# A METHOD FOR SPECIFYING COMPLEX REAL-TIME SYSTEMS WITH APPLICATION TO AN EXPERIMENTAL VARIABLE STABILITY HELICOPTER

by

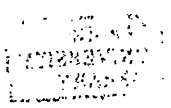
**Roy Bradley, B.Sc., M.Sc., F.I.M.A., C.Math.**

Dissertation submitted to the Faculty of Engineering, University of Glasgow, for the Degree of Doctor of Philosophy

April 1992

## CONTENTS





## Acknowledgements

I wish to thank the following persons for their advice and encouragement during the period of the research: Dr Gareth Padfield of the Flight Systems Department, Defence Research Agency for his unflagging enthusiasm for the ACT Lynx project, Mr John Cameron of the erstwhile Michael Jackson Systems Ltd. for his inspirational course and valuable discussions on Jackson methods, and Mr Peter Wright of Westland Helicopters Ltd. for his patient explanations of helicopter systems.

## Abstract

Engineering systems increasingly contain a significant element of embedded software. The specification of such systems causes problems because of the diversity of the sub-systems which they contain. For example, in modern aerospace systems a combination of mechanical, electrical, hydraulic and digital sub-systems need to function together in a safety-critical manner. The need is for a uniform means of specification which spans the whole diversity of sub-systems and which serves both to verify and to validate the functional aspects of the total system.

Verification is concerned with the consistency and completeness of the specification, while validation is a broader concept concerned with fitness-for-purpose, which is the ultimate requirement. The validation of large systems causes difficulties because of the variety of specialists who need to be satisfied of a system's acceptability; reliability engineers, pilots and actuator engineers are among those who may require different views of the system in order to perform their validation, yet underlying the whole system there must be a single definitive specification.

The development of a specification method for an active flight control system for a variable stability research helicopter is described. Software engineering techniques and computer based tools are employed to specify the whole system - not merely the embedded software. The software specification method used is Jackson System Development, which for this application needs to be enhanced by the concept of 'units', so that a large system is reduced first to a network of loosely coupled units with a restricted type of intercommunication. The decomposition of a system into units produces a significant reduction of complexity and leads naturally into a further decomposition based on redundancy, and to support by additional software engineering tools.

The reduction in complexity accruing from the unit approach has the benefit of highlighting several important design issues which include the fault management strategy and the integrity of system control. The software engineering approach develops a design which has integral consistency checks for verification and which allows a direct translation into a working simulation of the proposed system. In this context the value of automatic code generation is emphasised. The resulting simulation contributes to validation because non software-specialists can effectively interact with the specification to investigate its acceptability. The disciplined development of a specification which integrates text, JSD design and simulation is termed a 'living specification'.

## Abbreviations

There are a number of abbreviations used in the text which avoid repetition of often lengthy phrases. For reference, a complete list is given below.

| | |
|---|---|
| ACP | Activity/ Channel/ Pool |
| ACT | Active Control Technology |
| ADME | Actuator Drive and Monitoring Element |
| ADMU | Actuator Drive and Monitoring Unit |
| ADSE | Air Data Sensing Element |
| AE | Actuator Element |
| AFCS | Automatic Flight Control System |
| AMSE | Aircraft Motion Sensing Element |
| AWACS | Airborne Warning and Control System |
| BITE | Built in Test Equipment |
| CASE | Computer Assisted Software Engineering |
| CLE | Control Law Element |
| CLISE | Control Law Input Support Element |
| CLOSE | Control Law Output Support Element |
| CORE | Controlled Requirements Expression |
| CSE | Crew Station Element |
| DRA | Defence Research Agency |
| DS | Data Stream |
| ESSE | External System Support Element |
| FCC | Flight Control Computer |
| FOFS | Fail-Operative/ Fail-Safe |
| FSM | Finite State Machine |
| HDD | Head Down Display |
| HOOD | Hierarchical Object Oriented Design |
| HUD | Head Up Display |
| IDA | Intercommunication Data Area |
| JSD | Jackson System Development |
| JSP | Jackson Structured Progamming |
| JWB | Jackson Work Bench |
| LBMS | Learmonth Burchett Management Systems |
| MASCOT | Modular Approach to Software Construction, Operation and Test |
| MOD | Ministry of Defence |
| MODAS | Modular Data Acquisition System |
| MP | Menu Panel |
| MSP | Mode Select Panel |
| NOE | Nap of the Earth |
| OO | Object Oriented |
| PCP | Pilots Control Panel |
| PDF | Program Design Facility |
| PFCU | Primary Flight Control Unit |
| RAE | Royal Aerospace Establishment |
| RP | Repeater Panel |
| SADME | Series Actuator Drive and Monitoring Element |
| SADMU | Series Actuator Drive and Monitoring Unit |
| SADT | Structured Analysis and Design Technique |
| SE | Sensor Element |
| SE | Structured English |
| SID | System Implementation Diagram |
| SND | System Network Diagram |
| STD | State Transition Diagram |
| SVI | State Vector Inspection |
| WHL | Westland Helicopters Ltd |

# CHAPTER 1

# INTRODUCTION

## 1.1 AIMS OF THE RESEARCH

The aim of the research described in this thesis is to develop a method for the specification of complex real-time systems and to demonstrate its application to a representative engineering system. Some of the terminology in the aim, as stated, is particular to this research therefore it is apposite to expand briefly on the terms used in the statement above. The first to be considered is "specification"; a specification is a statement of what is required - posed in such a way that it can be implemented, that is, constructed if it is a piece of hardware such as an actuator, or coded if it is software - although in this context even the latter must have have its final manifestation as an embedded system on a piece of hardware. There are two major problems which can arise in this area. The first is that the final delivered system may not be what the specifier actually wanted - in other words the specification is deficient. The terms often used to avoid this situation is that the specification must be complete and unambiguous. The second problem is that it may not be possible to build the system as specified at all. There are aspects of completeness here but the most useful classification is that the specification is some general way contradictory when it should be consistent. The theme of how to develop a method which results in a complete, unambiguous, consistent maintainable specification was a persistent one throughout the programme of research.

A "complex system" in the context of this work is one that has major elements that are not digital, that is, are not dependent on software. No satisfactory general terminology has been discovered to describe a system which may involve hydraulic components, mechanical levers, electrical actuators and relays, analogue electronics and even human operators. The terms "mixed-motivator", "dissimilar", "mixed-media" have been among the many candidates discussed and discarded. Complexity can arise in different ways, of course, but within the confines of this work there is no confusion in using the term in this specialised manner. In particular, the emphasis is on systems which have a significant digital component embedded within them, so that software specification is a major factor.

The term "real-time" had only one meaning until recently , and that was that all calculated quantities within a system had to be updated at a pace that kept step with the

real world, and that there should be no accumulating backlog of information. The classical application is a ground-based, piloted flight simulator where the positional and attitude information must be delivered to the motion system at a rate representative of a real aircraft. A real-time application closer to the current study is an aircraft digital autostabiliser. Unfortunately the term has been hi-jacked by the data-processing fraternity who use it to refer to reservation systems and similar applications where there is no precise response-time requirement so that a better terminology here would be "while-you-wait". Accordingly the alternative terminology "time-critical" has grown up in the aerospace area. In the present study there are elements of the two interpretations. The software for the control laws and autostabiliser are real-time in the time-critical sense, whereas the control law and disturbance selections do not have response deadlines of the same kind.

As will be described more fully in the remainder of this thesis, the need for precision and integrity directed the development of the specification method towards the techniques of software engineering. However another factor, that of accessibility, influenced its development but received little formal acknowledgement during the course of the work. The need for precision drives the specification towards greater formality but it is that very formality which begins to deny access to the traditional engineer. The challenge is to find a method of description which can be accepted by a wide cross section of engineers and yet contain within it sufficient formality to dictate the details of implementation. Graphical representations can be a valuable supplement in this context and they played a major role in the development of the method presented herein.

The method and its supporting techniques have been developed to have general applicability but the need to specify an Active Control Technology (ACT) system for a Westland Lynx helicopter stimulated the research in the first instance and it is to this complex system that the method has been applied. In the discussion to be found in later chapters it will become clear that the ACT Lynx system is a demanding application and a stern test for any specification method.

The use of the term "methodology" to describe a harmonious combination of isolated techniques, methods and philosophy to form a procedure identifiable in its own right can be irritating to those who prefer to retain its proper meaning as a study of methods. Therefore the latter meaning will be used in this thesis and "method" will have a broader meaning to include aspects of philosophy, approach, and extensions to established techniques.

## 1.2 STRUCTURE OF THE THESIS.

There is a strong narrative theme to the development of the ACT Lynx Requirements Specification which needs to be appreciated for a full understanding of the problems encountered and solved in the course of this research. A concise account has been presented in References 1 and 2. However, it would not be appropriate to base the format of a thesis on this narrative thread, so the course adopted in this work has been to place the narrative in section 1.4 of this introduction after a discussion of the background of the project. The technical issues raised in the course of the narrative are then discussed individually in subsequent chapters and, where appropriate, in more detailed appendices. It is necessary to introduce some preparatory material and devote Chapter 2 to the Jackson System Development (JSD) method for software system development. A full tutorial would be out of place, but it is necessary to draw out those features of the method which have particular relevance to the theme of this work. Consequently Chapter 3 is concerned with the early activity of creating a representation of the ACT Lynx system which could be used to plan subsequent development work. The control of the ACT system is discussed in Chapter 4. During the project the design of the software behind the pilot's various interfaces for controlling the system became known as "the supervisor issue"; the problems associated with it and their resolution have general applicability. Chapter 5 describes the essence of the method adopted for the application of JSD to a complex system. It is concerned with the rationale of the method, which is at heart very simple, as applied to the ACT Lynx system. Its application was a substantial task and the content and structure of the resulting specification is described in Chapter 6. In practical terms, the end product of this stage was a written, structured specification with much of the JSD underpinning quite separate. The development of a JSD specification using CASE tools and a parallel development of an Ada simulation is described in Chapter 7. Several of the components of the Ada simulation are mentioned only in outline, with details being confined to specific appendices. Issues surrounding the incorporation of a semi automatic system test are discussed separately in Appendix 4. The combination of a structured specification, in text, implemented as an Ada simulation using JSD as a common expression of the design is the practical target of this stage of the ACT Lynx project. The achievement is summarised in Chapter 8, followed in Chapter 9 by critical evaluation of those achievements. The evaluation leads to some proposals for future work, both in the particular area of the application and on the method in general.

## 1.3 BACKGROUND

The specification method was developed in response to the needs of the ACT Lynx project, which has as its aim the procurement of an experimental variable stability helicopter since in-flight simulation provides the ultimate validation test of a new flight control concept. The realism of flight test overcomes the deficiencies of ground based simulation associated with cue fidelity and modelling inaccuracies. On the other hand, cost and safety issues constrain what is achievable in experimental flight test. A balance must be struck between ground and flight test in order to mature a control concept fully. In the field of helicopter flight control and handling qualities, the potential benefits offered by Active Control Technology (ACT) are considerable [3] and results derived from ground and in-flight simulation in Europe and North America have demonstrated benefits at moderate performance levels. Future military rotorcraft will need to operate at considerably higher performance levels and in tougher environments than currently achievable. Additionally, reduced manning may be imposed in order to save weight and logistical support costs. To support the development of appropriate handling criteria, carefree manoeuvring features, and the associated technologies in controls and displays, a number of research laboratories are exploring the options for enhanced in-flight facilities. In the UK, at the Royal Aerospace Establishment (RAE), now the Defence Research Agency (DRA), attention has been focussed on studies for the development of an ACT system for a research Lynx. Studies include the development of a rationale for a programme of research [4,5], and establishing the procedures for managing the life-cycle of control laws from initial design through to flight [6,7]. Particularly relevant to the present work was a series of studies to demonstrate the practical feasibility of modifying the RAE AH7 Lynx into a full authority flight-test facility [8]. The concept is illustrated in Figure 1.1. The helicopter is flown by one of two pilots - the safety pilot and the evaluation pilot. When the ACT system is not engaged, the safety pilot controls the helicopter using the conventional inceptors and mechanical control runs ( 'inceptors' is a generic term used in the UK for controls: for example, levers and joy-sticks). When the system is engaged, the evaluation pilot flies the helicopter by using a side-stick controller which provides inputs into the Flight Control Computer (FCC). The demands produced by the FCC are taken to an Actuator Drive and Monitoring Unit (ADMU) which generates the drive signals for the parallel actuators which are connected to the mechanical control runs. The system is engaged by energising the parallel actuators and, when energised, the parallel actuator back-drives the safety pilot's inceptors and at the same time forward-drives the existing Lynx Primary Flight Control Units (PFCUs). There is provision for filtering out the high frequency component of the demands and taking it directly to the series actuator of the PFCU in order to avoid a high frequency

component in the backdriven control runs. The triplex architecture of the FCC and the dual duplex nature of the ADMU ensure a fail-operative / fail-safe (FOFS) system. The FOFS architecture is necessary to provide safe experimental flight at the edges of the flight performance envelope and in the nap-of-the-Earth (NOE) environment.

The impact of this total functionality on the system requirements is considerable and RAE identified a need for a precise yet versatile specification of the system. The specification should address functionality (for normal and failed states), operation of the integrated system, together with interfaces, constraints and testing requirements. It should also be fit for establishing realistic development costs and timescales. This last consideration introduces another facet of the background which was important in influencing the direction in which the project developed. At this time the Ministry of Defence had had unfortunate experiences as a procurer for major projects, particularly when a significant element of novel software was involved. There was a history of cost over-runs and non-completions which culminated in the AWACS / Nimrod fiasco. The MOD have subsequently responded to this situation by imposing a strict regime for the development of new systems. This climate extended to the research establishments and from the outset RAE were determined that the specification should be the basis of a well managed procurement exercise, and as such, should solve all of the outstanding design issues of the system. Potential suppliers would then be able to assess accurately the costs of supplying the various components of the system, since the possibility of being involved in expensive open-ended design work would be eliminated. Also by solving the outstanding design problems *ab initio*, RAE would be sure that the system could actually be supplied in accordance with the specification.

Therefore, as RAE prepared to place a contract for the production of a requirements specification, they were clear about what was needed from the specification, but were uncertain on how it was to be achieved. It was clearly necessary to provide some guidance for the contractors about the approach to be taken.

## 1.4 DEVELOPMENT OF THE SPECIFICATION.

An area where the preparation of specifications has received a great deal of attention is that of software. The production of complete, unambiguous specifications and their managed development into functioning systems raised the possibility of using similar disciplines on complex systems such as the ACT Lynx. The techniques of software engineering would ensure a rigorous development of the design and the associated CASE tools would assist in maintaining the precision and integrity of the specification.

