE MEMORY - SEGMENT LENGTH
          PRESENT-IN-MEMORY-SWITCH := TRUE
          MAKE THE STATUS OF ABORTED PROCESS:= READY
          ADD TO THE MISSING COUNT OF SEGMENT OWNED BY THAT PROCESS
          GOTO PERFORM-MEMORY-FUNCTIONS
       }
       IF SHUFFLING NOT PERFORMED THEN
          CALL SHFFLE
          SHUFFLING := TRUE
      }
     ELSE
     {
      CALL REPLACE-SEGMENT
      IF NO SEGMENT CAN BE REPLACED THEN
         CHANGE THE OWING PROCESS STATUS TO ABORTED
         UPDATE THE OWING PROCESS EXECUTION TIMING
         GOTO PERFORM-MEMORY-FUNCTIONS
      IF THE PROCESS IS NOT WAITING TO BE KILLED THEN
         IF THE SEGMENT IS NOT THE DRUM THEN
            CHANGE THE STATUS OF THE PROCESS TO BE ABORTED
            CALL PASS-MSG TO DRUM-MANAGER
            WAIT FOR REPLY FROM THE DRUM-MANAGER
            UPDATE TABLES
 }
 STATIC-PROTECTION:
    IF SEGMENT'S ACCESSING PROCESS INTENTION RIGHTS DOES NOT
         MATCH THE SEGMENT PERMISSION RIGHTS THEN
       {
       CALL PROTECTION-VIOLATION-RULE
       GOTO DYNAMIC-PROTECTION
       }
    ELSE
       GOTO PERFORM-MEMORY-FUNCTIONS
 DYNAMIC-PROTECTION:
  IF THE REQUESTING PROCESS POSSESS PERMIT INTENTION AND
   ACCESSING A FORBIDDEN SEGMENT THAT BELONG TO ANOTHER PROCESS THEN
    IF LIMITED-SHARING-NEEDED THEN
       CALL DEATH-SENTENCE /* TO ABORT THE REQUESTING PROCESS*/
    LOOP
       SCAN PLS FOR THE SEGMENTS THAT POSSESS A PASS RIGHT AND
       BELONG TO OWNER OF THE FORBIDDEN SEGMENT
    IF FOUND THEN
       TRANSFER RIGHT FROM THAT SEGMENT TO THE REQUESTING PROCESS
    IF NEW INTENTIONS MATCHING THE RIGHTS OF THE FORBIDDEN SEGMENT
       THEN
       CALL PROTECTION-SYSTEM-ALLOW-ACCESS
    END-LOOP
   GOTO STATIC-PROTECTION
}
```

Chapter 7

EXPLORING THE DYNAMIC BEHAVIOUR OF THE ACTIVITY STRUCTURES BASED POSSIBILISTIC GENERATOR OF COMPUTER SYSTEMS

7.1 An Overview

In this chapter, we are concerned with exploring the behaviour of the possibilistic generator. This is a goal-directed activity of the designer, required to force the possibilistic generator to act in an 'interesting' way, by performing certain changes both user-oriented and machine-oriented.

Exploring new computer architectures, such as the architectures that can be generated by the possibilistic generator (i.e. an activity structures based architectures) is not a straightforward task. In our opinion, only with the intensive or concentrated extensive analysis (c.f. section 4.6.4) of the execution behaviour the effectiveness of new architectures can be significantly improved. Without an understanding of micro- and macro-execution behaviour, potential benefits from enhancements of computer architectures will be lost.

> "Models of execution behaviour are needed to connect measurement to theory".
> (Browne 1984)

The shells capable of capturing the dynamics of changes of behaviour seem to be of much greater practical importance for the design and implementation of cooperating environments than the majority of other types of software products that concentrate on actual product static behaviour quality (Riddle 1979). This is because the following typical difficulties arise in the design and implementation of system with cooperating environments:

1. Without an execution behaviour model, it is very difficult to foresee all the possible modes of operations that can occur due to varying synchronisation between the communicating environments or their subsystems, possible malfunctions of the communication medium, and/or error recovery actions initiated by one or both environments (c.f. Bochmann 1978).

2. With cooperating environments incorporating both a local learning mechanisms and a global learning mechanism, it is very difficult to detect whether the local learning mechanisms are not in conflict, and cooperate towards the global learning goal (i.e. the self-regulating behaviour). Only very recently, Gaines and Shaw (1986) pointed out the great importance of such a problem. They provided, however, only the models for verifying the logic of such cooperative environments, without going into the details of their implementation and of performance measurement.

3. Without execution behaviour models, it is impossible to tune the generator behaviour to the required behaviour of the problem environment (c.f. Sakamura et al 1979).

4. Cooperating environments impose the distribution of processes, and with it faults and deadlocks, which can give rise to time dependent errors that are very difficult to detect and locate by simple run time tests. Hence, the execution behaviour models may be of great help. Unfortunately these models must be empirical, at least for the time being, since there is no existing theory for modelling distributed computer systems. Klienrock (1985) summarises succinctly the present situation:

   > "We do know some thing about the way distributed activities and distributed systems behave precious few though may be. The most interesting thing about them is that they come to us from different fields of study. Unfortunately, the collection of results is just that – a collection, with no fundamental models or theory behind it".

Unfortunately the execution behaviour description models/measures are not simple or straightforward. These models/measures are often derived from multiple, conflicting, and usually nonlinear/ multi-variant situations which should reflect the system's design specifications. The difficulty of extracting such models/measures can be depicted from the Figure 7.1 showing the variety of factors contributing to one of the most important measures used for constructing models for execution behaviour, that is the response time. We approached the problem, however, from a different angle. We are not going to model the distributed system dynamics, since its execution structure may possibly vary according to the conflicting strategies of the conversational environment. This is an almost impossible case as Sinha recently reported (1985).

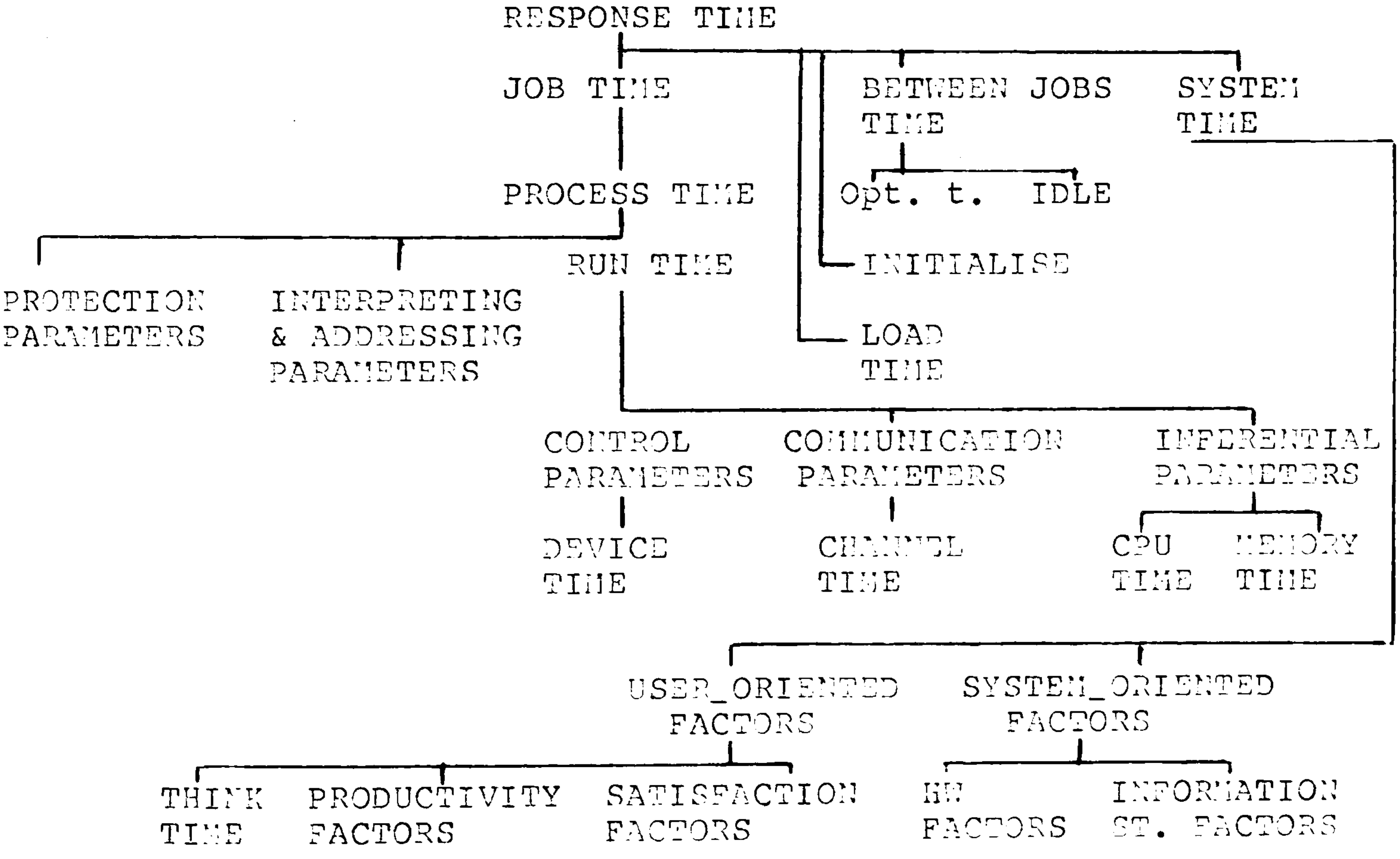


Figure 7.1: Factors affecting the response time measure.

In our work we take the execution structure to be unknown. Then we learn how to control this unknown system through a framework that utilises only the input-output experiments. The input represents a synthesised sequence of design data, selected possibly from the experience with another system, taken to systematically force the possibilistic generator to act in an interesting way. Then, by monitoring the input-output execution behaviour (via probes inserted within the required components needed to be analysed), the execution behaviour can be modelled using the 'performoact modelling' framework (see sections, 7.2 and 7.6). This is the framework that we develop. The significant thing about this framework is, that it represents the nature of the environments in cooperation using representative measures. This representation can help in identifying the admissible/nonadmissible situations that contribute to the behaviour exhibiting self-regulation of cooperating environments. Hence using the performoact modelling framework the designer learns how to control the executing system and what design data (or sequence of events) produce the admissible models. By using the design data for these admissible models the designer can tune the system. This process of monitoring-modelling-tuning can be repeated several times until the required system fine tuning is reached. The system must be stable (i.e. self-regulating) and possibly to match the required behaviour.

The framework was developed employing two practical design modelling principles: the theory of system identification (Gaines 1977) and the productivity theory (Mason 1979). In section 7.6 we describe the strategy that we used for modelling the execution behaviour.

In our case, we must select the synthesised input data to be the design data that can be effectively used to verify the logic of our possibilistic generator. For this purpose, we selected the data for priming the design from a similar highly constrained existing computer system. In this chapter in particular, we selected the priming data from within the class of highly constrained systems which contributed to the design of our possibilistic generator namely the NUKE (Crowley 1981). Using this data for priming, and by systematic variation of this data, we generated various simulation runs, that were systematically monitored. This identification process[1] yielded what is called the Nuke *operational descriptions*[2] see section 7.6 and the appendix. The performoact modelling utilises the operational descriptions to produce ***behavioural description models***[3]. We used our performoact modelling for identifying

[1] In the sense of admissible models of Zadeh (1962)

[2] It provides the details of how the system performs by monitoring the activities of its basic components using certain software probes (c.f. Robinson and Torsun 1977).

[3] It captures descriptively those essential of the system's behaviour so that it can be used it can be used to make some future predictions about the behaviour of the system or its parts (c.f. Gomma 1977).

the admissible design data that derive the Nuke family of systems within a particular class of user environments (see section 7.7).

7.2 A Framework for The Behavioural Description of The Activity Structures Based Computer Designs

In this section we are concentrating on the issue of modelling the input-output execution behaviour of our possibilistic generator.

Our work here is based on the conceptual scheme and the general- purpose measures of the productivity theory (Mason 1979). The works of Brian Gaines on system identification (1977) has considerably enhanced our method and understanding of its principal results. Mason's productivity theory is mainly concerned with design of economical systems organisation; we transfer his main concepts into the domain of computer systems design. In this thesis we refer to our developed framework as the *performoact modelling.*

Gaines system identification scheme (1977) originally developed from the system theory of Zadeh (1962). It attempts at extracting certain behavioural description models from observing the class of possible structures of behaviour and to identify a membe: from this class that is most likely to enforce the behaviour of interest. The Gaines scheme is a method of approximation which leads to the identification of certain admissible models called the *space of admissible models.* By using the space of admissible models a description of the system execution behaviour can be represented, and hence this description can be used to make future predictions. The design data that are used to generate the admissible models then can be used to tune the original system behaviour. It should be noted that also the priming of adaptive behaviour has been introduced by Gains (1972).

On the other hand, the productivity theory of Mason's provide us with certain general-purpose measures that can be used to measure the effectivity of cooperating environments, such as our conversational environment (Mason 1979). In our framework Mason's measures identify a specific evaluation space (i.e. a framework of admissible models) which consists of the *system effectiveness* (i.e. the machine environment productivity) in the relation to the *demand* (the user environment workload) consumed during the production period. As noted previously, the productivity theory has originated and is used only in the economic and managerial literature (Craig and Harris 1973, Gold and Soesan 1976).

The two above mentioned theories the system identification, and the productivity theory, have been used by the author to develop the scheme of the performoact modelling which can be conceptualised easily. Figure 7.2 illustrates this scheme. In a finite distributed computing system, any increase in the demand (i.e. concurrent transactions or processes) will lead to an increase in the system effectiveness (e.g. system throughput). If the increase in demand is homogeneous that is, always of the same proportion or consisting of the same mixture– then demand and effectiveness will be related by a straight line (OA). From the experience with performoact models this not an interesting behaviour, since the system (i.e. the waiting time) will be a linear-oriented function. Real computing systems are, of course, finite. At some point heavily used system component will limit the effectiveness (c.f. Barber (1979), Kupka (1974)). This limit is represented in the figure by the line BC. The limitation point, may be the point of system thrashing. Further, any real computing system will include shared resources (descriptors, processor, segments, etc) which at any demand other than zero will generate *queues*. This smoothes out the sharp knee in the figure, and real systems would behave according to a curve such as OD. The sharpness of the knee in the curve, is dependent however, on the utilisation of the shared resources, and therefore the speed with which the queues develop as the demand increases. In a real systems, the increase in the demand generates an additional component of work as these become loaded– the *overhead* as the result of managing a great deal of concurrent activities. Within the certain limits this overhead may increase more quickly than effectiveness (i.e. the speed of servicing the queues); in which case there is a net loss of throughput. This effect is illustrated by the curve OE in the figure. System equilibrium then, can be defined as the balance between queueing and overhead. The initial slope of the curve will be very close to the *straight line* of the equivalent *'infinite' computer*. The *asymptote of the curve* is the line corresponding to the saturation of the most heavily used and constrained system component (s). The actual curve is referring to the steady state which under special conditions can be considered as an admissible model. Among the non-linear curves, the most interesting one is the curve that expresses the balance between the queueing time and the overhead time. The case of balance defines the admissible model of behaviour for a running system, such as ours. It identifies the case where the system can balance the user environment activities with the computer machine activities and can survive. Survive in the sense that the system will achieve maximal utilisation and never reach thrashing situations. We believe that by monitoring the execution behaviour for the possibilistic generator for a sufficient period of time (it should not be less than 5 minutes according to Clapson (1979)), we can identify the type of behaviour and determine its admissibility. For this purpose, we fixed the execution period of the possibilistic generator to be 25 minutes (statistics monitored each 5 minutes and can be altered to any required interval) and we used the regression analysis to identify the type of the trend any execution behaviour may exhibit.

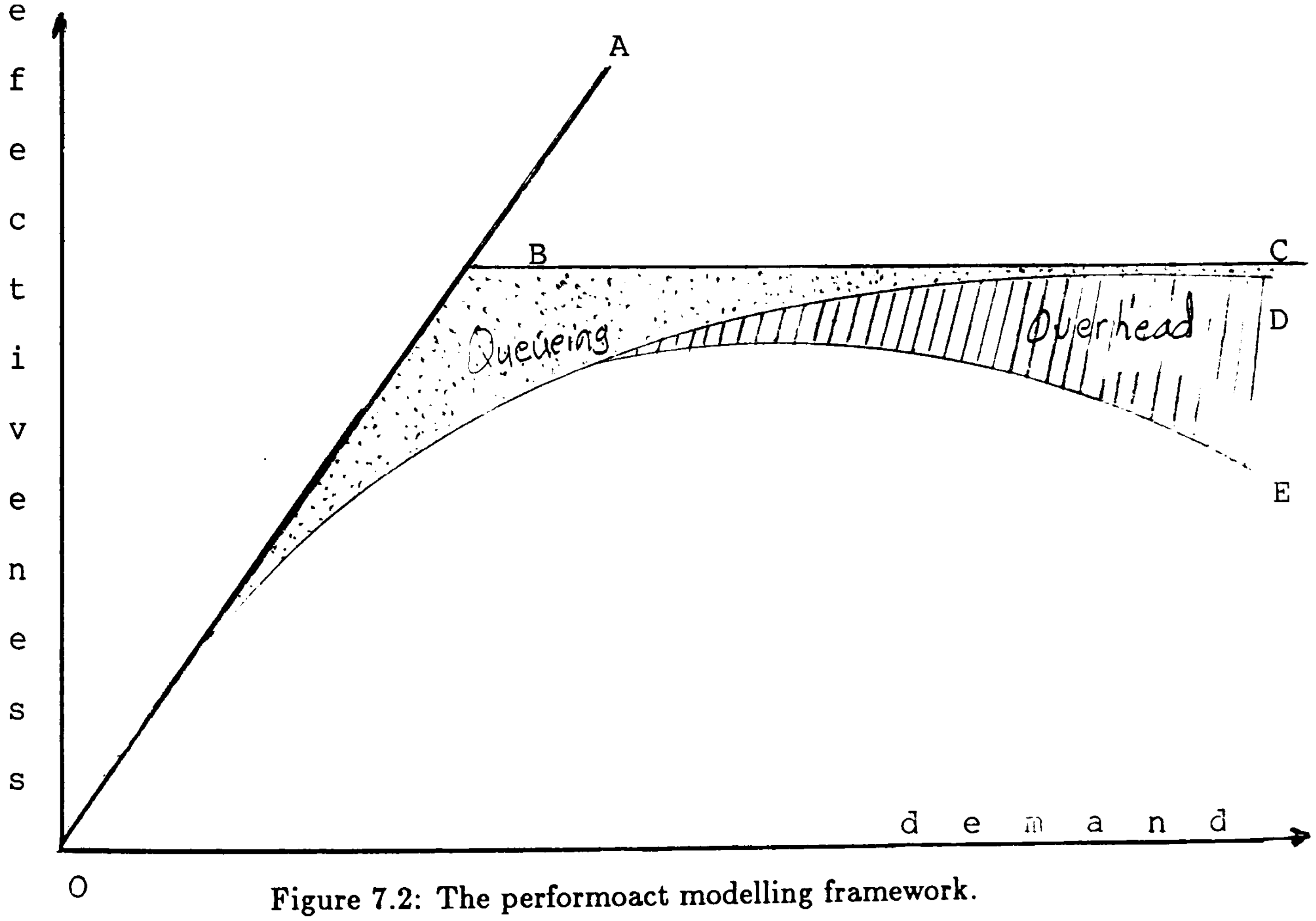


Figure 7.2: The performoact modelling framework.

We used the four (or eight if their negative is accounted) typical regression models that can capture effectively the trend of any data behaviour (c.f. Daniel and Wood 1971):

1. *The linear regression*: According to Figure 7.2, this regression possibly identifies a non-admissible behaviour if it proves having a large slope, the one that possesses the highest 'determination factor' or best fit. The negative case represents an impossible case in computer system (performance cannot be enhanced significantly with the increase of demand). The general equation of such trend is:

$$y = A + Bx;$$

2. *The logarithmic regression*: According to Figure 7.2, this regression identifies the admissible tyը type of trend that represents the balance between queueing and overhead times. The negative case of this trend identifies a non-admissible behaviour. The general equation of this trend is:

$$y = A + B \ln x$$

3. *The Exponential regression*: According to Figure 7.2, this regression possibly identifies an admissible trend of behaviour depending on whether the shape of the trend is increasing sharply or not, as well as if there is a certain intended low level threshold of demand. The negative may identify an admissible type of behaviour trend. The general equation for such type of trend is:

$$y = Ae^{Bx}$$

4. *The power regression*: According to Figure 7.2, this regression and its negative are unlikely to identify an admissible trend only if this trend grows slowly. The general equation of this trend is:

$$y = Ax^{B}$$

The indices x (demand of user environment) and y (effectiveness of the computer machine environment) are the parameters representing the admissible graph dimensions. These indices must be representative of user interaction and machine environments. We can select a vast number of indices representing both environments, but since we want to present selected examples in this thesis, we have chosen the most general-purpose and sensitive indices. These are the number of concurrent processes representing the demand of the user environment and the response time (if it is with only foreground blackboard) and the system throughput otherwise, representing the effectiveness of the computer machine environment. Vast number of the design changes can be studied within this framework.

7.3 Strategy for The Behavioural Description And Performoance Measurement

The use of measurement probes to record the operational behaviour of software systems is not new. Several researchers have used this technique to monitor the operational behaviour (or dynamic behaviour) of several computer languages programs (e.g. Knuth 1971, Page and Benson 1974, and Torsun and Al-Jarrah 1981). However, the technique that we adopted is rather different from those that have been used for languages. The later operate by using a preprocessor for inserting measurement probes and their routines within the given program; then using the language compiler and running its generated code, the measurements are collected and the report is produced using a postprocessor.

In our case, the monitoring technique is rather simpler because the distinction has been made between the outer simulation representing the user interaction environment and the inner simulation representing the machine environment. This situation, indeed, allows the overall system performance to be measured by direct experimentation on both environments. Based upon this distinction, and in order to let this monitoring technique be coherent, it must:

1. include in all the exploration experiments a specific central nucleus of design parameters, to act as a control data set.
2. include appropriate measurement probes within both, the outer simulation (user-oriented) and within the inner simulation (system-oriented).

There are various goals behind using our performoact modelling. Basically there are four main goals: performance projection, performance monitoring and behaviour description, and the evaluation the structural changes. The last two which take into account the performance as a criterion for behaviour description even when structural changes are performed, are the most frequent goals and also forms the background of this chapter.

As mentioned earlier in this section, the performance of a computer system depends on its special application. It is therefore important to have a critical knowledge of the workload and the other characteristics of the system need to be evaluated. We shall consider the case when a system and already in use is to be upgraded or replaced by a new one. That means a real job profile and the system characteristics are known and can therefore be explored, analysed and extrapolated.

There exists information on many computer systems designs and their formal descriptions is available in the literature. This can be used as our control data set as well as to act as our special verification data.

Using the design data for an existing target computer system of known performance results, not only verification can be achieved but also the viability of the activity structure based possibilistic generator can be estimated and compared to the known performance of the target system. Once verified, the possibilistic simulator can generate diagnostic data to be used in the evaluation of any design claiming the family membership.

The most suitable computer systems design data, with functionalities matching our simulator functional structures, that can be used as control data set which we found in the current literature are the following systems from the class of the highly constrained systems:

- Nuke system (Crowley 1981),

- Thoth system (Cheriton 1979), and

- Gutenberg system (Stemple et al 1982).

Although the Nuke system design data is the only one used in this chapter for verification, the method is general, and can be applied to other (not necessarily existing) computer systems designs. The main reasons for selecting the Nuke system in verification are as follows:

1. Both the Nuke system and the possibilistic generator run under the same bare architecture: the Virtual Address extension of PDP-11 (i.e. VAX-11 architecture).
2. Both the the Nuke system and the possibilistic simulator are written in C programming language.
3. NUKE architecture was chosen by the US Army and Navy CFA (Computer Architecture Family project) committee as their standard tactical military computer (Fuller et al 1977). It was given the preference and selected from a long list of architectures (see section 2.3) after very careful experimentation with specific evaluation of the functionality of the architectures concerned.
4. The Nuke system possess the richest functional structures e.g.
   (a) a message passing communication structure,
   (b) static protection structure,
   (c) I/O control structure
   (d) effective interpretive structure (portable with effective addressing mechanisms), and
   (e) sophisticated information structures.

Assuming that existing computer system data has verified the behaviour of the possibilistic generator. The generator then can also be used for verifying many similar existing computer systems designs, which has been done previously only by using certain formal description languages (such as ISP (c.f. Bell and Newel 1971). There were used to simulate the design from the computer description provided by the manufacturers manuals. These manuals are supposed to contain the true specification and design information of the computer system. Unfortunately, the quality of the informal specification of the manufacturers manuals varies hence using the formal design languages does not provide reliable verification. Our possibilistic generator, however, utilises only the design data and does not require the full design description details. The design description of our possibilistic generator is quite general (refer to chapters 5 and 6). It is believed to contain the most sophisticated design description of most of the current computer systems. We consider the design description to be trivially embedded in the simulator structure.

7.4 A Brief Overview of The Nuke System

NUKE is a kernel-based, message-passing emulation of the UNIX kernel. NUKE consists of

1. a kernel that provides environments, first-level interrupt handling, process dispatching, and process communication via messages
2. several system processes that implement the UNIX system calls. The system processes are implemented exactly in the same way as regular UNIX processes. Figure 7.3 shows the structure of the NUKE. The following processes are included in NUKE:
    (a) The process manager handling process traps and the process related system calls.
    (b) The memory manager allocating and free memory, reallocating and moves stacks, and providing address translation services.
    (c) The memory scheduler process handling swapping.
    (d) The clock process handling clock interrupts and all timing services for the system both internally and through system calls.
    (e) The file system process handling the file and I/O system calls.
    (f) The device deriver processes (e.g., disk drivers, tape driver, tty driver) handling devices and interfaces to device controllers.

User processes make normal UNIX system calls which trap to the kernel all the defaults. The Kernel in turn converts them to messages and directs them to the system process that handle that system call. In course of handling the system calls, system processes will make the kernel calls, requesting services from the kernel and sending messages to other

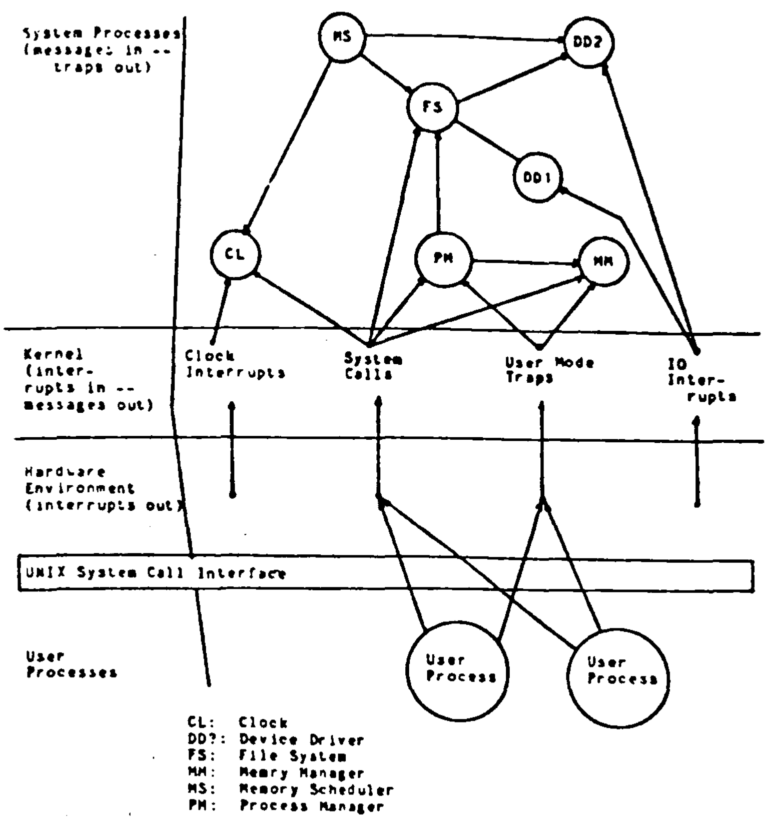


Figure 7.3: The Nuke Functional Framework

system processes which request services from them. The Nuke system, however, is an interactive computer system with no dynamic memory protection. It has only an overall static protection mechanism.

7.5 Collecting The Nuke-Oriented Design Data for The Study of Behaviour of The Possibilistic Generator

For the purpose of verifying our possibilistic generator, the simulator parameters (in PRE-PROCESSOR, POSSIBILISTIC SIMULATOR SWHW SETTING, POSSIBILISTIC SIMULATOR LOADING) were set to model VAX-11 computer system operating under NUKE (Crowley 1981, DEC 1977, DEC 1979).

Setting the design data is divided into two parts. The first part is to set the system-oriented parameters of the hardware and operating system assuming a general-purpose model with only static protection. The second part is to provide the *proper seeds* that can generate randomly realistic user-oriented (i.e. workload) parameters. Preparing the first part of the design data is quite straightforward, following the design manuals (Croweley 1981, DEC 1977, DEC 1979) and using certain general purpose design information when other required information is missing from the manuals (see Shaw 1974, London 1973, Yourdon 1972, Lewellyn 1976, Siewiork et al 1982, Hellerman and Conroy 1975). Table 7.1 lists the NUKE system-oriented parameters, selected for our possibilistic simulator verification. These values represent either the typical manual design settings or the average design values of certain typical general-purpose computer systems.

The user-oriented data, which form the second part of the design data, determines the system workload along with its characteristics. The workload is defined as collection of all individual jobs that are processed by the computer system during the specified period of time. The workload characterises the demand imposed by the jobs on various system resources. Realistic verification of our possibilistic generator requires that the submitted jobs possess the characteristics of a typical or representative sample of the jobs that the actual system would have to run.

In order to achieve this, the various properties of jobs must be simulated (or generated). Jobs may have several characteristics, but the most meaningful, easy to assess, and relevant to this work are listed in Table 5. The empirical distributions of most of these jobs characteristics were given by Robert Brundage (1974). It was measured from a programs sample that was run under the Burroughs B5500 computer system. The sample included 36 distinct Algol programs, concerned primarly with scientific and engineering applications. Among these were: a compiler, a linear programming package, and a variety of statistical programs. The programs and their associated data were chosen to keep the sample small enough, so that the trace data could be processed, yet varied enough to illustrate a number of significant features.

The Brundage data is well suited to our simulation for several reasons:

1. B5500 process structure closely approximates the process structure that one would expect in any descriptor-oriented computer system like our possibilistic simulator.

2. The B5500 uses a pure segmented virtual memory. Every array and code for each procedure is placed in a separate segment. This is exactly what we would expect in a system that enforces dynamic protection mechanism.

3. If one wishes to use empirical data to simulate segment reference (for memory protection), the Brundage data study contains the only available data.

Using the Brundage data we can extract the average value of each job characteristic distribution. These average values represent the random-number seeds in our simulation. The random number generator routine uses these seeds as well as their corresponding ranges to generate random values within the prescribed ranges.

However, not all the jobs characteristics (in Table 5) were we able to extract from Brundage data. The characteristics that we couldn't measure from Brandage data were:

- Average user think time,
- Average user faulty accesses,
- Mean number of User productivity,
- Mean number of user satisfaction, and
- User loss function average parameters.

The average values of these remaining parameters have been extracted two sets of other empirical data: Raymond Barber empirical data collected from University of New York (1979) and Chouinard Lewellyn empirical data collected from University of Illinos at Urbana-Champain (1976). Except for the user loss function parameters that we set the parameters intuitively. All the user-oriented parameters are listed in Table 7.2.

The remaining parameters of our possibilistic simulator are set at the system loading time, as shown below:

- Is spooling is required ? If yes then

  P53 Input spooler capacity 40 jobs

  P54 Output spooler capacity 50 jobs

- Is dynamic protection of the memory required ?
- Process table size ? P55 PTS 151 records

- Page/Segment table size ? P56 PSTS 301 records

Using all these parameters, the process of verifying the possibilistic generator can start.

Hardware Components Characteristics

Id	*Purpose*	*Setting Value*
P1	CPU Time to move one byte into core	0.1 msec
P2	CPU Context switching time	0.004 msec
P3	CPU Process dispatching time	0.1 msec
P4	CPU Primitive calling time	0.1 msec
P5	CPU Time to service kernel (timer interrupt)	0.05 msec
P6	CPU Time service kernel (job arrival)	0.1 msec
P7	CPU Time to service kernel (interrupt completion)	0.1 msec
P8	CPU Time to service kernel (faulty access)	0.1 msec
P9	CPU Time to service kernel (abort routine)	0.1 msec
P10	CPU Time to service kernel (halt routine)	0.1 msec
P11	CPU Time to service kernel (send msg)	0.6 msec
P12	CPU Time to service kernel (receive msg)	0.7 msec
P13	CPU Time to service kernel (call-activity)	0.8 msec
P14	CPU Time to service kernel (delete port)	0.1 msec
P15	CPU Time to service kernel (create port)	0.1 msec
P16	CPU Time to service kernel (changing eligible processes set)	0.1 msec
P17	CPU Time to service kernel (starting a process)	0.12 msec
P18	CPU Time to service kernel (stopping a process)	0.1 msec
P19	CPU Time to service kernel (modify access rights)	0.1 msec
P20	Memory size for non resident processes	131972 bytes
P21	Page and the average segment size	4096 bytes
P22	Disc transfer time (in msec/byte)	0.0033
P23	Disc seek time (in msec)	0.75
P24	Disc latency time (in msec)	0.12
P25	Disc record size (in bytes)	32767
P26	Drum transfer time (in msec/byte)	0.0083
P27	Drum seek time (in msec)	0.00
P28	Drum latency time (in msec)	8.00
P29	Drum record size (in byte)	32767
P30	Card reader transfer time (msec/byte)	0.75
P31	Card reader positioning time (msec)	0.0
P32	Card reader latency time (in msec)	0.0
P33	Card reader record size (in char/line)	80
P34	Line printer transfer time (msec/byte)	0.45
P35	Line printer positioning time (in msec)	0.0
P36	Line printer latency time (in msec)	0.0
P37	Line printer record size (char/line)	132
P38	Number of active terminals	32

Table 7.1: NUKE System-oriented Verification Parameters.