Once the introduction of software engineering techniques into the project had been argued as a potential method of providing the specification with verification, it was of interest to survey what methods were available to the systems designer, bearing in mind that what was contemplated was an extension, in some manner as yet unknown, into the non-software parts of the system. So at the very least the method to be selected must be compatible with those parts even if it had little design input to offer. There is a wide range of design methods competing for the attention of the system developer: Freeman [9] lists 24 techniques and the Department of Industry [10] lists 21 potential candidates. A particularly useful contribution is provided by Birrel and Ould [11] who apply a selection of methods to an image processing system in a comparative case study. Their criteria for a short list include:

(a) Public availability of sufficient information to allow potential users make an initial value judgement.

(b) There must be commercial support or substantial coverage in the literature.

The issue of public availability is an important one. One may sympathise with the commercial need for confidentiality, but design is concerned with communicating a way of solving problems and an 'open systems' approach to methodology would benefit the whole software development community. The commercial aspects could then be satisfied by consultancy expertise and by the marketing of CASE products. To the criteria above, one may add for the present application:

(c) The method of expressing the design must easily make sense to a wide community of engineers - not solely the software personnel.

(d) Adequate CASE support for large systems is vital.

(e) There must be sufficient detail required by the method to ensure proper verification.

The scope for the method must include the definition, design and production phases, where the last named phase largely satisfies criterion (e). The phases are not to be interpreted too dogmatically since there can be some overlap of phases between methods. In view of the criteria above, one may confine discussion to the following candidates.

Consider first Structured English (SE). The method uses a number of sentences derived from templates in a hierarchical structure to give successive levels of refinement. Ostensibly limited to the definition phase and considered suitable only for small systems, SE does at least offer readability by a non software-specialist, even if the language is rather stilted. It is also attractive for the non-software parts of a complex specification since it has some similarity with the traditional specification couched as far as possible in standard phraseology.

System definition is also the objective of Controlled Requirements Expression (CORE) [12]. CORE uses a hierarchy of viewpoints as one of its principal concepts. For each viewpoint, a table is compiled of activities, inputs and outputs. Then, on the table, connections are drawn between each input and every associated activity; the outputs are treated similarly. This information, once captured, can be processed for internal consistency to ensure that, for example, each input has a corresponding output. The hierarchical structure requires that a record is maintained of the data-flow decompositions, and CASE support is normally used for this purpose. The identification of an appropriate set of viewpoints is essentially a modelling exercise, and for a complex system the viewpoint would probably be related to the items of hardware comprising the system. It is important to select a set of viewpoints which do not overlap but which give a complete coverage of the system.

CORE is often partnered by MASCOT [13,14] for the design and production stages. The Mascot design phase produces an activity/channel/pool (ACP) diagram based on the information compiled during the system definition. The ACP diagram is a network of processes communicating via channels (message queues) and pools (reference data). More recent terminology [14,15] refers to 'activities' rather than processes and to 'intercommunication data areas' (IDAs) rather than pools and channels, but the distinction between pools and channels should be recalled when Jackson System Development (JSD) [16,17] is described in Chapter 2. Strictly speaking, the ACP diagram is not hierarchical but for large systems it is convenient to group together elements of the network and define subsystem IDAs. Such 'chunking' [18] is an indispensable part of coping with large systems, but how to select the chunks is not always obvious. JSD also suffers from an absence of a natural hierarchy and one of the results of the present work is a method for selecting the chunks in a complex system in a way that is compatible with JSD (Chapters 5 and 8). In the production phase MASCOT offers a set of constructional tools and mechanisms to verify the design before the code is ported to the target system.

A long established method is the Structured Analysis and Design Technique (SADT) [19] which has benefited from early CASE support. The method spans the definition and design phases, constructing a hierarchy of formal diagrams by functional decomposition. A SADT diagram consist of a small (<= 6) boxes connected by arrows. The boxes represent activities, the arrows entering them are either inputs, control or mechanisms, and the arrows leaving them represent outputs. Decomposition takes one of the boxes and expands it into several smaller activities linked by more arrows. Naturally, for consistency, the number of arrows leaving and entering the diagram must be the same as for the original box. The separation into 'control' and 'other' data loosely parallels the channels and pools of the original MASCOT. Clearly, the success of the SADT method depends on the confidence with which functional decomposition can be applied, and the ease with which control and other data can be separated.

A popular set of similar methods are due to Yourdon [20], De Marco [21], and Ward and Mellor [22] which are discussed here in the context of their application to real-time systems. The methods use a network of processes, depicted as circles, connected by flows of data. The network is termed a dataflow diagram (DFD) or bubble diagram and several types of characteristic network elements, such as transformations and transactions, may be identified. In real-time applications there is also a flow of control information which is typically processed by a Finite State Machine (FSM) approach (the FSM has certain advantages and was seriously considered for the present work - as described in Chapter 4 & Appendix 2). The dataflows and activities are decomposed until a level is reached where implementation is straightforward. The popularity of the method may have come about due to the lack associated design discipline, so that there is considerable freedom left for the designer either to use or abuse. Unfortunately, when the initial design phase is neglected, it is possible for the decompositional sequence to produce modules which exhibit poor cohesiveness. Indeed, Thewlis [18] reports that the problem of lack of cohesiveness can become so serious that a wholesale modification of the way in which the method is applied is required.

Discussion of JSD and its relationship to the methods above is postponed until Chapters 2 and 9. At this stage one may merely note the common feature of a number of methods: a network of communicating processes with two types of data - control data and reference data.

Recently a design approach has gained currency which supplants the dynamic view of a system as suggested by 'process' or 'activity' with the more static terminology of 'object'. Object oriented (OO) methods are not new and they have their origins in SIMULA in the 1960's. The OO approach considers a system to be a set of interacting objects requesting services from other objects by issuing messages to them. An important advantage of the approach is that it encourages cohesion and additionally promotes software re-use by virtue of its encapsulation of data. Hierarchical Object Oriented Design (HOOD) is described in Reference 23. Modern object oriented languages such as $C^{++}$ [24] have also been a factor in the increasing popularity of of the OO approach, but its incorporation into large-scale system design has not yet matured. Cameron [25] reports the current incomplete developments with OOJSD. It is also certain that while OO languages and design may aid good system development and maintenance, there will always remain an initial system definition stage which is creative and not prescriptive.

For the ACT Lynx system, at the time of its definition the lack of support ruled out OO methods. Nevertheless, the conclusions reached in Chapters 5 and 8 appear to be in keeping with the anticipated developments of JSD. From the available contenders, JSD was selected partly because of its modelling emphasis and also because of compliance with contemporary policy in the relevant part of RAE. JSD is particularly suitable for a research environment where it is relatively common for new systems to be developed *ab initio.* The JSD method is described in Chapter 2, and some illustrations of the application of JSD within this project are also given there. However it should be remembered that in the early stages it was not known how to apply the method to those parts of the system which were not digital. Even the application to those parts which were likely to be digital was not trivial. Moreover, it was desirable that at the specification stage there should be some freedom as to the type of implementation ultimately chosen - leaving open, for example, the option of dissimilar implementations of redundant units. Exactly how to treat analogue, mechanical, hydraulic and even human components of the system within the context of JSD was not clear. Advice was sought from various sources about the viability of using JSD to specify the ACT Lynx system, and encouraged by the responses, the decision was taken to stipulate JSD for the preparation of the ACT Lynx Requirements Specification. In the light of subsequent events, and the problems encountered, it is significant that the sources concerned did not attempt to suggest *how* it might be done - only that it should be possible.

The description of one way in which complex systems may be specified, in particular using JSD, is the main result of this thesis. It will be shown that there is a simple approach - a principle - which can be successfully applied to a system as complex as the ACT Lynx. The essence of the method, together with the techniques required to implement it, are described in Chapter 5.

These ideas were not available at the time the contract was placed with Westland Helicopters Ltd (WHL) for the preparation of a Requirements Specification for an ACT Lynx using the JSD method. The author's direct involvement in the ACT Lynx project stems from this time when invited to act on behalf RAE in providing advice about the use of JSD, having collaborated with RAE in the use of JSD in a number of projects [26,27]. Once the contract was underway there was the usual regular sequence of progress and technical meetings which involved personnel from RAE and WHL together with a number of specialist advisers. For convenience this group will be referred to as 'the design team'

### 1.4.1 Preliminary Design

In preparation for the start of the contract some effort was expended by RAE in preparing an outline design in the form of a network of communicating processes along the lines of a JSD System Network Diagram. Chapter 3 reviews this early work in some detail and the network ultimately produced is shown in Figures 3.1-3.3. There was some concern at the time that the processes were not being identified and specified by a correct JSD analysis. This was indeed true but not appreciated by all concerned. Nevertheless the outcome was a useful overview of the design even if only interpreted as a signal flow diagram of the likely system. Dogma gave way to pragmaticism in this instance and, as will be made clear later, obtaining this overview of the total system is an essential step in applying JSD to complex systems. There are several points of interest which arise immediately from a study of Figures 3.1-3.3. First is the place of interfaces on on a process network. Since they do actually transform information there is a case for retaining them, however if the interface merely changes the form of representation of the same information they are unnecessary from a functional point of view. In subsequent developments they were omitted for the sake of clarity of the whole diagram unless they possessed some functional behaviour. Second, it is clear that the process labelled BITE (Built In Test Equipment - later denoted System Test) apparently has little interaction with the remainder of the system. The whole system test issue is described in Appendix 4 but it is worth highlighting that even at this early stage its poorly defined nature was identified by the simplest of analyses. Finally it can be noted that a single physical item can give rise to several processes on the network. For

example, a parallel actuator can give rise to a parallel actuator function process and a parallel actuator entity process. Again this is an early indication of an important ingredient in the method as finally employed, and is set out in full in Chapter 5.

### 1.4.2 Design Development

With this preliminary work available to them the design team considered how best to make use of it in pursuing the JSD specification. Indeed the whole question of the suitability of JSD for the specification exercise became an issue. At the root of the problem was the fact that the ACT system was complex in nature and that, even given a familiarity with the JSD method for software system design, the way to apply the method to such systems had not been established. What compounded the difficulty, was that the majority of the design team lacked experience in JSD, and, not unnaturally, felt a lack of confidence in tackling a large, novel application, when they would have found even a conventional application a demanding exercise. As is discussed in Chapter 2, the first, modelling, stage in JSD is one where beginners often lose heart. The level of detail that needs to be drawn in just, it seems, to get started is an anathema to 'top-down' and 'broad-brush' exponents. Seasoned practitioners recognise that valuable design issues are being resolved - 'the bad news now' syndrome - and develop the patience to see this stage through to a successful conclusion. Bradley [28] set out a strategy based on using JSD on an area of the system where it could be applied in a conventional manner. The pilot's control panel offered such an opportunity and some preliminary JSD studies were presented to form a starting point for the development of such a strategy. An example of an early modelling of pilot interaction is shown in Figure 1.2 . The argument for this method of progress was that by starting with a situation where JSD could be applied with confidence, design work of known value would be achieved and, further, by following the threads of the design it would naturally lead on to those areas where the method was less well defined. For example, in Figure 1.2 it would need to be established exactly what effect BITE was to have on the remainder of the system. Therefore the approach would at least indicate, in JSD terms, what was required of the remainder of the system and possibly give some clue as to how the design of the majority of the system should be developed formally. It is also worthy of note that Bradley [28] also observed, at this seminal stage, that once a JSD specification was available, if only in part, then it could be implemented as a prototype simulation with which to validate the specification and contribute to its continued development. In Chapter 7 an Ada Simulation is described which brings such ideas to fruition. The concept of specification through simulation, or 'living specification' as it is dubbed in the ACT exercise, has currently found favour for aerospace software systems [29], but the it is believed that the ACT application is the

first complex system to be specified in this way. In the event, these ideas did not hold sway with the majority of the design team at that time, but they were developed independently and their culmination is discussed in appropriate detail in Chapter 5.

### 1.4.3 Version 1

The design team preferred to progress in a more conventional manner and relied on a traditional format of a document composed of chapters containing numbered paragraphs of text supplemented by a set of technical illustrations and diagrams. Ultimately this work led to Version 1 of the Requirements Specification [30], which contained a wealth of technical detail upon which most of the later work was built. It is not appropriate to review every aspect of this document in detail; it would, for example, be out of place to examine the non-functional requirements such as the stipulated weight limits for a particular unit, when the present work is devoted to the requirements for functionality. Some aspects are, however, crucially relevant, and it is on these that attention will be concentrated.

The first important aspect that was a source of intense debate was the need in the specification to make the control of the ACT System, that is its interface with the pilot, highly visible in the specification so that it would be utterly clear to the prospective implementers exactly what was required. In addition, to guarantee the integrity of the implementation, the specification should be cast in a form from which software could be directly written. Also mooted was the possibility of retaining the whole of the control of the system in a single process, referred to as the Supervisor, so that resources could be directed to it at implementation time to ensure a very high degree of reliability. The Supervisor issue came to be studied in some depth and involved some important general principles, accordingly a separate chapter - Chapter 4, The Supervisor - is devoted to the subject. Eventually, as a result of continued study, the majority of the tasks originally associated with the supervisor, became reconciled into the various roles of the pilot entity, but this was not until Version 3 of the specification was produced.

The second aspect was the absence in the specification document of much of the rationale for detailed statements of the requirements, or explanation of why it was necessary or desirable for the system to work in a particular way. The inclusion of such *design* information was felt to be necessary in order to achieve the required versatility of the specification. If some component of the specification were to be changed as a consequence of alternative technology becoming available, for example a new type of parallel actuator, then the relationship with the rest of the system in terms

of the supply of demands, failure monitoring and reconfiguration etc. should be set out clearly and completely in the specification.

Finally, there was some concern that there was an absence of cohesiveness in the whole specification. It was difficult to appreciate from the text the relationship between the various components of the system and the communications between the individual units. This aspect is obviously related to the second one discussed above but here there was more emphasis on the need for formal descriptions including a diagrammatic representation of the system, for example an updated and elaborated version of the system diagram of Figures 3.1- 3.3 . It is in this area that an approach based on software design methods would be expected to make a significant contribution.

### 1.4.4 Version 2

The next issue, Version 2, of the specification [31] attempted to ameliorate the shortcomings identified above. In it, the control of the state of the ACT system was discussed via Finite State Machines (FSM) and transition diagrams. Interestingly, the inclusion of FSM material was a stimulant to constructive debate on the essentials of the specification, including the discussion on the Supervisor in Chapter 4. (The concepts and notation of FSM seem easily accessible to non-specialists and this is a valuable property not shared by Jackson techniques.) The criticism regarding the absence of a rationale for the selected design approach was met by including supplementary design information to accompany the specification text. The final point about design cohesiveness was met by including a partially developed De Marco [21] description of the ACT system. The method is hierarchical and decompositional, and so the decompositions of processes and data flows are performed on a heuristic basis until they are resolved at the lowest stage of the hierarchy. Therefore it is not until this lowest level is reached that the decompositions are justified. Unfortunately the decompositions were not taken down to such a level and the work in this area was not as valuable as it should have been, whereas JSD would ensure that these considerations were dealt with first.

### 1.4.5 Version 3

As a matter of deliberate policy Version 2 was subject to careful scrutiny in order to determine those areas where it could be significantly improved. In particular the possibility of using JSD was re-examined since in the context of the ACT Lynx application a method which was biased towards system development was considered to be more appropriate than a decompositional, hierarchical technique. A strength of the

Jackson method is that it spans the full range of activity from system definition to production of code [11], so that at one end it is concerned with modelling correctly, for example, the actions of the pilot when he uses the Pilot's Control Panel, illustrated in Figure 1.3, and at the other end, contains the level of detailed specification necessary to generate code. Such a level of detail ensures that the design problems of the specification have been addressed even if the software is not actually produced - in this case the further step was taken and is described in Chapter 7. These two areas - modelling and attention to detail - had not been given sufficient emphasis in the earlier versions and the disciplines of JSD would force attention to them.

The production of Version 3 of the specification is described in detail in Chapter 6 so will covered here with considered brevity. Employing the techniques described for complex systems in Chapter 5 was a substantial, practical test of the viability of the method. As will be made clear in Chapter 6 and 7 not only did the application of the method fit smoothly with the preparation of new specification but it opened the door to advances in generating an associated simulation.

The first step was to perform a decomposition of the total system into logical, largely hardware-based, elements as illustrated in Figure 6.1. They may be listed as:

(i) SE - Sensor Element
(ii) CSE - Crew Station Element
(iii) CLISE - Control Law Input Support Element
(iv) CLE - Control Law Element
(v) CLOSE - Control Law Output Support Element
(vi) ADME - Actuator Drive and Monitoring Element
(v) AE - Actuator Element
(vi) ESSE - External Support Element

Such a decomposition follows the principles for complex systems set out in Chapter 5. Each element - or unit if the element is composed of replicated units - was then specified under the following headings:

(i) TYPE
(ii) FUNCTION
(iii) OPERATION
(iv) PERFORMANCE
(v) INPUTS & OUTPUTS
(vi) INTERFACES
(vi) TESTING
(vii) FAILURE REPORTING & RECOVERY

These headings are also considered in detail in Chapter 6. Important headings include FUNCTION and OPERATION; the former being a statement of the tasks of that element or unit, and the latter being a detailed account of how those tasks are achieved. A substantial amount of preliminary JSD design was needed for the main processing elements to ensure that the narrative in the OPERATION section was a viable basis for a JSD specification using the CASE products. The end product was a written specification that had been carefully structured and compiled so as to be compatible with a full JSD specification based on the composite simulation approach of Chapter 5. Therefore the further step to prepare a full, computer-based JSD specification was one that could be faced with confidence and the generation of an associated simulation a straightforward matter.

### 1.4.6 Simulation

A full JSD specification and implementation of the associated simulation was prepared from Version 3 of the written specification by Learmonth Burchett Management Systems Plc (LBMS) under contract. The preparation of a full JSD specification gave an independent verification of the written text of Version 3, while the simulation offered the opportunity for hands-on validation of the specification. The development of the simulation is described in Chapter 7. A particular contribution of LBMS was expertise in code generation tools, where in addition to the Adacode tool for direct generation of Ada from the JSD database, extensions were developed to enable replicated units to be generated from a formal description. The importance of this work is emphasised in Section 5 of Chapter 7. Since the formal descriptions include a definition of the requirements for fault management and consolidation it is possible to include the software for these features as part of the automatic generation. In turn this led to a decompositional view of the system based on the replication of the components of the system as well as its logical elements. The resultant unit networks, Figures 7.9 - 7.11, provide high level views of the system to give an overall perspective but ,when

viewed from below by an implementer, define the structure of the code to be generated. Therefore this phase of the work provided not only the combination of specification and simulation called a living specification (as introduced in Chapter 5) but also contributed to the overall method by formalising the unit definition for the hardware structure which could then be mapped directly onto the JSD Units of Chapter 5. The production of the simulation marked the end of the activities associated with Version 3 of the specification. The results of the verification and validation exercises will be collated and incorporated into a final, definitive version of the text. The objectives established by the review of Version 2 will then have been achieved.

## 1.5 THE MATURE METHOD

From a review of Section 1.4, above, one can identify the important ingredients of the method leading to a living specification. They may be summarised as follows:

(a) Decompose the system into significant elements. The decomposition should be tailored to the application but one based on hardware subsystems and redundancy is probably common to most applications.

(b) Prepare a natural language specification for that decomposition. Chapter 6 provides a useful prototype for the structure of the description. The specification should be underpinned by appropriate JSD design studies using the techniques of composite simulation developed in Chapter 5.

(c) Use CASE tools to prepare a full JSD specification. This step provides verification of the text prepared in step (b) and any inadequacy therein should be remedied by formal revision of the text.

(d) Use code generation to prepare Ada code for the specification. Novel code generation facilities may be necessary for a new application area to reflect any new decompositional criteria.

(e) Compile the code into a runnable simulation for validation of the original specification. Again, any inadequacy should be dealt with by formal revision of the text.

To these steps one should attach the rider that the specifier should not be too constrained by them. The success of this work has depended on adapting accepted

orthodoxy and taking a flexible approach. The application of a few simple principles is the appropriate way to tackle large complex systems, adjusting the detail to suit the particular system. JSD and Ada have been used in this project but the ideas should transfer to any equally mature design and code generation environment.

Finally, although it is not listed above, the validation stage should be considered when preparing the JSD design; the data necessary for validation must be produced by the simulation. A specific element of the ACT Lynx system was dedicated to data acquisition and management.

The method, as described, has developed contemporaneously with Reference 29 concerning the use of simulation in the specification of software systems and with Reference 32 relating to the development of an avionics-systems simulation environment. The present method, it is asserted, is more comprehensive than the former and more mature than the latter. The specific achievements surrounding the development of the method are considered in Chapter 8 where the benefits accruing, particularly in relation to making a formal specification accessible to the non-specialist, are emphasised. A critical analysis of the method and the outlook for future developments are undertaken in Chapter 9. In particular, the potential for further developing the unit / connection descriptions and networks is explored, and the need for incorporating the text specification within the JSD database is argued.

## 1.6 CONCLUSIONS

This introductory chapter has stated the aims of the research and related them to needs of the ACT Lynx application. The application background has been discussed in order to demonstrate the nature and architecture of the system and justify its classification as a complex system . The development phases of the specification have been discussed in detail in order to place in context the advances that have been made. The discussion also places on record the special difficulties encountered and those approaches which did not prove fruitful. The specification method has been summarised as a sequence of steps to be considered for general application to complex systems.

# CHAPTER 2

# JACKSON SYSTEM DEVELOPMENT

Summary

The elements of Jackson System Development (JSD) are described in this chapter in a manner which relates to the ACT Lynx specification project. Jackson Structured Programming (JSP) is described first since it is a prerequisite to an understanding of JSD. JSD is introduced via modelling of external entities, including the concept of roles of an entity. The use of data (or static) entities is emphasised since they play a crucial part in the subsequent development of the ACT Lynx specification. Finally, network development and implementation are treated in the conventional manner.

## 2.1 AIMS AND ORIGINS OF JSD.

The development of systems covers a range of activities concerned with specifying and implementing computer systems. In an ideal application the starting point is a written explanation of what the system is to do, conventionally termed a narrative, together with some background material. The quantity of such material (and its quality) naturally varies with the sophistication of the application, but one of the features of the JSD technique is that attention is given at an early stage to resolving ambiguities and vaguenesses so that subsequent development can proceed on a secure foundation. From the initial definition, JSD constructs a network of communicating processes, with each process fully defined using Jackson Structured Programming JSP (see 2.2 below) and although there may be alternative methods of implementing the system, possibly reflecting different hardware configurations, there is the opportunity to generate code from the network / process definition. Therefore JSD has a remarkable scope since it involves several activities, including those termed systems analysis, systems design, and programming, and as such covers the whole spectrum of development from system definition to production of code [11].

Historically, JSP originated in the observation by Jackson [33] that the data structures associated with the input and output of a program must be reflected in the structure of the program itself, and that any processing of information can be attached directly to that structure.The crucial development into JSD came about [16,17] when it was realised that operator and system actions in real time applications had the same effect, so that a system could be viewed as a collection of programs (processes) each designed by JSP. Often a major difficulty lies in interpreting or 'modelling' the input

actions of the outside world, so that JSD, as taught, sets great store by the initial modelling phase.

The two features of JSD that had a deleterious influence on the initial development of the ACT Lynx specification are: (i) the ability to generate code implies a level of detailed treatment which was unacceptable to the original design team and (ii) the emphasis on modelling confuses the issue when data entities predominate (section 2.3).

## 2.2 JACKSON STRUCTURED PROGRAMMING

There are tutorial texts available on JSP, for example [34], so only an abbreviated discussion, specific to the ACT Lynx System, is considered in this section.

### 2.2.1 Data Structure

Consider a program required to control the engagement state of the system, so that when the state is engaged then the parallel actuator is connected to the control run (or equivalently the hydraulics of the parallel actuator are pressurised). The actions, or input records, to be processed by the program are an ENGAGE message from the pilots, a DISENGAGE message from the pilots or the redundancy management system, an ARM message from the pilot, and an ARMED message from the process which matches actuator position to ACT demand. JSP asks what the ordering constraints applicable to this situation are, and requires them to be expressed in a structure diagram. Figure 2.1 shows the appropriate structure, where the three basic substructures of the JSP notation may be observed. First, the engagement sequence ARM, ARMED, ENGAGE, DISENGAGE is represented by the sequence of boxes so labelled . This *sequence* construct imposes the correct order of actions for the pilot to engage the system. In addition to the sequence substructure, there is the *selection* substructure which indicates mutually exclusive alternatives. For example, there is a selection between the two possible sorts of cycle. One is the normal sequence just described, the other is one where a DISENGAGE has interrupted the sequence. The symbol (o) in the boxes for Normal Cycle and Early Cycle denotes the fact that the box Engage Cycle is either a normal or an early disengage cycle. The final substructure is the *iteration*, which is a repetition of none or more occurrences of the box indicated by the symbol (*), so that Pilot Engagement is a repetition of Engagement Cycle. The box named Actions in the Early Disengage sequence represents either an Arm, or an Arm followed by an Armed, that is, the part of the normal sequence which is then

interrupted. As will be made clear below there is no need to elaborate the detail at this stage.

### 2.2.2 Program Structure

As is often the case when using JSP in a JSD context the program structure is easy to derive. Since the program is to be used merely to hold the system state as a reference for other processes, then the input structure can be adopted as the program structure. The boxes or components of the diagram are used to indicate the flow of program control. The diagram requires some elaboration in order to direct the generation of code, as shown in Figure 2.2. Sequences are trivial since in all procedural languages, the order of the statements indicates the sequence of processing. An iteration requires a terminating condition (or more correctly, a continuation condition); this must in general be supplied in terms of a condition applied to data within the program. This may be either data which have been read in, or data derived from internal calculations such as counters. In the case illustrated, since the program is required to cycle continuously, the continuation condition is TRUE. This trivial situation is common in embedded systems where a process is required to run during the whole time the system is switched on. Each of the alternatives in a selection must have a condition attached to it; usually this is simple to achieve and the condition is determined by information immediately available within the program - often as a result of recent input. Some times, as here, there is a difficulty (a recognition difficulty) and when an ARM message arrives there is no way of knowing whether it will continue along the normal cycle or whether there will be an early disengage. The (?) in the selection boxes show that the selection is unknown and that the normal cycle is proceeded with (POSITED) but if a DISENGAGE message is detected after ARM or ARMED by the QUIT boxes marked (!) then there is a need to change control (ADMIT) to the early disengage cycle at an appropriate point. The appropriate point is the disengage box since the preceding actions have already been taken care of in the normal cycle (termed beneficial side effects). To show that the collection of odd actions need not be a concern when code is generated, the Actions box, now elaborated to show the types of actions to which it refers, is marked with a (-) and subsequently ignored. The processing required within the program is listed as a set of operations which are attached as sequence boxes to the program structure. In this instance they are few and simple, being merely message reading and setting the system state. An initial message is read to establish the disengaged state.

Figure 2.3 shows the basic data structure associated with the pilot initiated System Test. It has the same elements as the Pilot Engagement: a non-terminating iteration

waiting for the test to start then the body of the test - the result of which is undecided when the test begins. The elaboration of the system test body is a substantial item considered in Appendix 4.

## 2.3 JSD MODELLING.

The creation of a data structure based on input messages has been discussed in section 2.2.1 above. The data structure models the pilot engagement *entity* of the outside world and so, in JSD, the process derived from it is also termed an entity. The classical JSD modelling exercise compiles a list of all relevant actions - particularly externally applied actions - and attempts to group them so that they can be associated with an identifiable entity in the outside world. In 2.2.1 above the actions associated with pilot engagement have been grouped together in this way. Figure 2.4 shows part of the list of actions from which the pilots engagement actions were extracted. Usually for a beginner the modelling stage is an exercise fraught with difficulty and frustration as slow or even no progress is made. Correspondingly it is an area which receives particular attention in taught courses and the tutorial books (beginners experiencing the same difficulties may be observed in mathematical modelling workshops, which suggests that any formal modelling is a creative activity quite different to the usual technical exercise of skill). There are two aspects of modelling that require emphasis for the ACT Lynx system. The first point is that there can be more than one model associated with an outside world entity. In section 2.2.2 there have been descriptions of Pilot Engagement and System Test, which can also be ascribed to the pilot. The recent JSD terminology for this situation is that the entity has different *roles* and each role has its own independent, single threaded, structure. Bradley [27] used these ideas in the specification of the control of a piloted flight simulator and constructed models for several aspects of the user, including data-logging user, hardcopy user and pilot-rating user. At the time of that work the 'role' terminology and concept had not been developed in conventional JSD. The second point is fundamental to the development of the method applied to the ACT Lynx system in Chapter 5. A good example is taken from Bradley [26] which involved the application of JSD to the design of a device for injecting preprogrammed inputs into the series actuator of the Primary Flight Control Unit (PFCU), shown schematically in Figure 2.5(a). The modelling of the operator through a touch sensitive screen overlay provides a conventional modelling exercise of some interest, which is reported in Reference 26, but since the preprogrammed input may optionally be superimposed on the existing Automatic Flight Control System (AFCS) there is a need to model the actions of the AFCS by a suitable model structure. A non-JSD person would regard this as a trivial exercise and merely sample the AFCS demand values at an appropriate frame time using an analogue to digital converter. So

would a JSD person with any experience but often the means of explaining the situation has left much to be desired. The search for actions to attach to this entity led to such explanations as 'the pitch rate demand gets changed' is the action being modelled. Such convoluted explanations can be confusing to a beginner and it is possible that JSD has been resisted by some real-time practitioners because of its loss of credibility on this simple point. The appropriate data-structure for the model is shown in Figure 2.5(b) where the input message is simply the 'tick' of a frame time clock - Time Grain Marker is the respectable JSD term. The single operation, other than reading the tick message, is inspecting the value of the ADC and retaining the data for other processes in the system to use. The structure always has the form shown although sometimes data is sent out directly as messages. This type of model is very common in real time applications and has been called a Data Entity by the author to emphasise its effect of merely capturing and holding data for the rest of the system to use. Conventional JSD came to call such models Static Entities to reflect the relatively uninteresting structure. It is helpful in the ACT Lynx work to avoid viewing data entities as poor cousins of entities with more interesting structures. A viewpoint where data entities are the norm and occasional transitions in value can be interpreted as discrete events - more akin to analogue applications - is the key to a consistent approach to JSD for complex systems. Further discussion on this important aspect is postponed to Chapter 5.