User-Oriented Parameters

P39	Mean interarrival time between jobs	15 sec
P40	Mean CPU time required by a job	15 sec
P41	Mean Core space required by a job	16384 bytes
P42	Mean number of tasks required for a job	3
P43	Mean number of backing store files required by a job	3
P44	Mean Number of backing store input records required by a job	150
P45	Mean number of backing store output records required by a job	150
P46	Average think time	3×10^4
P47	User loss function parameters	
	th	600
	tinf	3.6×10^{-6}
	mh	0.01
	minf	0.2
P48	faulty accesses percentage	0.75
P49	Mean number of user productivity	30
P50	Max. Satisfactory Response time	1×10^4
P51	Max. Satisfactory Turnaround time	7×10^5
P52	Number of jobs	9999

Table 7.2: The User-Oriented Parameters.

7.6 Performoact Modelling: Towards Analysing The Behaviour of The Possibilistic Simulator

In performoact modelling we run the possibilistic generator under a variety of conditions/changes, in order to extract certain useful inferences and models for performance prediction. These changes are grouped into three main purpose-oriented analysis activities:

1. reduction-oriented (i.e. adding/removing software/hardware components) type,
2. user-oriented (i.e. changes in the demand or the workload) type, and
3. system-oriented (changes in the hardware/software capabilities).

Before we attempt to analyse the effect of each type of change, we introduce the basic principles of the performoact modelling. Performoact modelling identifies from the simulation runs, the main statistical trend of every single change, and according to a standard modelling framework. The performoact modelling consists of a number of steps:

1. *Definition of a non-specific evaluation graph:* The performoact is a relationship between DEMAND X EFFECTIVENESS. The former is a non-specific index characterising the user interaction environment, whereas the latter is a non-specific index characterising the machine environment.
2. *Particularizing the non-specific graph to a pair of specific indices:* By assigning a specific index to each demand x effectiveness we obtain a specific performoact graph. To the DEMAND we assign one of the parameters and indices probes (c.f. section 5.4.1). To the EFFECTIVENESS we assign, on the other hand, the parameters and indices measured by the machine oriented performance probes (c.f. section 5.4.2).
3. *Running the generator:* This is done by setting the total system time and the statistic gathering period. Setting also the machine and the user oriented parameters as well as the initialisation parameters using Table 7.1 and 7.2. Then several runs may start by changing systematically any particular parameter.

4. *Regression analysis:* Fit essential regression models to the results obtained by the step 3 above. The essential regression models are linear, exponential, logarithmic and power. For each run results, the extracted best regression model referred to as best) then can be used for future performance predictions. The criteria for deciding which of the formulae is the best, depends upon the statistical index called *factor of determination* (c.f. Steel and Torrie 1980). This factor represents the degree to which the extracted formula matches observed simulation results. The best fit model does not represent automatically the admissibility of the model. The best model from the regression models that are marked BEST of all the runs, must have the highest factor of determination among the others (is referred to as B-BEST).

5. *Definition of admissibility:* For each run results, the regression model is admissible (referred to as "admis") if it is the curve enforcing the balance state between the demand and the effectiveness. In our performoact modelling framework, this curve will in most cases be logarithmic. Other admissible models are briefly discussed in section 7.2. The most acceptable model from the regression models that are marked ADMIS of all the runs, must have the highest determination factor among the others (is referred to as B-ADMIS).

6. *Definition of the generator tuning:* The generator behaviour may be tuned to its average behaviour using all the parameters that are associated with B-Best models. This tuning is called 'AVERAGE TUNING'. The generator can also be tuned according to the criteria of self- regulation by chosing all the parameters that are associated with B-ADMIS. This tuning is called 'BALANCE-TUNING'.

7.6.1 Towards Experimentations: The Reduction Oriented Changes

For the purpose of analysing the effects of certain intrinsic changes within the structure of our possibilistic simulator, we initially run four different versions of the simulator. Using these versions we are trying to assess and model the effect of modifying the possibilistic simulator to include a dynamic protection mechanism within the memory system, as well as the effects of removing the management unit of background blackords or spoolers (i.e. transforming the simulator from a general-purpose one to an interactive type).

These versions are:

V1 A general purpose possibilistic simulator with only static protection (GPPSSP),

V2 A general purpose possibilistic simulator with static and dynamic memory protection (GPCDMP),

V3 An interactive constellation with only an overall static protection (ICSP), and

V4 An interactive constellation with static and dynamic memory protection (ICSDMP).

Each version is chosen by specifying the changes[4] (i.e. reduction data from V1 to V3 or addition data from V4 to V3) on the NUKE control data. This is done by the PREPROCESSOR which later selects the required system components via the use of `#include module` macro.

Studying such types of changes is important. For example, it may be necessary to cut costs by reducing equipments. In that case, before arbitrarily removing or adding some piece of equipment or a new capability, the designer should determine how the on-line and batch systems will be affected, and how protection might effect both systems. If problem areas can be identified prior to the removal of the equipment action can be taken to reduce any negative impact. The next decision may then involve the selection of equipment or capability to remove or add, respectively. This decision can not be made if comparative data on the performance of various configurations are not obtainable. We should note that our batch system is not a sequential batch system as those found in the third generation computer systems (c.f. Beaumont and Macaskill 1975). Our batch system is managed by the control structure of our activity structures based simulation using a common blackboard area similar to the background queue of the Unix system.

To analyse the effects of these changes, we performed our performoact modelling scheme after running the four simulator versions. Figures 7.4 to 7.6 show the linear regression

[4] Changes in the context of this section does not mean systematic or incremental changes. Hence, we can not option the B-Best and the B-Admis models of the performoact framework.

performoact models of the four various changes and compare the changes effects using three different effectiveness indices.

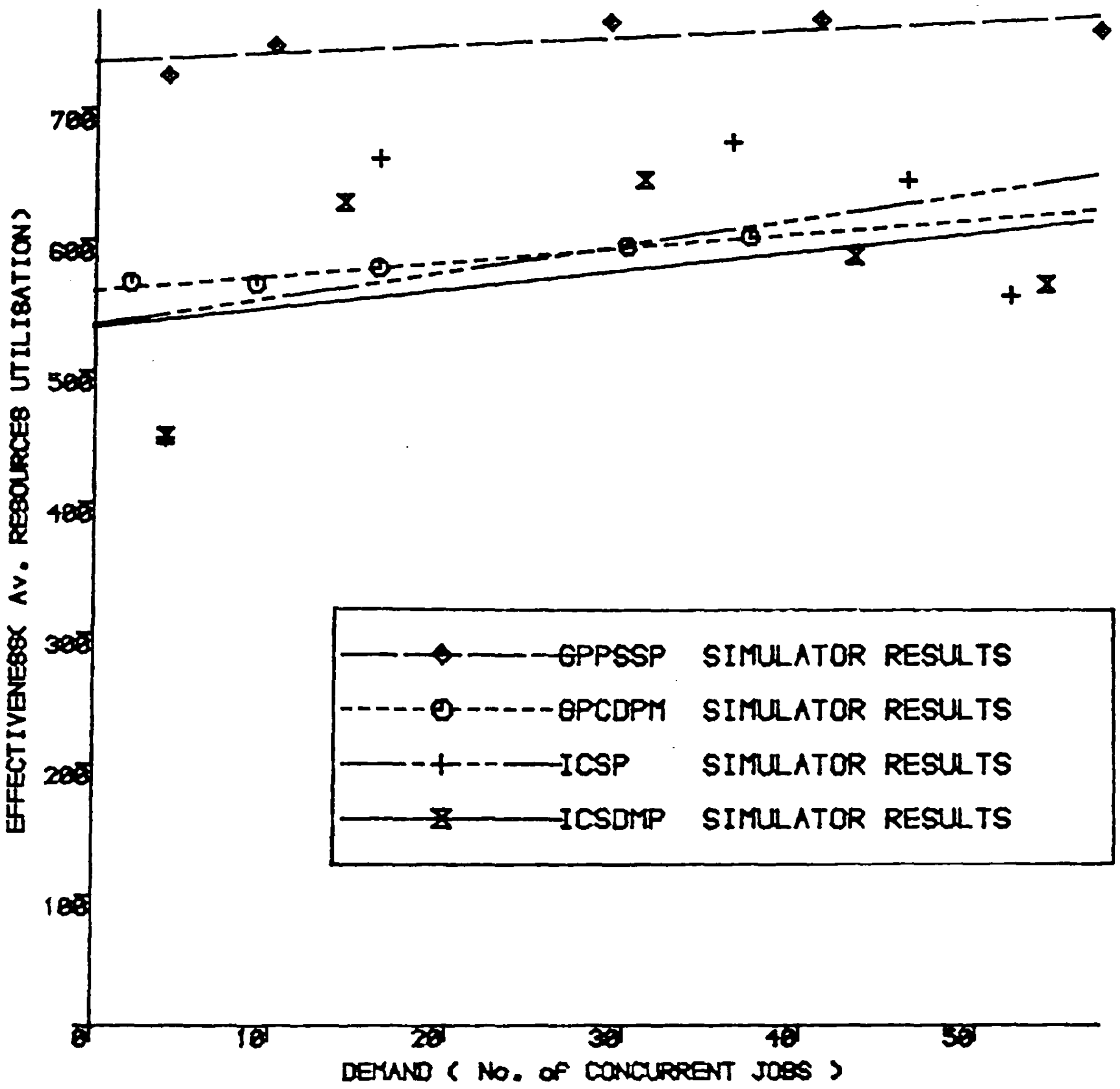


Figure 7.4: PERFORMOACT Modelling: Reductions Effects using Average Resource Utilisation

Tables 7.3 to 7.5 shows the other regression analysis models that can be used for finding the best description and possibly the admissible formulae.

The effects of these changes upon the average resources utilisation index reveals that by adding a dynamic protection mechanism it largely effects the utilisation of the general-purpose possibilistic simulator version (i.e. with spooling) and slightly effects the interactive version (i.e. without spooling). The equations marked 'best' describe these changes of behaviour and future predictions, as demand decreases or increases.

Table 7.4 shows that the average system throughput for the interactive versions (with

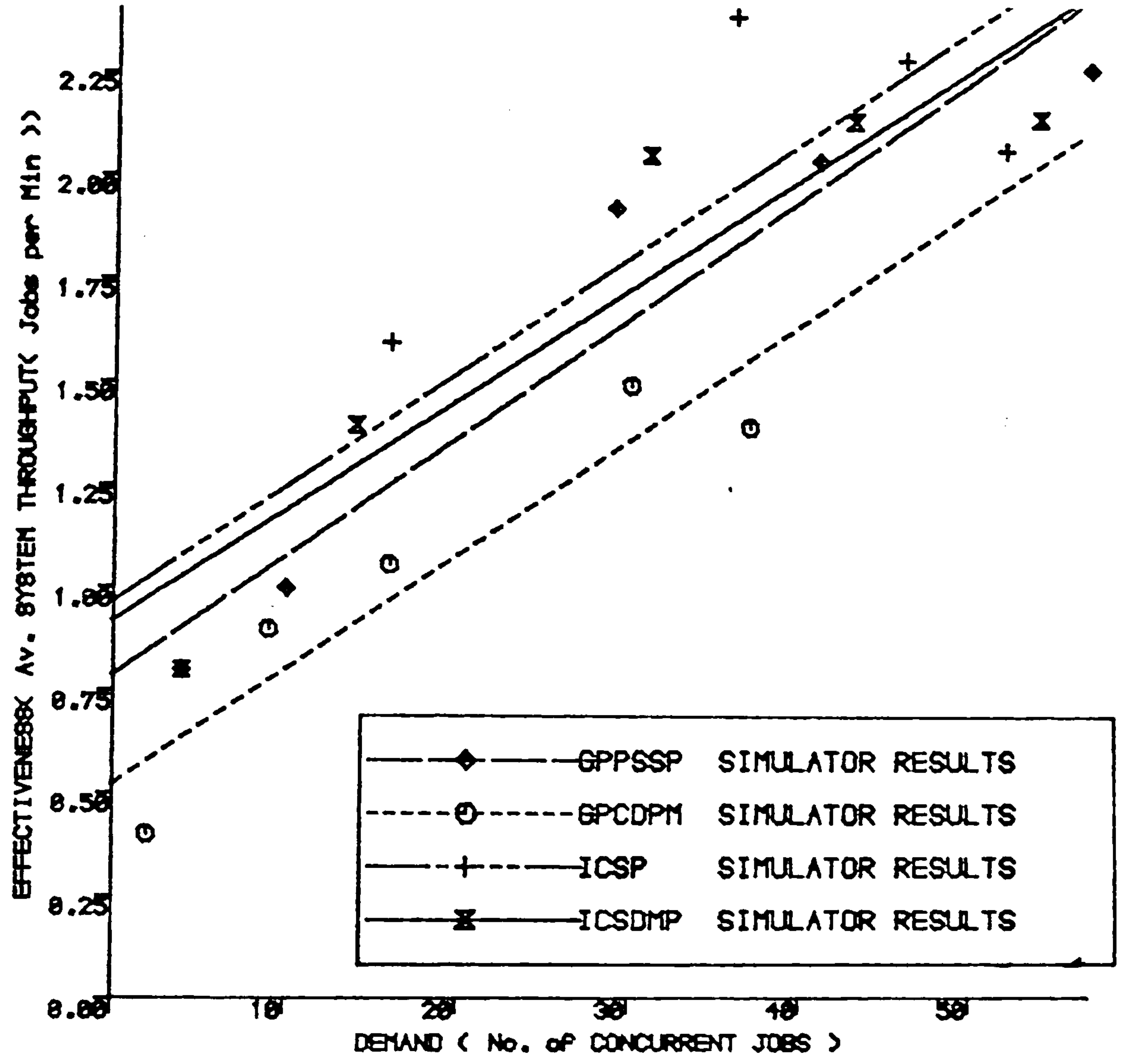


Figure 7.5: PERFORMOACT Modelling: Reductions Effects using Average System Throughput

or without dynamic memory protection) is higher than the average system throughput for the general purpose versions. We noticed that the dynamic protection mechanism has less impact upon the interactive version than the general-purpose version. We also noticed that the degradation impact, upon the average system throughput, of adding a spooling unit is greater than that of adding the dynamic memory protection unit.

Table 7.5 illustrates that the average response degrade by 5 seconds on average when a spooling unit is added. It degrade more when the dynamic memory protection is added to the general purpose simulator, and less when the same dynamic memory protection unit is added to the interactive version.

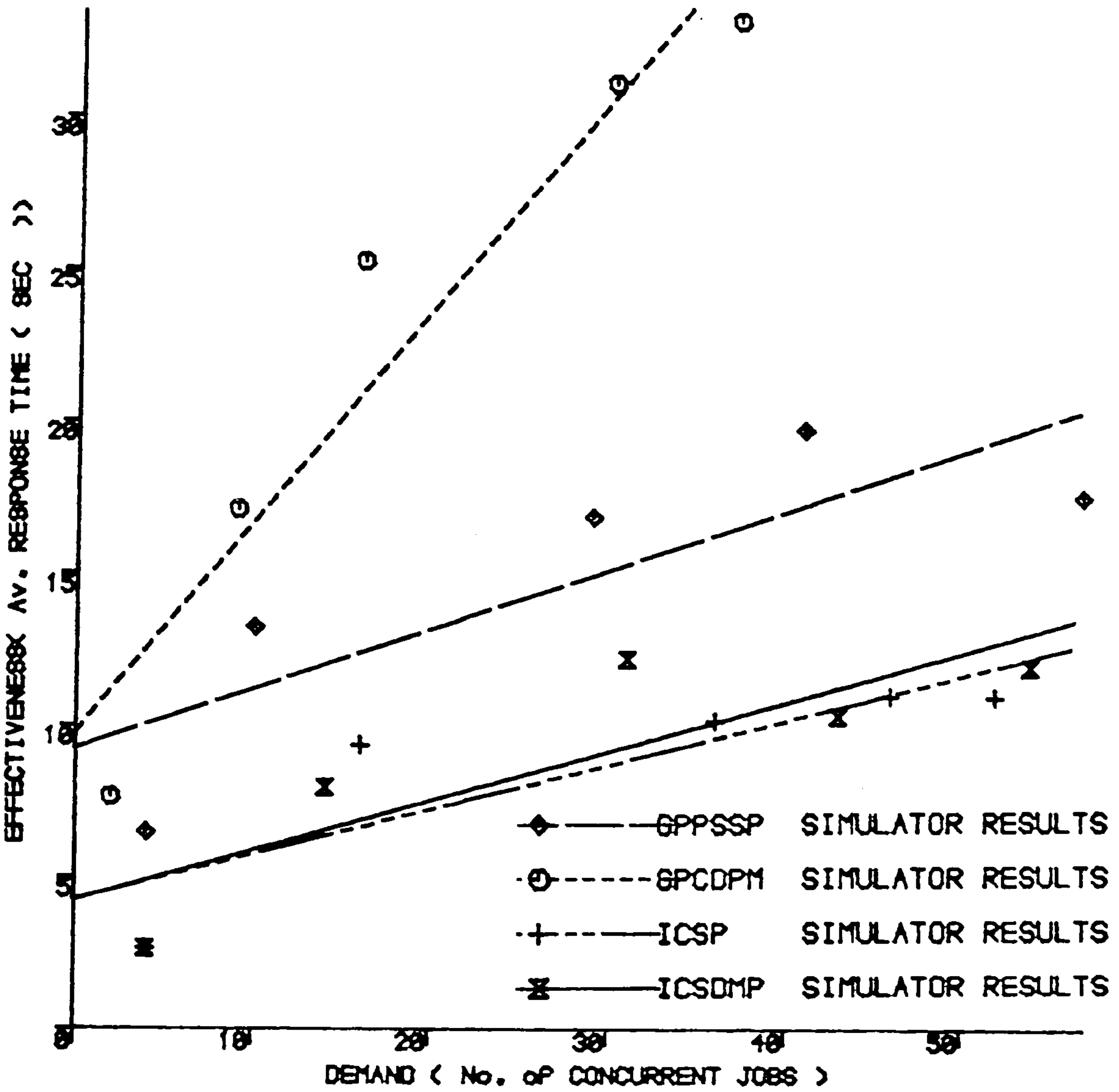


Figure 7.6: PERFORMOACT Modelling: Reductions Effects using Average Response Time

It is important to note that the performoact modelling equations describe the performance behaviour subject to certain changes and are independent of the time changing factor. On the other hand, the traditional simulation techniques always associate the variation of any performance index with the time factor (e.g. simulation time, cpu utilisation time, etc.). They prove very sensitive and confusing as many researchers note (see Mohamad 1981 for the survey). Figures 7.7 to 7.9 illustrate the way, the simulation based modelling describes the effects of the major reduction of activities upon the average response time index (note for example the difficulty of deciding which curve in figure 7.7 represent better response time).

It is important to note that in order to tune the behaviour of the possibilistic generator

```
----------------------------------------------------------------
MIN                                5       10       15       20       25
----------------------------------------------------------------
(V1)      OBSERVED DEMAND   4       10       29       41       57
(GPPSSP) OBSERVED EFFECT.  725.2   748.6    769.64   773.3    766.9
LIN PERFORMOACT MODEL : A= 735.7 B=.745      DF=.814
EXP PERFORMOACT MODEL : A= 735.6 B=.0009     DF=.813
LOG PERFORMOACT MODEL : A= 705.3 B=17.37     DF=.952 *ADMS*
PWR PERFORMOACT MODEL : A= 706.3 B=.023      DF=.953 *BEST*
----------------------------------------------------------------
(V2)       OBSERVED DEMAND  2       9        16       30       37
(GPPSDMP) OBSERVED EFFECT. 567.5   566.6    580.3    597.8    605.8
LIN PERFORMOACT MODEL : A= 561.1 B=1.196     DF=.982
EXP PERFORMOACT MODEL : A= 561.4 B=.0020     DF=.983 *BEST*
LOG PERFORMOACT MODEL : A= 550.8 B=12.93     DF=.855 *ADMIS*
PWR PERFORMOACT MODEL : A= 551.5 B=.0221     DF=.857
----------------------------------------------------------------
(V3)    OBSERVED  DEMAND    4       16       36       46       52
(ICSP) OBSERVED  EFFECT.    449.2   663.24   678.2    650.8    571.8
LIN PERFORMOACT MODEL : A= 535.4 B=2.12      DF=.448
EXP PERFORMOACT MODEL : A= 523.2 B=.004      DF=.484
LOG PERFORMOACT MODEL : A= 408.6 B=61.92     DF=.686  *ADMIS*
PWR PERFORMOACT MODEL : A= 414.1 B=.116      DF=.717  *BEST*
----------------------------------------------------------------
(V4)      OBSERVED DEMAND   4       14       31       43       54
(ICSDMP) OBSERVED EFFECT.   450.9   629.9    648.6    592.5    571.8
LIN PERFORMOACT MODEL : A= 533.6 B=1.545     DF=.407
EXP PERFORMOACT MODEL : A= 524.1 B=.003      DF=.444
LOG PERFORMOACT MODEL : A= 427.9 B=49.58     DF=.675  *ADMIS*
PWR PERFORMOACT MODEL : A= 428.8 B=.0959     DF=.705  *BEST*
----------------------------------------------------------------
Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
        THE AV. RESOURCES UTILISATION REPRESENTING EFFECTIVENESS;
        LINEAR PERFORMACT MODEL = EFF = A + B * DEMAND
        EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP( B* DEMAND)
        LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN( DEMAND)
        POWER PERFORMOACT MODEL = EFF = A*DEMAND**B
```

Table 7.3: PERFORMACT MODELLING OF THE MAJOR REDUCTION ACTIVITIES USING THE AVERAGE RESOURCES UTILISATION FOR EFFECTIVENESS

it is important to select those design data marked ADMIS (i.e. admissible). If the trend is marked ADMIS and BEST this means the system is already showing the required behaviour.

```
---------------------------------------------------------------
MIN                          5        10       15       20       25
---------------------------------------------------------------
(V1)      OBSERVED DEMAND    4        10       29       41       57
(GPPSSP) OBSERVED EFFECT.    0.8      1.0      1.93     2.05     2.28
LIN PERFORMOACT MODEL : A=  .78   B=2.92      DF=.96
EXP PERFORMOACT MODEL : A=  .83   B=.02       DF=.93
LOG PERFORMOACT MODEL : A= -.14   B=.59       DF=.97 *ADMIS*
PWR PERFORMOACT MODEL : A=  .42   B=.42       DF=.98 *BEST*
---------------------------------------------------------------
(V2)       OBSERVED DEMAND   2        9        16         30     37
(GPPSDMP) OBSERVED EFFECT.   0.4      0.9      1.06       1.5    1.4
LIN PERFORMOACT MODEL : A=  .52   B=.02       DF=.93
EXP PERFORMOACT MODEL : A=  .52   B=.03       DF=.87
LOG PERFORMOACT MODEL : A=  .11   B=.36       DF=.98 *ADMIS*
PWR PERFORMOACT MODEL : A=  .30   B=.44       DF=.99  *BEST*
---------------------------------------------------------------
(V3)   OBSERVED  DEMAND      4        16       36         46     52
(ICSP) OBSERVED  EFFECT.     0.8      1.6      2.4        2.3    2.08
LIN PERFORMOACT MODEL : A=  .96   B=.02       DF=.87
EXP PERFORMOACT MODEL : A=  .94   B=.01       DF=.86
LOG PERFORMOACT MODEL : A=  .006  B=.58       DF=.95  *ADMIS*
PWR PERFORMOACT MODEL : A=  .47   B=.41       DF=.96  *BEST*
---------------------------------------------------------------
(V4)     OBSERVED DEMAND     4        14       31         43     54
(ICSDMP) OBSERVED EFFECT.    0.8      1.4      2.06       2.15   2.16
LIN PERFORMOACT MODEL : A=  .92   B=.02       DF=.92
EXP PERFORMOACT MODEL : A=  .92   B=.01       DF=.89
LOG PERFORMOACT MODEL : A=  .005  B=.56       DF=.99  *BEST*ADMIS*
PWR PERFORMOACT MODEL : A=  .47   B=.40       DF=.98
---------------------------------------------------------------
Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
        THE AV. RESOURCES UTILISATION REPRESENTING EFFECTIVENESS;
        LINEAR PERFORMACT MODEL = EFF = A + B * DEMAND
        EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP( B* DEMAND)
        LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN( DEMAND)
        POWER PERFORMOACT MODEL = EFF = A*DEMAND**B
```

Table 7.4: THE EFFECTS OF THE MAJOR REDUCTION TECHNIQUES UPON THE AVERAGE SYSTEM THROUGHPUT INDEX

```
-----------------------------------------------------------------
MIN                            5        10       15       20       25
-----------------------------------------------------------------
(V1)     OBSERVED DEMAND      4        10       29       41       57
(GPPSSP) OBSERVED EFFECT.     6.49     13.26    16.96    19.93    17.72
LIN PERFORMOACT MODEL : A= 9.25  B=.19          DF=.82
EXP PERFORMOACT MODEL : A= 8.77  B=.01          DF=.78
LOG PERFORMOACT MODEL : A= 1.33  B=4.56         DF=.95  *BEST*ADMIS*
PWR PERFORMOACT MODEL : A= 4.46  B=.38          DF=.93
-----------------------------------------------------------------
(V2)       OBSERVED DEMAND  2        9        16        30     37
(GPPSDMP) OBSERVED EFFECT. 7.65     17.14    25.34     31.27  33.39
LIN PERFORMOACT MODEL : A= 9.78  B=.70        DF=.95
EXP PERFORMOACT MODEL : A= 10.10 B=.03        DF=.89
LOG PERFORMOACT MODEL : A= .18   B=8.9        DF=.98
PWR PERFORMOACT MODEL : A= 5.51  B=.51        DF=.99  *BEST*ADMIS*
----------------------------------------------------------------
(V3)   OBSERVED  DEMAND     4        16       36        46     52
(ICSP) OBSERVED  EFFECT.    2.67     9.73     10.25     11.11  12.28
LIN PERFORMOACT MODEL : A= 4.27  B=.15          DF=.85
EXP PERFORMOACT MODEL : A= 3.73  B=.02          DF=.80
LOG PERFORMOACT MODEL : A= -1.13 B=3.22         DF=.96  *BEST*ADMIS*
PWR PERFORMOACT MODEL : A= 1.46  B=.54          DF=.94
-----------------------------------------------------------------
(V4)      OBSERVED DEMAND   4        14       31        43     54
(ICSDMP) OBSERVED EFFECT.   2.65     7.99     12.28     11.43  12.09
LIN PERFORMOACT MODEL : A= 4.24  B=.16          DF=.85
EXP PERFORMOACT MODEL : A= 3.81  B=.02          DF=.81
LOG PERFORMOACT MODEL : A= -1.90 B=3.61         DF=.96  *BEST*ADMIS*
PWR PERFORMOACT MODEL : A= 1.38  B=.57          DF=.95
-----------------------------------------------------------------
Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
        THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS;
        LINEAR PERFORMACT MODEL = EFF = A + B * DEMAND
        EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP( B* DEMAND)
        LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN( DEMAND)
        POWER PERFORMOACT MODEL = EFF = A*DEMAND**B
```

Table 7.5: THE EFFECTS OF THE MAJOR REDUCTION TECHNIQUES UPON THE AVERAGE RESPONSE TIME INDEX

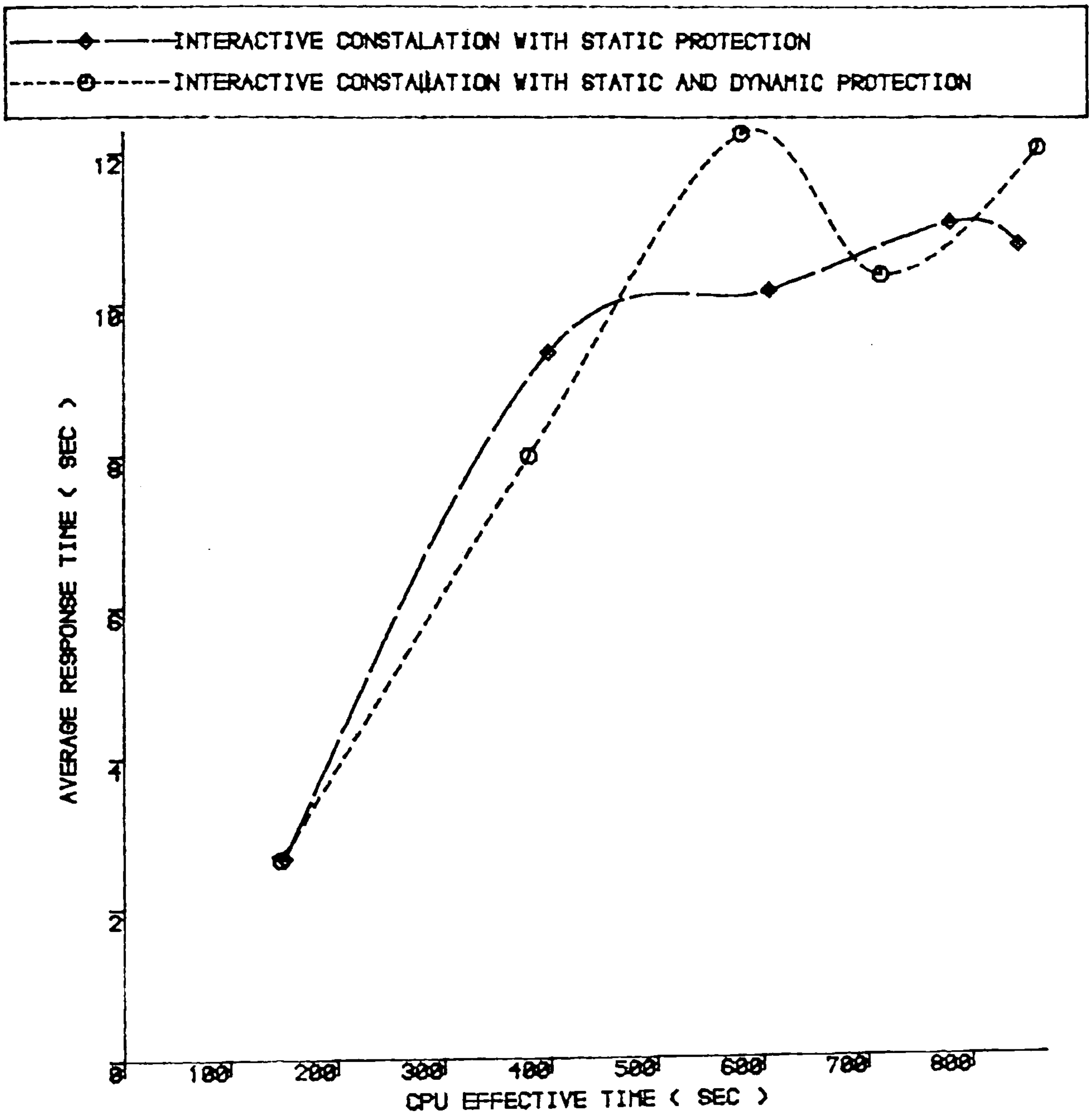


Figure 7.7: Adding Dynamic Memory Protection to The Interactive Constellation

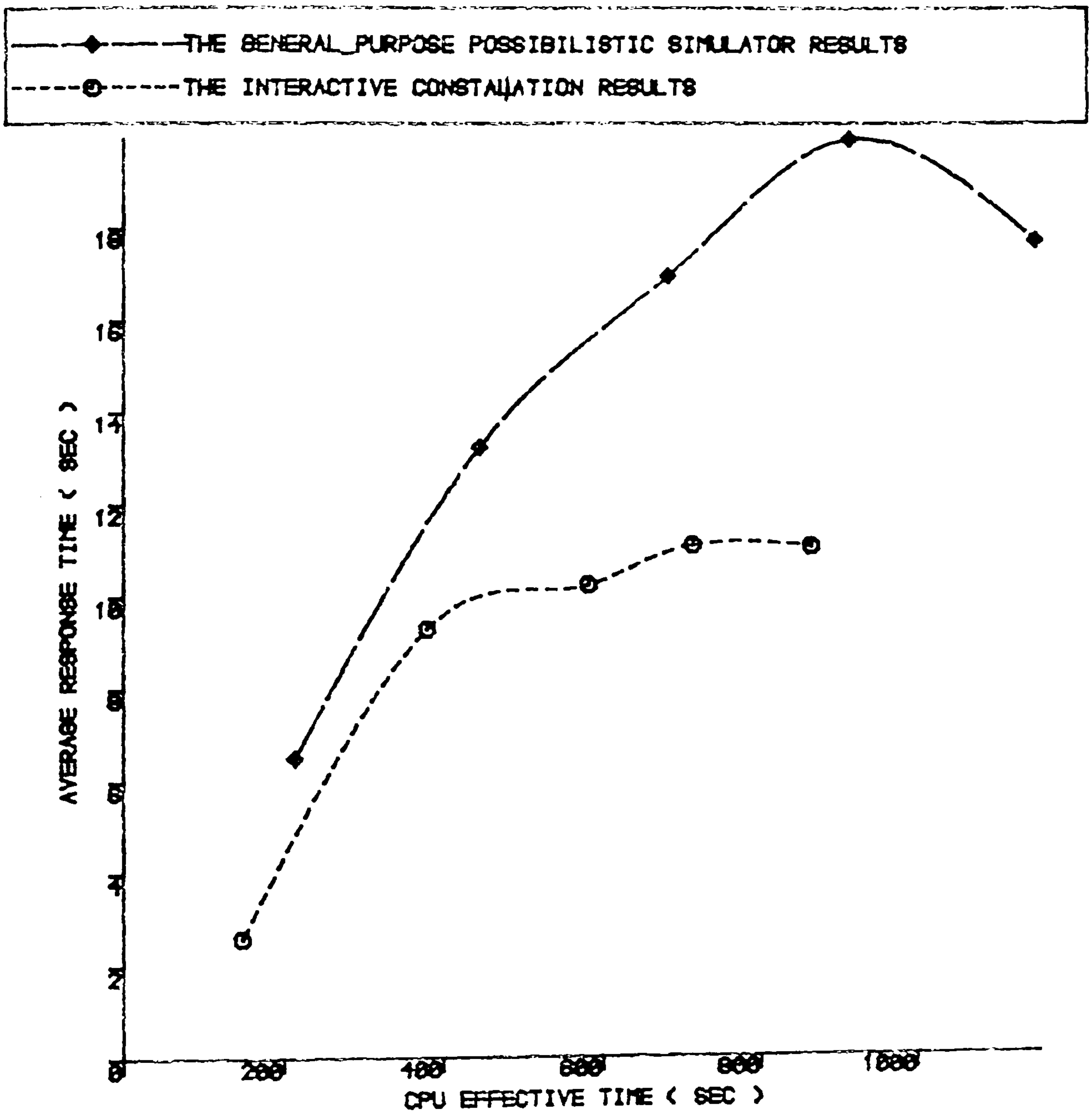


Figure 7.8: Reducing The General Purpose Simulator to An Interactive Constellation

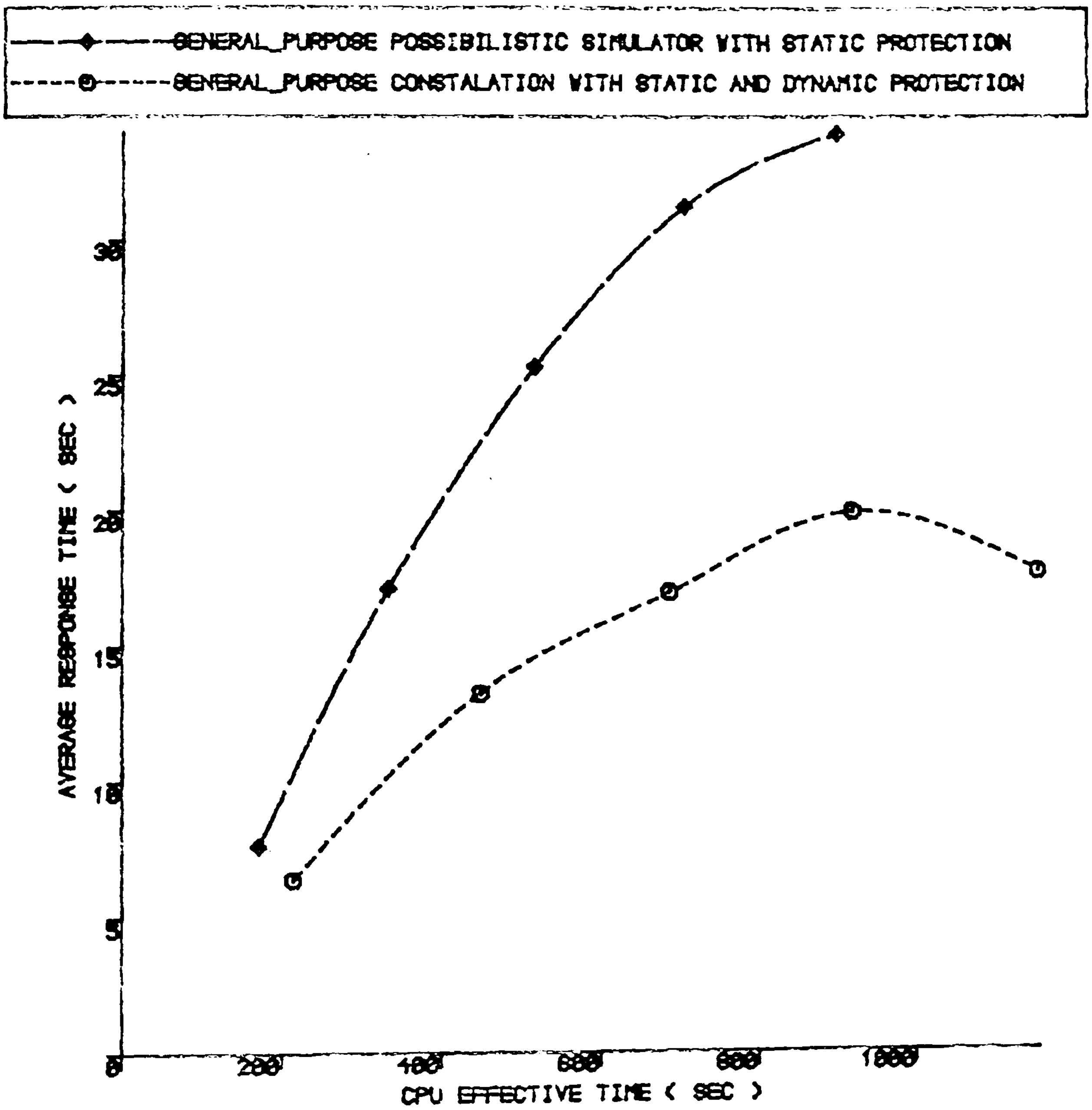


Figure 7.9: Reducing GPPS to a GPC with Dynamic Memory Protection

7.6.2 The results of experiments of performoact modelling:

In order to prepare for the performoact modelling, we need to gather data from running the simulator. For this purpose we set our system operation time to 25 minutes (i.e. 1.5×10^6 msecs) and the statistics gathering period is set at 5 minute ($= 3 \times 10^5$ msecs) intervals. The other parameters were set according to the NUKE control data of Tables 7.1 and 7.2.