## 2.4 THE NETWORK

Some important components of the JSD network diagram are shown in Figure 2.6(a), where processes are shown as boxes and message queues as circles. Message queues are referred to as data streams in JSD parlance. A simple situation is where a process P reads messages arriving on data stream DS1. Process P also writes messages on data stream DS2. Many programs are of this simple form; DS1 may be composed of records from a file or a keyboard, for example. The output DS2 may be composed of records being sent to another file or possibly a screen. In a system, we are concerned with the situation where the messages may originate from another process and be sent to yet others, as shown in Figure 2.6 (b), so that the whole forms a network of communicating sequential processes. Conventionally, the terminating data streams, where there is a connection with the outside world, are marked with a squiggle. The network in Figure 2.6(b) also shows that a process may merge the messages emanating from several data streams. Another network component , the diamond representing a state vector inspection, is shown in Figure 2.6(c). A state vector inspection (SVI) enables a process to inspect, but not modify, the internal variables of another process. The inspection requires no action on the part of the process being inspected and

consequently does not feature in the process description. As has been noted in Chapter 1, these two types of communication are common to many methods [13,19,22].

The development of the network in a JSD project starts with the boxes representing the entities, or roles of the entities if there are more than one. In Figure 2.7, some simplified examples are taken from the ACT Lynx. The pilot's inceptors are modelled, as is the engagement of the ACT system. In addition there is a modelling of the helicopter, in the sense that its attitude and rate information is sampled. As outputs or functions of the system there are demands for the series and parallel actuators and the operation of the clutch to connect the parallel actuator to the normal control runs. The outputs are shown as SVIs since, as will be discussed in Chapter 5, they are the most appropriate type in an application of this kind. A control law process is shown since such a process would be expected between the inceptors and actuators. The control laws would use sensor information too, but the nature of the connections - datastream or SVI is uncertain until the detail is worked out. The network stage of JSD is to discover what supplementary processes are needed to derive the required outputs from the inputs provided by the model processes. The messages between processes also need to be specified, as do any required state vector inspections. The designer looks to the narrative describing the requirements of the system to provide sufficient information to enable him to elaborate the network once the modelling phase is complete. For a large system elaborating the network can involve a substantial quantity of effort. However, apart from the usual difficulties caused by omission of information from the narrative, developing the network is a task enjoyed by software personnel, and uses skills that they are keen to exercise. There are some problems of managing a large network diagram, particularly the JSD type since it should be clear that it is essentially a two-level representation of the system, and not hierarchical in a similar way to the MASCOT network [13] described in Chapter 1. The top level is the process network, called the system network diagram (SND), and below that there is only the process level with its JSP structure diagrams. It is a situation which calls for computer assistance and there are several CASE tools available, some specific to JSD. Appendix 1 discusses CASE tools in the context of ACT Lynx specification. In practice the developer works on a reasonably sized segment of the whole diagram which then has to be integrated within the whole network. Figure 2.8 shows part of the ACT network associated with the control laws as an example. The process boxes can be seen, as can the datastream and SVI connections. The strokes across the connections to the Disturbance Imposer process, for example, indicate a multiple connection, that is, there are several copies of the Disturbance Imposer (represented by a single box for clarity since they have the same structure) and the connection is to each of them. A complete

set of connections is shown only for the Control Law Algorithm process. Other processes only have those connections shown which connect to it.

When complete, the network specification contains sufficient detail to be formally executed. This level of detail is one of the problems with developing a specification in JSD since only the problem of matching the specification to the target implementation environment remains. After that it is merely a case of generating code. The rationale is that it is not possible to specify a system until all of the relevant detail has been worked through. Nevertheless, it is common for contracts for software systems to be placed and accepted on the basis of a quite inadequate requirements specification. The hope is that there will be no intractable problem, and that those problems that do occur will be solved in a reasonable time. It is at the same time a strength and weakness of JSD that it cannot be used in this way. It is a strength because of the integrity of its detail: it is a weakness because purchasers of systems often cannot devote the resources necessary to develop a JSD specification, and prefer to risk the possibility of failure.

## 2.5 IMPLEMENTATION

A full discussion of the scope for matching the network to particular implementation environments is beyond the scope of this Chapter. Since the subject is fully covered elsewhere [16,35,36] the following discussion is purposely limited. The most straightforward implementation would be a multi-processor environment with one processor devoted to each JSD process. The processes could then run concurrently, and messages could be passed asynchronously via appropriate channels. That extreme implementation may be technically achievable at the present time, but appears never to have been chosen for a major application. The other extreme is to use a single processor and use it to run each of the processes in turn according to some stipulated schedule. Often the schedule is such that processes are waiting to read an input message. When one is supplied to a process it is executed until it reaches another read whereupon it suspends. It may, of course, be only partially through its thread of execution, and having suspended there must restart from that same position. Processes are usually implemented as procedures or subroutines and no current procedural language allows a convenient suspension and restart (although that is precisely what an operating system does). The device used to allow suspension and restart is called *inversion* [16] and, if necessary, preprocessors may be used on normal procedural code to implement it in a way invisible to a programmer. Interrupts can also be employed to execute processes according to an established priority.

A simple example of implementation will be discussed for Figure 2.9(a). The SVI operations are not important to the System Implementation Diagram (SID), and the first stage is to identify the input data streams, here labelled **a**,**b**,**c** and **d** . These data streams are then gathered up and the network allowed to 'hang' as shown in Figure 2.9(b). A scheduler process is introduced in place of the input data streams and the result is Figure 2.10(a). Messages from **a**,**b**,**c** and **d** are received by the scheduler and passed on to the appropriate processes **A**, **B** and **C** respectively via procedure calls. In turn, **A** and **B** pass the messages to **D** on data streams **e** and **f** respectively, also by procedure calls. This is a simple way of converting a network into a SID. It was used by Bradley for a programmable control input device [26]. The technique described above maps the SND into a hierarchy of procedures, but it should be noted that the hierarchy is not a tree. Another option is shown in Figure 2.10(b) where the inversion has been taken a stage further. In this scheme the messages to **D** from **A** and **B** are also handled by the scheduler, and the result is a simple, uniform structure to the SID at the expense of a more complicated scheduler. The ACT Lynx simulation described in Chapter 7 was implemented in a manner similar to this latter SID.

## 2.6 CONCLUSIONS

The foregoing sections of this chapter have contained a very brief summary of the main elements of JSD. The vocabulary and notation of the method have been introduced in the way that they will be used in the Chapters to follow. The use of the process and network diagrams have been explained in order to emphasise the level of detail and precision that is required of a JSD specification. The importance of static entities in real -time applications, and in particular the ACT Lynx, has been underlined. It should be borne in mind when reading this work and the referenced material, that JSD is a method which develops and continues to develop in a commercial environment. As a consequence the most recent changes to the jargon and philosophy may not be widely available. Often such changes do not affect the basic tenets of the approach, however it is likely in the near future that the method will be cast in an object oriented mould rather than one which is process oriented. Such an approach may appear, superficially, to be quite significantly different but it would not invalidate the essentials of modelling, network and implementation, and the methods described in subsequent chapters for handling complex systems would continue to apply.

# CHAPTER 3

# EARLY DESIGN STUDIES

Summary

This chapter is concerned with the collation of all of the information contained in the various feasibility studies and related work into an annotated diagram. JSD techniques were to be used and the resulting system diagram was planned to be the basis of the detailed specification prepared by the contracted design team. Difficulties associated with the use of JSD are described, and the options for making use of the network diagram are discussed

## 3.1 PURPOSE OF A SYSTEM DIAGRAM

The practicalities of modifying the RAE research Lynx into an ACT flight-test vehicle were examined in some detail [8] and several reports had been assembled concerning the probable form of the pilot interface, and the architecture of certain parts of the system. The task of the first design exercise was to take this accumulated material and to create a complete plan of the whole design. The purpose of this complete plan was two-fold. First it would enable RAE better to manage the development of the specification by having a clear view of the whole system, and secondly, it would enable different groups to work on the specification of different parts of the system in the confidence that the parts would ultimately combine to form a harmonious whole. The use of JSD would ensure that even at a detailed level there would be consistency between separately developed elements of the system because the interfaces between them would be defined precisely.

## 3.2 DIFFICULTIES WITH JSD

Direct application of the JSD method involves modelling the relevant parts of the outside world in order to obtain the inputs into a system. The outputs are then derived from the model processes via function processes, which may need information from intermediate processes of varying complexity. For the ACT system, the relevant world outside the system was reasonably well defined. The experimental pilot is 'seen' by the system through the movement of his inceptors, the setting of switches and the pressing of buttons. They constitute in JSD terms different roles of the pilot. Similarly the safety pilot has inceptors, switches and buttons which capture his actions for the system. The other main entity, which can easily be overlooked, is the helicopter itself. The manner in which it interacts with the system is that its kinematics may be used by

the control laws and so the attitude angles, rates etc. need to be captured for their use. This sampling activity is a classic 'data' or static entity formulation. Later, other features needed modelling such as the rotor brake, and rotor speed but they are essentially of the same nature and together form a 'helicopter' model for the system.

The outputs are no more difficult to identify. The principal output is the set of four PFCU actuator displacements to give the pitch of the rotor blades. The four control axes are, of course, collective, longitudinal and lateral cyclic, and tail rotor collective. In addition are the various displays associated with flight tests: head up (HUD), head down (HDD), and helmet. There are also some lamp displays on the ACT control panel which need to be serviced.

The difficulties start with the next stage. How can one derive the outputs from the given inputs, and so define the intermediate processes? Starting from a precise definition of the outputs required in terms of available inputs it should be possible to define the intermediate processes and their internal structure - after all that is what a feasibility study had shown. It is not guaranteed, of course, but even assuming that it is possible the result of the exercise would be the very specification that is the object of the exercise. Therefore one needs to do all the work before one starts (one may note that this situation is not uncommon when using Jackson techniques. Even when dealing with specifying a single piece of software to be contracted out, a JSP specification virtually requires it to be written in advance). Alternatively one might approach the difficulty by trying to establish the kind of network of processes likely to be suitable for the kind of outputs likely to be specified. This latter approach is not JSD of course, but if one sets the dogma aside for a moment, the situation is that it *is* known what components and sub-systems are likely to comprise the final system, and many of the interactions between them *can* be anticipated. Therefore based on this knowledge, Figures 3.1-3.3 were compiled from the available technical notes and reports. Together they were the first representation of total ACT System available for development. Each box on the diagram denotes the processing or transforming of information in some way. A table giving fuller names for the processes is provided in Figure 3.4-3.5.

The first observation that can be made about Figures 3.1-3.3 is that it is not JSD. The communications are shown simply by directed lines, they are neither data streams nor state vector inspections. The reason for this is that at the time the diagram was prepared it was not understood how to interpret data streams and SVIs in a situation where some processes are analogue, such as the Air Data Unit, and some are mechanical, such as the clutch. A method of resolving this problem is described in

Chapter 5. Advice at the time was that data streams should be used for any causal signal, but this interpretation is not helpful in this application. Also since the boxes do not have their internals specified there is no guarantee that they will be single thread in the final system, therefore one must accept that each box may be a small network of single thread processes - a top down approach which is an anathema to JSD orthodoxy.

The processes are labelled <letter> <letter> <number>, where the first letter D,A,M or H denote digital, analogue, mechanical or human respectively to denote the 'type' of the process. The designation of type is useful since it helps identify where interfaces are needed and conveys some of the implications for implementing the system as regards power supplies etc. However, it was the original intention that the specification should remain flexible and implementers should have the freedom to opt for either, say, digital or analogue components depending on their expertise. It proved difficult to retain such generality when producing a detailed specification and the decision was taken to stipulate the likely type of a process and give a blanket concession allowing alternative implementations which had equivalent functionality.

Interfaces are shown as processes in Figures 3.1-3.3 although, since they merely change the representation of information rather than modify it, there is a case for omitting them, and simplifying the diagram.

The mechanical control runs connect the safety pilot's inceptors to the PFCUs, but are moved by the parallel actuator when the clutch is engaged and then back drive the inceptors. This is a little complicated to handle in diagram notation since several arrows can change direction due to the system state. It can set a puzzle for a JSD representation, (and caused no little discussion) but is handled comfortably by the general principles outlined in Chapter 5.

## 3.3 LESSONS FROM THE NETWORK

The network diagram serves the purpose of showing clearly what the system is about. The general functioning of the system should be clear to a project manager, a systems engineer, a software engineer, and even a pilot. Generally, the inputs are shown on the left and the outputs on the right. There are several threads of activity. The experimental pilot (HE1) moves his inceptors (DE2), the movement is transmitted to the control laws (DP3) where sensor information (DE1) is also available. The demands produced by the control laws are split (DP6) into high and low frequency information (DP7,DP8) and the replicated signals are consolidated (DP32,DP33). The high and low

frequency information produce series actuator and parallel actuator demands (DF9, DF10) respectively. The parallel actuator (MP10) moves the control run (MP15) , provided the clutch (MP11) is engaged, to inject demands into the PFC (MF12) which also receives signals from the series actuator. At the same time the control run back drives the safety pilot inceptors (AE17). When the clutch is not engaged, the safety pilot (HE2) flies the Lynx via his inceptors (AE17) and the control run (MP15). The engagement of the clutch (DF20) is controlled by a button operated by the experimental pilot (DE2) and a requirement that there is synchronisation (DP17) between the control run position and the parallel actuator displacement. There is also an opportunity for the safety pilot to force 'break-out' via a switch on the control run. This is conveyed by connection 'C'. Both the series and parallel actuators are modelled (DP11, DP14) so that any failure of the actuator can force a compensating reconfiguration (DP22). The displays HDD, HUD, and Helmet (AF28,AF29 and AF30) receive the majority of their information from the sensors (DE1). Finally, the control and engagement of the ACT system (DE4) is indicated by the panel lights (DF21).

It is also clear from the diagram that although published notes indicate BITE (built in test equipment), initialisation and reset processes (DP5, DP28, DP30) corresponding to buttons on a draft of the control panel layout, their interaction with the rest of the system is minimally defined. The BITE is a test or check-out of the whole system prior to engagement, or take off. 'Initialisation' ensures that all of the monitoring and failure reporting is initialised in the correct sequence. 'Reset' takes the system back to its normal functioning after a monitored fault has caused a reconfiguration. These activities are potentially as complicated as any in the system yet no detailed information is available.

A possibly disturbing feature of the diagram is the appearance of multiple occurrences of the same object. For example, we have DF10 Parallel Actuator Demand, AP9 Parallel Actuator Demand Interface, MP10 Parallel Actuator, AF13 Parallel Actuator Interface, DE16 Parallel Actuator, and even DP14 Parallel Actuator Model. The placing of the interfaces suggests why this occurs. DE16 is a model in software of the 'real' parallel actuator MP10, serviced by the data emanating from the interface AF13. It will be shown in Chapter 5 that even when the interfaces are absent it is crucial to retain a similar pattern of multiple representations of the same object when applying JSD to complex systems.

## 3.4 DEVELOPMENTS

The compilation of the network shown in Figures 3.1-3.3 is a summary of the components, processing and communications within the ACT system. For the progress of the specification it was necessary to establish whether it was a good foundation on which to build future work, and if so in which direction should it be developed, bearing in mind that it was not clear how to weave into this activity a closer relationship with 'correct' JSD. The options for immediate action were in essence:

(a) Apply JSD to those areas where it can be applied directly such as the inceptors, switches and control panel, and simultaneously within the design team work at the specification of the control panel functions such as BITE and Reset, and at the general philosophy of applying JSD to complex systems. This approach was argued by the author [28].

(b) Elaborate each of the functional boxes with a formal (mathematical) statement of the processing to occur in the box, implementing a kind of successive refinement.

(c) Do no further work on the network representation but use it for preparing a detailed specification in a traditional, text dominated, format using numbered paragraphs.

In fact, course (c) was chosen as the main thrust of activity, mainly because of the lack of familiarity with JSD within the design team. Eventually, in the second phase of the evolution of the specification, when JSD was readopted this detailed text was a valuable source of information. It worth noting that course (b) would produce descriptions which could contribute to either (a) or (c) and is an exercise which would have to be done eventually when JSD was applied fully over the whole network.

Course (a) was pursued individually by the author and the results of the continuation of the JSD approach are described in the present work.

## 3.5 AFTERMATH

The creation of the network shown in Figures 3.1-3.3 proved valuable both in directing future studies and as a vehicle for communicating discussion. A second, revised, version responding to suggestions for modifications, was prepared with separate networks for the different types of process - digital, analogue, mechanical and human. The network was described using a database approach where each process is

described in a record, as shown in Figure 3.6 . One important element of the record is a list of the input and output connections together with a statement of the data flowing along the connection. This example is taken from the control panel roles of the pilot model. JSD is applicable in a straightforward manner and the process shown would be part of the input subsystem. The inputs and outputs can be classified relatively simply: the inputs are the SVIs of the state - open or closed - of the RESET, MODE-SELECT, BITE and INIT buttons. The outputs are the press/release messages which are interpreted from the button states and sent to the appropriate control process. A network fragment of the process is also incorporated in the record. An explicit statement of the multiplicity of the process occurs for the first time. Although simple, such a database is capable of elaboration to form a powerful tool. For example, if the process is of the single thread type then one field of the database can be used to point to some representation of the processing - such as a structure diagram. If the process is composed of several other processes then fields can points to those. If appropriate software is written then the database can be used to draw the network. There is nothing new in these ideas; many CASE tools provide such facilities. The package marketed for Jackson design techniques is Speedbuilder [37], which is currently being replaced by Jackson Workbench [38], but there are numerous others which provide similar facilities for maintaining a database of network information. (Jackson CASE tools also provide the ability to generate code from the process information.) Rather than invest time in developing the record format shown in Figure 3.6, Speedbuilder was ultimately used for the JSD work. The network fragments from each record were subsequently combined to produce version 2 of the system network diagram. Interestingly, this was the last time that a diagram of the whole system was compiled.

## 3.6 CONCLUSIONS

The early design studies, described in this chapter, impinged upon several important principles, whose value was only appreciated with hindsight, and which have general applicability to complex systems.

(a) A network diagram representing the whole system is a valuable document for focussing further development. This is true even if there are substantial areas of uncertainty within the network.

(b) A top-down approach, separating the system into its physical components is an appropriate way to begin to apply JSD to a complex system. (See Chapter 5).

(c) CASE tools can be applied at an early stage.

These are lessons to be carried forward to any comparable design and specification study.

# CHAPTER 4

# CONTROL OF THE ACT SYSTEM: THE SUPERVISOR

Summary

One area which, from the beginning, was subject to intense scrutiny was the need for the automatic control or overall supervision of the pilot interaction with the ACT Lynx system, so that the pilot would be prevented from taking actions which were invalid in the prevailing context. This chapter describes the type of actions which are relevant to this discussion, and considers a number of options which were explored for overall control. In the main these options were rejected for reasons which have general applicability and so the chapter concludes with a set of recommendations for systems which have a human interface of significant complexity.

## 4.1 CONTROL ACTIONS

Clearly, the controlling sequence for the engagement of the ACT system is a critical area where it is essential to get the specification and implementation absolutely correct. For example, engagement can only take place when there been a matching of the position of the ACT parallel actuator with the displacement of the conventional control runs. Other control activities may depend on the engagement state: for example, control law and parameter set selection cannot take place when the state is engaged, but control mode selection and disturbance injection can. System test can be initiated after power up and from the standby state, but the system test cannot be initiated when the rotor brake is off or the rotor is turning - and should terminate if these conditions occur during system test. The following is a list of the operations, in addition to system engagement, that have an interface with the pilot and are candidates for control by a global controller or supervisor process.

(a) Control law selection.
(b) Parameter set selection.
(c) Disturbance selection.
(d) Disturbance injection.
(e) Control mode selection.
(e) System test.

Once a decision has been made about which aspects of pilot interaction are to be subject to supervisory process, further decisions must be made about (i) how to specify it, and (ii) how to implement it. Both are considered below.

## 4.2 SPECIFICATION OF A SUPERVISOR.

### 4.2.1 Flow chart.

An example of an early model of pilot activity is shown as the flow chart in Figure 4.1, where the System Test, initiated by the pilot, if successful, is followed by a repetition of the arm, engage, disengage sequence of actions. While useful for conveying the general idea of the pilot's interaction in this area, it was not sufficiently precise to base software directly upon. For example, it is possible in the specification, to return to Standby through a disengage action without an engagement of the system. This path is not shown in the flow chart; neither is the opportunity to repeat the System Test from Standby. It is perfectly conceivable that code could be generated automatically from a suitably annotated flow chart, but the author is unaware of any such commercial product. Figure 4.1 therefore may be rejected because of the lack of CASE support and because the description is inadequate for the application.

### 4.2.2 Finite State Machine

To express the requirements in a precise manner finite state machines (FSM) were mooted and proved a very useful approach. It was observed that information expressed in FSM form was very accessible to both specialists and non-specialists, so that it is is attractive in applications such as the ACT Lynx project where personnel from different backgrounds are involved. The FSM shown in Figure 4.2 included the additional transitions to Standby omitted from the flowchart, but suffered from a shared disadvantage that the system test, itself, can include arm, engage, disengage sequences. However, FSMs have the advantage that they are readily transformed into software. To specify a FSM, a response to any possible message must be defined for each state. The response should consist of the transition (the next state) and the message to be output. The definition is easily processed either directly into code or indeed into a JSP tree diagram. The problems experienced with this approach were twofold. First, incorporating all of the possible states and transitions afforded by the pilot resulted in a very complex FSM; it was difficult to interpret and militated against a correct implementation. One can observe this in Figure 4.3 [39] which was a first attempt to provide a complete FSM for the the control of the ACT system. Since any FSM can be represented by a JSP structure diagram, it is clear that the structure diagram corresponding to Figure 4.3 would, in essence, be the Pilot model required for JSD.

This course was pursued to completeness by Bradley [40] and is reported briefly in Appendix 2. The second source of difficulty is that, as conventionally defined, transitions cannot be conditional on other information within the system and so their applicability is limited. The effect of this limitation can be seen in Figure 4.3 where, in order to prevent system test being initiated when the rotor brake is off, an explicit 'brake-off' message has to be incorporated. It would be preferable to merely ignore a system test message in those circumstances. In practice, it would be possible to enhance the FSM to include preconditions or even attach a filter for incoming messages. Nevertheless, for a monolithic supervisor process, the FSM was considered to have reached a size where it was too complex to be useful since its specification and maintenance would be prone to error.

## 4.3 SUPERVISOR IMPLEMENTATION

During the deliberations about the specification of the supervisor concern was beginning to surface about the implementation of a monolithic supervisor. It had developed into a complex process which had no natural location in the system as envisaged. Consequently, the possibility of making it a separate hardware item was mooted although viewed with apprehension.

## 4.4 JSD AND THE DEMISE OF THE SUPERVISOR

The discussions above relate to Versions 1 and 2 of the ACT specification. Once work started on Version 3 [41] then the modelling of the system using JSD principles naturally led to a different evolution of the specification. The different interfaces which are manipulated by the pilot each represents one of his roles. Therefore each role carries its own model process and tree diagram, for example Figures 2.1 and 2.3. The control of the ACT System is based on these model processes, so that the control is distributed rather than monolithic. Any conditions associated with an actions are incorporated by appropriate interlocks or common context filters using information from state vector inspections. It is now possible to relate this approach to earlier work. First, the complexity has been reduced by distributing the functionality among several processes. Second, there is the opportunity to generate code via the usual JSD/JSP method. Finally, and crucially in this context, it is clear that the specification must be done this way for optimum integrity. For example, the section of code exercised in the control of engagement and disengagement during system test is precisely that code which will be used in flight. For a monolithic, single-threaded, supervisor this is not necessarily the case (see Appendix 2). The same applies to the code for the other model processes, so the distributed approach encouraged by the 'roles' of the JSD method leads naturally to a desirable specification.

It should be noted that once the 'role' approach has been accepted then there is the opportunity to use FSMs within that role (bearing in mind its limitations expressed earlier). In fact, although not presented here, this was done as part of the preliminary studies for Version 3. The considerations discussed above led to the disappearance of the supervisor issue once the need for the independent integrity of each role was appreciated as an important factor.

## 4.5 CONCLUSIONS

As a result of the experiences associated with the specification of the ACT Lynx control processes the following recommendations can be made systems with a complex interface:

(a) Monolithic 'supervisors' should only be used when it can be implemented simple process.

(b) Control can be distributed among several process without compromising the integrity of the specification.

(c) It is essential to separate out test-processes, so that the remaining processes are subject to testing in as near to authentic conditions as possible.

Having such a set of recommendations to hand at an early stage of the Lynx study would have prevented much sterile discussion.

# CHAPTER 5

# JSD FOR A COMPLEX SYSTEM.

Summary

This chapter introduces a procedure for interpreting the elements of a complex system in such a way that their functioning can be represented by JSD notation. Initially the ideas are described in relation to analogue components of a flight control system and then extended to more general systems including mechanical and hydraulic. The main principle behind the application of JSD to such systems is quite straightforward, possibly deceptively so, consequently the underlying disciplines are emphasised. Examples of the approach are taken from the ACT Lynx system.

## 5.1 JSD FOR FLIGHT CONTROL SYSTEMS

The Jackson method for structured programming, JSP, has its origins in commercial data-processing. The sequential processing of one or more files to produce other sequential files or sequential outputs is a traditional data processing application which is well served by the techniques of JSP. Surprisingly, the same approach to the sequential processing of actions, messages or events gave the key to the successful development of commercial 'real-time' systems such as library systems - a standard case study in a JSD course - and reservation systems. JSD has not achieved the same success in time critical applications such as flight control. There are several factors which explain the lack of penetration of JSD techniques in this area.

One factor is that much of the processing is algorithmic and inappropriate to JSD analysis. For a flight control engineer the algorithmic aspects - their precision and their accuracy - dominate his attention. Yet despite this antipathy there are good reasons for at the very least wanting to use JSD notation in the design of flight control software. The first is that the algorithmic part is almost certainly embedded in a system which has a significant element of system-control software involving interaction with the pilot. This system-control software is ideally suited to the JSD treatment, and therefore for consistency there is a strong motivation for employing JSD for the whole. If this is done then documentation is treated in a unified way and the same CASE tool can be used for design and automatic code generation. A second reason is that the JSD network stage is a representation of the design which can be used to implement the system on multiple processors so if a multi-processor implementation is envisaged, there is a natural advantage from using JSD techniques.

A second factor is that, traditionally, control systems have been designed as a network of transfer function blocks of the kind shown in Figure 5.1. This representation has a direct implementation in analogue components, as indicated in Figure 5.2. The design representation shown in Figures 5.1 and 5.2 has been the traditional approach of control engineers. It is appreciated that, more recently, design procedures based on discrete formulations more appropriate to digital implementations are becoming accepted, but that is not the issue here - this discussion concerns the application of JSD to an area apparently unsuited to it. Cameron [42] formulates a novel approach based on the JSD analysis of 'events' and communicating sequential processes [43]. The view taken is that an event is the arrival of a data value (i) at the input of Figure 5.1, (ii) at each of the transfer function boxes, and (iii) ultimately at the output of the system. The arrival of an event at a box 'fires' appropriate processing activity, and the modified data from the process is dispatched to 'fire' the next box in the network. Further, there are synchronising events where paths in the network join. It is interesting that the approach taken is akin to the 'actions' used in JSD modelling and is a refinement of it for the particular context. Recalling the comments made in Chapter 2 about actions not always being the correct modelling response to a situation, it is instructive to examine this application from a less sophisticated modelling viewpoint.