The changes include systematic variations of both, hardware and software parameters. We decided to use one version of the possibilistic simulator, namely, the interactive constellation with dynamic memory protection. Also for a more narrow assessment of performance we decided to use the average system response time index as the most representative index of the factors involving the matching environment, and the number of concurrent jobs representing the user environment see Figure 7.10). Response time has also a great impact upon the user productivity.

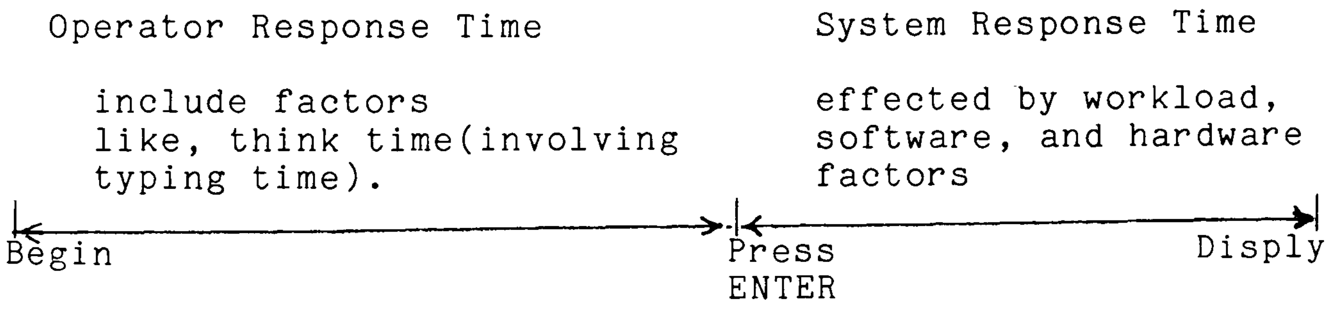


Figure 7.10: The transaction structure and its contribution to the response time index.

Here the reader should note the the original NUKE-based control data will be marked as "CON" at the left side of each performoact table to provide the clear indication of any variation from the original set of data.

There are indeed a large number of experiments that can be studied and analysed using the performoact framework. We are presenting in this section only one experiment and a table 7.6 which summarises the data associated with the B-Admis models as well as the data associated with B-Best models. The experiment introduced in this section is used to demonstrate the idea of performoact and the table list the two tuning data sets that will be used in the next section for validation. The details of the experiments (2-20) that contributes to the results listed in Table 7.6 can be obtained from the Appendix. For Experiment 1 see the sequel.

Exp No	B_BEST MODEL	B_ADMIS MODEL	B_BEST Values	B_ADMIS Values
1	LOG	LOG	24 Terminal	24 Terminal
2	LOG	LOG	75 Transaction	75 Transaction
3	LOG	LOG	8.0 Sec	8.0 Sec
4	LOG	LOG	.25 Rate	.25 Rate
5	PWR	LOG	10 Sec	20 Sec
6	LOG	LOG	2 Tasks	2 Tasks
7	LOG	LOG	3 Sec	3 Sec
8	PWR	LOG	15 Sec	15 Sec
9	LOG	LOG	15384 Byte	15384 Bytes
10	LOG	LOG	400 Records	400 Records
11	LOG	LOG	3072 Bytes	3072 Bytes
12	LOG	LOG	151552 Bytes	151552 Bytes
13	LOG	LOG	0.4 Msec	0.4 Msec
14	LOG	LOG	0.0066 Msec	0.0066 Msec
15	LOG	LOG	16 Msec	16 Msec
16	LOG	LOG	1 Pass	1 Pass
17	LOG	LOG	SRR	SRR
18	LOG	LOG	0.5 Msec	0.5 Msec
19	LOG	LOG	1.5 Msec	1.5 Msec
20	LOG	LOG	0.3 Msec	0.3 Msec

Note that experiments 2-20 are in the appendix.
B_BEST refers to the average tuning values
B_ADMIS refers to the balance tuning values

Table 7.6: Summary of the experimentation.

These tuning values will be used in the next section for validation.

Experiment 1: Number of terminals vs Av. response time. Using the performoact modelling scheme after running the interactive constellation with different number of active terminals attached to it, table 7.7 is produced.

The average response decreases as the number of terminals decreases (in average 2 seconds for each eight terminals). Figure 7.11 illustrates the linear performoact models of different active terminals setting, attached to the interactive constellation. In this experiment, the logarithmic curve of 24 terminal prove to be the best for both, the average and

```
-----------------------------------------------------------------
MIN                              5        10        15        20        25
-----------------------------------------------------------------
(16)      OBSERVED DEMAND    4        13        27        40        46
Trmnls.) OBSERVED EFFECT.  2.38     3.89      6.13      5.28      6.7
LIN PERFORMOACT MODEL : A= 2.56  B=0.08       DF=.89
EXP PERFORMOACT MODEL : A= 2.64  B=0.02       DF=.88
LOG PERFORMOACT MODEL : A= 0.02  B=1.64       DF=.94  *ADMIS*
PWR PERFORMOACT MODEL : A= 1.39  B=0.40       DF=.96  *BEST*
-----------------------------------------------------------------
(24)       OBSERVED DEMAND  4        14       30        42       50
Trmnls.)  OBSERVED EFFECT. 2.3       5.94    8.62       8.55    9.18
LIN PERFORMOACT MODEL : A= 3.06  B=.13      DF=.91
EXP PERFORMOACT MODEL : A= 2.97  B=.02      DF=.86
LOG PERFORMOACT MODEL : A=-1.38  B=2.75     DF=.99  *BEST*ADMIS*
PWR PERFORMOACT MODEL : A= 1.18  B=.54      DF=.97
-----------------------------------------------------------------
(32)   OBSERVED  DEMAND     4        14       31        43       54
CON)   OBSERVED  EFFECT.    2.65     7.99     12.28     10.43  12.00
LIN PERFORMOACT MODEL : A= 4.24  B=.16        DF=.85
EXP PERFORMOACT MODEL : A= 3.81  B=.02        DF=.81
LOG PERFORMOACT MODEL : A= -1.90 B=3.61       DF=.96  *BEST*ADMIS*
PWR PERFORMOACT MODEL : A= 1.38  B=.57        DF=.95
-----------------------------------------------------------------
(40)      OBSERVED DEMAND   4        14       32        45       56
Trmnls.) OBSERVED EFFECT.  3.39     8.19     12.65     12.76  13.13
LIN PERFORMOACT MODEL : A= 4.61  B=.17        DF=.90
EXP PERFORMOACT MODEL : A= 4.46  B=.02        DF=.85
LOG PERFORMOACT MODEL : A= -1.87 B=3.88       DF=.98  *BEST*ADMIS*
PWR PERFORMOACT MODEL : A= 1.79  B=.52        DF=.97
-----------------------------------------------------------------
Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
        THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS;
        LINEAR PERFORMACT MODEL = EFF = A + B * DEMAND
        EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP( B* DEMAND)
        LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN( DEMAND)
        POWER PERFORMOACT MODEL = EFF = A*DEMAND**B
           : B-BEST
           : B-ADMIS
```

Table 7.7: The Effects of Changing The Average Number of Terminals on The Average Response Time Index

balance tunings. linear performoact models of different active terminals settings attached to the interactive constellation.

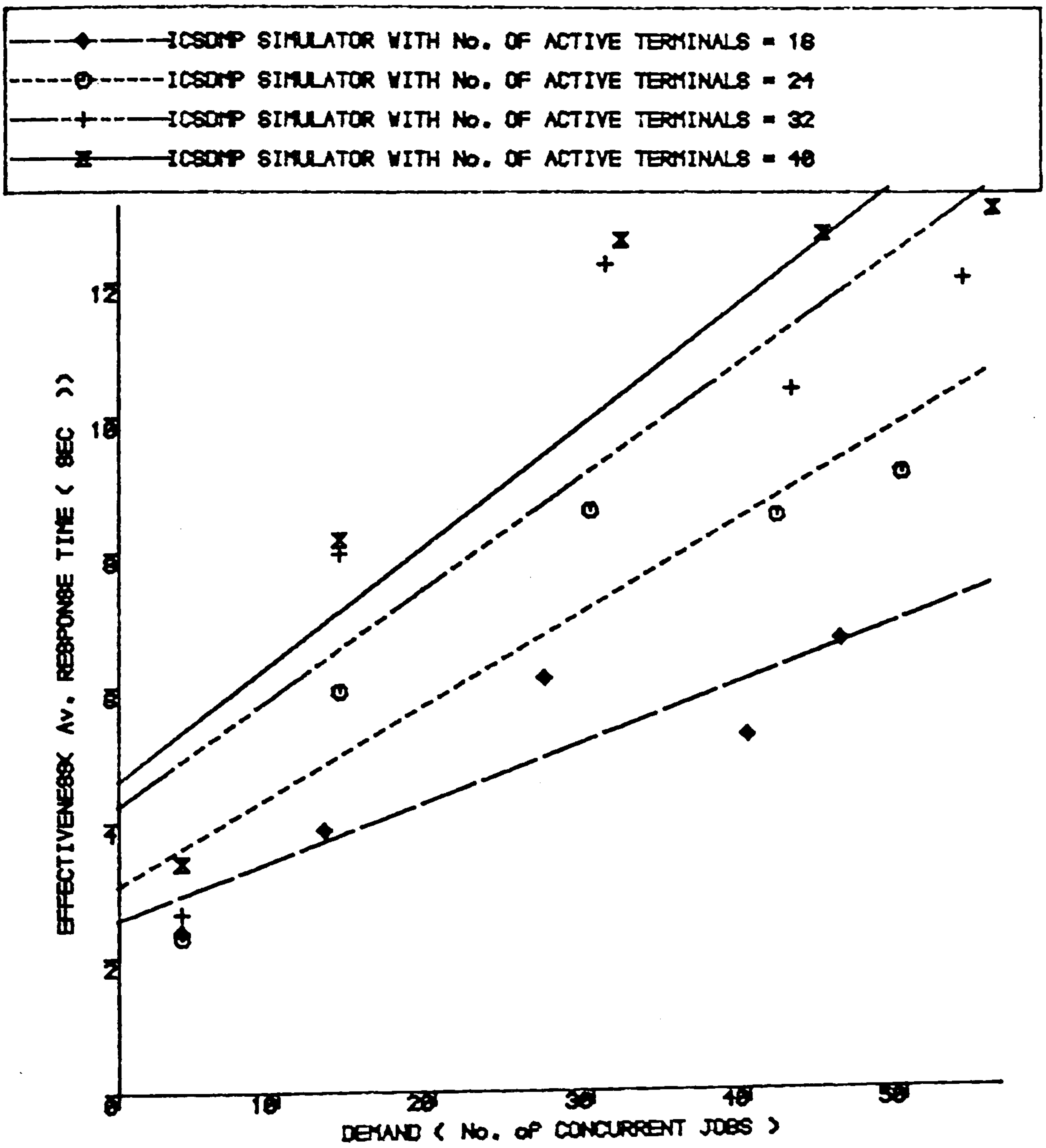


Figure 7.11: PERFORMOACT Modelling: The Effects of Adding Terminals

7.7 The Use of Admissible Models and The Validation Issue

It was of a great interest to see the effects of chosing the design data of the balance tuning (B-ADMIS) and those associated with the average tuning (B-BEST) for tuning the possibilistic generator. This can be done by re-runing the possibilistic generator according to these design data. It was of interest also to validate our tuned results (using Tables 7.6, 7.1, and 7.2) against the actual results obtained from a similar system (i.e. DEC-Nuke-oriented system) as reported by Penny and Sheedy (1980). For this purpose, we run our possibilistic generator for the same period monitored by Penny and Sheedy, 90 minutes, the response time was monitored each 5 minutes. The running results and the comparison is shown in Figure 7.12.

It is clear from Figure 7.12 that our system with the balance tuning data in most of the cases provide better response time than the actual system and less effectively of that with the average tuning data and showing both greater stability than the actual system results. This due mainly to the identification of the admissible behaviour of the cooperating environment and tuning the system accordingly. The correlation coefficient between the actual system response times and our possibilistic simulator balance-tuning response times is 0.5933 and with our average-tuning response times 0.5317 which are relatively high considering that our possibilistic simulator possess the self- regulating behaviour due to the learning mechanisms adopted in the complete shell of the possibilistic generator that the actual system does not implicitly have.

It is an established fact, however, that the *validation* of a complex software system such as the possibilistic generator, is a complicated process (c.f. Hughes 1981, Theory 1975). That is beside the fact that the activity structures simulation is not meant to simulate existing systems so that we need to provide rigorous validation proofs. Activity structures simulation has been performed to show the applicability of the activity structures methodology as well as to show the effectivity of its design principles in producing high-performance architectures for highly constrained systems. The validation of our possibilistic simulator mainly depends upon the verification of the logic of the simulator C programs.

However, we may perform certain degree of validation using theoretical models. Unfortunately, there are no theories to design and analyse distributed computer systems such as our possibilistic generator (c.f., Klienrock 1985). However, we can direct the comparison to a different type of theoretical models; those extracted from empirical situations. For this purpose, we selected two notable empirical models, one based the constant model of Boyse and Warn (1975) and the second based on a model extracted the operational laws of Buzen (1979).

For this case study we run the generator by using the data mentioned in tables 7.1 and 7.2, in which the only variable for further runs is the number of active terminals in the system and the only performance index is the average response time. The relevant data in Tables 7.1 and 7.2 are used by the two empirical models. The empirical models define the response time as follows:

BOYSE AND WARN EMPIRICAL MODEL:

C the average CPU time required by a job

c the average CPU time period between I/O operations

i the average service time of an I/O request

M the effective degree of multiprogramming

N the number of active terminals

Z the average user think time

K number of devices

U CPU utilisation

$= 1 - \frac{(M-K)!}{M!} \left(\frac{i}{C}\right)^k$

R Response time

$= \frac{NC}{U} - Z$

THE OPERATIONAL LAWS OF BUZEN:

N is the number of active terminals

Z the user average think time

K number of devices

h is a normalisation factor (Williams and Brandiwad 1976)

$X = \frac{h(N-1,K)}{h(N,K)} \frac{N}{Z}$

R Response time

$= \frac{N}{X} - Z$

Figure 7.13 illustrates the first case study results.

The comparison results show close agreements between the results (correlation coefficient between the possiblistic generator response times and the Boyse and Warn response times is 0.9090 and between the possibilistic generator response times and the operational analysis response times is 0.9357), indicating that our possibilistic simulator can be trusted for conventional computer systems design.

The other dimension of performing some validation on our possibilistic generator can be performed in the direction of comparing the performance of specific components or structures of the possibilistic generator with an existing, similar type computer system components.

We picked performance data from the activity of the memory system utilising a disc available and from two well-protected computer systems, the CAP computer (Wilkes and Needham 1979) and the HYDRA computer (Cohen et al 1974) in which also they are exceptionally successful. Figure 7.4 illustrate the comparison.

Figure 7.14 shows a relatively close agreement (the correlation coefficient between the possibilistic simulator disc utilisations and the Hydra system disc utilisation is 0.5215 whereas the correlation between the possibilistic simulator disc utilisations and the CAP system disc utilisation is 0.5404). This indicates that our possibilistic simulator disc is slightly over utilised compared to the two notable computer systems. This is again may provide further confidence in our possibilistic simulator results.

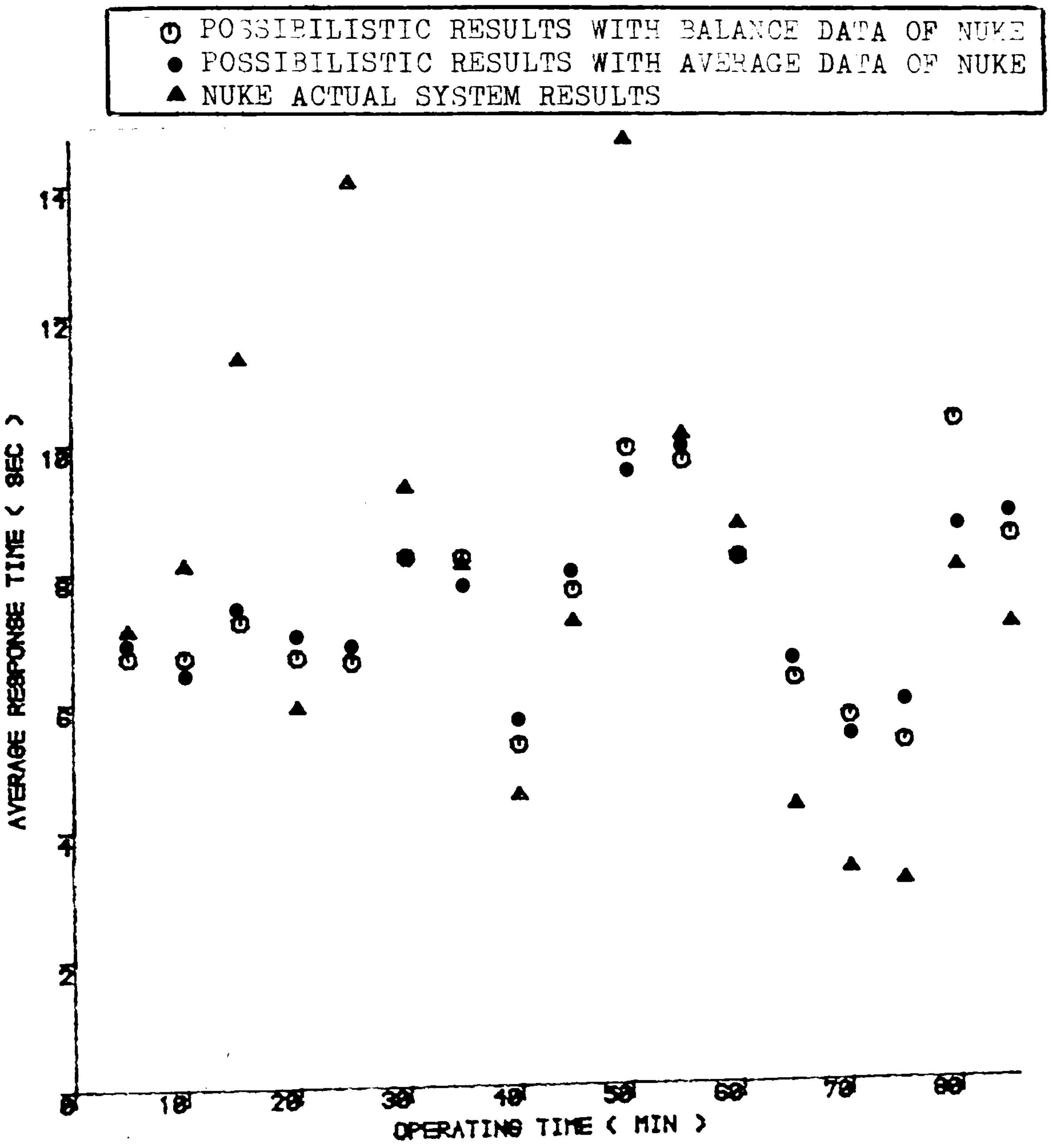


Figure 7.12: Simulation vs Real System Results: A Validation Case Study

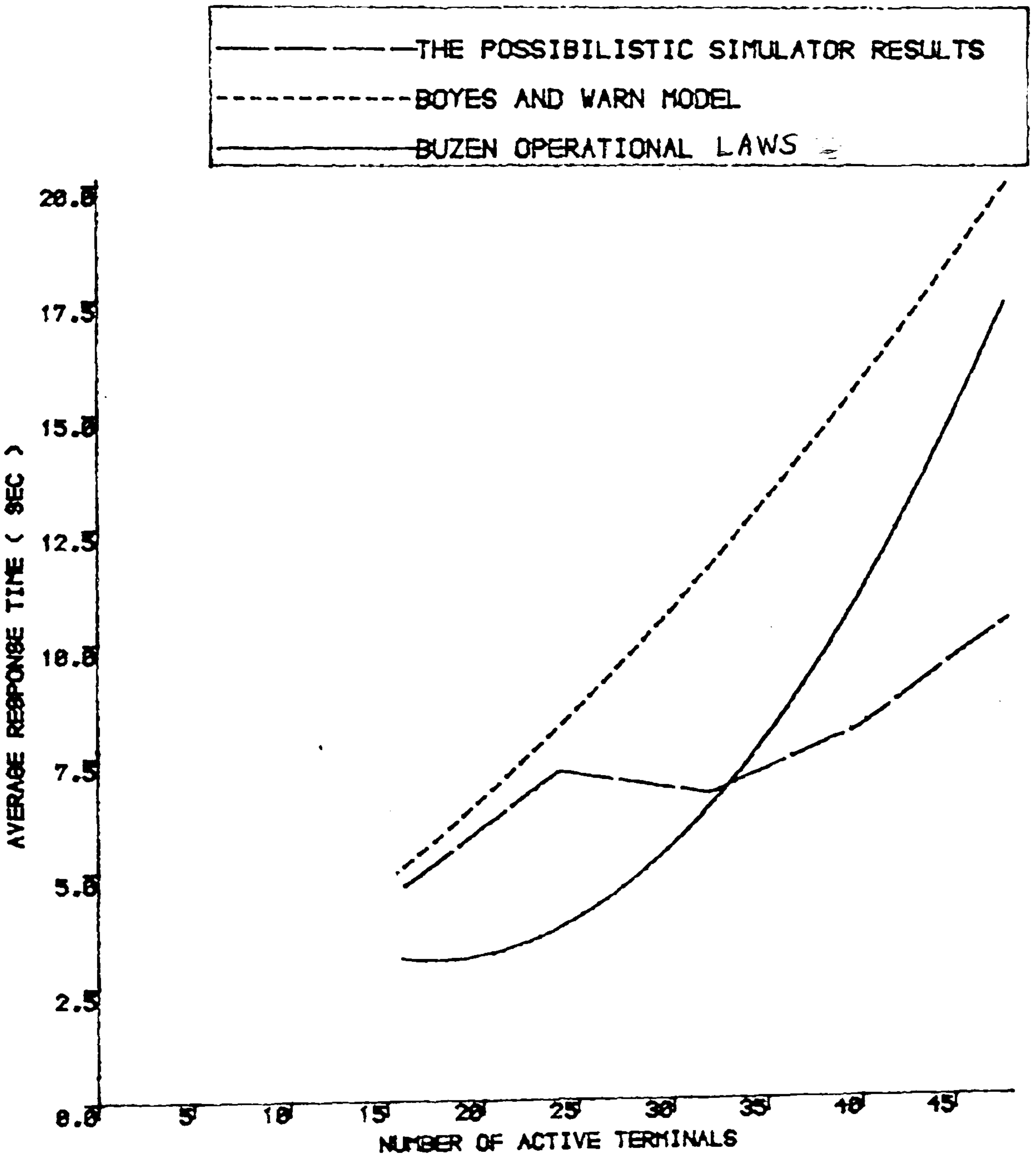


Figure 7.13: Validating The Simulator Results with some Theoretical Models

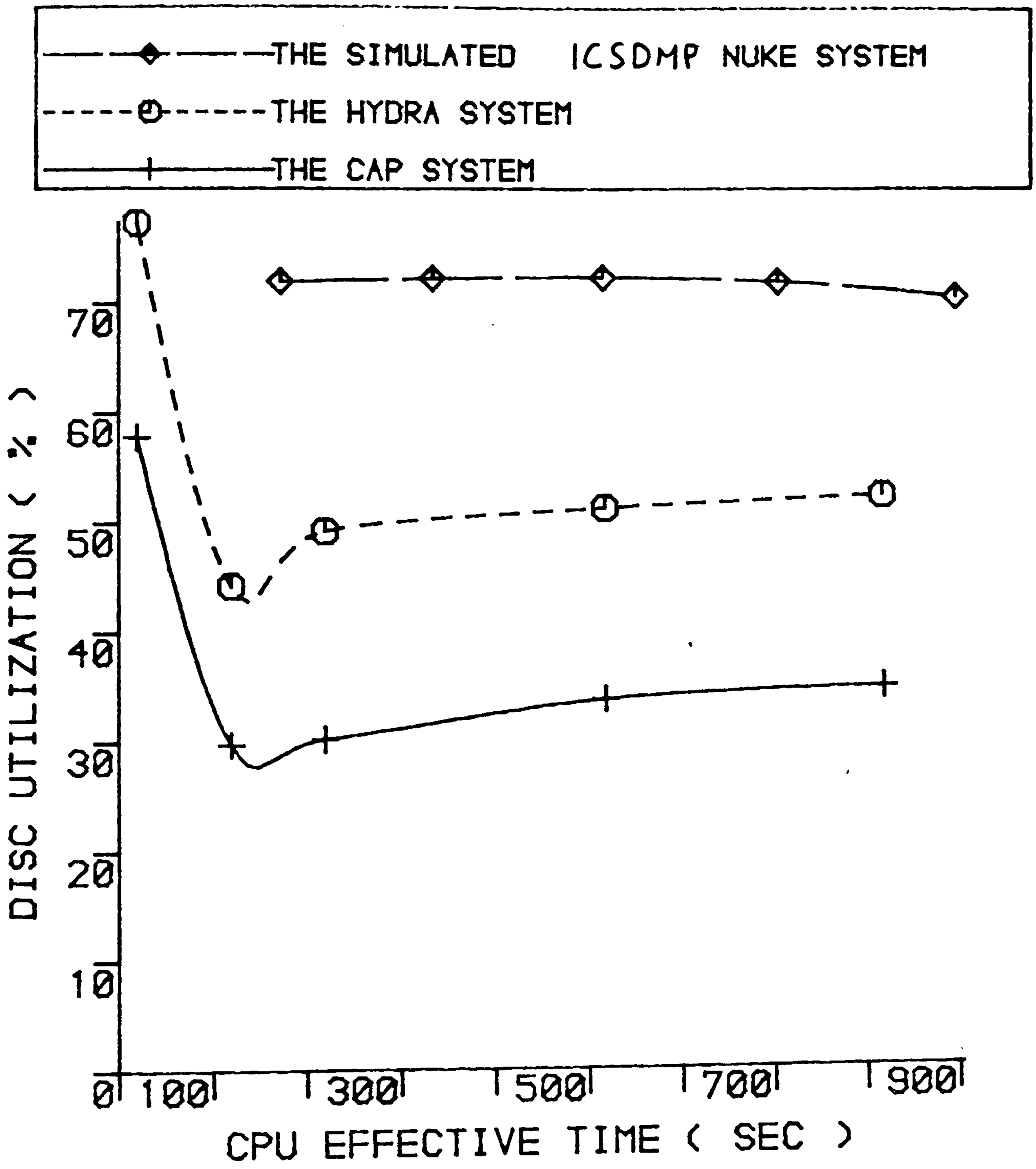


Figure 7.14: A Comparison of Protected Systems Disc Utilisation Performance

Chapter 8

SUMMARY, CONCLUSIONS AND FUTURE RESEARCH

8.1 Summary And Conclusions

In this thesis I have proposed an activity structures based methodology which can be used to design and construct highly constrained computer systems. The motivation for such design and construction methodology emerges from an entirely different paradigm than the conventional paradigm of designing computer system. Based on the general/meta systems design paradigm, we have been studying the problem of designing computer systems employing the analogy with the brain system. The brain is a maximally constrained system. Although such maximally constrained systems appear to be limited in their possible behaviour, they can function in a way that introduces a limitless plasticity in their behaviour without breaking their constraints. In a very recently published article Gains and Shaw (1986) expressed a view similar to that which motivated my work. Formalised conventional computer systems design techniques are thus often seen as threatening and unneeded.

In our opinion, the conventional design techniques do not captures adequately the dynamics of the human-factors involved in the computer interaction with its user environment. Computer systems must always be designed as coupling devices that coordinate planning and improve control and performance of both the user interaction and machine environments. By-in-large, our design and construction methodology attempts to capture the dynamics of the essential interrelated and mutually adjusting design factors: problem, technology, people, and function structures. The complex nature of interaction of these factors in computer system design is pictured in Figure 8.1

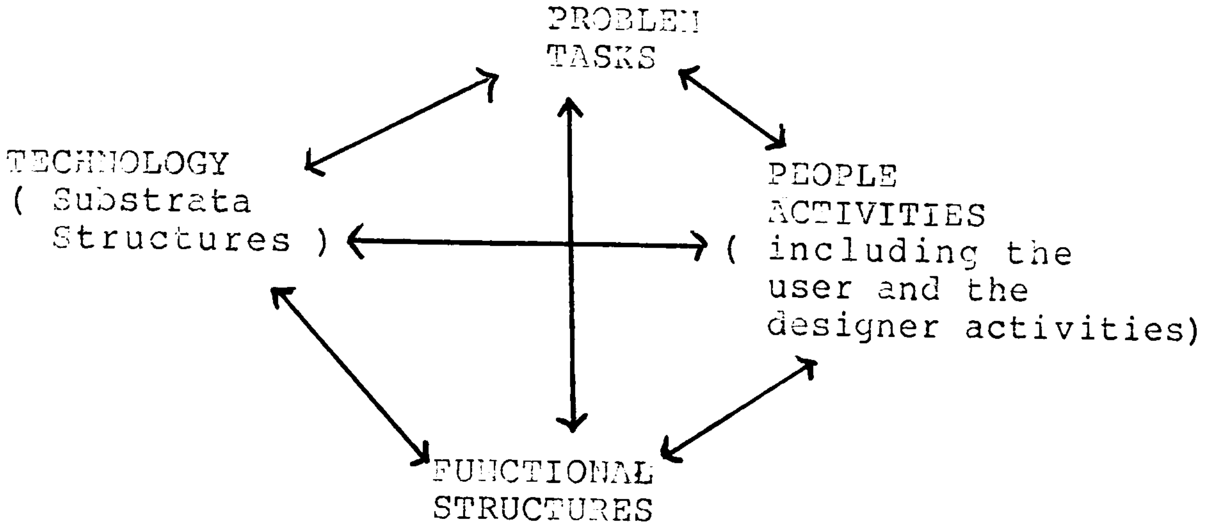

PROBLEM TASKS
TECHNOLOGY (Substrata Structures)
PEOPLE ACTIVITIES (including the user and the designer activities)
FUNCTIONAL STRUCTURES

Figure 8.1: Design factors of our computer design methodology.