### 5.1.1 Component Simulation.

The standard way of providing a digital implementation of systems such as Figure 5.1 would be to express the transfer functions in differential form and formulate the whole system as a set of differential and algebraic equations to which a standard solution technique would be applied. The result would be a single process hosting a complicated algorithm. Set this approach aside for the moment and, in addition, ignore any indeterminacies caused by cross connections or non-linearities in the network. Such features need to be addressed whatever the implementation approach. Consider instead the analogue circuit of Figure 5.2, and ask the question whether its behaviour is *caused* by the inputs arriving on the left hand side. Causality is fundamental to the event-driven interpretation of JSD modelling, and an examination of causality was suggested as possible way forward for developing JSD for the ACT Lynx specification. Tracing causality in a large system, which can involve a significant element of feedback can be difficult and unproductive task. An alternative approach is not to look for directly causality at all, and merely view each analogue component as evolving in time subject to the constraints imposed by the network. In other words the causation is time and the circuits impose constraints on the data in the system. This

view is easily simulated and the network simply expressed in JSD form, as in Figure 5.3. Each component is an independent process which implements a discrete algorithm which corresponds to the transfer function or gain of the component. Each process is as simple as the original component from which it originated. Note the essential features of this mapping. Each process is driven only by time since the only data stream input to it is a time grain marker. The communications with other components are by state vector inspection. Therefore each process is essentially polling other processes in the system to obtain the data to update its own. The processes have the same concurrency as the original analogue system. The only question to resolve, as in any digital version of a continuous system, is to minimise the truncation error to an acceptable value. In the spirit of the component simulation described above, the truncation error can be reduced by a sufficiently frequent clocking of the time grain marker. For an application such as a flight control system an increase in the clocking rate to satisfy the requirements for truncation error may not be not be attractive but in principle it can be done. It is this concept of component simulation with independant evolution in time which is carried forward to the general situation. Before the general situation is considered, an example of a component simulation is discussed in some detail to emphasise how a desire to economise on clock rate leads to a tighter coupling between components of the systems and eventually to event driven, data-stream communications.

### 5.1.2 Example of Component Simulation

In order to demonstrate the concepts associated with component simulation the simple lag component of Figure 5.4 is considered. The figure shows (a) the transfer function, (b) the analogue equivalent, and (c) the equivalent JSD component simulation. The single process has only one input data stream, and that is a regular time grain marker (tgm) which prompts a frame of calculation of the process. The structure diagram of the process is the familiar polling iteration with only three operations: (1) Read the tgm tick, (2) Get the most recent value for $x$ from the upstream process and (3) perform one step of an algorithm designed to represent a simple lag and make the output $y$ available for downstream processes. Consider a suitable algorithm for the lag. For simplicity, low order methods are discussed without any loss of generality. In any event, it is often the case that the discontinuities in the system make high order schemes inappropriate.

From:

$$\frac{a}{s+b} = \frac{y}{x}$$

the differential form is:

$$\frac{dy}{dt} + by = ax$$

which may be discretised as:

$$\frac{y_{n+1} - y_n}{\delta\tau} + b(\alpha y_{n+1} + (1-\alpha)y_n) = ax^* \qquad (5.1)$$

where $\alpha$ is a constant, $0 <= \alpha <= 1$, and n is an index of frame time, so that $\tau_n = n\delta\tau$, and $y_n$ approximates $y(\tau_n)$ etc.

The LHS is a second order trapezoidal form for $\alpha = 1/2$, but the position in time of the term on the RHS is not specified. Since the value of $x^*$ is the latest available from a state vector inspection the whole method cannot be guaranteed to be more than first order, as can be seen from the rearrangement:

$$y_{n+1} = \frac{(1-(1-\alpha)b\delta\tau)}{(1+\alpha b\delta\tau)} y_n + \frac{\delta\tau a x^*}{(1+\alpha b\delta\tau)} . \qquad (5.2)$$

One possible way of maintaining the consistency of the scheme and improving the order is for the process A which makes $x^*$ available is to additionally keep the associated value of time, that is the process maintains a set of the pairs $(x_m, \tau_m)$ for appropriate values of m. If m=n and m=n+1 are available then the use of

$$\alpha x_{n+1} + (1-\alpha)x_n = x^*$$

in Eqn. 5.1 above will match the truncation error of the LHS. If the process has not yet generated the n+1th value, then an alternative discretisation can be employed. For example if the n th and n-1th values are available and $\alpha$=1/2 for a second order integration method, the order is retained by the form

$$(3/2)\, x_n - (1/2) x_{n-1} = x^*.$$

The method can be adapted, in principle, to suit dynamically whatever information is available from process A, and the order of method required. The result is that the need for both precision and economy has resulted in a transfer of additional information which has tightened the coupling between the processes.

The coupling is further tightened by eliminating the need to allow for a variation in the time attribute of the available values; if $x_n$ and $x_{n+1}$ are always available then the update calculation Eqn 5.2 always has the same form and variations need not be considered. The processing required is thus minimised at the expense of a further tightening of the coupling between processes. One way of guaranteeing that the same values are always available to update process T is to send them as a message in a data stream as shown in Figure 5.5. This also guarantees their processing by process T and eliminates the time grain marker so that process T is driven by process A. Process A is driven by its time grain marker or an equivalent driving process.

To summarise, the use of component simulations of the kind shown in Figures 5.3 and 5.4, where the processes are driven independently by time and have loose interconnections, is a justifiable modelling method which captures the essence of the analogue systems being simulated. It is common, in a practical exercise, to couple the processes of the simulation together in a tighter manner in order to achieve benefits in terms of performance and economy. When faced with problems in the application of JSD to new area there turn out to be advantages in regressing to the simplicity of the loose coupling.

## 5.2 SPECIFICATION OF COMPLEX SYSTEMS BY COMPOSITE SIMULATION.

The question posed at the beginning of this work was whether it is possible to use software engineering techniques to specify complex systems, which, we recall, may include non-software elements such as hydraulic actuators, mechanical levers and gears, and analogue implementations of transfer functions. If the answer is in the affirmative then there is a secondary question about the technicalities of using JSD in this context. Since the stimulus for this work was the initial failure to make progress on the specification of the ACT Lynx it is possible that at the time there was to much concern with the intricacies of JSD and too little consideration of the general problem of describing non-software systems through software methods. Firstly, it is important to realise that the specification is concerned with functional matters - the relevant activities of the system rather than the non-functional properties such as size and weight. One may then pose the question whether would be possible to create a computer simulation of the functional behaviour of the system being considered. Invariably, in the circumstances of a real situation, the reply will be in the affirmative, so that we may assume the existence of a piece of code that simulates, in a relevant manner, the subject system. Now any piece of code in a procedural language may be described in terms of a software specification and, in particular, in JSD notation. The mapping may not be one-to-one but, no matter, the existence of a specification has been demonstrated, albeit

informally. The answer, therefore, to the original question is that one can indeed specify non-software systems and that one should do it by specifying an equivalent simulation. That such a specification exists has not been proved but has been given plausibility by the discussion above. The demonstration of existence, of course, is not sufficient although it is encouraging; it leads to the more specific question about how to build a simulation of a non-software system, and it is at this point that JSD, and its notation, have a contribution to make. The basic ideas are those expressed in 5.1 above extended to a wider range of systems. Bradley [44] summarised the approach to be taken when creating simulations of this kind. For convenience they are called *composite simulations*. There are certain rules to be followed which expedite the formulation of composite simulations and these are discussed in the following section in relation to JSD. A final, and important, point to be made at this juncture is that once such a specification is available then it is readily implemented to form an executable simulation of the specification. Such an approach to specification is being proposed for software systems [29] but it is believed that the extension to complex systems is being addressed here for the first time in the current work. It is also worth noting, as mentioned in the introduction, that it was recognised very early in the project [28] that a side effect of using JSD would be a working, *living*, version of the specification. The term *living* is used in Reference 2 to describe a specification which has been developed in the manner described in this work. From a properly structured, text specification of a system, a corresponding JSD design is produced, from which code can be generated. When compiled into an executable form it can be run and used as an interactive form of the specification - and indeed is part of the specification itself. In such a way the sequence of four steps - text, JSD, code, and finally a working simulation - makes the specification live. Another view of the word *living* is that a disciplined implementation of the sequence of steps allows the specification to evolve in response to changing requirements with a corresponding simulation being generated as a matter of course. Naturally, the validity of the approach depends on the integrity of each of the four steps. It is possible to automate the steps from JSD design to simulation and thus eliminate any ambiguity. For the ACT Lynx system the production of a simulation is described in Chapter 7. The earlier steps: text and JSD at the present time are not automated and require the adoption of a disciplined procedure to maintain integrity. Chapter 6 describes how this was successfully achieved for the ACT Lynx system. It is important to distinguish between the well defined mapping of text into simulation via the steps described above and what is usually termed prototyping. Prototyping has gained a reputation as high level 'hacking' of code to obtain an acceptable system. With a living specification, changes start at the specification level so that the integrity of the whole is maintained.

Having indicated the *existence* of a JSD specification for a complex system, and its role in providing an interactive manifestation of a written, text specification, the following section is concerned with the *construction* of a JSD specification. That is, a statement of how to create a JSD specification for a complex system.

## 5.3 COMPOSITE SIMULATION USING JSD

The method of constructing a composite simulation follows the development described in Reference 44. The principle is simple and this simplicity gives it two important attributes. First, it is clear how to start, and second, it is very adaptable in its application - even to the extent that a fresh start is readily made when it is clear that an alternative approach would have been preferable. The total system which will be considered - the *complex* system - is considered to be a representative modern aerospace system with a significant software based digital component. This forms part of total system which may contain mechanical, hydraulic or electrical components, for example. The section above has indicated that the non-digital elements are to be simulated, therefore the whole simulation is a composite arrangement comprising several connected simulations, and a software component which may be considered a prototype of the software for the real system - and in some circumstances could even be used in the real system. It is convenient to denote a digital component of the whole an *internal* system and any other component an *external* system. The method starts by selecting a decomposition of the total system.

### 5.3.1 Decomposition

The total system is decomposed into internal and external systems in a commonsense manner - that is, based on its hardware characteristics. For example, in the ACT Lynx system one might identify the parallel actuator - a hydraulic component - as an external system and the Flight Control Computer (FCC) as an internal system. The initial decomposition is top-down in the sense that there is a partitioning of the whole without detailed attention to its functionality. However, by decomposing the system according to its type one would expect some natural cohesiveness to result. Further decompositions may be subsequently dictated by the particular application; in Chapter 6 a decomposition of the FCC is specified in order to make the control law part readily replaceable, which again emphasises the role of the hardware in determining the decomposition. The decomposition step is not difficult since it is done in terms of the hardware architecture and this would be familiar to an engineer.

### 5.3.2 Specification

After decomposition comes the specification step. Each of the external and internal systems is specified using JSD. For the internal systems this is a normal specification exercise since they are known to be digital. The external systems must be specified as a simulation of the real external system. Therefore each of the external and internal systems is specified as a JSD network. It is convenient and very relevant to the development of the treatment of complex systems to give a network produced in this way a special name, therefore the term *JSD unit* is introduced for this purpose. The process is shown schematically in Figure 5.6 where a total system composed of a digital part embedded in external hardware is mapped into a JSD unit representation to form a network of communicating units. For convenience in subsequent discussions, the qualifiers internal and external are used to describe units derived from internal and external systems respectively. In keeping with the principles of component simulation outlined above there are restrictions on the unit network:

(a) All units may only communicate with other units by State Vector Inspection.

(b) All units have only one data stream input and that is a time-grain-marker.

Figure 5.7 shows a network of this type. The specification of the internals of a unit is a detailed task which is highly dependent on the particular application. For the ACT Lynx system, Chapters 6 and 7 deal with this aspect, but there are important modelling principles to maintain and these are now considered.

### 5.3.3 Inter-unit Connections

Of particular concern is the connection between units which form the boundary of the internal and external systems. JSD modelling principles dictate that within the internal unit there should be a model of the external system using data from the corresponding external unit. In addition there may be a function process in the internal unit to provide data for the external unit. Therefore the configuration shown in Figure 5.8 may be expected to be generic. An example taken from the ACT Lynx system will make this point clear. Suppose a digital actuator drive and monitoring unit (ADMU) is connected to an actuator unit, then one would expect one function of the ADMU would be to supply the demand (or drive) signals to the actuator. In addition, the ADMU must contain a model of the actuator if it is to detect disengagement. In fact, one must expect units to contain appropriate models of their neighbours and for them to have functions to provide neighbouring units with data. Therefore, referring back to Chapter 3, the

occurrence of the same name in several boxes in the preliminary system diagram, Figures 3.1-3.3, is not only acceptable - it is essential.

Another important principle is that models should only use data that is available in the real system. Again the ACT Lynx system provides an illustrative example: if the generation of the drive signal for an actuator requires the position of the actuator as feedback information, then in the real system this information is not directly available and a position pick-off would be provided. It is important that the pick-off is used in the model and if there are a number of redundant lanes driving the actuator then each lane should have and independent pick-off signal if this is the case with the real system. Related to the principle above, is the further principle that external systems should not be over-modelled. For example, suppose one of the external systems is a simple control panel connected to an internal system which interprets the button state and powers the lamps. Then the corresponding external unit should merely offer open / closed information about the buttons to the internal unit and illuminate lamps in response to the signals received. There should be no modelling of button-press actions or pilot entities in the external unit.

## 5.4 CONCLUSIONS

The principles of specifying complex systems through the specification of an associated composite simulation using a network of JSD units have been set out above. The principles are deliberately simple in order to make the ideas accessible to a wide range of engineers. It is also believed that simple principles are needed to deal with complex systems, and the method outlined adds very little overhead to JSD as taught - in fact it simplifies certain of the modelling issues. Chapters 6 and 7 describe the application of the method to the ACT Lynx system, which is believed to be sufficiently complex as to provide a stringent test of the method. It will be seen that bonuses accrue from the use of the method since the fault tolerant architecture of the system is not only accommodated by the method but is able to influence the decomposition process.

It is appropriate to conclude this chapter by recalling an earlier difficulty encountered in the ACT Lynx specification and showing how the method described in this chapter deals with it. In Chapter 3 the puzzle of the causality of the control run was described. When the system is engaged, the control runs are moved by the parallel actuators and the safety pilot's inceptors are backdriven. When the system is disengaged, the control runs are moved by the safety pilot's inceptors and the parallel actuators follow this movement. Addressing the causality by the use of JSD data streams proved difficult and unhelpfully complicated, but the situation is greatly

simplified if the control runs are mapped into an external unit. At each tick of the unit's time-grain-marker, using data from neighbouring units, the control runs' positions are set to those of the parallel actuators if the system is engaged, otherwise they are set to the positions of the safety pilot's inceptors. Moreover, the dynamics of the control runs are easily incorporated if more sophisticated modelling is required.