When one of the interaction of the design factors is changed, the other factors should adjust to diminish the impact of that change. This simply means that the basic concepts of design such as relativity, uncertainty, ability to change, conversation and learning of both the user and machine environment, should be naturally encompassed by the design methodology. As argued in chapters 2 and 3, this were not present in the conventional computer systems design theories and techniques. The major implication of the conventional methodologies is that the complexity of a computer system is best controlled by designing it in a structural way so that the smallest possible design components can be represented. From the conventional design methodologies point of view, the design can be performed best by segregating the design task into design three directions; namely, ***software enginnering***, ***computer architecture***, and ***knowledge engineering***. Each segregation is responsible for producing certain design product that ***can be matched in some way to the others***. Accordingly, the software engineering techniques are used to produce the computer software, such as the operating system; the computer architecture techniques produce the computer organisation and hardware; and finally the knowledge engineering provide *"intelligent"* application programs that can operate on that computer system and communicate with its users.

The segregation we use is entirely different. For us segregation is the task of identifying the sets of functions which are mutually independent and complimentary. To this effect our design methodology segregates the following functions (see chapter 4):

1. the user requirements versus the designer construction steps,
2. the user environment activities versus the machine environment activities,
3. design functional duties (i.e. the functional structures) versus the the implementation media (i.e. the substrata),

Although these segregated functions often seem to be *conflicting* in the traditional design, our methodology use them to design and construct computer systems. For us segregation is the result of understanding and cooperation that identifies the essential parts of the design and the simple relations between its parts. In other words, the design can best be regarded as a web which express the essential parts which are delicately pieced together by simple links. Indeed, if we express a design as web of ideas, we can emphasise its properties in a natural and satisfying way (see the properties of our design as it were echoed by the postulates in chapter 4).

It is the fundamental premise of this thesis that the purpose of computers is to provide effective computational media within any given user environment. That solution can be best attained through the understanding and realisation of the following issues we are proposing in (see chapters 4 and 5):

1. *Performing the Essential Design Activities*: These activities aim at producing an activity structures based computer system. Starting by ***eliciting the design requirements*** from the user. Then by ***identifying the relevant design features***, the designer decide upon the way he/she selects the relevant and essential functional structures (representing the design essential elements and the required constraints). This step is followed, then, by ***selecting the matching substrata*** for realising the essential functional structures (i.e. forming the computer machine). In our case we selected two ***extensible substrata structures***: the ***coroutines*** and the ***descriptor-oriented architecture*** (i.e. in order to construct a possibilistic generator of computer designs). The selecting

of such extensible substrata for implementation, produce a possiblistic system that can be optimised easily. Beside selecting the matching substrata, we distinguished between activities generated by the user environment (essentially randomised with certain user learning capabilities) and activities generated by the computer environment (essentially randomised with certain machine learning capabilities).

2. *Exploring the Resulting Possibilistic Design*: In this stage the designer explore the computer system activities according to the given design requirements. The designer in the exploration stage perform the following operations (see chapter 4 and 7):

   (a) ***observe the behaviour*** of the possibilistic design according to the given requirements,

   (b) ***identify*** from the observed behaviour the ***admissible design data*** that produce ***interesting behaviour***,

   (c) ***tune*** the possibilistic design using the admissible data obtained,

   (d) ***perform*** 2a, 2b, and 2c until the system reaches the state of ***self-regulation or survival.***

3. *Adopting Cooperating Learning Mechanisms*: These mechanisms are not essential only for the actual representation of computer systems design, but also for enforcing the criteria of self-regulation or the dynamics during system-user interaction that was theoretically pointed out by Kupka (1974) and experimentally validated by Barber (1979) on computer systems. In our case, the user learning mechanism changes the average user intentions in certain directions that causes maximisation/minimisation of the user activities according to the machine responses. Similarly, machine learning mechanisms (i.e. our inferential structures) try to enhance performance with the increasing user productivity power until certain threshold is reached, by then their performance degrades, signaling to the user environment to that it should decrease its activities. When the user environment learns this, the computer machine environment enhances its performance again, and so the cycle repeats itself, causing the

enforcement of the self- regulating criteria which provide the fine tuning.

In chapter 5, we presented the abstract features of activity structures based designs (forming a design shell). The design of the shell represents a contribution within the area of distributed systems design. The distribution of the shell structures was mainly upon the control, communication, inferential and protection structures. Chapter 5 describes certain essential new design concepts, such as the ***communication distributed modules*** (using message-passing, loosely coupled modules), the ***communication participants***, the ***inferential/learning memory and processor mechanisms***. It also contains further less essential contributions. Particularly in the design of sharable information structures (based on the ***descriptor-oriented architecture***) and the enforcement of both the static protection (using the ***port-oriented mechanism***) and the enforcement of the ***protection dynamics***. Chapter 5 concludes with the issue of selecting the C programming language as a ***multiparadigm high-level language for implementation***.

In chapter 6, we performed the main steps of the designer activities leading to the realisation of the shell. This chapter contains the main algorithms that we used for the purpose of our implementation. Chapter 6 clearly demonstrates that the design structure, presented in chapters 4 and 5 can be successfully embeded in a workable implementation.

Chapter 7 illustrates the case of using the shell to simulate an existing highly constrained computer system, the NUKE. Exploring the activity structures Nuke based design, we used the performance modelling technique that we developed to identify certain admissible/nonadmissible design features that enhances/degrade the overall system performance. Briefly, this chapter study the ***validation*** issue of the activity structures Nucke based design.

Therefore, the main contributions of this thesis can be summarised as follows:

1. I presented a ***total*** system design framework which overcomes the conventional computer systems design inadequacies which are caused by the lack of the design and constraint picture of the whole design. The need for total design framework has been expressed recently by Roman et al (1984).

2. Using this framework, we developed an activity structures based method for producing

accurate, effective, highly constrained, realistic and practical computer systems (chapter 5). The method produces computer architectures that are machine-independent, display stability of performance within acceptable regions (self-regulating), virtual memory, multiprocessing, blackboard, decentralised functional structures, descriptor-oriented, message-passing. The other outcome that we do not end with the final product but also with a reusable design shell (basically new designs can be obtained by changing the design data from the knowledge structures). This shall captures the design experience during the use of other product.

3. The full scale implementation of a computer design tool codified using the C programming language. This implementation is runnable under the VAX 11/750 computer and can be easily ported to other suitable computer systems.

4. The development of a new framework called performoact modelling, which can be used for evaluating the effects of the design parameters on the criteria of self-regulation.

8.2 Future Research

Future research could continue in the following directions:

1. *The Empirical Analysis of Performance* :

   The simulation study of the activity structures based shell presented in chapter 6 and 7 could easily be extended to investigate the effect of different machine learning or inferential mechanisms, different memory architectures (i.e. different hierarchy), different device characteristics, different communication styles (i.e. asynchronous communication), different addressing mechanisms, additional protection mechanisms (e.g. static and dynamic access control protection upon files), different message structures, different memory and processor scheduling policies, different activity generators random distributions, additional constraints (e.g. reliability factors), and different user environment activities, etc.

2. *The Empirical Complexity Analysis of the Shell* :

   It will be interesting to make an extensive empirical study of the complexity of the C program of the activity structures based shell (e.g. program size, static and dynamic statement percentages, program style, etc). in comparison with a similar purpose software (e.g. the UNIX operating system version 5). The reason for studying the complexity is it has been shown that the implications of different programming techniques upon the resulting computer architecture efficiency is quite considerable (c.f. Tanenbaum 1978). In order to achieve this we need only to modify an existing C compiler or write our own software (for aiding dynamic and static analysis) and write a preprocessor and postprocessor (for aiding the dynamic analysis). To start such research the reader is referred to Robinson and Torsun (1976, 1977); Berry and Meekings (1984). However, empirical complexity analysis may also be done at a different abstraction level using the Halstead theory of software science (1977). For this purpose, we refer the reader to the following essential references which used of such approach within the area of operating systems and computer architecture quality evaluation (Pashtan 1985, Kavipurapu and Frailey 1979).

3. *Structural Synthesis of the Shell* :

   In our design the shell synthesis was manual and performed by the designer. However, structural synthesis represent the automatic decomposition of the given shell into a set of systems-components, which after connection behave similarly to a decomposed system (c.f. Pichler and Ottendorfer 1978). The approach may require to develop certain complexity measures that can be associated with the process of synthesis, such as the complexity measures developed by the reconstructibility analysis (Cavallo and Klir 1981), or the heuristics synthesis techniques (Abd-Alla and Karlgaard 1974). The process of synthesis should also be accompanied by an approach to performing the automatic synthesis of user intentions. For this purpose, we refer the reader to Haring et al (1978).

4. *Developing a Theoretical Approach* :

   This is an interesting future development that we aim to achieve within our future research work: The development of theoretical modelling techniques for the design and evaluation of activity structures based computer systems. This should produce quite original research, since there is no existing theoretical technique for the design and evaluation of distributed computer systems (such as activity structures based systems) (c.f. Klienrock 1985). For this purpose, we believe that the theory of dynamic systems (Jacak and Sierocki 1985), the formal theory of modelling (Zeigler 1972), and the theory of system behaviour description (Gaines 1977, Gaines 1976, Witten 1977, Riddle 1979) can be used along with the *system connection analysis* (Yuval 1980) to produce an effective design and analysis activity structures based computer models. However, simpler modelling technique can produced by using the concurrent system design (Hartmann 1983, Clements 1977) along with the theory of relational products (Bandler and Kohout 1980).

5. *Stability Analysis of the Shell Self-Regulation* :

   In this respect, we propose to extend our performoact modelling framework to a more formal analysis by expressing the criteria of self-regulation using the stability analysis (Perlis and Ignizio 1980) or the *adaptivity theory* of Gaines (1972, 1974) or the learning metaphors of Carrol and Mack (1985). We believe this is quite possible way, since it was previously used to express the criteria of self-tuning of certain conventional computer systems (c.f. Von Mayrhauser 1979).

6. *Distributed Descriptor-Oriented Architectures* :

   In spite of the amount of work recently devoted to distributed systems, distributed applications are relatively rare (c.f. Ellis 1985). One hypothesis that explains this scarcity is lack of experience with algorithm design techniques that are tailored to a design environment in which out-of-date information is the rule. Since the design of data structures is an important aspect of traditional algorithm design, we feel that it is an important to consider the problem of distributed data-structures (or data types).

Specially, the effects of different organisations of the distributed descriptor-oriented architecture. As a starting point I believe it would be to extend the techniques of Mohamad and Cavouras (1984), Booth and Wiecek (1980), Giloi and Berg (1977).

7. *Further Design Features Taken from the Brain Analogy* :

   For this extension, I propose to study the possibility of adding/changing parts of the shell structures in order to provide new Brain-like features. Examples of such new features that could enhance the criteria of shell 'reliable' operations are the representation of the *Brain casual structures* (Rosen 1986) or the *Brain by-pass* mechanism (Jugeli 1980).

8. *The Systematic Analysis of Dialog Shell Design* :

   Since the interaction level of the shell with its human users is of interest for the design of sixth generation computer systems (Gaines and Shaw 1986), then the replacement of our user environment by a shell that systematically captures the foundation for dialog engineering, will be of considerable advantage. For the purpose of this replacement we recommand the adoption of the theoretical models oh human-machine communication of Oberquelle et al (1983).

To conclude, we believe that our presented design tool supports a methodological approach for designing maximally constrained, high-performance computer systems that provide elegant solutions to several problems which previous fragmented attempts have handled only in an ad hoc fashion.

REFERENCES

ABD-ALLA, A. M. and KARIGANRD, D. C. 1974, Heuristic synthesis of microprogrammed computer architecture, IEEE Trans. on Computers, Vol. C-23, No. 8, pp. 802-807

ALEKSANDER, I. 1982, Mind, brain, structure and function, kybernetes, Vol. 11., pp. 249-253

ALLEN, J.R. and KENNEDY, K. 1985 A parellel programming environment, IEEE Software, Vol. 2, No. 4, pp.175-189

ALLISON, D. R. 1977, A design philosophy for microcomputer architectures, IEEE Computer, Vol. 10, No. 2, pp. 35-41

AMARI, S. 1983, Field theory of self-organization neural nets, IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-13, No. 5, pp. 741-478

AMBLER, A. L. and HOCH, C. G., 1979 A study of protection in programming language, Sigplan Notices, Vol. 12, No. 3, pp. 25-40.

ANDERSON, H. A. and SARGENT, R.G. 1974, Investigation into scheduling for an inneractive computing system, IBM J. of Reasearch and development, March issue

ANDERSON, J. , KOHOUT, L.J. et al 1985, A knowledge-based clinical decision support system using new techniques: CLINAID, Proc. AAMSI Congress

ANDERSON, B. 1980 Type syntax in language "C". Sigplan Notices, Vol. 15, No. 3, pp. 21-27

ANDREAE, J. H. and CLEARY, J. G. 1976, A new mechanism for a brain, Int. J. Man-Machine Studies, Vol. 8, pp. 89-119

ARBIB, M. A. 1975, From automatic theory to brain theory, Int. J. Man-Machine Studies, Vol. 7, pp. 279-295.

ANTONELLI, S. and IAZEOLLA, G. 1983, Multiple access control policies in capability-based protection systems, J. of Information Processing, Vol. 6, No. 1, pp. 16-22

BACKUS, J. 1981 Functional level programs as mathematical objects, Proc. of the conf. on functional programming languages and computer architecture, October, USA.

BACKUS, J. 1985 Function-level computing, In Next-Generation Computer (ed. Torrero), IEEE press, Spectrum Series.

BAER, J. 1974, Models for the design, simulation and performance of distributed-function architecture, IEEE Computer, Vol. 7, No. 3, pp. 25-30

BAILES, P. A. 1985, A low-cost implementation of coroutines for C, Software-Practice and Experience, Vol. 15, pp. 379-395

BALZER, R., 1983, Software technology in the 1990's: Using a new paradigm, IEEE Computer, Vol. 16, No. 11, pp. 39-45.

BANDLER, W. 1978 Some Espmathematical Uses of Category Theory, In Klir's Applied General Systems Research, Plenum Press.

BANDLER, W. and KOHOUT, L. J. 1979, Activity structures and their protection in: improving the human condition: quality and stability in social systems, Proc. Int. Meeting of SGSR, Louisville, pp. 240-246.

BANDLER, W. and KOHOUT, L. J. 1980, Fuzzy relational products as a tool for analysis and synyhesis of the behaviour of complex natural and artificial systems, Ed. Wang, P.P. and Chang, S.K., Plenum Press, New York and London

BARBACCI, M. R. 1975, A comparison of register transfer language for describing computers and digital systems, IEEE Trans. on Computers, Vol. C-24, No. 2.

BARBACCI, M. R. 1977, An architecture facility, ISP descriptions, simulations, data collection, Proc. of AFIPS, Vol. 46, pp. 161-173.

BARBACCI, M. R. and UEHARA, T. 1985, Computer hardware description languages: The bridge between software and hardware, IEEE Computer, Vol. 18, No. 2, pp. 6-8.

BARBER, R. E. 1979, Response time, operator productivity and job satisfaction, New York University, Graduate School of Business Administration, Ph.D. Thesis.

BARON, R. et al 1985 Mach-1: An operating environment for large scale multiprocessor application, IEEE Software, Vol 2, No 4, 1985

BARTER, C. J. 1983, Communications policy for composite processes, The Australian Computer J., Vol. 15, No. 1, pp. 9-16

BASKETT, F. and et. al., 1975, Open, closed, and mixed networks with different classes of customers, ACM J., Vol. 22, No. 2, pp. 248-260.

BAYLIN, E. 1984, Functional modeling of the business organization, Cybernetics and systems : An Int. J., Vol. 15, pp. 259-291

BAYLIN, E. N. 1986, Identifying system functions, Int. J. General Systems, Vol 12, pp. 7-38

BEAUMONT, W. P. and MACASKILL, J. L. C. 1975 Studies in the simulation of computers, The Australian computer J., Vol. 7, No. 1, pp. 7-11

BELL, C. G. and NEWELL, A. 1971, Computer structures: Readings and Examples, McGraw-Hill Book Company, New York

BEER, S. 1972, Brain of the firm, ltarmondsworth: Penguin

BERG, H., A computer architecture based on ordered sets as a primitive data entities, Ph.D. Thesis, Minnesota University.

BERRY, R. E. and MEEKINGS, B. A. E. 1984, A book on C, Macmillan pub. Co.

BERNSTEJN, N. A. 1967, The Co-ordination and regulation of movements, Oxford:Pergamon Press

BERNSTEIN, A. J. and SHARP, J.C. 1971, A policy-driven scheduler for a time-sharing system, CACM, Vol. 14, No. 2, pp. 74-78

BHANDIWAD, R. A. and WILLIAMS, A. C. 1974, Queueing network models of computer systems, Proc. of the third Texas Conf. on Computing Systems.

BIC, L. 1982, A protection model and its implementation in a dataflow system, CACM, Vol. 25, No. 9, pp. 650-658.

BISHOP, J. M. and BARRON, D. W. 1981, Principles of descriptors, The Computer J., Vol. 24, No. 3, pp. 210-221.

BLANK, J. and KRIJGER, M. J. 1983, Software engineering methods and techniques, John Wiley & Sons Pub. Co.

BLUNDEN, G. P., and KRASNOW, H. S., 1967, The process concept as a basis for simulation modelling, Simulation, Vol. 9, No. 2, , pp. 89-93.

BOCHMANN, G. V. 1978, Combining assertions and states for the validation of process communication, Constructing Quality Software, P.G. Hibbard/S.A Schuman (Eds.) IFIP, North-Holland Pub. Co.

BOHM, A. et al. 1985 Hardware and software enhancement of the Manchester Data Flow machine, Digest, Copacon Spring Feb 85.

BOOTH, T. L. and WIECEK, C. A. 1980, Performance abstract data types as atool in software performance analysis and design, IEEE Trans. on Software Engineering, Vol. SE-6, No. 2, pp. 138-151

BOSE, P. and DAVIDSON, E. S., 1984, Design of instruction set architechtures for support of high-level languages, ACM SIGARCH Newsletter, Vol. 12, Issue 3, pp. 198-206.

BOULTON, P. I. P. and GOGUEN, J. R., 1979, A machine description language, The Computer J., Vol.22, No.2, pp. 132-135.

BOYSE, J. W. and WARN, R. D. 1975, A straightforward model for computer performance prediction, ACM computer Surv., Vol. 7, No. 2, pp. 73-93

BRAD, Y., 1971, Performance criteria and measurements for a time sharing system, IBM Sys. J., Vol. 10, pp. 193-219.

BRITTON, K. H. et. al., 1982 A procedure for designing abstract interfaces for device interface modules, Proc. Fifth Int., Conf. on Software Engineering, IEEE Computer Society Order, No. 332, pp. 195-204.

BROCKMEYER, E. el al. 1948 The life and work of A.K. Earlang, Trans. of Danash Academic Tech. Sci. (In English), Vol. 2.

BROWNE, J. C. 1984, Understanding execution behaviour of software systems, IEEE Computer, Vol. 17, No. 7, pp. 83-87

BRUNDAGE, R. E. 1974, A study of process bahaviour in virtual computer systems, Ph.D. Thesis, university of Virginia

BURR, W. E. et. al. 1977, Overview of the military computer family architecture selection, AFIPS, Vol. 3, pp. 131-137.

BURROUGHS, C. 1961, The descriptor-a definition of the B5000 information processing system. Detroit: The Burroughs Corporation, Bull. no. 5000-20002-p

BUZEN, J. P., 1971, Queueing network models of multiprogramming, Ph.D. Thesis, Div. Eng. and Applied Physics, Harvard University, Cambridge.

BUZEN, J. P., 1973, Computational algorithms for closed queueing networks with exponential servers, CACM, Vol. 16, No. 9, pp. 527-531.

BUZEN, J. P., 1976, Fundamental operational laws of computer system performance, Acta-informatica, Vol. 7, pp. 167-182.

BUZEN, J. P., 1976a, Operational analysis: the key to the new generation of performance prediction tools, Proc. IEEE COMPCON, pp. 166-171.

BUZEN, J. P., 1978, Operational analysis: an alternative to stochastic modelling, Proc. Int. Conf. Performance of Computer Installations pp. 175-194.

BUZEN, J. P., 1979, The predictable problem, Computing Surveys, Vol. 11, No. 1, pp. 70-72.

BUZEN, J. P. and DENNING, P. J., 1980, Operational treatment of queue distributions and mean-value analysis, Computer Performance, Vol. 1, No. 1, pp. 6-15, IPC.

CANTRELL, H. N. and ELLISON, A. 1968, Multiprogramming system performance and analysis, AFIPS, pp. 213-221.

CAPPER, L. 1986, A philosophy for teaching of computer science information technology, The Computer J., Vol. 29, No. 1, pp. 83-89.

CARLSSON, C. 1979, Cybernetic modelling through hierarchical multilevel systems and adaptive control, Kybernetes, Vol. 8, pp. 91-103

CARROL, J. M. and THOMAS, J. C. 1982, Metaphor and cognitive representation of computing systems, IEEE Trans. on Systems, Man, and Cybernetics, Vol. 12, pp. 107-116

CARROLL, J. M. and ROBERT, L. M. 1985, Metaphor, computing systems, and active learning, Int. J. Man-Machine Studies, Vol. 22, pp. 39-57

CAVALLO, R. E. and GEORGE, J. K. 1981, Reconstructability analysis: Overview and Bibliography, Int. J. General Systems, Vol. 7, pp. 1-6.

CAVOURAS, J. C. and DAVIS, R. H., 1981, Simulation tools in computer system design methodologies, Comput. J., Vol. 24, No. 1, pp. 25-28.

CHANDY, K. M. and SAUER, C. H., 1978, Approximate methods for analysis queueing network models of computer systems, ACM Computing Surveys, Vol. 10, No. 3, pp. 281-317.

CHANDY, K. M. and MISRA, J. 1979, Distributed simulation: A case study in the design verification of distributed programs, IEEE Trans. on Software Engineering, SE-5, No. 5

CHECKLAND, P. B. 1975, The development of systems thinking by systems practice-A methodology from action research programme, Progress Cybernetics and Systems Research, Vol. 11, Ed. by R. Trapple and F. de P. Hawika, Hemisphere, Washington DC, pp. 278-283

CHERITON, D. R. 1982, The Thoth system: multi-process structuring and portability, Elsevier Pub. Co.

CHU, Y. 1965, An Algol-like computer design language, CACM, pp. 607-615.

CHU, Y. 1977, Direct-execution architecture, Information Processing 77, IFIP North-Holland Pub. Co.

CHU, Y. 1981, High-level computer architecture, Computer, Vol. 14, No. 7, pp. 7-8.

CLAPSON, P. J. 1977, Toward performance science: a comparative analysis of computing systems, The Computer J., Vol. 12, No. 4, pp. 308-315

CLEMENTS, D. P. 1977, Fuzzy ratings for computer security evaluations, Ph.D. Diss., University of California, Berkeley

COHEN, E. and DAVID, J. 1975, Protection in the hydra operating system, Proc. of the Fifth Symp. on Operating Systems Principles, University of Texas at Austin, Nov. 19-21, (ACM & Operating Systems Review, 9:5), pp. 141-60

CONRAD, M. 1985, On design principles for a molecular computer, CACM, Vol. 28, No. 5, pp. 464-480

CONWAY, M. E. 1963, Design of a separable transition-diagram-compiler, CACM, Vol. 6, No. 7, pp. 396-408

COOK, D. J. 1978, The evaluation of a protection system, Ph.D. Thesis, University of Cambridge.

Goguen, J.a. 1986 Reusing and interconnecting software components, IEEE Computer, vo. 21, No. 1, pp. 16-28

GRAIG, C. E. and HARRIS, R. C. 1973, Total productivity measurement at the firm level, Sloan Management Review, Vol. 14, No. 3

CRAIG, F. D. 1986, The ariadne-1 blackboard system, The Computer J., Vol. 29, No. 3, pp. 235-240

CROWLEY, C. 1981, The design and implementation of a new Unix kernel, Proc. AFIPS Conf., Vol. 50, pp. 265-171

CURNOW, H. J. and WICHMANN, B. A. 1976, A synthetic benchmark, The Computer J., Vol. 19, pp. 43-49

DAVIES, N. R., 1976, A modular interactive system for discrete event simulation modelling, in Proc. 9th Hawaii Int. Conf. in System Sciences, Western Periodical, pp. 296-299.

DAVIS, R. and SHROBE, H. 1983, Representing structure and behaviour of digital hardware, IEEE Computer, Vol. 16, No. 10, pp. 75-82

DEC, 1977, VAX-11 Software handbook, Digital Equipment Corporation Press

DEC, 1979, VAX-11 Architecture handbook, Digital Equipment Corporation Press

DECARVALHO. R. S. and CROOKES, J. G., 1976, Cellular simulation, Oper. Res., Vol. 27, No. 1, pp. 31-40.

DENNING, P. J. 1968, The working set model for program behaviour, CACM, Vol. 11, pp. 323-333

DENNING, P. J. and BUZEN, J. P., 1977, An operational overview of queueing networks, in Infotech State of Art Repore on Performance Modelling and Prediction, U.kk, pp. 75-108.

DENNING, P. J. and BUZEN, J. P., 1978, The operational analysis of queueing network models, Computing Surveys, Nol. 10, No. 3, pp. 225-261.

DENNIS, J. B. and VANHORN, E. C. 1966, Programming semantics for multiprogrammed computations, CACM, Vol. 9, No. 3, pp. 143-155.

DENNIS, T. 1980, A capability architecture, Ph.D. Thesis Purdue University.

DIETHELM, M. A. 1977, An empirical evaluation of analytical models for computer system performance prediction, Computer Performance, K.M. Chandy and M. Reiser (Eds.), North Holland Pub. Co.

DIETTERICH, T. G. and BUCHANAN, B. G. 1983, The role of experimentation in theory formation, Proc. Int. Machine Learning Workshop, University of Illinois, Department of Computer Science, Urbana, ILL

DIETZ, W. and SZEWERENKO, L. 1979, Architectural efficiency measures: An overview of three studies, IEEE Computer, Vol. 12, No. 4, pp. 26-33.

DIJKSTRA, E. W. 1968, The structure of the "T.H.E." -multiprogramming system, CACM, Vol. 11, No. 5

DJORDVEVIC, J. 1985, A PMS level notation for the description and simulation of digital systems, The Computer J., Vol. 28, No. 4, pp. 357-365.

DORAN, R. W. 1979, Computer architecture: A structured approach, Academic Press Inc.

DOUGLASS, R.J 1985 A qualitative assessment of parallelism in expert systems, IEEE Software, Vol. 12, No. 3, pp. 70-80

DOWNS, D. D. 1984, Operating systems key security with basic software mechanisms, Electronics, Vol. 57, No. 5, pp. 122-130.

DUFF, M.J.B. 1985 The CLIP parallel processor, Computer Bulletin, Vol. 1, Part 4, pp. 26-27

DULEY, J. R. et. al. 1969, A digital system design language (DDL), IEEE Trans. on Computers, C-17, pp. 850-861.

DUNSMUIR, M. R. M. and DAVIES, G. J. 1985, Programming the UNIX system, Macmilian Pub. Co.

ELLIS, C. S. 1985, Distributed data structures: A case study, IEEE Trans. on Computers, Vol. C-34, No. 12, pp. 1178-1190

ENGLAND, D. M. 1974, Capability concept mechanisms and structure in system 250, International Workshop on Protection in Operating Systems IRIA/LABORIA Rocquencourt, France, Aug. 13-14, pp. 63-82.

EVANS, D.J. et al 1981 A guide to using the Neptune parallel processing system, Loughbourough University of Technology, Computer Studies Department, Technical Report.

FABRY, R. S. 1967, A user's view of capability, ICR Quarterly Report No. 15, Institute for Computer Research, Universuty of Chicago.

FABRY, R. S. 1974, Capability-based addressing, CACM, Vol. 17, NO. 7, pp. 403-412

FAGIN, R. 1978 On an authorisation mechanism, ACM Trans. on Database Systems, Vol. 3, No. 3, pp. 310-319

FARRARI, D., 1978, Computer systems performance evaluation, Prentice-Hall, Pub. Co.

FRARRI, D. 1986 Considerations on the Insularity of Performance Evaluation, IEEE Trans. on Software Engineering, Vol. SE-12, No. 6, pp.678-683

FEIERTAG, R. J. and NEUMANN, P. G. 1979, The foundations of a provably secure operating system (PSOS), AFIPS Conf. Proc. 1979 National Computer Conf., Vol. 48, pp. 329-334

FERNANDES, S. T. 1982, Computer architecture: description and interpretation, Imperial college, London University, Ph. D. Thesis.

FEUSTED, E. A. 1973, On the advantages of tagged architecture, IEEE Trans. on Computers, Vol. C-22, No. 7, pp. 644-656

FITZPATRICK, D. et. al. 1981, A RISCy approach to VLSI, VLSI design, Vol. 2, No. 4, pp. 14-20.

FLOYD, C. 1981, A process-orinted approach to software development, ICS, The Int. Computing Symp. Proc. of the 6th ACM European regional conference, pp. 285-294.

FLYNN, M. J. 1972, Some computer organizations and their effectiveness, IEEE Trans. on Computers, Vol. 21, No. 9, pp. 948-960.

FLYNN, M. 1980, Directions and issues in architecture and language, IEEE Computer J., Vol. 13, No. 10, pp. 5-22.

FOX, G.C. 1985 Using the Caltech hypercube, California Institute of Technology, Booth Computing Center,158-79 pasadena, CA 91125.

FRANKOWSKI, E. N. and FRANTA, W. R., 1980, A process oriented simulation model specification and documentation language, Software-practice and Experience, Vol. 10, No. 9, pp. 721-742.

FRIDRICH, M. and OLDER, W. 1985 Helix: The architecture of the XMS distributed file system, IEEE Software, Vol. 2, No. 3, pp. 21-29

FRIEDLAND, P. and KEDES, L. H. 1985, Discovering the secrets of DNA, CACM, Vol. 28, No. 11, pp. 1164-1186

FULLER, S. F. et. al. 1977, Evaluation of computer architectures via test programs, AFIPS, Vol. 3.

GAMMAGE, N. and CASEY, L. 1985 XMS: A rendezvous-based distributed system software architecture, IEEE Software, Vol. 2, No. 3, pp. 9-19

GAINES, B. R. and ANDREAE, J. H. 1966, A learning machine in the context of the general control problem, Proceedings of the 3rd Congress of the Int. Federation for Automatic Control.

GAINES, B. R. 1972, The human adaptive controller, Ph.D. Thesis, Cambridge University, Cambridge

GAINES, B. R. 1972a, Axioms for adaptive behaviour, London: Butterworths, Int. J. of Man-Machine Studies, Vol. 4, No. 2, pp. 169-199

GAINES, B. R., 1973, Computer technology and its utilization today and tomorrow, National Engineering Laboratory Conference, Glasgow 27-29 March.

GAINES, B. R., et. al. 1974, Design objectives for a descriptor-organised minicomputer, The European Computing Congress, Brunel University, 13-17 May.

GAINES, B. R. 1974, Training, stability and control, Instructional Science, Vol. 3, No. 2, pp. 151-176

GAINES, B. R. and KOHOUT, L. J. 1975, The logic of automata, Int. J. of General Systems, Vol. 2, No. 4, pp. 191-208

GAINES, B. R. and FACEY, P. V. 1975, Minicomputers in security dealing, IEEE Computer, Vol. 9, No. 9, pp. 6-15.

GAINES, B. R. 1976, System identification, approximation and complexity, Int. J. Gen. System, Vol. 3.

GAINES, B. R. 1977, System identification, approximation and complexity, Int. J. General Systems, Vol. 3, pp. 145-174

GAINES, B. R. 1978, Man-computer communication- what next ?, Int. J. Man-machine Studies, Vol. 10, pp. 225-232

GAINES, B. R. and SHAW, M. L. G. 1984, The art of computer conversation, Prentic Hall International

GAINES, B. R. and SHAW, M. L. G. 1986, From timesharing to the sixth generation: the development of human-computer interaction. Part I, Int. J. Man-Machine Studies, Vol. 24, pp. 1-27

GAINES, B. R. and SHAW, M. L. G. 1986a, Foundations of dialog engineering: the development of human-computer interaction. Part II, Int. J. Man-Machine Studies, Vol. 24, pp. 101-123

GANE, C. and SARSONS, T. 1979, Structured systems analysis: tools & techniques, Mcdonnell douglas Pub. Co.

GEHRINGER, E. F. 1979, Functionality anf performance in capability-based operating systems, Ph. D. Diss., Purdue University.