# CHAPTER 6

# SPECIFICATION STRUCTURE

Summary

This chapter describes the structure of Version 3 of the ACT Lynx specification. It is put forward as a prototype of the kind of specification which would blend well with the principles for the application of JSD as described in Chapter 5. First, there is a decomposition into major components; the components correspond to the JSD Units introduced in Chapter 5. Each component is then described under a number of prescribed headings, and the influence of JSD at this stage is examined in detail. The chapter concludes with a critical review of the achievements of such a specification.

## 6.1 INITIAL DECOMPOSITION.

The specification structure describes the system in terms of its major functional elements. This decomposition was the only one that was imposed on the system *a priori* and reflects a separation based on major hardware components. Such a subdivision is in keeping with the principles of Chapter 5, but does not preclude further subdivisions should they evolve from the design and specification process. The outcome is shown in Figure 6.1, where the square and rectangular components are those relevant to the specification exercise. The bold rectangles are referred to as processing elements embodied in a Flight Control Computer (FCC) although such terminology was not used in the specification.

The elements of the system are described in the order of the primary flow of the signal information illustrated by the arrows in Figure 6.1.

(i) Sensor Element (SE). This leading element contains the aircraft motion sensors - attitude, heading and rate gyros and accelerometers, and also the air data units for obtaining velocity components, pressure and temperature information.

(ii) Crew Station Element (CSE). The other leading element incorporates the conventional controls for the safety pilot and a versatile side arm controller facility for the experimental or evaluation pilot. For convenience these inceptor components were subsequently grouped together as an Inceptor Element (IE). The

CSE also contains the various interfaces for the pilot to engage, operate and be cued by the ACT system (Figure 1.3) as follows:

(a) Pilots Control Panel (PCP) - used by the Evaluation Pilot for engagement and disengagement and also for conducting the system-test sequence. Engage and Disengage operations would normally be performed using switches on the pilot's controls.

(b) Repeater Panel (RP) - provides a copy of the displays for the Safety Pilot.

(c) Menu Panel (MP) - provides other ACT interactions, such as selecting one of the available control laws and sets of parameter values. The same panel provides the interface for injecting preprogrammed disturbances into the system, as part of a flight-test facility used, for example, in the validation of the helicopter mathematical models and in demonstrating compliance with handling qualities requirements of new control laws.

(d) Mode Select Panel (MSP) - available for in-flight selection of control modes, for example, height-hold and speed-hold.

Clearly the interactions associated with the CSE would be expected to feature significantly in any JSD modelling exercise, with the pilot assuming a number of different roles as he interacts with different components of the system. The modelling would be situated in the component which provides the system interface to the CSE.

(iii) Control Law Input Support Element (CLISE). This element has the main purpose of processing and managing the information from the Crew Station and Sensor Elements, and since it provides the system interface to the CSE it will host the JSD modelling. It also contains the process for the scheduling of a comprehensive system test. The details of the system test are considered in Appendix 4.

(iv) Control Law Element (CLE). This element is supplied with inceptor, sensor, mode selection and related information by the CLISE. The CLE is the raison d'etre of the ACT Lynx since it hosts the experimental control laws which are to be evaluated. It is this element that the user of the ACT Lynx, the handing qualities engineer or flight dynamicist, will interact with. Carefully verified and validated control law software [6] will be plugged into and unplugged from this element. Typically six

control laws will be selectable by the experimental pilot with an additional choice of up to six sets of parameters within each law. The demands produced by the CLE for each of the four axes may be separated into low and high frequency demands, if required, which are destined for the parallel and series actuators respectively (an option being currently evaluated). The separation algorithm is part of the user supplied CLE software. Alternatively this function could be achieved in software and a combined signal fed to full authority actuators.

(v) Control Law Output Support Element (CLOSE). The element following the CLE interfaces the demands produced to the remainder of the system. It also provides a selectable limiter on the demands produced by the control law as additional protection against immature software.

(vi) Actuator Drive and Monitoring Element (ADME). The final element to provide processing takes the demands from the CLOSE and produces drive signals for the parallel actuators resident in the Actuator Element, and the series actuators in the Primary Flight Control Units (PFCU). The ADME also manages the engagement of the ACT system through the energising of the parallel actuators, and supplies a normal autostabilisation function when the ACT system is not engaged.

(vii) Actuator Element. The parallel actuator system is last in the sequence. The parallel actuators are connected to the conventional control runs from the safety pilot; when the actuators are engaged (hydraulically powered), the controls are back driven to provide the safety pilot with essential control position cues and to aid in recoveries, and forward driven to the existing Lynx PFCUs.

(viii) External System Support Element (ESSE). In support of this network of elements is an element which essentially provides a catchment for all of the significant data in the system. It interfaces with the standard on-board data acquisition system MODAS [45] and also with the experimental displays such as helmet mounted or head down displays. A record of all system related events such as engagement, disengagement, and diagnostic messages is retained in a System Journal.

The decomposition is clearly based mainly on the nature of the hardware architecture but the division of what is ostensibly a Flight Control Computer into CLISE, CLE, CLOSE requires further comment. This subdivision is dictated by the need to make the user replaceable part of the system a separate item of hardware. This

specialised requirement is easily accommodated by the techniques of Chapter 5. The revised system architecture is shown in Figure 6.2 which provides an update to Figure 1.1. The CLISE, CLE and CLOSE are each triplex with one to one interconnections between the elements. The connections to the dual duplex ADME are full cross connections, that is, each sub-lane of the ADME is connected to each lane of the CLOSE. Figure 6.3 further illustrates the connectivity of the system. The connections shown are those from the triplex inceptor element, via the CLISE, CLE and CLOSE, to the ADME. It is necessary to separate out the series actuator part of the ADME since the series actuator must operate when the ACT system is unpowered in order to provide autostabilisation. The dual duplex ADME is composed of two identical lanes each containing an ADMU and a SADMU (Series Actuator Drive and Monitoring Unit). Each lane is composed of two ADMU sub-lanes and two SADMU sub-lanes which have a one-to-one connection. Both the ADMU and SADMU sub-lanes are treated as JSD Units in the specification.

## 6.2 ELEMENT DESCRIPTIONS AND JSD UNITS

Version 3 of the specification [41] contains a detailed description of each of the elements identified above. As far as possible, the recommendations of the STARTS [46] guide relating to the procurement of real time systems were followed in the preparation of the specification. Further, each element is described in detail under the headings Type, Function, Operation, Performance, Inputs & Outputs, Interfaces, Testing, and Failure Reporting & Recovery. Where a particular element is composed of replicated units, so that several **units** together comprise an **element**, the replication of units in the element is stated and the unit itself is described under the same headings. For example, the CLISE is a triplex element composed of three identical CLISUs (Control Law Input Support Units). In the transformation to JSD each unit of this type becomes a JSD Unit. In detail the descriptions are:

TYPE - Some indication is given here of whether implementation is anticipated as an analogue, digital, mechanical, hydraulic, electro-mechanical or human process. The suggested implementation is not intended to exclude alternatives if a supplier possesses a particular specialism or preferred approach. The view was taken, after some deliberation, that it was better to make specific recommendations rather than to leave the 'type' issue open. A general allowance could then be made for variations that nevertheless complied with the functional aspects of the specification.

FUNCTION - Under this heading is a complete statement of the tasks of the unit , that is,a statement of what job the unit has to do. For example, one of the tasks of the CLU (a unit of the CLISE) is inceptor management; the entry reads: "The inceptor displacements and inceptor switch positions shall be processed to provide consolidated signals for the associated Control Law Unit (CLU)"

OPERATION - This sub-section is concerned with how the unit will achieve its functions. This is done by detailed description, in text, of the processing required for each function. For the CLISE example above, the full details of the processing of the triplex signals would be supplied, including the consolidation algorithms for fault tolerance. The narrative under this heading is used to build the JSD Specification; the full JSD is not held within the text of Version 3, but sufficient initial design work was undertaken to be confident that a JSD specification could be derived from the narrative, as discussed in Section 6.3 below.

PERFORMANCE - This deals with 'how much' and 'how well' issues, including a statement of the times within which the tasks must be completed and, where appropriate, the accuracy that must be achieved. For example, a certain part of the system test must be performed within a stipulated time. The sampling rates for the unit would be specified here. One important defined constraint in the ACT Lynx system is that the total system time delay should be less than 25 ms.

INPUTS & OUTPUTS - This contains a list of all signals received by the unit and those transmitted by it. It includes the source of a received signal and the destination of a transmitted one. This information is also presented in diagrammatic form, Figure 6.4, for example, where the connections to neighbouring units are shown in JSD notation. There is, of course, a need to maintain consistency here, since for each input listed there must be a corresponding output on some other unit. Such consistency is easily maintained by a CASE tool such as Jackson Work Bench (JWB) [38].

INTERFACES - A list of the units and their types, both internal and external, to which the subject unit is connected. The purpose of this information is to identify the interfacing requirements between units - analogue to digital, for example.

TESTING - A statement of how the function, operation and performance of the unit is verified. In particular this may be done at a system test invoked prior to take off, or by the inbuilt monitoring.

FAILURE REPORTING AND RECOVERY- A statement of how errors, produced by a fault and having been detected, are reported within the system. Usually they are reported to the pilot via the Menu Panel, and sent to the system journal part of the ESSE. Cautions and Warnings may also be raised through the Central Warning System. In addition, a statement of the recovery of the system may be required; often the recovery is by returning to Standby via a controlled disengage - as would be the case when one of the monitoring tolerances within the system has been exceeded.

6.3 JSD NARRATIVE.

The material included in the OPERATION section of each unit description provides the narrative for the design and specification of the JSD network describing the internals of the corresponding JSD Unit. The question arises as to whether it is, in fact, possible to achieve the functions of the unit - one of the vital specification issues. This can only be answered with certainty when a full design has been completed, but an experienced person can identify the crucial parts of the unit's network and develop those in detail to give a reply with a high degree of confidence. Such a study in depth was made of those units with a significant quantity of system functionality, and the detailed text specification was based on these studies. The units treated in this way include:

(1) The Mode Control Panel (3.2.2)
(2) The Menu Panel (3.2.3)
(3) Pilots Control Panel and Repeater Panel (3.2.4)
(4) The Control Law Input Support Element (3.4.1)
(5) The Control Law Element (3.4.2)
(6) The Control Law Output Support Element (3.4.3)
(7) The Actuator Drive and Monitoring Element (3.4.4)
(8) The External System Support Element (3.4.5)

where the figures in parentheses indicate the section of Reference 41 where resulting specification may be found; it amounts to 35 pages of text. It is instructive to examine a small sample of the network fragments produced during this evolutionary stage, although some of the terminology has been superceded. Figure 6.5 shows three such

fragments - for the Parallel Actuator Drive, Series Actuator Drive and the Engage and Disengage Control and Monitoring. All three show how the component simulation approach described in Chapter 5 is useful in such contexts for demonstrating a plausible network without needing to specify detail. In Figure 6.5(a) the drive signal is simply obtained from the difference between the demand and the positional feedback. For the series actuator, Figure 6.5(b) a smooth transition is required at disengage so there is a blend between the ACT demand and the Automatic Flight Control System (AFCS) demand. The EDCM in Figure 6.5(c) contains the engage/disengage role of the parallel actuator entity as well as the engagement function - again in the spirit of Chapter 5. The Pilot Control (PC) and the ADMU finite state machine (A_FSM) are shown in Figure 6.6. Essentially PC is a context filter for the finite state machine which represents the Pilot_Engagment role. Since FSMs proved to be popular with the initial design team they were retained as a specification method for the engagement control software for a considerable time. Ultimately they were discarded in the interest of uniformity, and in any event they would need to be converted to JSP for Jackson CASE tools. Apart from some variation in terminology, the only significant difference between the FSM of Figure 6.6 and the pilot model of Figure 2.1 - which was ultimately adopted - is in the handling of the ARMED signal (the instant when the low frequency demand is brought to match the parallel actuator position). Here the alignment signal is handled by the PC filter so that an engage signal is blocked until alignment is achieved and the system is ARMED. In Figure 2.1 the ARMED is handled explicitly. In Figure 6.6, ROTOR refers to the rotor brake, FCLE is the flight control law element (the 'F' being subsequently dropped), FCSE is the flight control support element which became the CLISE and the CLOSE, PEB is the pilot's engage button which was incorporated in the Inceptor Element of the CSE, and the PCP is the Pilots Control Panel. Therefore the FSM of Figure 6.6 may be used to specify the detail of the control of system engagement. Figure 6.6 also indicates from where in the system the information to be used by the PC is to come.

## 6.4 SYSTEM TEST AND FAULT MANAGEMENT.

A significant part of the specification was concerned with ensuring the correct operation of the ACT System. There are two aspects to this. First is the fail operative / fail safe requirement that demands a certain level of fault tolerance. The second is a check of the system before take off - or System Test as it is called. There are a possibilities for duplication between the two aspects in as much as the system test could be required to test every aspect of the system including the redundancy within elements. The view taken in the specification was that the pre-flight system test should be solely concerned with the pilot's interface with the system and consist of a check that

the inceptors, panels, switches - particularly disengage switches-etc. are functioning correctly. The remainder of the system should be continually and automatically monitored for faults as part of the fault management strategy. The general principle of the fault management has three layers:

(1) fault tolerance,

(2) fault monitoring,

(3) reconfiguration.

using the terminology of Reference 47. The fault tolerance is obtained by accepting the median of a set of redundant continuous data. For example, the median value of the triplex parallel actuator demands from the CLOSE is used in subsequent processing by the units of the AMDE. For discrete data, such as the system engagement state, a majority vote is taken if the connection is triplex. The variation between the data on a connection and the voted value is an error which is used to monitor the data's validity. If it exceeds a certain tolerance and persists more than a certain time then a fault is assumed. The fault is reported to the pilot 's Menu Panel and to the reconfiguration mechanism which then causes all data from the faulty source to be ignored in subsequent processing. This reconfiguration can only apply sensibly to a single failure in a triplex connection so the safety pilot should respond to a reported fault by disengaging the ACT system and taking over manual control. However, should a further fault be detected prior to disengagement by the pilot then the system makes an automatic disengagement. The precise mechanism by which automatic disengagement should take place was not included in the text of Version 3 and had to be supplied when the detailed JSD design, described in Chapter 7, was being prepared. The principle of the mechanism was to maintain a map, in the ADME, of the validity of every unit of the whole system upstream of the ADME. The map is scanned regularly and if the integrity of the system is compromised then automatic disengagement is invoked. The integrity is compromised when valid information is no longer available from an element. For example, when two lanes of the triplex Inceptor Element are faulty then the monitoring cannot identify a valid lane and when this information is relayed to the map the next scan will induce a disengagement. In a general system there may be several ways of maintaining the validity map through the network of inter-unit connections; in the ACT Lynx application there is an obvious flow of information from the inceptors and sensors through to the ADME. In fact, a distributed map was implemented where the CLISE maintained and scanned the sensor and inceptor information, for example, and the ADME managed the CLOSE/ CLE/ CLISE validity status. The concept of a validity map simplifies a potentially complex area. It also allows for a subsequent enhancement of diagnostic facilities, since connection failures may be deduced from some combinations of unit failure messages.