GILOI, W. K. and BERG, H. 1977, Introducing the concept of data structure architectures, Proc. Int. Conf. on Parallel Processing, USA, pp. 44-51 (IEEE CATALOG No. 776H-1253-4C)

GILOI, W.K. and GUETH, R. 1982 Concepts and realisation of a high-performance data type architectures, Int. J. of Computer and Information Sciences, Vol. 11, No. 1,pp. 25-54

GOLD, S. E. B. and SOESAN, J. 1976, Applied poductivity analysis for industry, Pergamon Int. Library, Oxford

GOMMA, H., 1976, A Modelling approach to the evaluation of computer system performance, Ph.D. Thesis, Imperial College of Science and Technology.

GORDON, C. et. al. 1971, Computer structures: Reading and examples, New York, McGraw-Hill Pub. Co.

GORDON, W. J. and NEWALL, G. F., 1967, Closed queueing systems with exponential servers, Oper. Res., Vol. 15, pp. 254-265.

GOTTLIEB, J.R. et al 1985 The NYU Ultra computer designing an MIMD shared memory parallel computer, IEEE Trans. Computer, Vol. C-32, No.2, pp.175-189

GRAHAM, B. W. 1969, Dynamic protection structures, AFIPS, 35, Fall Joint Computer Conf., pp. 27-38

GRAHAM, B. 1984, Hints for computer system design, IEEE Software,Vol. 1, No. 1, pp. 11-28.

GRENANDER, V. and TSAO, R., 1972, Quantative methods for evaluating computer system performance: A review and proposals, Statistical Computer Performance Evaluation, W. Freiberger (Ed.), Academic Press, pp. 3-24.

GRIFFITHS, P.P. and WADE, B.W. 1976 An authorisation mechanism for relational database system, ACM Trans. on Database Systems, Vol. 1, No. 3, pp.242-255

GROSSBERG, S. 1982, Studies of the mind and brain; neural principles of learning, perception, development cognition and motor control, Reidel, Hingham, Mass

HABERMANN, A. N., FLON, L. and COOPRIDER, L. W. 1976, Modularization and hierarchy in a family of operating systems, CACM, Vol. 19, No. 5, pp. 266-272

HAC, A. 1982, Computer system simulation in Pascal, Software-Practice and Experience, Vol. 12, pp. 777-784.

HAILPERN, B. 1986, Multiparadigm languages, IEEE Software, Vol. 3, No. 1, pp. 6-8

HAJEK, P. and HAVRANEK, T. 1978 The GUHA Method: Its aims and techniques, Int. J. of Man-Machine Studies, Vol. 10, No. 1, pp. 3-22

HALSTEAD, M. H. 1977, Elements of software science, Elsevier North-Holland Pub. Co.

HANSEN, B. 1973, Operating system principles, Prentice-Hall Pub. Co.

HARING, G. et. al. 1978, The use of a synthetic job stream in performance evaluation, The Computer J., Vol. 22, No. 3, pp. 209-219.

HARTMANN, C. L. 1983, Concurrent systems design with access graphs, IEEE Second Annual Phoenix Conference, March 14-16, Phoenix, Arizona

HAUSEN, H. L. and MULLERBURG, M. 1981, Architecture of software systems in the context of software enginering environments, The Int. Computing Symp. Systems Architecture, Proc. of the Sixth ACM European Reginonal Conference

HAYES, I. J. 1983, Computer architecture: The hardware-software interface, Ph.D. Thesis, University of New South Wales, Sydney, N.S.W., Australia.

HEINRICH, F. R. and KAUFMAN, D. J. 1976, A centralized approach to computer network scurity, AFIPS National Computer Conf.

HELLERMAN, H. and CONROY, T. F. 1975, Computer system performance, McGraw-Hill Pub. Co.

HELLMAN, M. E. 1978, Security in communication networks, National Computers Conf. AFIPS, pp. 1131-1134

HERBERT, A. J. 1978, A new potection architecture for the cambridge capability computer, ACM Operating System Review, Vol. 12, No. 1, pp. 24-28

HILL, F. J. 1975, Updating AHPL, Proc. of the Int. Symp. on Hardware Description Languages and Their Application, New York, pp. 22-29.

HILLIS, W.D. 1985 The connection machine, The MIT Press.

HOARE, C. A. R. Oct. 1974, Monitors: Operating system structuring concept, CACM Vol. 17, No. 10, pp.95-125

HOARE, C.A.R. 1978 Communicating Sequential Processes, CACM Vol. 21, No. 8,pp.666-677

HOEVEL, L. W. and FLYNN, M. J. 1979, A theory of interpretive architectures: Some notes on design and a Fortran case study, Computer Systems Laboratory, Tech. Report No. 171, Stanford University, Stanford, CA 91305

HOLBAEK-HANSSEN, E., et. al.,1977, System description and the delta language, Rep. 4, Norwegian Computing Center, Oslo.

HSAIO, D. K. 1968, A file system for a problem solving facility, Ph.D. Thesis, University of Pennsylvania, Philadelphia

HUGHES, P. H. and MOE, G., 1973, A structural approach to computer performance analysis, in PROC. AFIPS National Computer Conf., Vol. 42, AFIPS Press, Montvale, N.J., pp. 109-119.

HUGHES, H. D. 1981, A highly parameterized tool for studying performance of computer systems, ACM Perf. Eva, Rev., Vol. 10, No. 2, pp. 48-65

HUTCHINSON, G. K., 1975, Introduction to the use of activity cycles as a basis for system's decomposition and simulation, Simuletter, Vol. 7, No. 1, pp. 15-20.

IEEE SOFTWARE 1985 National Supper Computer Research Centers, IEEE Software, Vol 2, No. 6, pp. 55-68

ILIFFE, J. K. 1961, The use of the genie system in numerical computation, Annual Review in Automatic Programming, Vol. 2, New York: Pergamon Press, pp. 1-28.

INMOS 1984 Occam programming manual, Printice-hall Pub. Co.

JACAK, W. and SIEROCKI, I. 1985, Level of structural decomposition of dynamical systems, Int. J. General Systems, Vol. 10, pp. 177-186

JACKSON, J. R., 1963, Jobshop like queueing systems, Manage. Sci., Vol. 10, pp. 131-142.

JAMSA, K. A. 1984, Object oriented design vs structured design – A student's perspective, ACM SIGSOFT Software Engineering Notes, Vol. 9, No. 1, pp. 43-49.

JONES, A. 1977, The narrowing gap between languages systems and operating systems information processing 77, IFIP North-Holland Pub. Co.

JONES, A. K. et. al. 1979, StarOS, a multiprocessor operating system for the support of task forces, in Proc. 7th Symp. Operating Systems Principles (ACM).

JONES, A. K. and SCHWARZ, P. 1980, Experience using multiprocessor systems- A status report, Computing Surveys, Vol. 12, No. 2, pp. 121-165

JONES, B. 1983 General Systems Theory and Algorithm Theory, Int. J. General Systems, Vol. 9, pp. 157-160

JONES, G. 1985 Programming in "occam", Oxford University Computing Laboratory, Technical Monograph No. 41.

JUGELI, E. P. 1980, The process of dynamic interference in memory organization, kybernetes, Vol. 9, pp. 33-36

KAMBAYASHI, Y. 1981 Centeralised Dynamic Authorisation Mechanisms, Preceedings of the IBM computer science symposium, Amagi, Japan.

KAISLER, S. H. 1983, The design of operating systems for small computer systems, John Wiley & Sons Pub. Co.

KAVI, K. M. and KRISHNAMOHAN, K. 1984, Architecture quality, Operating systems review, Vol. 18, No. 1, pp. 11-13.

KAVIPURAPU, K. M. 1979, Quantification of architectures using software science, ACM Sig. on Computer Architecture News, Vol. 7, pp. 2-6

KEED, J. L. 1979, On structuring operating systems with monitors, Vol. 13, No. 1, pp. 5-9

KERNIGHAN, B. W. and RITCHIE, D. M. 1978, The C programming language, Prentice-Hall Pub. Co.

KERRIDGE, J. and SIMPSON, D. 1986, Communicating parallel processes, Software-Practice and Experience, Vol. 16, No. 1, pp. 63-86

KICKERT, W.J.M. 1980, Organization of decision making, Amsterdam: North-Holland

KIMBLETON, S. R., 1975, A heuristic approach to computer systems performance improvement: A-I fast performance prediction tool, AFIPS, pp. 839-846.

KINDLER, E.,1979, Dynamic systems and theory of simulation, Kybernetika, Vol. 15, No. 2, pp. 77-87.

KLEINE, H., 1977, SDDL: Software design and documentation language, Publication 77-24, Jet Propulsion Lab., Pasadena, Calif.

KLEINROCK, L., 1975, Queueing systems: Theory, Vol. 1, J. Wiley Pub. Co.

KLEINROCK, L., 1976, Queueing systems: Computer applications, Vol. 2, J. Wiley.

KLEINROCK, L., 1985, Distributed systems, CACM, Vol. 28, No. 11, pp. 1200-1213.

KLIR, G. J. 1969, An approach to general systems theory, Van Nostrand Reinhold, New York

KLIR, G. J. 1975, On the representation of activity arrays, Int. J. General Systems, Vol. 2, pp. 149-168.

KLIR, G. J. 1976, Identification of generative structures in empirical data, Int. J. of General Systems, Vol. 3, pp. 89-104

KLIR, G. J. and WAY, E. C. 1985, Reconstructability analysis: Aims, Results, Open problems, Int. J. of General Systems, Vol. 2, No. 2, pp. 141-163.

KNUTH, E. 1969, The art of computer programming, Addison-Wesley Pub. Co.

KNUTH, D. E.,1971, An empirical study of Fortran programs, Software-Practice and Expereince, Vol. 1, pp. 105-133

KOBAYASHI, H., 1978, Modelling and analysis: An introduction to system performance evaluation methodology, Addison-Wesley.

KOHOUT, L. J., 1974, Algebraic models in computer-aided medical diagnosis, Medinfo, pp. 575-579.

KOHOUT, L. J. 1976, Methodological foundation of the study of action, Tech. Rep. MMS-EES-ACT 76, Man-Machine System Laboratory, Department of Electrical Engineering Science, University of Essex, Colchester, U.K. pp. 270

KOHOUT, L. J. and GAINES, 1976, Protection as a general systems problem, Int. J. of General Systems, Vol. 3, pp. 1-21.

KOHOUT, L. J. 1977, Functional hierarchies in the brain: a methodology for their identification, Int. Conf. on Applied General Systems Research, State University of N.Y. at Birmingham, 15-19 Aug.

KOHOUT, L. J. 1978, Analysis of computing protection systems by means of Multi-Valued-Logic, in IEEE Proc. 8th Int. Symp. on MVL, New York

KOHOUT, L. J. 1978a, Methodological foundation of the study of action, Ph.D. Thesis, Man-Machine System Laboratory, Department of Electrical Engineering Science, University of Essex

KOHOUT, L. J., 1981, Control of movement and protection structures, Int. J. Man-Machine Studies, Vol. 14, pp. 397-422.

KOHOUT, L. J. and BANDLER W., 1981, Analysis of capability-based computer protection models by means of fuzzy logic, Proc. 11th Int. Symp. on Multiple-Valued Logics, IEEE Comput. Soc., Los Angeles, pp. 95-105.

KOHOUT, L. J. 1982, Lecture notes on activity structures, Workshop of Man-Computer Studies Group, Brunel University

KOHOUT, L. J. and BANDLER, W. 1982, Fuzzy expert systems, BCS Expert System 82 Conf., Brunel University, 14-16 Sept.

KOHOUT, L. J. 1983, A perspective of intelligent systems, Man-Computer Studies Group, Brunel University, pre prit

KOHOUT, L. J. KERAVNOU, E. and BANDLER, W., 1984, Automatic documentary information retrieval by means of fuzzy relational products, TIMS/Studies in the Management Sciences 20, pp. 383-404.

KOHOUT, L. J. and et. al. 1984, Construction of an expert therapy adviser as a special case of a general system protection design, Cybernetics and Systems Research 2, R. Trappl (Ed.), Elsevier Science Pub. B.V. (North-Holland).

KOHOUT, L. J. 1984, Activity structures: A tool for the design of information systems: Module 5 for M.Sc. in Computing System for Advance Information Technology, Department of Computer Science, Brunel University

KOHOUT, L. J., et al 1985, CLINAID: a knowledge-based DSS for use in medicine, In Mitra, G. (Ed.), Computer models for Decision Making, North-Holland

KOHOUT, L. J. 1986, Perspective on intelligent systems a framework for analysis and design, Abacus Press.

KOHOUT, L. J., 1986a, Activity structures: A general systems construct for design of technological artifacts, 8th European Meeting in Sybernetics and Systems Research, Vienna.

KOHOUT, L. J. and MOHAMAD, S. M. A. 1986, Activity structures: A general system framework for the design of information processing machines, To appear.

KRAMER, J., MAGEE,J., and SOLMON, M. 1984 A software architecture for distributed computer control systems, Automatica, Vol. 20, No. 1, pp. 93-102.

KUPKA, I. 1974, The syntax of dialog, Lecture Notes in Computer Science, Springer Pub. Co., Vol. 7, pp. 45-54

LAMPORT, L. 1978. Time, clocks and ordering of events in a distributed system, CACM, Vol. 21, No. 7

LAMPORT, L. 1985, Solved problems, unsolved problems and non-problems in concurrency, SIGOP Review, Vol. 19, No. 4, pp. 34-44.

LAMPSON, B. W. 1969, Dynamic protection structures, AFIPS, Vol. 35, pp. 27-38.

LAMPSON, B. W. and STURGIS, H. E. 1976, Reflections on an operating systems design, CACM, Vol. 19, No. 5, pp. 251-265.

LAMPSON, B. W., 1984, Hints for computer system design, IEEE Software, Vol. 1, No. 1, pp. 11-28.

LANDWEHR, C. E. 1981, Formal models for computer security, Computing Surveys, Vol. 13, No. 3, pp. 247-278.

LANTZ, K. A. et. al., 1982, Rochester's intelligent gateway, IEEE Computer, Vol. 15, No. 10, pp. 54-67.

LAUER, H. C. and NEEDHAM, R. M. 1979, On the duality of operating system structures, Vol. 13, No. 2, pp. 3-19

LEA, R. M., 1983, Scape: a single chip array processing element for image analysis,Lea, VLSI 83, Elsevier Science Pub. B.V. (North-Holland), F. Anceau and E.J. Aas (Eds.).

LEWELLYN, C. P. 1976, The statistical estimation of throughput and tornaround functions for a university computers syetem, Ph. D. Thesis, University of Illinois at Urbana Champaign

LEHMAN, M. M., 1980, Programs, programming and the software life cycle, Rep. 80/6, Dept. of Computing and Control, Imperial College of Science and Technology, London.

LELANN, G. E. 1977, Distributed Processing Towards a formal apprach, Proc. IFIP

LENDARIS, G. G. 1980, Structural modeling - A tutorial guide, IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-10, No. 12, pp. 807-839

LEVINE, A. P., 1984, Design overview of an expert system for computer performance modeling, The 12th European Computre Measurement Association (ECOMA-13), Munich, pp. 23-26.

LILIENFEID, D. 1978, The rise of system theory, Wiley Pub. Co., New York

LINDSAY, B. 1973, Suggestions for an extensible capability-based machine architecture, Grenoble, France, pp. 1-20

LINDSTROM, H. and SHANSHOLM, J. 1981, How to make your own simulation system, Software-Practice and Experience, Vol. 11, pp. 629-637

LINDQUIST, T.E. 1985 Assessing the usability of human-computer interfaces, IEEE Software, Vol 2, No. 1, pp. 74-82

LISKOV, B. et. al. 1977, Abstraction mechanisms in CLU, CACM, Vol. 20, No. 8, pp. 564-576.

LISTER, A. M. 1975, Fundamentals of operating systems, Macmillan Pub. Co.

LOCKETT, J. A. 1974, Computer performance analysis in mixed on-line batch workloads, National Computer Conf., AFIPS, pp. 671-676

LONDON, K. R. 1973, Techniques for direct access: hardware systems programming, Auerbach Pub. Co.

LUSCHEL, E.C 1962 The logical systems of Leniewski, North-Holland Pub. Co.

LUNDE, A. 1977, Empirical evaluation of some features of instruction set processor architectures, CACM, Vol. 20, No. 3, pp. 143-153.

MacDOUGALL, M. H. 1970, Computer system simulation: An introduction, Computing Surveys, Vol. 2, No. 3, pp. 191-209

MADNICK, S. E. and DONOUAN, J. J. 1974, Operating system, McGraw-Hill Pub. Co.

MADSEN, J. 1981, A computer system supporting data abstraction, SIGOP, Vol. 15, No. 1, pp. 45-72.

MAMRAK, S. and RANDAL, J. 1977, An analysis of a software engineering failure, The Computer J. , Vol. 20, No. 4, pp. 316-320.

MAPLES, C. 1985 Analysing software performance in a multiprocessor environment, IEEE Software, Vol. 2, No. 4, pp.21-29

MARKOWITZ, H. M., 1979, Simscript: Past, present, and some thoughts about the future, in Current Issues in Computer Simulation. N.R. Adam and A. Dogramaci, (Eds.), Academic, New York, pp. 27-60.

MASON, R.O. 1979, A general system theory of productivity, Int. J. of General Systems, Vol. 5, pp. 17-30

MATHEWSON, S. C. and ALLEN, J. H., 1977, DRAFT/GASP- A program generator for GASP, in Proc. 10th Annual Simulation Symp., (Tampa, Fla.), pp. 211-225.

McCAULEY, E. J. and DRONGOWSKI, P. J. 1979, KSOS - The design of a secure operating system, AFIPS National Computer Conf.

McKEON, B. 1983, A small-C Operating system, Dr. Dobb's J., Vol. 8, No. 77, PT3, pp. 36-61

MCLAREN, F. W. and MACEWAN, G. H. 1981, Computer security enhancement; an external kernel approach; Int. J. Mini-& Microcomputer, Vol. 3, No. 2, pp. 30-34.

McLEOD, J., 1973, Simulation: From art to science for society, Simulation, Vol. 21, No. 6, pp. 77-80.

MILLER, R. E. 1973, A comparison of some theoritical models of parallel computation, IEEE Trans. on Computers, Vol. C-22, No. 8, pp. 710-717

MILS, J. A. 1985, A pragmatic view of the system architect, CACM, Vol. 28, No. 7, pp. 708-717.

MOHAMAD, S. M. A. ,1981, A comparison of some performance evaluation techniques, M.Sc. Thesis, Glasgow University.

MOHAMAD, S. M. A. and CAVOURAS, J. C., 1982, Performance models of computer systems, MCSG/TR18, Computer Science Dept., Brunel University.

MOHAMAD, S. M. A. 1982, A guideline to the problem of designing well-protected computer systems, MCSG/TR19, Computer Science Dept., Brunel University.

MOHAMAD, S. M. A. 1983, Descriptor-oriented system: A protection architecture for performance, MCSG/TR28, Computer Science Dept., Brunel University.

MOHAMAD, S. M. A. and OHIORENOYA, 1983, Design criteria for expert systems, NAFIP II Conference, New York, June 29, 30 and June 1

MOHAMAD, S. M. A. and CAVOURAS, J. C. 1984, Performance study of descriptor roiented architectures, Computer Performance J., Vol. 5, No. 1, pp. 14-22.

MOORE, C. G., 1971, III Network models for large-scale time sharing systems, Tech. Rep. 71-1, Dept. Industrial Eng., University, Michigan, Ann Arbor, Ph.D. Thesis.

MORI, K. and et. al 1985, Autonomous decentralized computer system and software structure, Vol. 1, No. 1, pp. 17-22

MOTO-OKA, T. and STONE, H. 1984 Fifth Generation computer systems, IEEE Computer, Vol. 17, No. 3, pp.6-13

MULLERY, A. P. et. al., 1963, ADAMA problem-oriented symbol processor, AFIPS SJCC, pp. 367-380.

MUMFORD, E. et. al. 1978, Participative systems design, Computer weekly, Nov/Dec Part1-Part5.

MUNTZ, R. R., 1979, A predictable problem, ACM Computing Surveys, Vol. 11, No. 1, pp. 70-72.

MYERS, G. J. 1978, Advanves in computer architecture, Wiley Pub. Co., New York.

NANCE, R. E., 1971, On time flow mechanisms for discrete event simulation, Manage. Science, Vol. 18, No. 1, pp. 59-93.

NEEDHAM, R. M. 1972, Protection systems and protection implementations, AFIPS, Vol. 41, pp. 571-578.

NEEDHAM, R.M. 1977 The CAP project: an interim evaluation, proc. 6th Symposium on operating systems principles, Purdue University, 16-18 Nov., pp.17-22

NEWMAN, I. A. and Woodward, M. C. 1981, Performance degredation due to common memory access in multiprocessor systems, TR 41, Computer Science Dept., Loughborough University.

NIELSEN, J. 1986, A virtual protocol model for computer-human inferaction, Int. J. Man-Machine Studies, Vol. 24, pp. 301-312

NILSSON, N. J. 1965, Learning machines, McGraw-Hill Pub. Co., New York

NURMI, H. 1978, On strategies of cybernetic model-building, Kybernetes, Vol. 7, pp. 13-18

NUTT, J. G. 1978, A case study of simulation as a computer system design tool, IEEE Computer, Vol. 11, No. 10, pp. 31-36

NYGAARD, K., and DAHL, O., 1978, The development of the SIMULA languages, Sigplan Notices, Vol. 13, No. 4, pp. 245-272.

OBERQUELLE, H. et. al 1983, A view of human-machine communication and co-operation, Int. J. of Man-Machine Studies, Vol. 19, pp. 309-333

OHIORENOYA, M. A. and MOHAMAD, S. M. A. 1983, Software tools for decision support systems: Review and proposals, IUCC Conference, Glasgow University, 13-16 September

OLSON, R. 1985, Parallel Processing in a Message-Based Operating System, IEEE Software, Vol. 2, No. 4, pp.39-49

OREN, T. I., 1984, GEST-A modelling and simulation language based on system theoretic concepts, in Simulation and Model-Based Methodology: An Integrative View, T.I. Oren et. al.. (Eds.), Springer-Verlag, New York.

OREN, T. I. and ZEIGLER, B. P., 1979, Concepts for advanced simulation methodologies, Simulation, Vol. 32, No. 3, pp. 69-82.

OVERSTREET, C M. and NANCE, R. E., 1985, A specification language to assist in analysis of discrete event simulation models, CACM, Vol. 28, No. 2, pp. 190-201.

PAIGE, M. R. and BENSON, J. P. 1974, The use of software probes in testing Fortran programs, IEEE Computer, Vol. 7, No. 7

PLAM, G. 1982 Neural Assemblies: An alternative Approach to Artificial Intelligence, Springer-Verlag.

PARNAS, D. L. and DARRINGER, J. A., 1967, SODAS and a methodology for system design, Proc. AFPIS FJCC, Vol. 13, AFPIS Press Montvale, N. J. pp. 449-474.

PARNAS, D. and SIEWIOREK, D. 1975, Use of the concept of transparency in the design of hierarchically structured systems, CACM, Vol. 18, No. 7, pp. 401-408.

PARNAS, D. et. al. 1976, Design and specification of the minimal subset of an operating system family, IEEE Trans. on Software Engineering, Vol. SE-2, No. 4, pp. 301-307.

PASHTAN, A. 1985, Operating system models in a concurrent Pascal environment: Complexity and performance considerations, IEEE Trans. on Software Engineering, Vol. SE-11, No. 1, pp. 136-141.

PASHTAN, A. and UNGER, E. A. 1985, Resource monitors: A design methodology for operating systems, Software-Practice and Experience, Vol. 14, No. 8, pp. 791-806.

PATTERSON, D. and PIEPHO, R. 1982, RISC assessment: A high-level language experiment, Proc. 9th Int. Symp. Computer architecture, Apr. 26-29.

PEACOCK, J. K. et al 1979, Distributed simulation using a network of processors, Computer Networks, Vol. 3, No. 1

PENNY, J. P. and SHEEDY, C. R. 1980, Measurement of response time performance in small time-sharing systems, The australian Computer J., Vol. 12, No. 1, pp. 15-22

PERLIS, J. H. and IGNIZIO, J. P. 1980, Stability analysis: An approach to the evaluation of system design, Cybernetics and Systems: An Int. J., Vol. 11, pp. 87-103.

PETER, W. 1971, Data structure models for programming languages, Sigplan Notices, Vol. 6, No. 2, pp. 1-39

PETERSON, J. and SILBERSCHATZ, A. 1983, Operating system concepts, Addison-Wesley Pub. Co.

PETRI, C. A. 1966, Communication with automata, Tech. Rep. RADC-TR-65-337, Vol. 1, Griffiss Air Force Base, New York, N. Y.

PICHLER, F. and OTTENDORFER, W. 1978, Decomposition of general dynamical systems, Int. Conf. Systems Science IV, Wroclaw

PICHLER, F. 1983 Systematic Manipulation of Systems Models, In Oren et al: Simulation and Models Based Methodologies, Springer Verlag.

PILOTY, R. 1975, Segmentation constructs for RTS111, Proc. of the Int. Symp. on Hardware Description Languages and Their Application, New York, pp. 115-124.

PRATT, T.W. 1985 Pisces: An environment for parallel scientific computation, IEEE Software, Vol. 2, No. 4, pp. 7-20

PRICE, W. R. and PARNAS, D. L. 1973, The design of the virtual memory aspects of a virtual machine, Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems.

PRITSKER, A. A. B. 1981, The GASP IV Simulation language, Wiley, New York.

POTTER, J.L. (ed) 1985 The Massively Parallel Processor, The MIT Press.

REEVES, A.P. 1985 Parallel Pascal and the Massively Parallel Processor, In the Massively Parallel Processor (ed. Potter), MIT Press.

REINER, D. 1980, A method for adaptive performance improvement of operating systems, Ph.D. Thesis, University of Wisconsin-Madison.

RIDDER, T. 1986, Coroutines for C reconsidered, Software-Practice and Experience, Vol. 16, No. 3, pp. 301-302

RIDDLE, W. E. 1979, An approach to software system behaviour description, Computer Languages, Vol. 4, pp. 29-47

RISER, M., 1979, Mean-value analysis of queueing networks: A new look at old problem, Proc. 4th Int. Symp. on Modelling and Performance Evaluation of Computer System, Vienna, pp. 63-77.

RISER, M. and LAVENBERG, S. S., 1980, Mean-value analysis of closed multichain queueing networks, CACM, Vol. 27, No. 2, pp. 313-322.

ROBERTS, Z.A. 1986 The functional structure of control in expert systems, M.Sc. Diss., Florida State University.

ROBINSON, S. K. and TORSUN, I. S. 1976, An empirical analysis of Fortran programs. The Computer J., Vol. 19, No. 1, pp. 56-62

ROBINSON, S. K. and TORSUN, I. S. 1977, Dynamic analysis of program performance (DAP) in a FORTRAN batch environment, Software-Practice and Experience, Vol. 7, pp. 307-315.

ROMAN, G. et. al. 1984, A total system design framwork, IEEE Computer, Vol. 17, No. 4, pp. 15-26.

ROSE, C. S., 1976, Validation of queueing model with classes of customers, Proc. ACM Computer Performance Symp. pp. 318-326.

ROSEN, R. 1986, Causal structures in brains and machines, Int. J. General Systems, Vol. 12, pp. 107-126

ROSS, D. T. and SCHOMAN, 1977, Structured analysis for requirements definition, IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, pp. 6-15.

RUNES, D. D. 1942 Dictionary of Philosophy, The Philosophical Library.

RUSHBY, J. M. and RANDELL, B. Feb. 1983, A distributed secure system, Tec. Report 182, Computing Laboratory, University of NewCastle Upon Tyne, England,

SAKAMURA, K. et al 1979, Automatic tuning of computers architectures, National Computer Conf., AFIPS, pp. 499- 512

SCHERR, A. L., 1967, An analysis of the shared computer systems, MIT Press, Cambridge, Mass.

SCHLAEPPI, H. P. 1964, A formal language for describing machine logic, timing and sequencing (LOTIS), IEEE Trans. on Electronic Computers, Vol. EC-13.

SCHORR, H. 1964, Computer-aided digital system design and analysis using aregister transfer language, IEEE Trans. on Electronic Computer, Vol. EC-13.

SCHROEDER, M. D. and SALTZER, J. H. 1972, A hardware architecture for implementing protection rings, CACM, Vol. 15, No. 3, pp. 157-170.

SCHRUBEN, L., 1983, Simulation modelling with event graphs, CACM, Vol. 26, No. 11, pp. 957-963.

SEVCIK, K. and MITRONI, I., 1978, The distribution of queueing network states at input and output instants, Research Report no. 307, IRIA Recquencourt, France.

SHARP, D.A. 1984 The Role of functional programming in the design of structures for man-computer interactions, fourth year project, computer science department, Brunel University.

SHAW, A. C. 1974, The logical design of operating sytems, Prentice-Hall, Inc.

SEVCIK, K. C. and KLAWE, M. M., 1979, Operational analysis versus stochastic modelling of computer system, 12 Annual Symp. on the Interface of Computer Science and Statistic, University of Waterloo.

SHAW, M. L. G. 1979, Conversational heuristics for eliciting shared understanding, Int. J. Man-Machine Studies, Vol. 11, pp. 621-634

SHIMIZU, T. and SAKAMURA, K., 1983, Microprogram development based on knowledge engineering MIXER, Trans. Inst. Electron & Commun. Eng. Jpn. Part D, Vol. J66D, No. 7, pp. 864-871.

SHOOMAN, M. L. 1983, Software Engineering, McGRAW-HILL Pub. co.

SHORT, K. W. 1980, Data type interfaces in ad hoc decision support systems, Ph.D. Thesis, University of East Anglia.

SHUEY, R. 1986, Data engineering and information systems, IEEE Computer, Vol. 19, No. 1, pp. 18-30.

SIEWIORELC, D. P. et al 1982, Computer structures: Principles and examples, McGraw-Hill Pub. Co.

SIMPSON, H. R. and JACKSON, K. 1979, Process synchronisation in MASCOT, The Computer J., Vol. 22, No. 4, pp. 332-345.

SNYDER, L. 1981 Formal Models of Capability-Based Protection Systems, IEEE Trans. on Computers, Vol C-30, No. 3, pp.172-181

SIMON, H.A. 1986 Whether software enginnerring needs to be artificially intelligent, IEEE Trans. on software engineering, vol. SE-12, No. 7, pp.726-732

SINHA, M. K. 1985, Atomic actions and resource coordination problems having nonunique solutions, IEEE Trans. on Software Engineering, Vol. SE-11, No. 5, pp. 461-471

SMITH, C. 1980, The prediction and evaluation of the performance of software from extended design specifications, Ph.D. Diss., University of Texas at Austin.

SMITH, C. V. and BROWNE, J. C. 1982, Performance engineering of software systems: a case study, AFIPS, Vol. 51, pp. 217-224.

SORASEN. O. and SOLBERG, B., 1983, VLSI-implemented systolic array for vector processing, VLSI 83, Elsevier Science Pub. B.V. (North-Holland), F. Anceau and E.J. Aas (Eds.).

SPECTOR, A. and GIFFORD, D. 1986, A computer science prespective of bridge design, CACM, Vol. 29, No. 4, pp. 268-283

SPIER, M. J. 1973, An experimental implementation of the kernel/domain architecture, Proc. of the Fourth Symp. on Operating Systems Principles, Yorktown Heights, New York, October 15-17, pp. 8-21.

STEEL, R. G. D. and TORRIE, J. H. 1980, Principles and procedures of statistics abiometrical approach, McGraw-Hill Pub. Co.

STEMPLE, D. et al 1983, Operating system support for abstract database types, 2nd Int. Conf. on Database Cambridge, UK, Aug 30-sep 3

STEMPLE, D. et al 1982, Preliminary design of a port oriented operating system, Report No. 82-24, University of Mass. Computer and Information Science Department.

STEUSLOFF, H. U. 1984, Advanced real-time languages for distributed industrial process control, IEEE Computer, Vol. 17, No. 2, pp. 37-46

STOCKENBERG, J. and VAN DAM A. 1978, Vertical migration for performance enhancement in layered hardware/firmware/software systems, IEEE Computer, Vol. 11, pp. 35-50.

SVOBODA, A. 1964, Behaviour classification in digital systems, information processing machines, No. 10, pp. 25-42.

SZYMANSKI, A. 1980, Computer simulation of human thought- its perspectives and constraints, Kybernetes, Vol. 9, pp. 9-13

THEAKER, C. J. and BROOKES, 1983, Apractical course on operating systems, Mcmillan Press LTD

THEOREY, T. J. 1975, Validation criteria for computer system simulation, Symp, Simulation of Computer Systems III

TORSUN, I.S. and AL-JARRAH, M. M. 1978, Dynamic analysis of COBOL programs, Software-Practice and Experience, Vol. 11, pp. 949-961

TRELEAVEN, P. and LIMA, I. 1982, Fifth generation computer systems, IEEE Computer, Vol. 15, No. 8.

TULLY, C. J. 1985, Information, human activity and the nature of relevant theories, The Computer J., Vol. 28, No. 3, pp. 206-210.

TURING, A. M. 1936, On computable numbers with an application to the entscheidungs problem, Proc. Lond. Soc., Ser. 2, Vol. 4, pp. 230

UNGER, B., et. al., 1983, JADE: A simulation and software prototyping environment, Res. Rep. 83/133/22, Dept. of Computer Science, University of Clagary, Alberta.

USTER, M.J. 1984 Information theory for information technologists, MacMillan Pub. Co.

VAN GIGCH, J. P. 1979, A methodological comparison of the science systems and metasystem paradigms, Int. J. Man-Machine Studies, Vol. 11, pp. 651-663

VAN GIGCH, J. P., 1984, Epistemological questions raised by the metasystem paradigm, Int. J. Man-Machine Studies, Vol. 20, pp. 501-509

VICK, C. et. al. 1980, Adaptable architectures for supersystems, IEEE Computer, Vol. 13, No. 11, pp. 17-34.

VISONTAY, G. and CSAKI, P., 1979, DOCUM-For automatic documentation of SIMULA programs, SIMULA Newsletter, Vol. 7, No. 2.

VON BERTALANFFY, L. 1968, General system theory, George Braziller, New York

VON MAYRHAUSER, A.E.K., 1979, Performance-oriented design of interactive computer systems, Ph.D. Diss., Duke University.

WATSON, R. W. 1970, Time sharing system design concepts, McGraw-Hill Pub. Co.

WATSON, D.J. 1978 An approach to protection through capabilities, Ph.D. Thesis, Cambridge University.

WELSH, J. and BUSTARD, D.W. 1979 Pascal-Plus: another language for moduler multiprogramming, Software-Practice and Experience, Vol. 9, pp.947-957

WELCH, T. A. 1976, An investigation of descriptor oriented architecture, ACM Sig. Computer architecture News, Vol. 4, PT 4, pp. 141-146

WHITE, S. M. et al 1985, Embedded computer system requirements workshop, IEEE Computer, Vol. 18, No. 4, pp. 67-70

WIENER, R. and SINCOVEC, R. 1984, Software engineering with modula-2 and ADA, John Wiley & Sons Pub. Co.

WILKES, M. V. 1968, Time sharing computer systems, American Elsevier Pub. Co., New York.

WILKES, M. V. and NEEDHAM, R. M. 1979, The Cambridge CAP computer and its operating system, Elsevier/North Holland, Netherlands.

WILKES, M. V. 1984, Security management and protection, The Computer J., Vol. 27, No. 1, pp. 3-7.

WILLIAMS, M. H. 1983, The problem of absolute privacy, Information Processing Letters, Vol. 17, No. 3, pp. 169-171

WILLIAMS, T., 1972, Computer systems measurements and evaluation, BSC The Computer Bulletin, No. 16, pp. 100-104.

WILLIAMS, A. C. and BHANDIWAD, R. A. 1976, A general function approach to queueing network analysis of multiprogrammed computers, Networks, Vol. 6, No. 1, pp. 1-22

WITTEN, I. H. 1977, Exploring, modelling and controlling discrete sequential environments, Int. J. of Man-Machine Studies, Vol. 9, pp. 715-735

WOOD-HARPER A. T. and et. al 1982, A Taxonomy of Current approaches to systems analysis, The Computer J., Vol. 25, No. 1, pp. 12-16

WULF, W. et al, 1974, Hydra: The kernel of a multiprocessor operating system, CACM, Vol. 17, No. 6, pp. 337-345

WULF, W. A. et. al. 1976, An introduction to the construction and verification of ALPHARAD programs, IEEE Trans. on Software Engineering, Vol. SE-2, No.4, pp.253-265.

WULF, W.A. 1981 Fundemental structures of computer science, Addison-Wesley Pub. Co.

YOURDON, E. 1972, Design of on-line computer systems, Prentice-Hall Pub. Co., Englewood Cliffs, New Jersey

YUVAL, A. 1980, System contention analysis- An alternate approach to system tuning, IBM Syst J., Vol. 19, No. 2, pp. 208-228

ZAVE, P. 1984, The operational versus the conventional approach to software development, CACM, Vol. 27, No. 2, pp. 104-118.

ZEIGIER, B. P. 1972, Towards a formal theory of modeling and simulation: structure preserving morphisms, Journal of the Association for Computing Machinery, Vol. 19, No. 4, pp. 742-764

ZEIGLER, B. P. 1975, Statistical simplification of neural nets, Int. J. Man-Machine Studies, Vol. 7, pp. 371-393

ZEIGLER, B. P., et. al., 1980, ESP: An interactive tool for system structuring, in Proc. European Meeting Cybernetics and Systems Research, Vienna, Hemisphere, New York.

ZEIGLER, B. P., 1984, Multifacetted Modelling and Discrete event simulation, Academic, New York.

ZEMANEK, H. 1979, Abstract Architecture: General Concepts for Systems Design, The Copenhagen Winter School Proceedings on Abstract Software Specifications.

Appendix

Analysing the effects of changes within the user environment: We analyse the user effectiveness and its relation to the system effectiveness measure of the average response time. Programmer effectiveness can apparently be measured in terms of work units (Jones and Schwarz 1980). It can be increased with scheduling dexterity (Doherty and Kelisky 1979), and it can be constrained by the interactive environment (Barber 1979). For this purpose we performed a series of experiments to test the user effectiveness upon our interactive constellation simulator using the user effectiveness parameters mentioned in table 10 .

A.1 Experiment-2: USER PRODCTIVITY VERSUS AVERAGE RESPONSE TIME

We run the interactive constellation with average user productivity from 25 to 100 transactions per login period in increments of 10. Using the scheme of performoact we modelled the effects of these changes upon the average response time index. Refer to Table 1 and Figure 1. Indeed, increasing the user productivity sharply increases the average response time index which has an increasing slop depending on the concurrent jobs allowed to be entered to the system. The 'best' equations in Table 1 provide an approximate description of the behaviour of the system, and according to different user productivity rates.

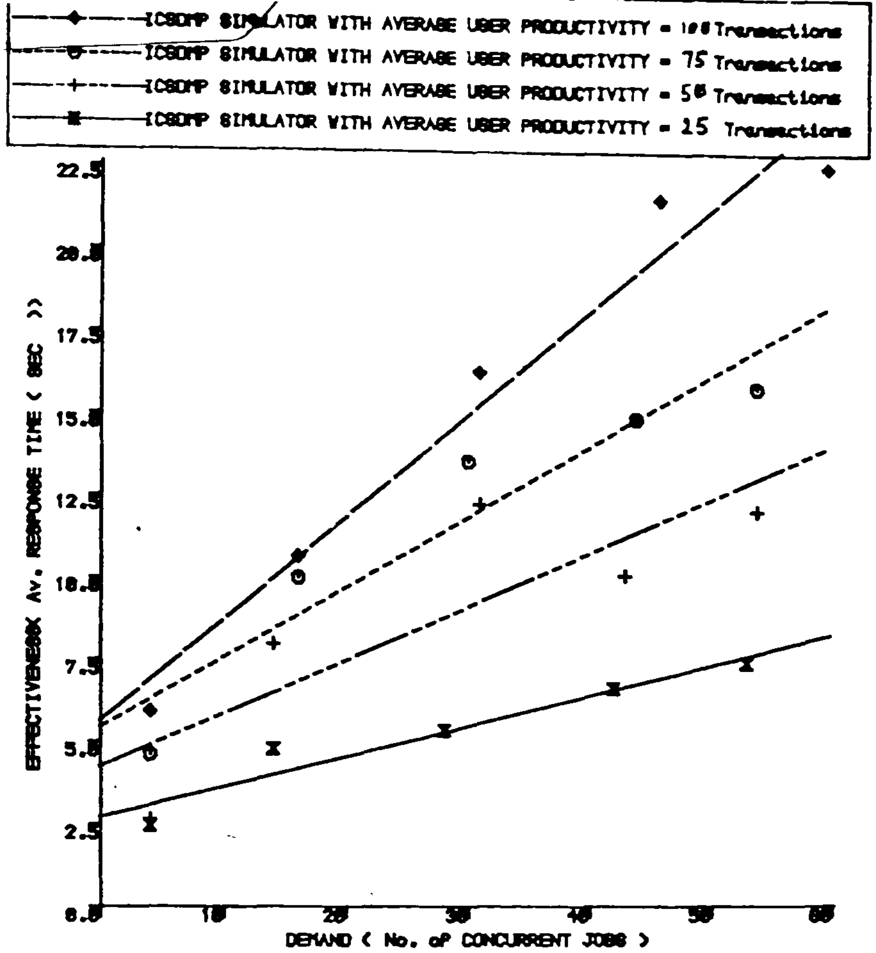


FIGURE 1: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(100 OBSERVED DEMAND	4	16	31	46	60
transaction) OBSERVED EFFECT.	5·93	10·67	16·28	21.57	22.57
LIN PERFORMOACT MODEL :	A=5·65	B=·31	DF=.98		
EXP PERFORMOACT MODEL :	A=6·58	B=·02	DF=.94		
LOG PERFORMOACT MODEL :	A=-4.30	B=6·35	DF=.96	*ADMIS*	
PWR PERFORMOACT MODEL :	A=2.80	B=·51	DF=.99	*BEST*	
(75 trans. OBSERVED DEMAND	4	16	30	44	54
CON) OBSERVED EFFECT.	4·63	10·02	13·58	14·9	15·84
LIN PERFORMOACT MODEL :	A=5·45	B=·21	DF=.94		
EXP PERFORMOACT MODEL :	A=5·58	B=·02	DF=.89		
LOG PERFORMOACT MODEL :	A=-1·56	B=4·35	DF=.997 ▸■	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A=2.48	B=·48	DF=.992		
(50 OBSERVED DEMAND	4	14	31	43	54
trans.) OBSERVED EFFECT.	2.65	7.99	12.28	10·43	12.09
LIN PERFORMOACT MODEL :	A=4·24	B=·16	DF=.85		
EXP PERFORMOACT MODEL :	A=3·81	B=·02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1·90	B=3·61	DF=.96	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A=1·38	B=·57	DF=.95		
(25 OBSERVED DEMAND	4	14	28	42	53
trans-) OBSERVED EFFECT.	2·45	4·81	5·39	6·67	7·45
LIN PERFORMOACT MODEL :	A=2·73	B=·09	DF=.96		
EXP PERFORMOACT MODEL :	A=2·85	B=·01	DF=.91		
LOG PERFORMOACT MODEL :	A=-·14	B=1·82	DF=.98	*ADMIS*	
PWR PERFORMOACT MODEL :	A=1·44	B=·41	DF=.99	*BEST*	

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 1: THE EFFECTS OF User Productivity Variation UPON THE AVERAGE RESPONSE TIME INDEX

△ : B-Best
□ : B-Adms

A.2 Experiment-3: USER AV. SATISFACTORY RESPONSE TIME vs. AVERAGE RESPONSE TIME

It is generally known that users are more productive which shorter average response time index. In our activity structures based design the inferential structure mechanisms make more frequent inspections whenever the key performance indices degrade (i.e. the average response time index for the interactive version and average system throughput index for the general purpose). This experiment examines address the problem of the effects of varying the average satisfactory response time parameter of the user environment. Refer to Table 2 and Figure 2. The result of increasing the average satisfactory response time parameter reduces the average system response time index slightly (1 sec approximately for each two seconds of the increase in the average user satisfactory response time parameter).

A.3 Experiment-4: Faulty Intention Rate vs. Av. Response Time

In our interactive constellation design, faulty access intentions can be generated at different rates. Of interest here is the question whether or not an increase in the number of the user faulty access intentions affects the system performance. In this case, the rate of the value of the user faulty intentions access was varied from 15% to 30% in increments of 5%. Using the performoact scheme the results of running the interactive constellation are described in Table 3 and Figure 3. The average response time is slightly decreased with the increase of the faulty access rates due to the overahead time spent by the dynamic protection mechanism to search whether any other user job is allowed to pass the right access rights to that particular user job. That explains why the increase in the faulty access intention rate didn't produce an equivalent reduction in the average response time (only.5 sec for increase in the rate of 5%).

USER WORK UNITS vs. AV. RESPONSE TIME User work units can be expressed using several parameters in our interactive simulation model. The parameters are:

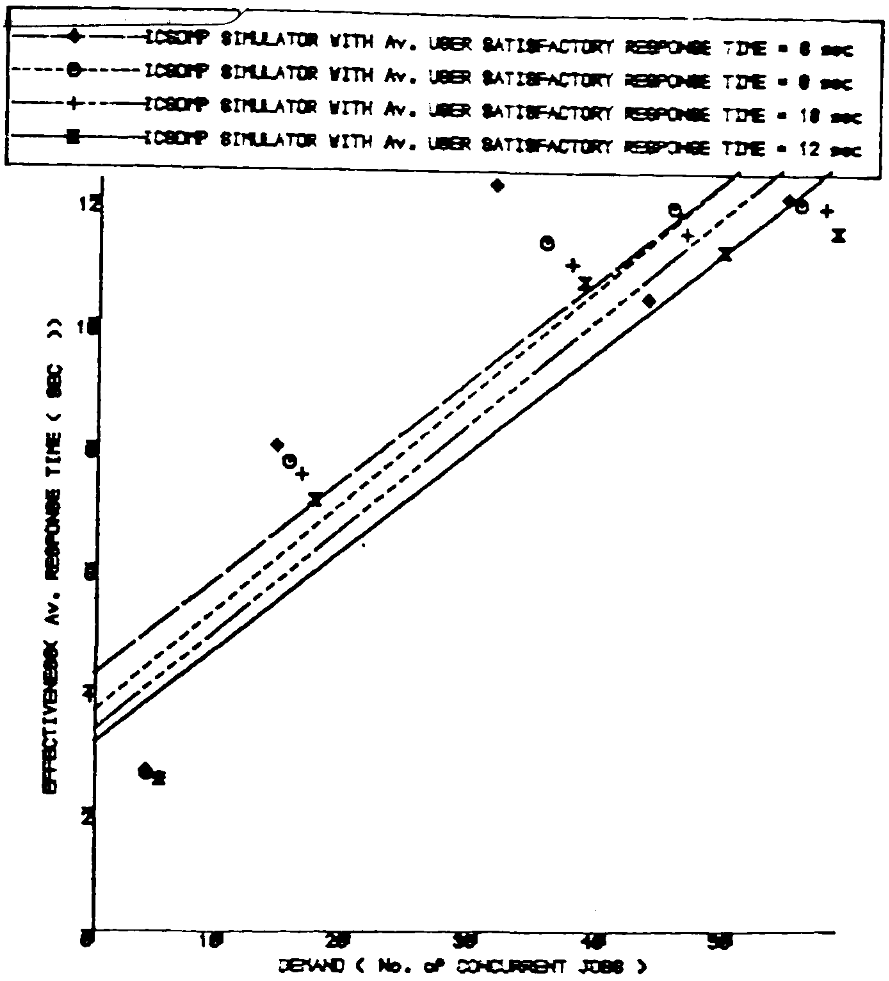


FIGURE 2 : PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(6.0 Sec OBSERVED DEMAND	4	14	31	43	54
CON) OBSERVED EFFECT.	2·65	7·99	12·28	10·43	12·09
LIN PERFORMOACT MODEL :	A= 4·24	B= ·16	DF=.85		
EXP PERFORMOACT MODEL :	A= 3·81	B= ·02	DF=.81		
LOG PERFORMOACT MODEL :	A= −1·90	B= 3·61	DF=.960	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·38	B= ·57	DF=.95		
(8·0 Sec OBSERVED DEMAND	4	15	35	45	55
) OBSERVED EFFECT.	2·60	7·73	11·35	11·93	12·0
LIN PERFORMOACT MODEL :	A= 3·63	B= ·17	DF=.92		
EXP PERFORMOACT MODEL :	A= 3·49	B= ·02	DF=.86		
LOG PERFORMOACT MODEL :	A= −2·48	B= 3·75	DF=.992	▲■ *BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·26	B= ·59	DF=.97		
(10·0 Sec OBSERVED DEMAND	5	16	37	46	57
) OBSERVED EFFECT.	2·53	7·52	11·0	11·51	11·93
LIN PERFORMOACT MODEL :	A= 3·32	B= ·17	DF=.93		
EXP PERFORMOACT MODEL :	A= 3·30	B= ·02	DF=.66		
LOG PERFORMOACT MODEL :	A= −3·67	B= 3·96	DF=.991	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·02	B= ·64	DF=.97		
(12·0 Sec OBSERVED DEMAND	5	17	38	49	58
) OBSERVED EFFECT.	2·51	7·11	10·71	11·22	11·55
LIN PERFORMOACT MODEL :	A= 3·11	B= ·16	DF=.94		
EXP PERFORMOACT MODEL :	A= 3·17	B= ·02	DF=.88		
LOG PERFORMOACT MODEL :	A= −3·58	B= 3·80	DF=.991	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·00	B= ·62	DF=.98		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 2 : THE EFFECTS OF Average User Satisfactory Response Time UPON THE AVERAGE RESPONSE TIME INDEX

▲ : B - BEST
■ . S - Admis

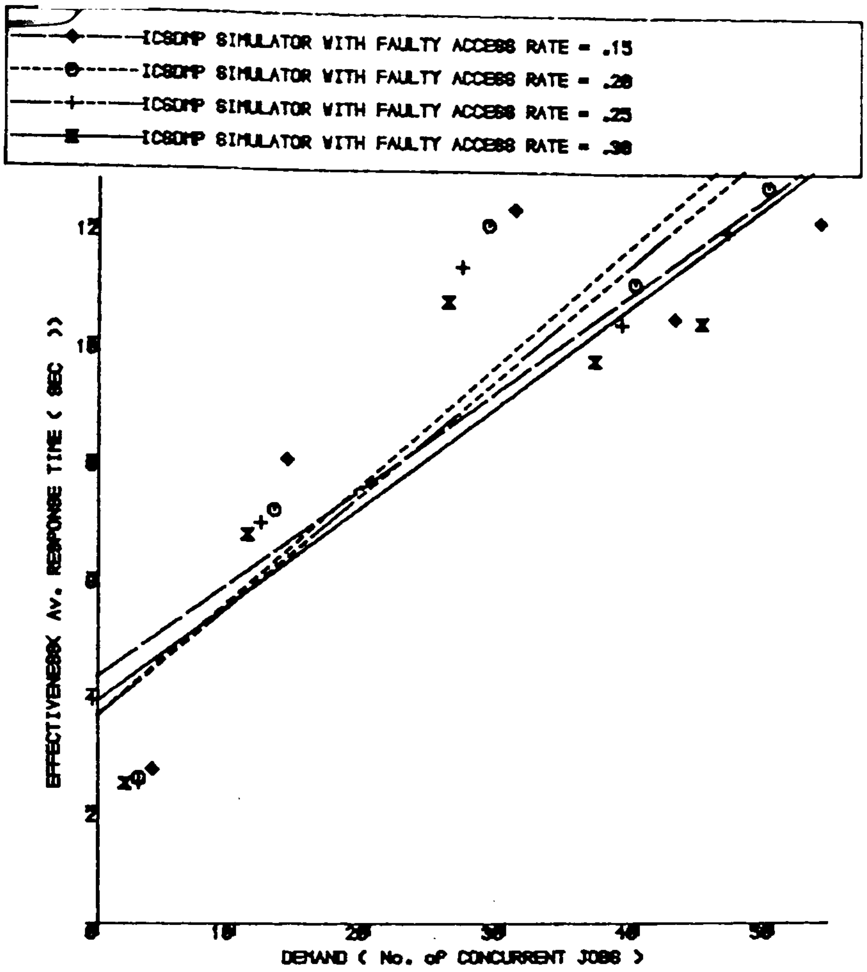


FIGURE 3 : PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(·15 rate OBSERVED DEMAND	4	14	31	43	54
CON)OBSERVED EFFECT.	2·65	7·90	12·28	10·43	12·09
LIN PERFORMOACT MODEL :	A= 4·24	B= ·16	DF=.85		
EXP PERFORMOACT MODEL :	A= 3·81	B= ·02	DF=.81		
LOG PERFORMOACT MODEL :	A= -1·90	B= 3·61	DF=.96	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·38	B= ·57	DF=.95		
(·20 rate OBSERVED DEMAND	3	13	29	40	50
) OBSERVED EFFECT.	2·51	7·12	12·01	11·02	12·7
LIN PERFORMOACT MODEL :	A= 3·58	B= ·20	DF=.91		
EXP PERFORMOACT MODEL :	A= 3·44	B= ·03	DF=.86		
LOG PERFORMOACT MODEL :	A= -1·61	B= 3·65	DF=.97	*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·42	B= ·58	DF=.98	*BEST*	
(·25 OBSERVED DEMAND	3	12	27	39	47
rate) OBSERVED EFFECT.	2·42	6·9	11·3	10·32	11·92
LIN PERFORMOACT MODEL :	A= 3·57	B= ·19	DF=.90		
EXP PERFORMOACT MODEL :	A= 3·37	B= ·03	DF=.85		
LOG PERFORMOACT MODEL :	A= -1·31	B= 3·43	DF=.98 ▲■	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·41	B= ·57	DF=.97		
(·30 OBSERVED DEMAND	2	11	26	37	45
rate) OBSERVED EFFECT.	2·41	6·7	10·7	9·7	10·35
LIN PERFORMOACT MODEL :	A= 3·83	B= ·17	DF=.87		
EXP PERFORMOACT MODEL :	A= 3·45	B= ·02	DF=.83		
LOG PERFORMOACT MODEL :	A= ·62	B= 2·66	DF=.97	*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·86	B= ·48	DF=.98	*BEST*	

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 3 : THE EFFECTS OF Faulty Accesses Rate Variation UPON THE AVERAGE RESPONSE TIME INDEX

▲ : B-Best
■ : B-Admis

1. User average think time,

2. Number of tasks (e.g. edit, compile, execute, etc),

3. Job arrival speed (i.e. mean interarrival time),

4. Job speed (i.e. average processor time required by a user job),

5. Average memory size required by a user job, and

6. Average number of backing store records required for a user job.

A.4 Experiment-5: Average Think Time vs. Av. response time

With the increase in the user average **think time** (a setp of 10 sec) we noticed very oscillating behaviour. First of all an increase in the think time increases the average response time and possibly we got later a reduction because of the frequent inspections of our inferential structure (which have the effects of improving performance). Then with a later increase in the average user think time the average response time increases (see Figure 4 and Table 4).

A.5 Experiment-6: No. of Tasks vs Av. Response Time

With the increase of the average **number of tasks** the user might produce, we notice a sharp increase in the average response time (around 1.5 second for each task increased) (see Figure 5 and Table 5).

A.6 Experiment-7: Job Arrival Speed vs Av. Response Time

With the increase in the job **arrival speed** of user jobs (i.e. lower mean of jobs interarrival time), the average response time gets slightly worse. It increases about.5 seconds for each 1.5 msec increase in the arrival speed) (Figure 6 and Table 6).

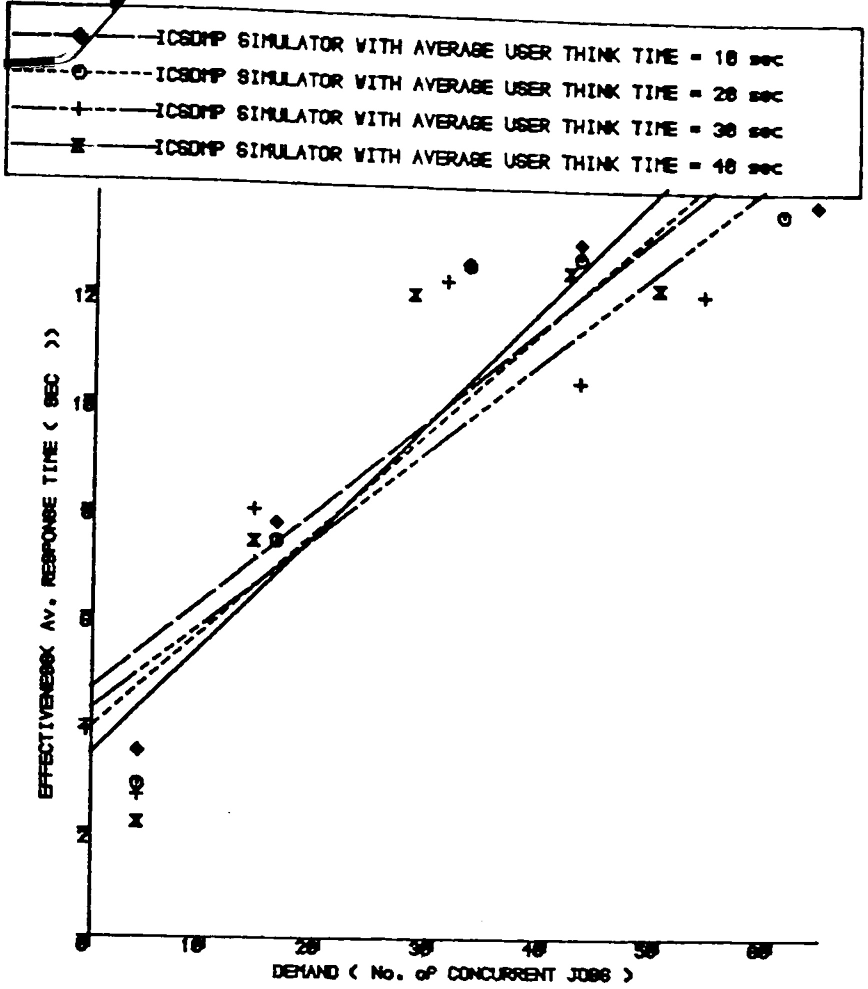


FIGURE 4 : PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN		5	10	15	20	25
(10	OBSERVED DEMAND	4	16	33	43	64
Seconds)	OBSERVED EFFECT.	3·47	7·75	12·62	13·02	13·78
LIN PERFORMOACT MODEL :		A=4·62	B=.17	DF=.91		
EXP PERFORMOACT MODEL :		A= 4.54	B=·02	DF=.86		
LOG PERFORMOACT MODEL :		A=-2·28	B=3·98	DF=.98	*ADMIS*	
PWR PERFORMOACT MODEL :		A= 1·74	B=·52	DF=.99	*BEST*	
(20	OBSERVED DEMAND	4	16	33	43	61
Sec.)	OBSERVED EFFECT.	2·85	7·42	12·58	12·74	13·61
LIN PERFORMOACT MODEL :		A=3·90	B=·18	DF=.91		
EXP PERFORMOACT MODEL :		A=3·83	B= ·02	DF=.86		
LOG PERFORMOACT MODEL :		A=-3·21	B=4·20	DF=.981	*ADMIS*	
PWR PERFORMOACT MODEL :		A=1·31	B= ·60	DF=.99	*BEST*	
(30 Sec.	OBSERVED DEMAND	4	14	31	43	54
CON)	OBSERVED EFFECT.	2·65	7·99	12·28	10·43	12·09
LIN PERFORMOACT MODEL :		A=4·24	B=·16	DF=.85		
EXP PERFORMOACT MODEL :		A=3.81	B=·02	DF=.81		
LOG PERFORMOACT MODEL :		A=-1·90	B=3·61	DF=.96	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :		A= 1·38	B=·57	DF=.95		
(40	OBSERVED DEMAND	4	14	28	42	50
Sec.)	OBSERVED EFFECT.	2·14	7·40	12·03	12.49	12.20
LIN PERFORMOACT MODEL :		A=3·41	B= ·21	DF=.89		
EXP PERFORMOACT MODEL :		A=3·12	B= ·03	DF=.83		
LOG PERFORMOACT MODEL :		A=-3·65	B= 4·29	DF=.980	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :		A= ·92	B= .71	DF=.96		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table4 : THE EFFECTS OF Average Think Time Variation UPON THE AVERAGE RESPONSE TIME INDEX

△ : B-Best
□ : B-Admis

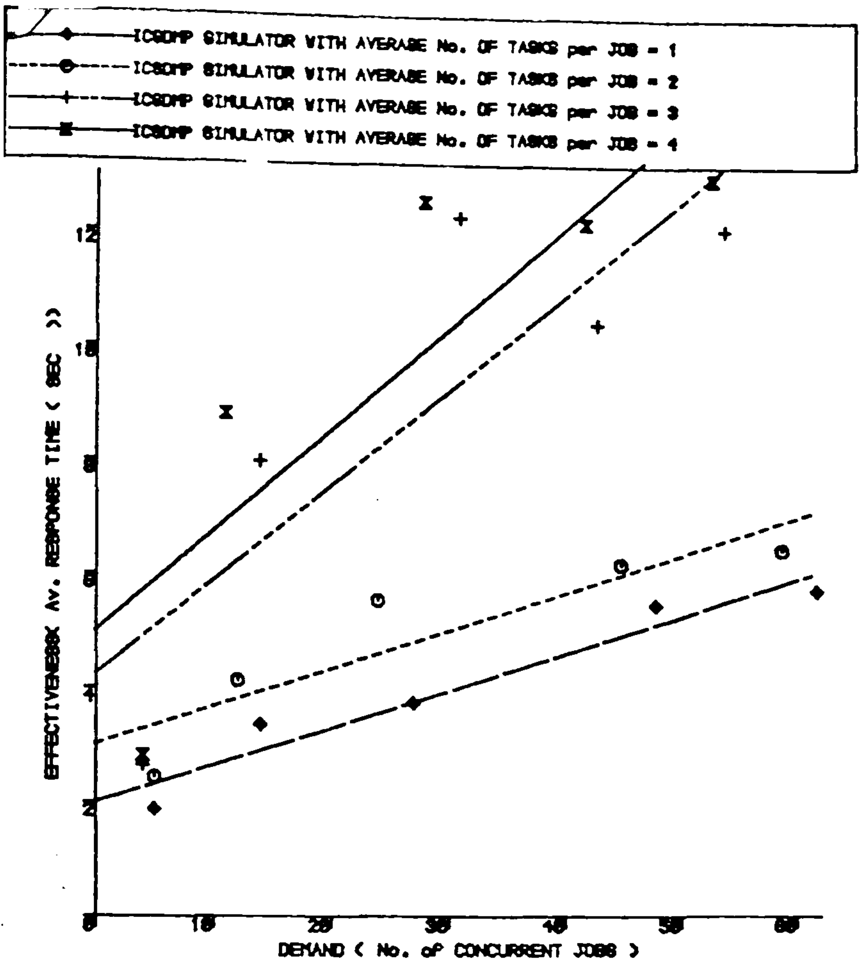


FIGURE 5 : PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(1 task OBSERVED DEMAND	5	14	27	48.	62
) OBSERVED EFFECT.	1·88	3·37	3·76	5·50	5·76
LIN PERFORMOACT MODEL :	A=2·00	B=·06	DF=.96		
EXP PERFORMOACT MODEL :	A=2.16	B=·01	DF=.92		
LOG PERFORMOACT MODEL :	A=-.74	B=1·54	DF=.97	*ADMIS*	
PWR PERFORMOACT MODEL :	A= ·95	B= ·44	DF=.98	*BEST*	
(2 task OBSERVED DEMAND	5	12	24	45	59
) OBSERVED EFFECT.	2·44	4·14	5·56	6·21	6·47
LIN PERFORMOACT MODEL :	A=3·02	B=·06	DF=.90		
EXP PERFORMOACT MODEL :	A=3·01	B=·15	DF=.85		
LOG PERFORMOACT MODEL :	A=-4·23	B=1·65	DF=.99 ▵■	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A=1·43	B=·39	DF=.97		
(3 task OBSERVED DEMAND	4	14	31	43	54
CON) OBSERVED EFFECT.	2·65	7·99	12·28	10·43	12.09
LIN PERFORMOACT MODEL :	A=4·24	B=·16	DF=.85		
EXP PERFORMOACT MODEL :	A=3·81	B=·02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1·90	B=3·61	DF=.96	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A=1·38	B=·57	DF=.95		
(4 task OBSERVED DEMAND	4	11	28	42	53
) OBSERVED EFFECT.	2·82	8·82	12·56	12·20	12·98
LIN PERFORMOACT MODEL :	A=5·11	B=.16	DF=.82		
EXP PERFORMOACT MODEL :	A=4·43	B=·02	DF=.76		
LOG PERFORMOACT MODEL :	A=-1·37	B=3·76	DF=.95	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A=1·66	B=·55	DF=.92		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 5 : THE EFFECTS OF Average No. of Tasks per Job UPON THE AVERAGE RESPONSE TIME INDEX

▵ : B-Best
■ : B-[illegible]

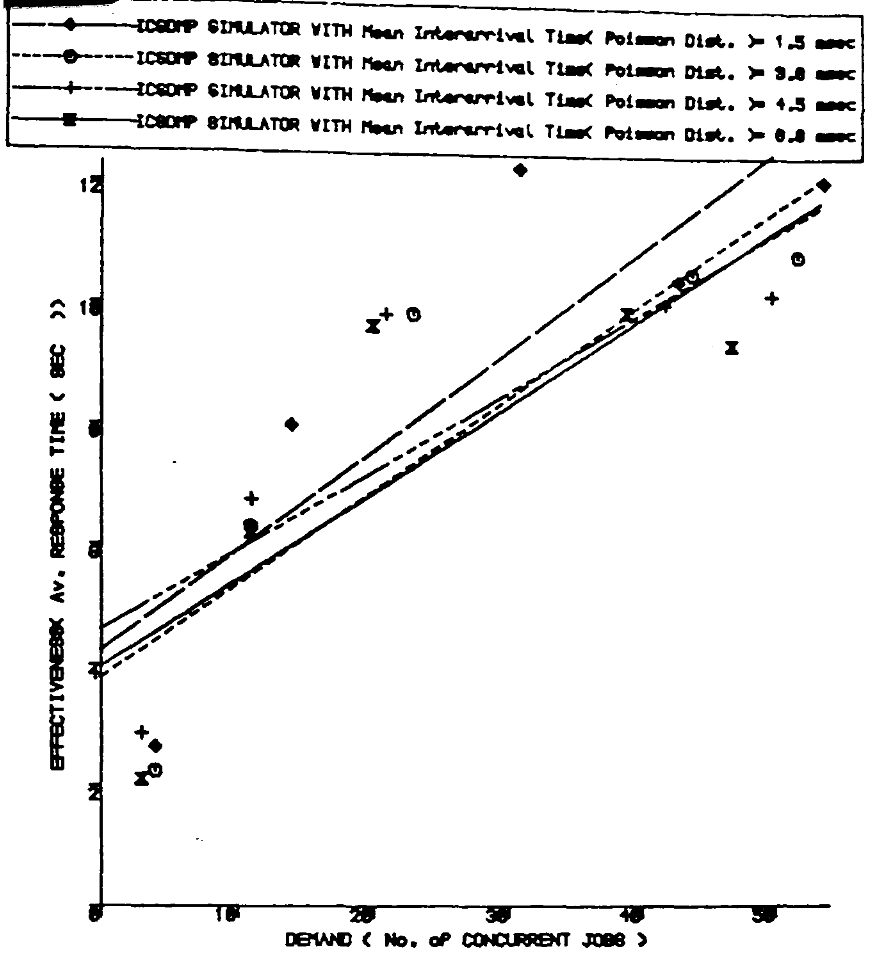


FIGURE 6: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25	
(1.5 msec. OBSERVED DEMAND	4	14	31	43	54	
CON)OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09	
LIN PERFORMOACT MODEL :	A= 4.24	B= .16	DF= .85			
EXP PERFORMOACT MODEL :	A= 3.81	B= .02	DF= .81			
LOG PERFORMOACT MODEL :	A= -1.90	B= 3.61	DF= .96			*BEST*ADMIS*
PWR PERFORMOACT MODEL :	A= 1.38	B= .57	DF= .95			
(3.0 msec. OBSERVED DEMAND	4	11	23	44	52	
) OBSERVED EFFECT.	2.25	6.3	9.85	10.54	10.85	
LIN PERFORMOACT MODEL :	A= 3.79	B= .15	DF= .87			
EXP PERFORMOACT MODEL :	A= 3.46	B= .02	DF= .80			
LOG PERFORMOACT MODEL :	A= -2.01	B= 3.40	DF= .98 ▲■			*BEST* ADMIS*
PWR PERFORMOACT MODEL :	A= 1.20	B= .59	DF= .94			
(4.5 OBSERVED DEMAND	3	11	21	42	50	
msec.) OBSERVED EFFECT.	2.87	6.76	9.85	10.03	10.19	
LIN PERFORMOACT MODEL :	A= 4.59	B= .13	DF= .83			
EXP PERFORMOACT MODEL :	A= 4.20	B= .02	DF= .78			
LOG PERFORMOACT MODEL :	A= .31	B= 2.68	DF= .96			*BEST* ADMIS*
PWR PERFORMOACT MODEL :	A= 1.99	B= .45	DF= .95			
(6.0 OBSERVED DEMAND	3	11	20	39	47	
msec.) OBSERVED EFFECT.	2.12	6.21	9.65	9.90	9.37	
LIN PERFORMOACT MODEL :	A= 3.96	B= .14	DF= .80			
EXP PERFORMOACT MODEL :	A= 3.41	B= .02	DF= .76			
LOG PERFORMOACT MODEL :	A= -.55	B= 2.85	DF= .95			*BEST*ADMIS*
PWR PERFORMOACT MODEL :	A= 1.37	B= .55	DF= .94			

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 1 : THE EFFECTS OF The Mean Interarrival Time UPON
THE AVERAGE RESPONSE TIME INDEX

▲ : B. Best
□ : B- Admis

A.7 Experiment-8: Job Speed vs Av. Response Time

With the increase of the job speed (i.e. the average processor time required for the user job to finish), we notice a sharp increase in the average response and with a potential increase and also the increase in the concurrency level (Figure 7 and Table 7).

A.8 Experiment-9: Memory Required per Job vs Av. Response Time

With the increase of the user demand for memory usage (average memory required for user job), there is a slight increase in the average response time, but highly affected by the increase in the concurrency level (see Figure 8 and Table 8).

A.9 Experiment-10: Backing Store Records Required vs Av. Res. T.

With the increase in the backing store records aquired by the user, we noticed a sharp increase in the average response time with respect to a slop caused by the level of concurrency in the constellation (see Figure 9 and Table 9).

Analysing the effect of changes within the machine environment: Now let us consider some changes to the hardware/software capabilities to the initial configuration of our interactive constellation. These changes are made by changing the parameters mentioned in Table 7.1. There are many possible alternatives that may produce different successful versions to the original NUKE-oriented data (called as general families of NUKE in this case). Here we performed two major experiments to analyse both hardware and software changes upon the average response time.

Hardware changes vs. Av. response time: The hardware-dependent parameters considered for this experiment include the following:

1. Average memory segment size,

2. Total memory size,

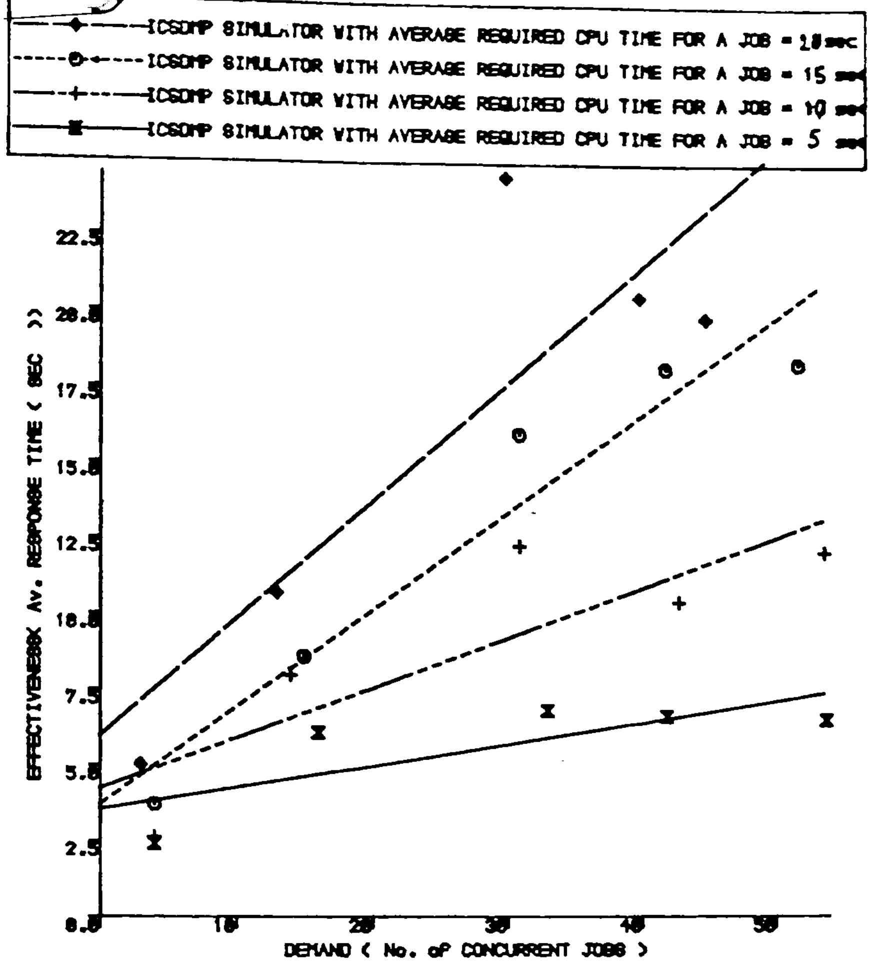


FIGURE 7 : PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN		5	10	15	20	25
(20	OBSERVED DEMAND	3	13	30	40	45
Second)	OBSERVED EFFECT.	5·02	10·70	24·44	20·50	19·81
LIN PERFORMOACT MODEL :		A=5·98	B=·38	DF=.86		
EXP PERFORMOACT MODEL :		A= 5·97	B=·03	DF=.88		
LOG PERFORMOACT MODEL :		A= -2·80	B=6·48	DF=.91	*ADMIS*	
PWR PERFORMOACT MODEL :		A= 2·73	B=·56	DF=.96	*BEST*	
(15 Sec.	OBSERVED DEMAND	4	15	31	42	52
CON)	OBSERVED EFFECT.	3·73	8·60	15·97	18·14	18·31
LIN PERFORMOACT MODEL :		A=3·71	B= ·32	DF=.96		
EXP PERFORMOACT MODEL :		A= 4·36	B= ·03	DF=.92		
LOG PERFORMOACT MODEL :		A=-5·65	B=6·11	DF=.97 ■	*ADMIS*	
PWR PERFORMOACT MODEL :		A= 1·51	B= ·65	DF=.99▲	*BEST*	
(10	OBSERVED DEMAND	4	14	31	43	54
Sec.)	OBSERVED EFFECT.	2·65	7·99	12·28	10·43	12·09
LIN PERFORMOACT MODEL :		A=4·24	B=·16	DF=.85		
EXP PERFORMOACT MODEL :		A= 3·81	B=·02	DF=.81		
LOG PERFORMOACT MODEL :		A=-1·90	B= 3·61	DF=.96	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :		A= 1·38	B= ·57	DF=.95		
(5 Sec.	OBSERVED DEMAND	4	16	33	42	54
)	OBSERVED EFFECT.	2·42	6·08	6·84	6·67	6·53
LIN PERFORMOACT MODEL :		A=3·56	B=·07	DF=.77		
EXP PERFORMOACT MODEL :		A=3·23	B= ·01	DF=.75		
LOG PERFORMOACT MODEL :		A= ·59	B= 1·66	DF=.93	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :		A= 1·59	B= ·39	DF=.92		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 7: THE EFFECTS OF The Av. Required CPU Time UPON THE AVERAGE RESPONSE TIME INDEX

▲ : B-Best
■ : B-Admis

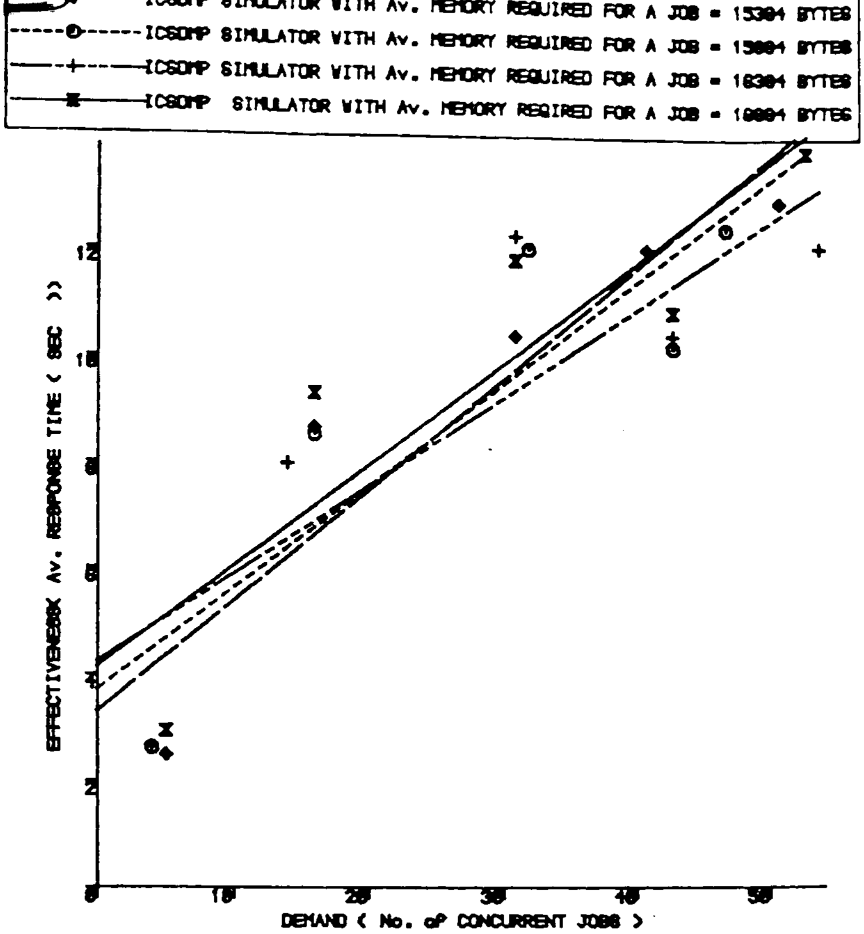


FIGURE 8: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25	
(15384 OBSERVED DEMAND	5	16	31	41	51	
bytes)OBSERVED EFFECT.	2.5	9.67	10.4	12.05	12.95	
LIN PERFORMOACT MODEL :	A=3.31	B=.20	DF=.93			
EXP PERFORMOACT MODEL :	A=3.30	B=.03	DF=.85 Δ□			
LOG PERFORMOACT MODEL :	A=-4.28	B=4.39	DF=.99			*BEST*ADMIS*
PWR PERFORMOACT MODEL :	A= .93	B=.69	DF=.96			
(15884 OBSERVED DEMAND	4	16	32	43	47	
bytes) OBSERVED EFFECT.	2.63	8.54	12.04	10.19	12.44	
LIN PERFORMOACT MODEL :	A=3.72	B= .19	DF=.87			
EXP PERFORMOACT MODEL :	A=3.44	B= .02	DF=.84			
LOG PERFORMOACT MODEL :	A=-2.24	B= 3.74	DF=.962			*BEST*ADMIS*
PWR PERFORMOACT MODEL :	A= 1.26	B= .60	DF=.961			
(16384 bytes OBSERVED DEMAND	4	14	31	43	54	
CON) OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09	
LIN PERFORMOACT MODEL :	A=4.24	B= .16	DF=.85			
EXP PERFORMOACT MODEL :	A=3.81	B= .02	DF=.81			
LOG PERFORMOACT MODEL :	A=-1.90	B=3.61	DF=.96			*BEST*ADMIS*
PWR PERFORMOACT MODEL :	A= 1.38	B= .57	DF=.95			
(16884 OBSERVED DEMAND	5	16	31	43	53	
bytes) OBSERVED EFFECT.	2.95	9.31	11.83	10.87	13.10	
LIN PERFORMOACT MODEL :	A=4.35	B= .17	DF=.87			
EXP PERFORMOACT MODEL :	A=4.03	B= .02	DF=.81			
LOG PERFORMOACT MODEL :	A=-2.91	B=4.02	DF=.96			*BEST*ADMIS*
PWR PERFORMOACT MODEL :	A= 1.30	B= .60	DF=.94			

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^B

Table 5: THE EFFECTS OF Av. Memory Required by a Job UPON THE AVERAGE RESPONSE TIME INDEX

Δ : B-Best
□ : B-Admis

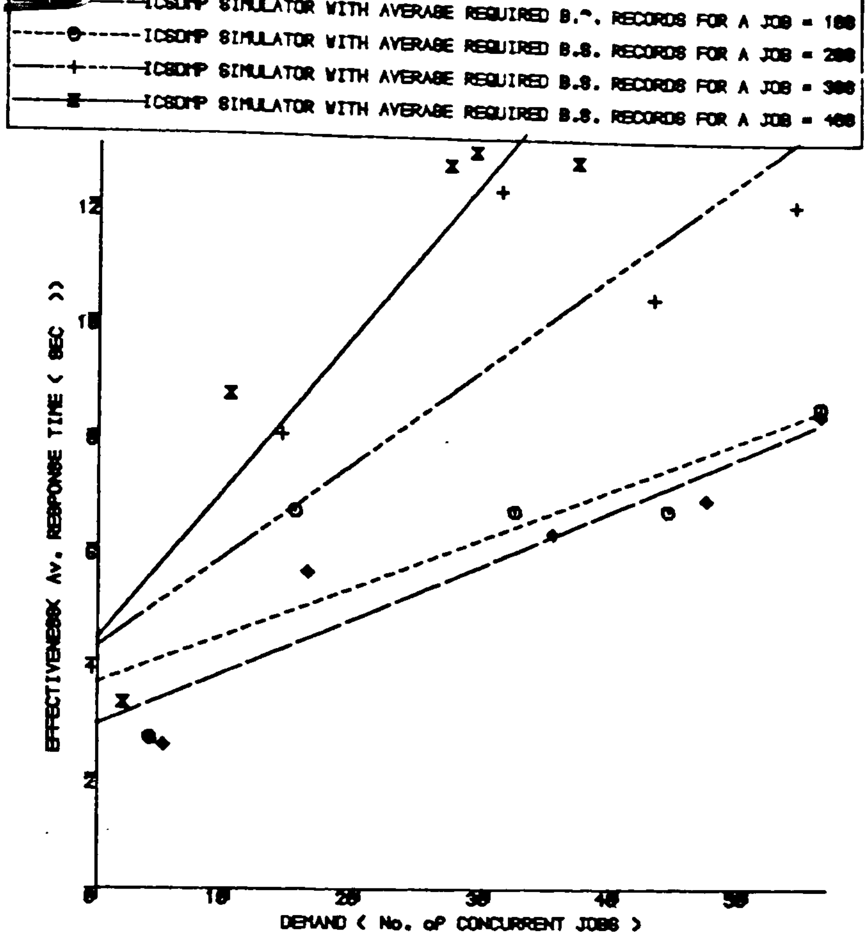


FIGURE 9 : PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(100 records) OBSERVED DEMAND	5	16	35	47	56
OBSERVED EFFECT.	2.53	5.58	6.28	6.89	8.40
LIN PERFORMOACT MODEL :	A=2.88	B=·09	DF=.93		
EXP PERFORMOACT MODEL :	A=2.98	B=·01	DF=.88		
LOG PERFORMOACT MODEL :	A=−·77	B=2.12	DF=.97	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.32	B= ·45	DF=.96		
(200 records) OBSERVED DEMAND	4	15	32	44	56
OBSERVED EFFECT.	2.65	6.65	6.67	6.70	8.54
LIN PERFORMOACT MODEL :	A=3.61	B=·08	DF=.84		
EXP PERFORMOACT MODEL :	A=3.46	B=·01	DF=.80		
LOG PERFORMOACT MODEL :	A= ·43	B=1.89	DF=.93	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.72	B= ·39	DF=.92		
(300 rec OBSERVED DEMAND	4	14	31	43	54
CON) OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B= ·16	DF=.85		
EXP PERFORMOACT MODEL :	A=3.81	B=·02	DF=.81		
LOG PERFORMOACT MODEL :	A= −1.90	B= 3.61	DF=.96	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.38	B= ·57	DF=.95		
(400 records) OBSERVED DEMAND	2	10	27	29	37
OBSERVED EFFECT.	3.26	8.68	12.73	12.97	12.82
LIN PERFORMOACT MODEL :	A=4.38	B=·27	DF=.93		
EXP PERFORMOACT MODEL :	A=4.21	B=·03	DF=.88		
LOG PERFORMOACT MODEL :	A= ·83	B=3.48	DF=.99	Δ□ * BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 2.47	B= ·48	DF=.98		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 9 : THE EFFECTS OF The Av. Required Backing Store Records UPON THE AVERAGE RESPONSE TIME INDEX

Δ : B-Best
□ : C-Admis

3. Processor context switching time,

4. Disc transfer time, and

5. Drum latency time.

A.10 Experiment-11: Average Segment Size vs Av. Response Time

With an average segment size of 1024 bytes the average response time decreases even when the degree of concurrency increases, but when we increase the average segment size we notice an increasing trend in the average response time highly effected by an increasing slop of the degree of concurrency in the system (see Figure 10 and Table 10).

A.11 Experiment-12: Total Memory Size vs Av. Response Time

When we increased the **total memory size** available to the non residential processes in the system, we noticed a slight change in a lower level of concurrency and the response time starting to increase when the memory size decreases and the concurrency level increases (see Figure 11 and Table 11).

A.12 Experiment-13: Context Switching Time vs Av. Response Time

With the increase of **processor context switching time** (i.e. the processor speed to switch from one process to another), we noticed slight increase in the average response time and the major effective factor seems the concurrency level (see Figure 12 and Table 12).

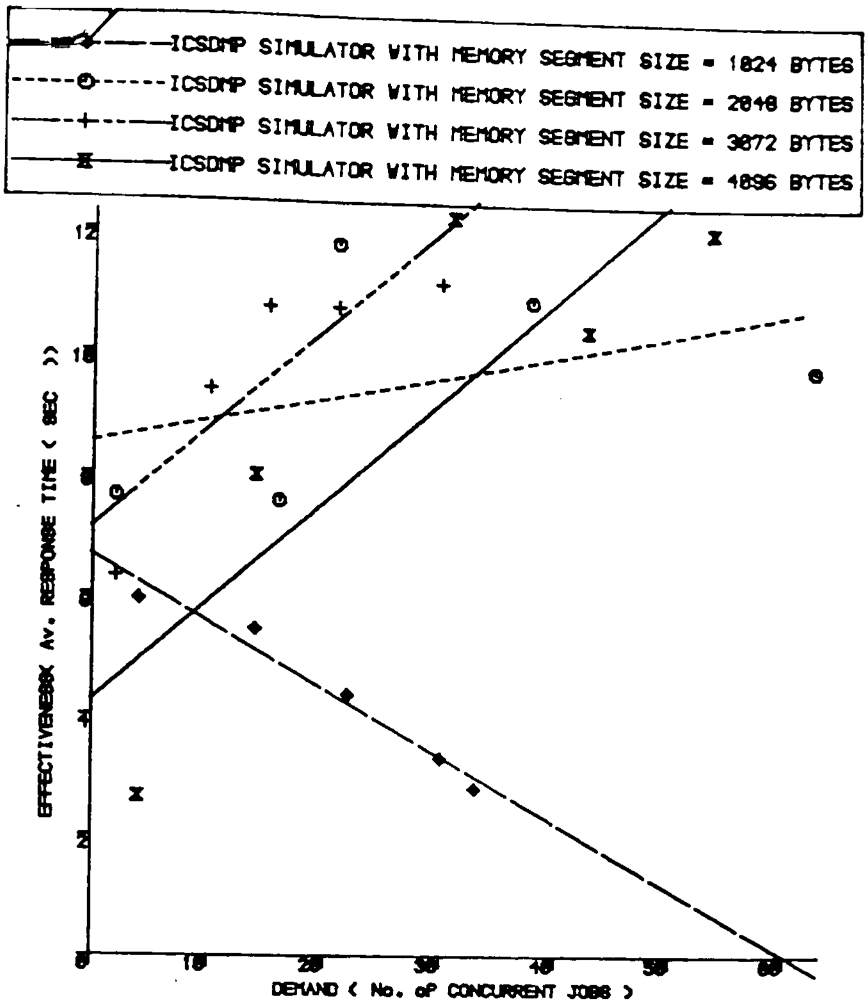


FIGURE 10 : PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(1024 OBSERVED DEMAND	4	14	22	30	33
Bytes)OBSERVED EFFECT.	5.92	5.44	4.36	3.31	2.8
LIN PERFORMOACT MODEL :	A=6.64	B=-.11	DF=.98	*BEST*	
EXP PERFORMOACT MODEL :	A=7.16	B=-.02	DF=.96		
LOG PERFORMOACT MODEL :	A=8.25	B=-1.38	DF=.89	*ADMIS*	
PWR PERFORMOACT MODEL :	A=10.24	B=-.31	DF=.85		
(2048 OBSERVED DEMAND	2	16	21	38	63
Bytes) OBSERVED EFFECT.	7.63	7.57	11.8	10.9	9.8
LIN PERFORMOACT MODEL :	A=8.51	B=.03	DF=.44		
EXP PERFORMOACT MODEL :	A=8.32	B=.004	DF=.49		
LOG PERFORMOACT MODEL :	A=7.03	B=.87	DF=.60	*ADMIS*	
PWR PERFORMOACT MODEL :	A=7.10	B=.09	DF=.63	*BEST*	
(3072 OBSERVED DEMAND	2	10	15	21	30
Bytes) OBSERVED EFFECT.	6.3	9.4	10.76	10.76	11.18
LIN PERFORMOACT MODEL :	A=7.12	B=.16	DF=.87		
EXP PERFORMOACT MODEL :	A=7.05	B=.01	DF=.84		
LOG PERFORMOACT MODEL :	A=5.11	B=1.87	DF=.99△■	*BEST*	*ADMIS*
PWR PERFORMOACT MODEL :	A=5.52	B=.22	DF=.98		
(4096 Byts OBSERVED DEMAND	4	14	31	43	54
CON) OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B=.16	DF=.85		
EXP PERFORMOACT MODEL :	A=3.81	B=.02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1.90	B=3.61	DF=.96	*BEST*	*ADMIS*
PWR PERFORMOACT MODEL :	A=1.38	B=.57	DF=.95		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^B

Table 10: THE EFFECTS OF The Av. Segment Size Variation UPON THE AVERAGE RESPONSE TIME INDEX

△ : B_Best
□ : B-Adm's

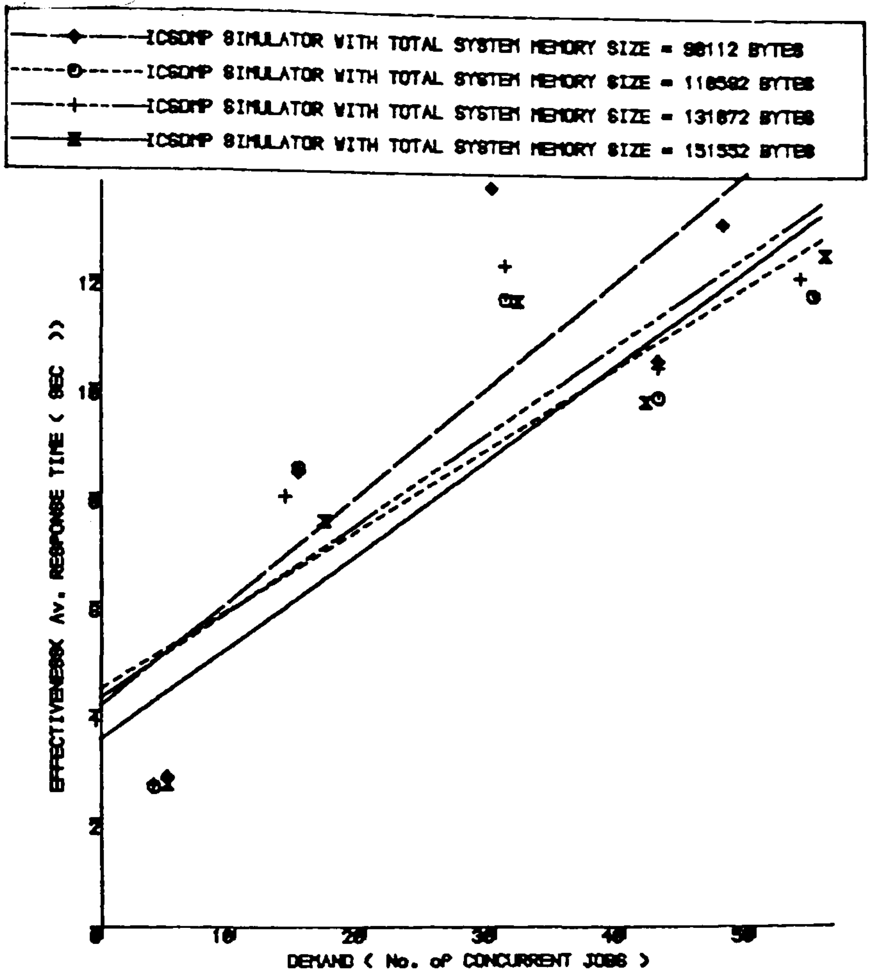


FIGURE 11: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(90112 OBSERVED DEMAND	5	15	30	43	48
Bytes)OBSERVED EFFECT.	2.79	8.44	13.72	10.56	13.10
LIN PERFORMOACT MODEL :	A=4.10	B=·19	DF=.82		
EXP PERFORMOACT MODEL :	A= 3.74	B= ·02	DF=.81		
LOG PERFORMOACT MODEL :	A=-3.63	B=4.35	DF=.92	*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.14	B= ·65	DF=.93	*BEST*	
(110592 OBSERVED DEMAND	4	15	31	43	55
Bytes) OBSERVED EFFECT.	2.61	8.52	11.65	9.87	11.76
LIN PERFORMOACT MODEL :	A=4.41	B= ·15	DF=.82		
EXP PERFORMOACT MODEL :	A=3.85	B= ·02	DF=.78		
LOG PERFORMOACT MODEL :	A=-1.47	B=3.38	DF=.95	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.40	B= ·56	DF=.94		
(131072 B OBSERVED DEMAND	4	14	31	43	54
CON) OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B= ·16	DF=.85		
EXP PERFORMOACT MODEL :	A=3.81	B= ·02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1.90	B=3.61	DF=.960	* BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.38	B= .57	DF=.95		
(151552 OBSERVED DEMAND	5	17	32	42	56
Bytes) OBSERVED EFFECT.	2.65	7.54	11.63	9.79	12.51
LIN PERFORMOACT MODEL :	A=3.48	B= ·17	DF=.89		
EXP PERFORMOACT MODEL :	A=3.44	B= ·02	DF=.84 ▲■		
LOG PERFORMOACT MODEL :	A=-3.59	B= 3.96	DF=.961	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.05	B= ·63	DF=.97		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 11 : THE EFFECTS OF Total Memory Size Variation UPON THE AVERAGE RESPONSE TIME INDEX

▲ : B-Best
□ : B-Admis

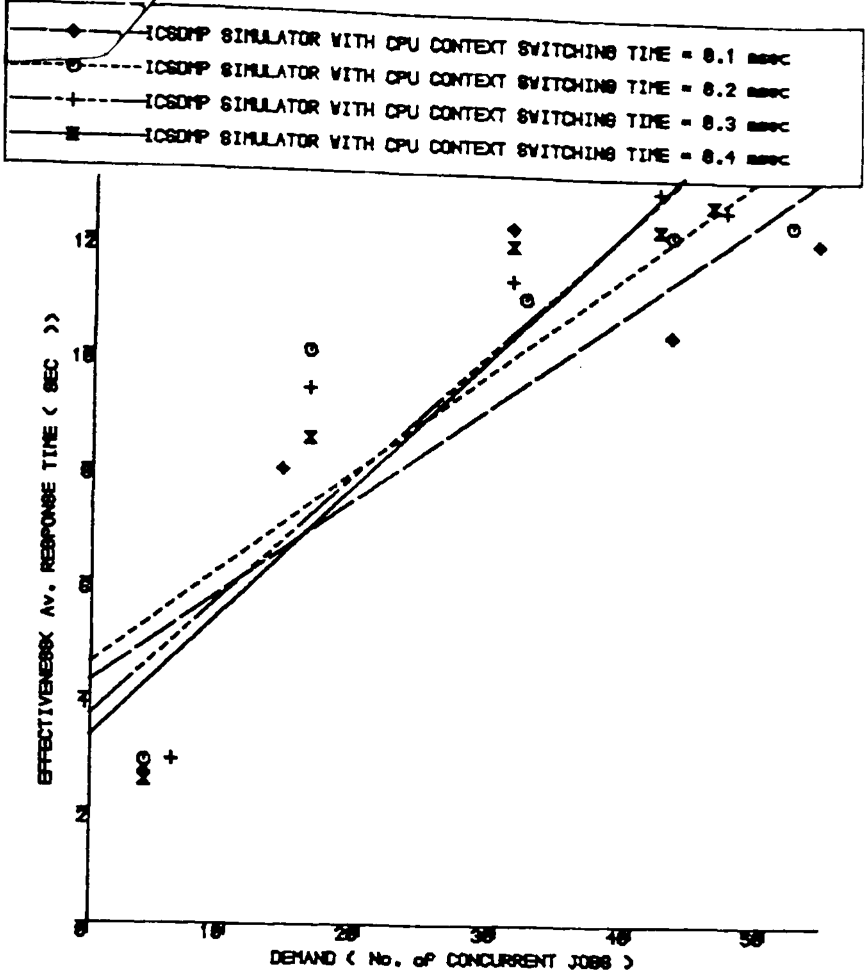


FIGURE 12: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(0.1 msec. OBSERVED DEMAND	4	14	31	43	54
CON)OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B=.16	DF=.85		
EXP PERFORMOACT MODEL :	A=3.81	B=.02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1.90	B=3.61	DF=.96	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.38	B=.57	DF=.95		
(0.2 msec. OBSERVED DEMAND	4	16	32	43	52
) OBSERVED EFFECT.	2.86	10.07	11.05	12.20	12.42
LIN PERFORMOACT MODEL :	A=4.56	B=.17	DF=.86		
EXP PERFORMOACT MODEL :	A=4.03	B=.02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1.65	B=3.70	DF=.97	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.50	B=.57	DF=.95		
(0.3 OBSERVED DEMAND	6	16	31	42	47
msec.) OBSERVED EFFECT.	2.89	9.40	11.35	12.97	12.66
LIN PERFORMOACT MODEL :	A=3.66	B=.21	DF=.90		
EXP PERFORMOACT MODEL :	A=3.63	B=.03	DF=.84		
LOG PERFORMOACT MODEL :	A=-5.04	B=4.77	DF=.98	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=.97	B=.70	DF=.95		
(0.4 OBSERVED DEMAND	4	16	31	42	46
msec.) OBSERVED EFFECT.	2.52	8.54	11.97	12.30	12.76
LIN PERFORMOACT MODEL :	A=3.28	B=.22	DF=.93		
EXP PERFORMOACT MODEL :	A=3.23	B=.03	DF=.87		
LOG PERFORMOACT MODEL :	A=-3.27	B=4.25	DF=.99 △□	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.09	B=.66	DF=.97		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^B

Table 12 : THE EFFECTS OF Context Switching Time UPON
THE AVERAGE RESPONSE TIME INDEX

△ : B- Best
□ : B - Admis

A.13 Experiment-14: Disc Transfer Time vs Av. Response time

Increasing the **disc transfer time**, we noticed the average response time increases and become more higher with the increasing level of concurrency (see Figure 13 and Table 13).

A.14 Experiment-15: Drum Transfer Time vs Av. Response time

Finally, when the **drum latancy time** is increased, the average response time again increased slightly and become higher as the level of concurrency increases (see Figure 14 and Table 14).

Software-based changes vs. Av. response time: The software-dependent parameters changed in this experiment includes:

1. the average number access right passes a user process allowed to pass during the process life time,
2. the processor scheduling policies,
3. the inferential inspectiou period,
4. memory system swapping time, and
5. processor system primitive calling time.

A.15 Experiment-16: No. of passing rights vs. average response

With increasing the **number of rights** a process allowed to pass to other processes (a parameter belong to the memory system (the dynamic memory protection mechanism), the average response time increases in ratio of 1.5 seconds per one extra pass (refer to Figure 15 and Table 15).

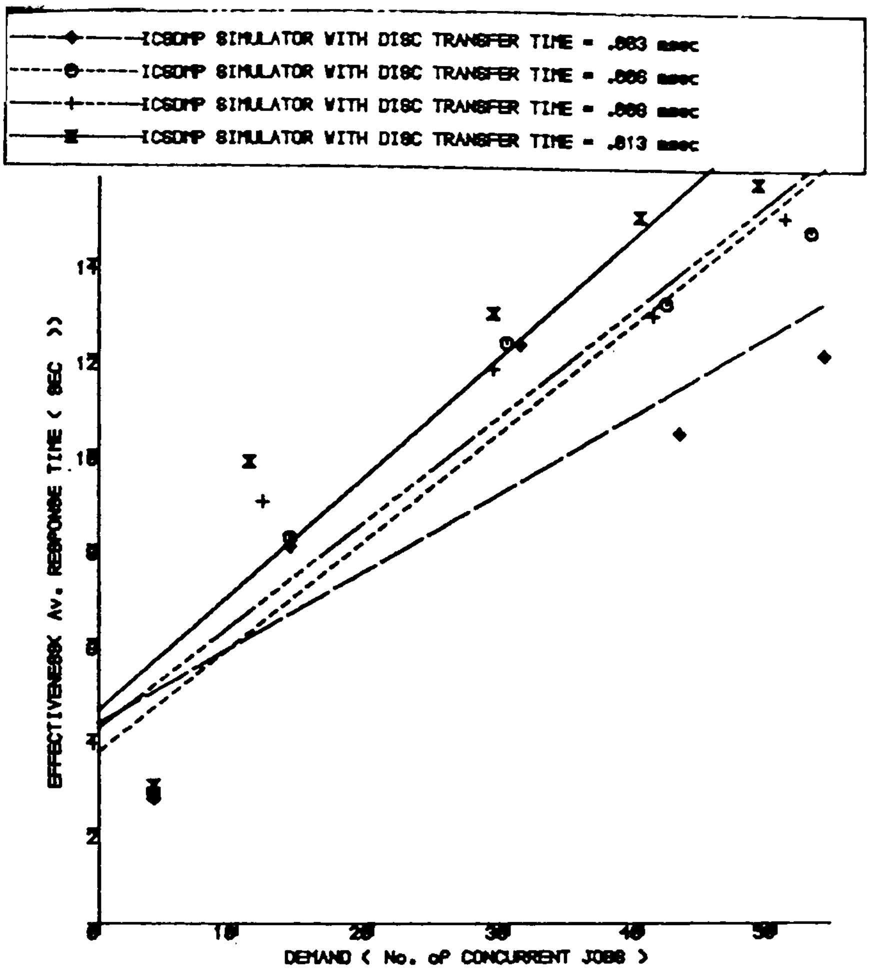


FIGURE 13: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(0.0033 msec OBSERVED DEMAND	4	14	31	43	54
CON) OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B= ·16	DF=.85		
EXP PERFORMOACT MODEL :	A= 3.81	B= ·02	DF=.81		
LOG PERFORMOACT MODEL :	A= -1.90	B= 3.61	DF=.96	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.38	B= ·57	DF=.95		
(0.0066 OBSERVED DEMAND	4	14	30	42	53
msec) OBSERVED EFFECT.	2.71	8.17	12.31	13.17	14.7
LIN PERFORMOACT MODEL :	A= 3.64	B=·22	DF=.94		
EXP PERFORMOACT MODEL :	A= 3.69	B=·03	DF=.87 △□		
LOG PERFORMOACT MODEL :	A=-3.77	B= 4.61	DF=.991	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.21	B=·65	DF=.98		
(0.0099 OBSERVED DEMAND	4	12	29	41	51
msec) OBSERVED EFFECT.	2.88	8.93	11.75	12.91	14.99
LIN PERFORMOACT MODEL :	A=4.14	B= ·22	DF=.93		
EXP PERFORMOACT MODEL :	A= 4.07	B= .02	DF=. 85		
LOG PERFORMOACT MODEL :	A= -2.92	B= 4.44	DF=.990	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.45	B= .61	DF=. 96		
(0.0132 OBSERVED DEMAND	4	11	29	40	49
msec) OBSERVED EFFECT.	2.93	9.77	12.94	14.99	15.72
LIN PERFORMOACT MODEL :	A= 4.52	B=.25	DF=.92		
EXP PERFORMOACT MODEL :	A= 4.29	B= ·03	DF=.83		
LOG PERFORMOACT MODEL :	A= -3.27	B= 4.93	DF=.990	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.50	B= ·63	DF=.95		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^B

Table 13 : THE EFFECTS OF Disc Transfer Time Variation UPON
THE AVERAGE RESPONSE TIME INDEX

△ : B-Best
□ : B-Admis

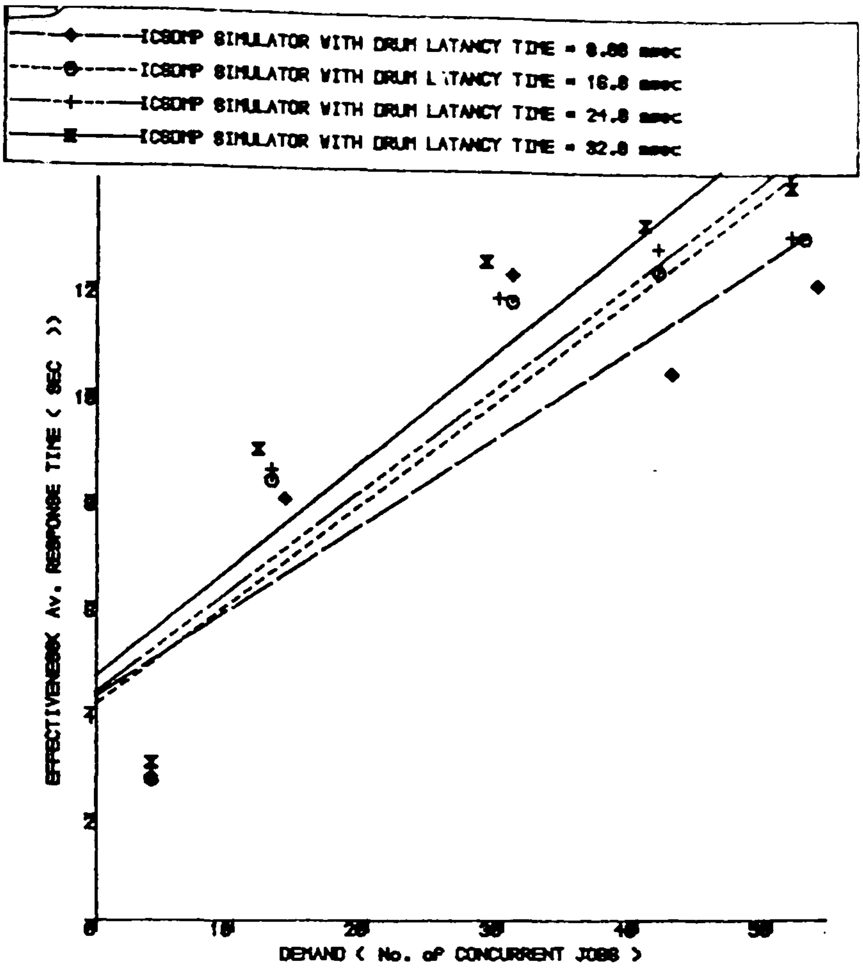


FIGURE 14: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(8·0 msec OBSERVED DEMAND	4	14	31	43	54
CON)OBSERVED EFFECT.	2·65	7·99	12·28	10·43	12·09
LIN PERFORMOACT MODEL :	A=4·24	B=·16	DF=.85		
EXP PERFORMOACT MODEL :	A=3·81	B=·02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1·90	B= 3·61	DF=.96	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·38	B= ·57	DF=.95		
(16·0 OBSERVED DEMAND	4	13	31	42	53
msec) OBSERVED EFFECT.	2·67	8·33	11·77	12·36	12·99
LIN PERFORMOACT MODEL :	A=4·09	B=·19	DF=.91		
EXP PERFORMOACT MODEL :	A=3·81	B=·02	DF=.83		
LOG PERFORMOACT MODEL :	A=-2·54	B= 4·03	DF=.992 △□ *BEST* ADMIS*		
PWR PERFORMOACT MODEL :	A= 1·34	B=·60	DF=.96		
(24·0 OBSERVED DEMAND	4	13	30	42	52
msec.) OBSERVED EFFECT.	2·91	8·54	11·83	12·79	13·03
LIN PERFORMOACT MODEL :	A= 4·31	B=·19	DF=.90		
EXP PERFORMOACT MODEL :	A=4·07	B=·02	DF=.84		
LOG PERFORMOACT MODEL :	A= -2·31	B=4·03	DF=.991	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.50	B=·58	DF=.96		
(32·0 OBSERVED DEMAND	4	12	29	41	52
msec) OBSERVED EFFECT.	2·99	8·92	12·52	13·25	13·96
LIN PERFORMOACT MODEL :	A=4·62	B=·20	DF=.90		
EXP PERFORMOACT MODEL :	A=4·31	B=·02	DF=.82		
LOG PERFORMOACT MODEL :	A=-2·43	B=4·28	DF=.991	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1·57	B=·58	DF=.95		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^B

Table 14: THE EFFECTS OF Drum Latancy Time Variation UPON THE AVERAGE RESPONSE TIME INDEX

△ : B-Best
□ : B-Admis

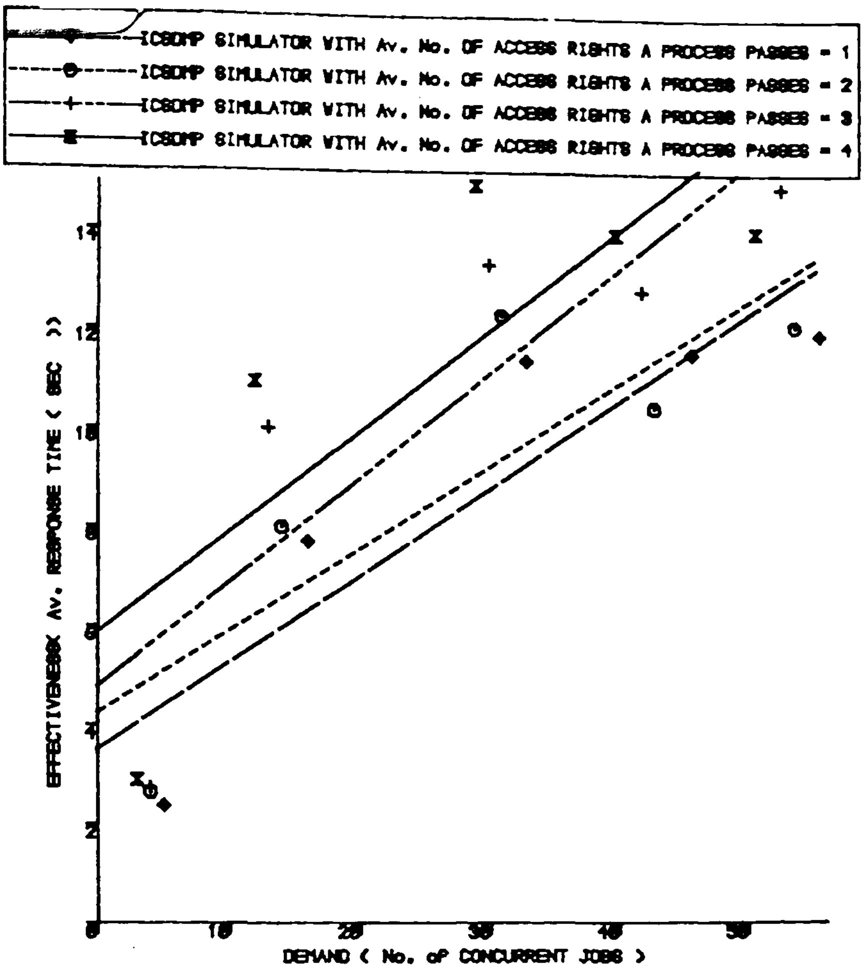


FIGURE 15: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(One pass OBSERVED DEMAND	5	16	33	46	56
)OBSERVED EFFECT.	2.37	7.71	11.37	11.53	11.91
LIN PERFORMOACT MODEL :	A=3.51	B=.17	DF=.90		
EXP PERFORMOACT MODEL :	A=3.29	B=.02	DF=.83		
LOG PERFORMOACT MODEL :	A=-3.88	B=4.08	DF=.98△□	* BEST * ADMIS*	
PWR PERFORMOACT MODEL :	A=.93	B=.67	DF=.96		
(Tow pass OBSERVED DEMAND	4	14	31	43	54
CON) OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B=.16	DF=.85		
EXP PERFORMOACT MODEL :	A=3.81	B=.02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1.90	B=3.61	DF=.96	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.38	B=.57	DF=.95		
(Three pass OBSERVED DEMAND	4	13	30	42	53
) OBSERVED EFFECT.	2.73	9.99	13.31	12.79	14.91
LIN PERFORMOACT MODEL :	A=4.77	B=.21	DF=.87		
EXP PERFORMOACT MODEL :	A=4.20	B=.02	DF=.80		
LOG PERFORMOACT MODEL :	A=-2.71	B=4.46	DF=.97	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.40	B=.62	DF=.94		
(Four Pass OBSERVED DEMAND	3	12	29	40	51
) OBSERVED EFFECT.	2.89	10.93	14.92	13.93	13.99
LIN PERFORMOACT MODEL :	A=5.87	B=.20	DF=.80		
EXP PERFORMOACT MODEL :	A=4.76	B=.02	DF=.76		
LOG PERFORMOACT MODEL :	A=-.62	B=4.10	DF=.95	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.90	B=.56	DF=.93		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = $A*DEMAND^B$

Table 15: THE EFFECTS OF No. of Passes Variations UPON THE AVERAGE RESPONSE TIME INDEX

△ · R_Best
□ : B_Admis

A.16 Experiment-17: Scheduling Policies vs Average Response Time

The **processor system scheduling policy** can be altered using the parameters (mh, th, minf, tinf) of the system loss function mentioned in section (originally developed based upon the scheme of Kleinrock 1970). Round Robin scheduling policy (1, 6000, 1, $3.6 * 10^6$) proved to produce the minimal average response time, followed by the Selfish Round Robin ($0.7, 6000, 0.7, 3.6 * 10^6$), the FCFS ($0.1, 6000, 0.1, 3.6 * 10^6$), and the Bulk Service ($0, 6000$, 0, $3.6 * 10^6$). This means for an optimal response a Round Robin policy should be adopted (refer Figure 16 and Table 16).

A.17 Experiment-18: Inferential Inspection Period vs A.R.T.

The **inferential inspection period** represent the processor system time slice. With the increase of this period, we notice an increase in the average response time becoming more higher with higher levels of concurrency (see Figure 17 and Table 17).

A.18 Experiment-19: Swapping Speed vs Average Response Time

With the increase of the **memory system swapping speed** or time with the backing store, we noticed an increase in the average response time (0.5 sec for 0.5 msec speed increase) becoming more higher as the level of concurrency increases (see Figure 18 and Table 18).

A.19 Experiment-20: Processor Speed vs Average Response Time

With the decrease of the processor system speed in invoking the primitives (system subroutines), we noticed a decrease in the average response time becoming a sharper decrease as the level of concurrency decreases (see Figure 19 and Table 19).

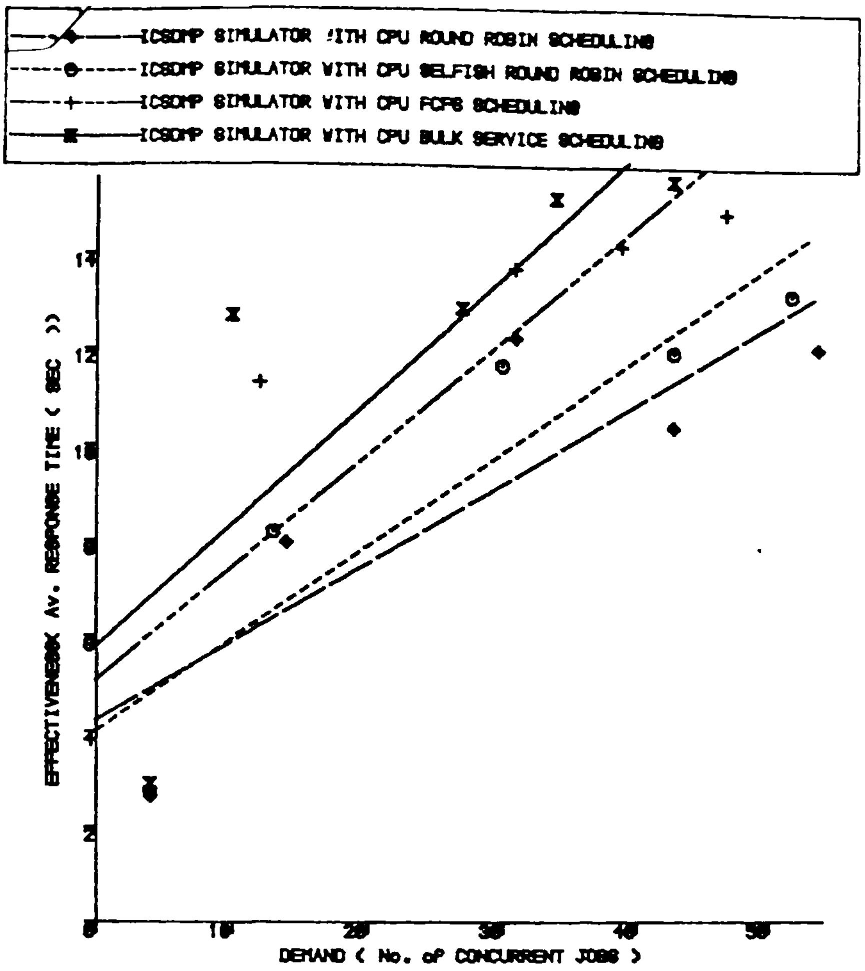


FIGURE 16: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(Round Robin OBSERVED DEMAND	4	14	31	43	54
CON)OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B=.16	DF=.85		
EXP PERFORMOACT MODEL :	A=3.81	B=.02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1.90	B=3.61	DF=.96	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.38	B=.57	DF=.95		
(Selfish R.R. OBSERVED DEMAND	4	13	30	43	52
) OBSERVED EFFECT.	2.70	8.21	11.71	11.99	13.20
LIN PERFORMOACT MODEL :	A=4.12	B=.19	DF=.91		
EXP PERFORMOACT MODEL :	A=3.81	B=.02	DF=.84		
LOG PERFORMOACT MODEL :	A=-2.54	B=4.01	DF=.99△□	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.36	B=.60	DF=.96		
(FCFS OBSERVED DEMAND	4	12	31	39	47
) OBSERVED EFFECT.	2.83	11.32	13.73	14.21	14.92
LIN PERFORMOACT MODEL :	A=5.08	B=.23	DF=.86		
EXP PERFORMOACT MODEL :	A=4.33	B=.03	DF=.79		
LOG PERFORMOACT MODEL :	A=-2.48	B=4.6	DF=.96	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.49	B=.63	DF=.92		
(Bulk OBSERVED DEMAND	4	10	27	34	43
Service) OBSERVED EFFECT.	2.91	12.71	12.91	15.21	15.59
LIN PERFORMOACT MODEL :	A=5.78	B=.25	DF=.81		
EXP PERFORMOACT MODEL :	A=4.74	B=.03	DF=.75		
LOG PERFORMOACT MODEL :	A=-1.77	B=4.77	DF=.91	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.69	B=.63	DF=.88		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = $A*DEMAND^B$

Table 6: THE EFFECTS OF CPU Scheduling Policies UPON
THE AVERAGE RESPONSE TIME INDEX

△ : E - Best
□ : B - Admis

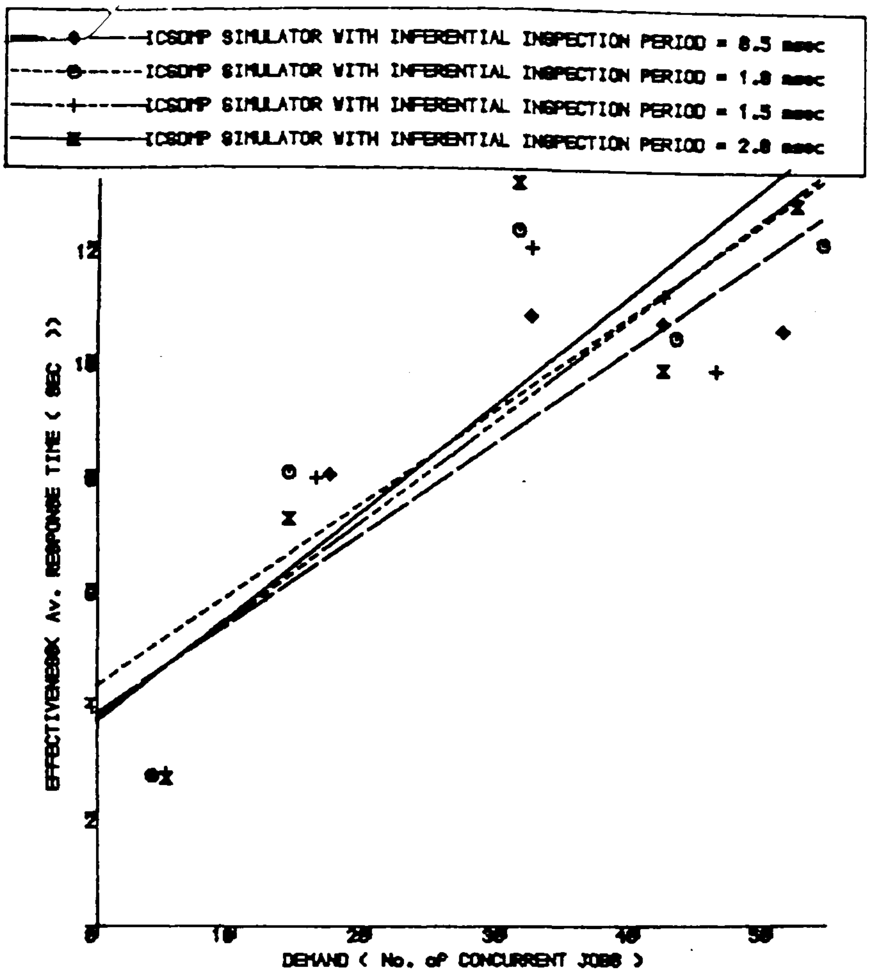


FIGURE 7: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(0.5 msec) OBSERVED DEMAND	4	17	32	42	51
OBSERVED EFFECT.	2.65	7.96	10.78	10.68	10.56
LIN PERFORMOACT MODEL :	A=3.76	B=.16	DF=.98		
EXP PERFORMOACT MODEL :	A=3.47	B=.02	DF=.84		
LOG PERFORMOACT MODEL :	A=-1.70	B= 3.33	DF=.98 △□	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.32	B= .56	DF=.97		
(1.0 msec CON) OBSERVED DEMAND	4	14	31	43	54
OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B=.16	DF=.85		
EXP PERFORMOACT MODEL :	A= 3.81	B= .02	DF=.81		
LOG PERFORMOACT MODEL :	A= -1.90	B= 3.61	DF=.96	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.38	B= .57	DF=.95		
(1.5 msec) OBSERVED DEMAND	5	16	32	42	46
OBSERVED EFFECT.	2.7	7.9	11.97	11.16	9.85
LIN PERFORMOACT MODEL :	A=3.68	B=.17	DF=.83		
EXP PERFORMOACT MODEL :	A=3.41	B= .02	DF=.82		
LOG PERFORMOACT MODEL :	A=-2.90	B= 3.76	DF=.93	*ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.12	B= .62	DF=.94	*BEST*	
(2.0 msec.) OBSERVED DEMAND	5	14	31	42	52
OBSERVED EFFECT.	2.61	7.89	13.11	9.86	12.77
LIN PERFORMOACT MODEL :	A=3.61	B=.19	DF=. 84		
EXP PERFORMOACT MODEL :	A= 3.47	B=.02	DF=. 83		
LOG PERFORMOACT MODEL :	A= -3.86	B= 4.22	DF=. 92	* ADMIS*	
PWR PERFORMOACT MODEL :	A= 1.04	B= .65	DF=.94	* BEST*	

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^B

Table 7 : THE EFFECTS OF The Inferential Inspection Period UPON THE AVERAGE RESPONSE TIME INDEX

△ : B_Best
□ : B_Admis

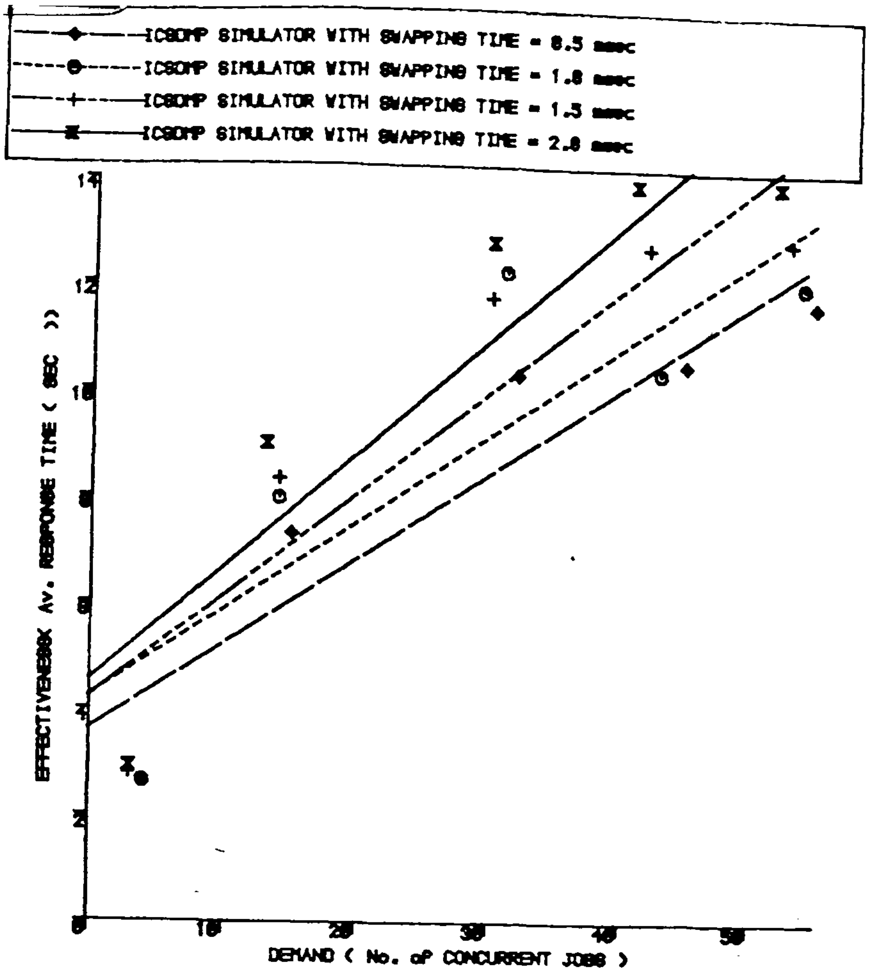


FIGURE 8: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25
(0.5 OBSERVED DEMAND	4	15	32	45	55
msec.)OBSERVED EFFECT.	2.66	7.33	10.35	10.59	11.73
LIN PERFORMOACT MODEL :	A=3.62	B=.16	DF=.92		
EXP PERFORMOACT MODEL :	A=3.53	B=.02	DF=.86		
LOG PERFORMOACT MODEL :	A=-1.99	B=3.42	DF=.991	*BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.34	B=56	DF=.97		
(0.1 mSec. OBSERVED DEMAND	4	14	31	43	54
CON) OBSERVED EFFECT.	2.65	7.99	12.28	10.43	12.09
LIN PERFORMOACT MODEL :	A=4.24	B=.16	DF=.85		
EXP PERFORMOACT MODEL :	A=3.81	B=.02	DF=.81		
LOG PERFORMOACT MODEL :	A=-1.90	B=3.61	DF=.96	*BEST*ADMIS*	
PWR PERFORMOACT MODEL :	A=1.38	B=.57	DF=.95		
(1.5 OBSERVED DEMAND	3	14	30	42	53
msec.) OBSERVED EFFECT.	2.80	8.36	11.79	12.77	12.91
LIN PERFORMOACT MODEL :	A=4.22	B=.19	DF=.91		
EXP PERFORMOACT MODEL :	A=3.94	B=.02	DF=.85		
LOG PERFORMOACT MODEL :	A=-1.20	B=3.67	DF=.992	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.67	B=.54	DF=.98		
(2.0 OBSERVED DEMAND	3	13	30	41	52
msec.) OBSERVED EFFECT.	2.93	9.01	12.83	13.95	13.99
LIN PERFORMOACT MODEL :	A=4.54	B=.21	DF=.91		
EXP PERFORMOACT MODEL :	A=4.20	B=.02	DF=.84		
LOG PERFORMOACT MODEL :	A=-1.41	B=4.05	DF=.991	* BEST* ADMIS*	
PWR PERFORMOACT MODEL :	A=1.75	B=.56	DF=.97		

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 9: THE EFFECTS OF Memory Swapping Time Variation UPON THE AVERAGE RESPONSE TIME INDEX

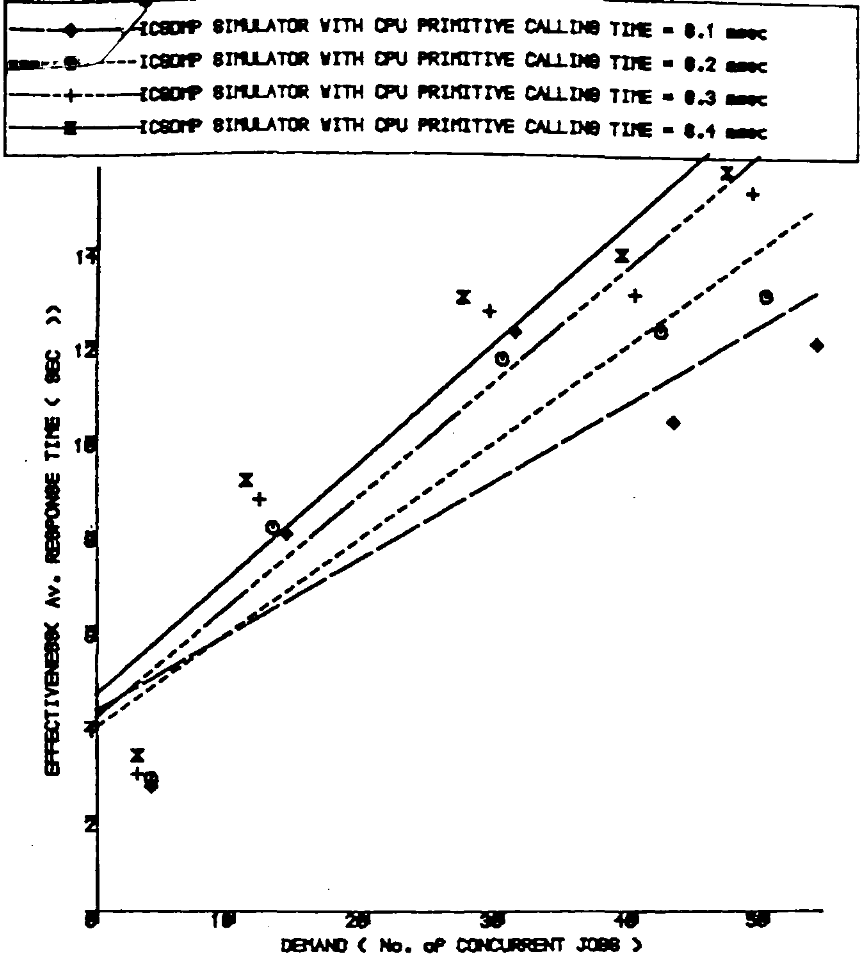


FIGURE 1C: PERFORMOACT MODELLING: DEMAND vs EFFECTIVENESS

MIN	5	10	15	20	25	
(0·1 msec. OBSERVED DEMAND	4	14	31	43	54	
CON)OBSERVED EFFECT.	2·65	7·99	12·28	10·43	12·09	
LIN PERFORMOACT MODEL :	A= 4·24	B=·16	DF=.85			
EXP PERFORMOACT MODEL :	A= 3·81	B= ·02	DF=.81			
LOG PERFORMOACT MODEL :	A= -1·90	B=3·61	DF=.96			*BEST* ADMIS*
PWR PERFORMOACT MODEL :	A= 1·38	B= ·57	DF=.95			
(0·2 OBSERVED DEMAND	4	13	30	42	50	
msec.) OBSERVED EFFECT.	2·81	8·1	11·7	12·33	13·11	
LIN PERFORMOACT MODEL :	A=3·89	B= ·20	DF=.92			
EXP PERFORMOACT MODEL :	A= 3·80	B= ·02	DF=.86			
LOG PERFORMOACT MODEL :	A= -2·63	B= 4·07	DF=.991			*BEST* ADMIS*
PWR PERFORMOACT MODEL :	A= 1·38	B= ·60	DF=.97			
(0·3 OBSERVED DEMAND	3	12	29	40	49	
msec.) OBSERVED EFFECT.	2·91	8·7	12·7	13·09	15·27	
LIN PERFORMOACT MODEL :	A=4·10	B=·24	DF=.94			
EXP PERFORMOACT MODEL :	A= 4·03	B= ·03	DF=.87			
LOG PERFORMOACT MODEL :	A= -1·78	B= 4·23	DF=.993 △□			*BEST* ADMIS*
PWR PERFORMOACT MODEL :	A= 1·70	B= ·57	DF=.98			
(0·4 OBSERVED DEMAND	3	11	27	39	47	
msec.) OBSERVED EFFECT.	3·3	9·1	12·99	13·93	15·72	
LIN PERFORMOACT MODEL :	A= 4·58	B= ·25	DF=.94			
EXP PERFORMOACT MODEL :	A= 4·52	B= ·02	DF=.87			
LOG PERFORMOACT MODEL :	A=-1·44	B= 4·35	DF=.990			*BEST* ADMIS*
PWR PERFORMOACT MODEL :	A= 2·00	B= ·55	DF=.98			

Where : THE NUMBER OF CONCURRENT JOBS REPRESENTING THE DEMAND
THE AV. RESPONSE TIME REPRESENTING EFFECTIVENESS
LINEAR PERFORMOACT MODEL = EFF = A + B * DEMAND
EXPONENTIAL PERFORMOACT MODEL= EFF = A EXP(B* DEMAND)
LOGARITHMIC PERFORMOACT MODEL= EFF = A +B* LIN(DEMAND)

POWER PERFORMOACT MODEL = EFF = A*DEMAND^{B}

Table 1: THE EFFECTS OF CPU Primitive Calling Time Variation UPON
THE AVERAGE RESPONSE TIME INDEX

△ : B-Best
□ : B-Admis