System Test and fault management are discussed further in Appendix 4 and Chapter 8 respectively.

## 6.5 CONSOLIDATION

There is a major problem with replicated asynchronous systems in a practical situation. It relates to the sampling of input data which is discrete - such as an OPEN/ CLOSED switch position or the number of a selected mode; one lane samples data differently to another and in extreme cases one lane may not detect data that another lane does detect and possibly acts upon. For example, in the ACT Lynx system the system test should abort if the rotor brake is released. Potentially, therefore with an asynchronous system there is the danger of one lane continuing with system test and another aborting it if the brake is applied at an instant which is close to the commencement of system test. This aspect of system design is referred to as consolidation; it must be distinguished from the occurrence of a fault since the lanes are operating correctly and a reconfiguration is not the appropriate response. Consolidation must ensure that if no lane acts on data unless all the lanes act on it. Silva [48] describes a method which has been adopted to consolidate signals within the EH101 helicopter flight control system. This method was specified for the ACT Lynx; it is concisely described as a sequence of steps:

(1) Each lane samples a value (the sample may be a voted value).

(2) Each lane calculates a detected-value as follows: if among the latest n samples there are n-1 contiguous samples with the same value then the detected-value is set equal to that common value otherwise the value is unchanged (n is termed the history length).

(3) Each lane calculates a consolidated value as follows: if the detected-value is identical to that of the other lanes (siblings) then the consolidated-value is set to the common detected-value otherwise it is unchanged.

This procedure guarantees that the lanes will change their consolidated values in a coordinated manner. The inter-lane (sibling) sampling required in step 3 is assumed to be done instantaneously when arguing the validity of the algorithm. Figure 6.7(a) illustrates the operation of the algorithm; it shows time running from left to right with sampled values for lanes A,B, and C indexed by frame. The value changes from 0

to 1 between C2 and A3 and returns to 0 between B5 and C5 (C2 indicates the 2nd sampling event of lane C etc.). The detected values are shown for n=3 as are the consolidated values derived from them. As one would expected, the processing of the data introduces a delay into the pulse of 1 values.

Unfortunately in Version 3 of the specification an important remark relating to additional monitoring was overlooked in Reference 48. It is important to avoid the situation where the changes in the consolidated values of all lanes are inhibited by one lane developing a fault and becoming 'stuck'. Silva [48] suggests that built-in monitoring should invoke a reconfiguration in order to isolate the faulty lane. Although such a reconfiguration is part of the fault management strategy adopted for the ACT Lynx, the first element of the strategy is a fault tolerant operation. Accordingly voting was introduced into step 3 so that if a majority of lanes agree then the consolidated value is changed. This is shown in the final row of Figure 6.7(a) where it can be seen that the effect has been to alter the position of the resultant pulse of consolidated values. Voting significantly alters the philosophy of the algorithm and one would expect its effects to be most pronounced for pulses of values where the duration of the pulse is close to the time history length. An example is shown in Figure 6.7(b) where a short duration pulse is detected by lanes A and B but not by C. However, C is brought into line by the voting. The result is that the pulse is recognised by all lanes whereas without voting it would have been recognised by none. Despite these differences it is possible to conclude on the basis of an experimental investigation that voting provides the required consolidation and in addition gives the first level of fault tolerance.

The same method was used to investigate the criticality of the need for instantaneous sibling sampling, and heuristically it appears that provided that the sampling time of propagation is not a significant fraction of the frame interval then, again, the behaviour of the algorithm is satisfactory.

As a final remark on consolidation, it is interesting to examine the implicit architecture that it imposes on the system. Figure 6.8 depicts the sampling and consolidation of a source by a triplex element. The source is marked as synchronised since the same information is available to each of the sampling units, but they only sample a value when a lane 'tick' arrives. Since the same 'tick' drives the consolidation as drives the sampling these two aspects work in mutual synchrony. It is this known mutual synchrony which is at the heart of the consolidation algorithm since it enables one lane to predict the behaviour of another.

## 6.6 REVIEW

The decomposition of the specification into elements based on the hardware of the system proved to be a turning point in its evolution. The composite simulation approach introduced in Chapter 5 successfully enabled the elements to be specified in detail using JSD analysis. Subsequently a full JSD specification using the Jackson CASE tools was undertaken by LBMS Plc. Therefore the test of Version 3 was whether the written material in the specification and, in particular, the detailed narratives in the OPERATIONS sections were adequate to assemble a full JSD specification. Only one major omission was subsequently identified. This was concerned with the voting arrangements of the consolidation algorithm and is discussed in Chapter 7. In fact, rather than an omission, it was an error in the specification. If it had not been recognised at the design stage then it would have been recognised during the evaluation of the simulation. There have been a number of areas where the specification has needed clarification, and additional material, but none where a significant revision of the specification has been necessary. The question arises as to whether it would have been possible, or even desirable, build the full JSD specification concurrently with the preparation of Version 3. In principle, the development of Text, JSD and simulation in parallel is the ideal situation. For the ACT Lynx application the software tools were not available to make simulation of such a complex system an integral part of the development of the specification - as will be emphasised in Chapter 7.

# CHAPTER 7

# THE ADA SIMULATION

Summary

This chapter is concerned with the implementation in Ada of a JSD specification of a complex system. The work undertaken in the ACT Lynx project is used to demonstrate the practical application of the principles involved. The crucial role of code generation in the implementation of substantial simulations is emphasised. The decomposition of complex systems on the basis of the underlying hardware receives renewed attention with the interconnections between hardware units providing an additional factor in a revised decomposition. Finally, there is a discussion of the additional software needed to support a system simulation by providing a correspondingly authentic real world to which the simulation can interface.

## 7.1 SIMULATION AIMS

The concept of a living specification - an integration of a written specification and a working simulation - was introduced in Chapter 5. Once the techniques of Chapter 5 have been used to prepare a JSD specification, the implementation of a simulation is, in principle, a routine exercise. All that is needed is for the target hardware environment to be selected and for a corresponding implementation strategy adopted for the JSD network. In fact, the size of the subject system can be such that traditional implementation techniques are prohibitively expensive and an automation of at least part of the implementation procedure is a necessity. The first consideration must be the choice of hardware platform, and this is substantially determined by the purpose of the simulation - that is, in what respect is it expected to animate the specification. For the ACT Lynx system the following aims for the simulation were identified [49].

(i) Control and pilot operation of the system. Pilot acceptance of the procedures for operating the system, for example, the arm/engage/disengage sequence can be evaluated through hands-on experience. Also suppliers can directly examine the nature of the interface between their equipment and the rest of the system.

(ii) Synchronised control information. The techniques for managing and synchronising control information within an asynchronous system or loosely synchronous system can be verified.

(iii) Establishing tolerances. An asynchronous system generally must allow some tolerance in the monitoring of the information from replicated units. Suitable tolerances can be verified or even derived.

(iv) Computational load. The processor power and memory requirements of the system can be more confidently deduced from a simulation than a paper specification. Alternative implementations can be evaluated for processing efficiency.

(v) Fault management. The mechanisms for reconfiguration and the issuing of caution and warning signals may be directly verified.

(vi) Design evolution. Alternative designs for the components of the system can be directly evaluated.

These aims fall into three categories which are relevant to the general principle of the living specification. The first is the validation of the specification. Aims (i),(ii) and (v) above fall into this category. Its purpose is to ensure that the specification is actually what is required by the specifier. It is by no means unusual for a strict interpretation of written text to be at variance with what the original author intended; neither is it unusual for the full import of a set of detailed statements to elude a less than fastidious reader. An example is the Pilots Control Panel of the ACT Lynx. The specification set out, clearly and unambiguously, the illumination of the lamps during the system test and engagement operations yet it was not until the simulation actually emulated the illumination of the lamps by putting them on a screen that the specified pattern was dismissed as being unacceptable to a pilot. Item (v) is unusual in that if the system and the hardware is functioning correctly there will be no faults to detect. Therefore to evaluate the fault management capability of the simulation it is essential to incorporate in the simulation a method of injecting faults into the system. Moreover, the need to inject faults brings with it a need to define the types of fault that are likely to occur and to find a method of modelling those faults. There are standard ways of modelling faults in digital systems [47]; the methods used in the ACT Lynx system specification are described in Appendix 6. The second category is the use of the simulation to enhance the specification, or fill in some of the details omitted due to lack of information. Items (iii) and (iv) fall under this heading; the requirements for processing power and memory size for the digital parts of the system may be used to improve the original written specification. The tolerances to be used for the monitoring

of faults in an asynchronous system are extremely difficult to adjust *a priori* so that obtaining the optimum position between too many nuisance disconnects - automatic disengagements invoked by the monitoring - and a dangerous tolerance to persistent discrepancies can be a useful contribution of a simulation. The final category is of particular importance to an evolving specification. Item (v) in the list of aims above emphasises the importance in a real-life project to be able to respond to modifications to hardware components. In the ACT Lynx, for example, the parallel actuators were originally planned to be electro-mechanical, then they became duplex valve hydraulic, and finally, as described in Version 3 of the specification, simplex, direct drive valves were the preferred candidate. For each of the candidate systems, a simulation allows a practical study of the integration of the new actuators into the remainder of the system, and enforces a thorough review of the associated fault management. Further, a working simulation gives the opportunity to assess the modifications in action, so no questions can be shirked and left unanswered. One could also examine how the existing hardware and fault management strategies would transfer to an entirely triplex architecture. Obviously, such a profound change, even though in principle it would be a prime role for a simulation, would entail a considerable burden of work, but as subsequent sections of this chapter will illustrate, automation of the production of a simulation can resolve this difficulty. Simulation, in brief, allows the specifier to get close to seeing how his various options will work in practice, and is an important feature in allowing a specification to evolve along a properly validated path.

## 7.2 TARGET SIMULATION HARDWARE.

The implementation stage is concerned with matching the JSD network to the target environment. Normally the flexibility which is available at this stage is a useful property of JSD in that it enables the specification to be verified prior to it being implemented on the target hardware. This becomes a crucial feature rather than a useful property in the living specification, where the target hardware may not be available. In the JSD network each process executes concurrently with every other. This is unlikely to be the case even in the target environment and the use of a limited number of processors has the implication that some of the processes must be suspended while others execute - Chapter 2 has considered the standard inversion technique for the suspension of processes. For a simulation, the simplest realistic target hardware is appropriate and a single processor would be used if possible - as was done for the ACT Lynx system simulation. It will be recalled that the ACT Lynx system is composed of a number of replicated units and the implications of using a single processor to emulate these are discussed in subsequent sections of this chapter. In fact, the target hardware for the ACT Lynx system simulation was an IBM PC, or compatible. Initially its

configuration was a Dell 310 (80386 processor with 80387 coprocessor) and 5 Mbytes of RAM - mainly to support the Alsys Ada compiler. Figures 7.1 to 7.4 show photographs of the screens, which largely correspond to the panels of Figure 1.3, together with a display of the series and parallel actuator displacements. The screens are toggled by presses of the Escape key. Other keys provide all of the functions of the pilots interaction. In addition to inceptor movements, this includes engagement and disengagement, mode selection and system test. The Menu Panel functions are also provided via the keys of the PC; these include control law selection and parameter set selection. The full interaction for the disturbance injection is available from the keyboard so that the type of input - doublet or frequency sweep, for example - may selected, and then injected into the system after a rehearsal has been observed. The nature of the menu panel interaction is rudimentary - but, nevertheless, is in keeping with the specification. Ultimately a more sophisticated interface, similar to that supplied in the work of Reference [26], will be included. The validation of the the specification proceeds by operating the simulation from the keyboard and observing the screens for compliance with the written requirements.

## 7.3 INCREMENTAL SIMULATION.

Project management is an inescapable factor in an engineering activity of any size. Indeed, it was the need for management control that was the driving force behind the insistence that modern specification techniques be used and, if necessary, developed for the ACT Lynx System specification. When the preparation of the full JSD specification and the implementation of a simulation was contracted to LBMS Plc it was recognised that the implementation must be conducted in a manner which was under the full control of the design team - particularly since the final product of the contract was to be a simulation and specification that the design team and suppliers could modify and rebuild as necessary in order to support developments in the specification. In this area too the decomposition method and JSD proved its worth.

The compositional, or "middle out", nature of the JSD method has the property that once a model has been built every new function added to it provides a potentially deliverable, working, system. In fact, at any stage of the development of the network it can be implemented. Incremental development takes advantage of this natural property of JSD and phases development of a system over a number of increments. The added functionality required from each increment is defined initially in outline, and as each increment is completed it is reviewed and the contents of future increments re-examined in the light of any modifications or additions that have been found to be necessary. The

development of a system is thus responsive to an evolving specification but at the same time allows the project to be managed on the basis of milestones actually achieved.

The ACT Lynx simulation was developed over a number of increments the material for the first six was distributed as follows:

Increment 1: A model of the pilot/ system interaction including engagement of the ACT system and inceptor movement. The Repeater Panel and a display of the control run position.

Increment 2: A model of the pilot/ system interaction as regards System Test, Control Law Selection, Disturbance Selection, Mode Selection, Parameter Set Selection. The Menu Panel, Mode Control Panel and Pilot's Control Panel.

Increment 3: A definition of a hardware description language for units and connections, and development of associated tools, as discussed in later sections of this chapter. The functionality of Increments 1 and 2 based on the specified hardware, including fault tolerance. Provision for injection of errors.

Increment 4: Completion of the Control Law Input Support Element including the development of a tool for building a System Test process from a non-procedural definition. The Aircraft Motion Sensor and the Air Data Elements

Increment 5: Completion of the Control Law Element and the Control Law Output Support Element.

Increment 6: Completion of the Actuator Drive and Monitoring Element and the Actuator Element. Further development of the System Test Builder.

It can be seen that the increments are based heavily on the element decompositions described in Chapter 6. Essentially, the CSE, CLISE, CLE, CLOSE, and ADME are developed in turn, but it can also be seen that the incremental method has allowed a rescheduling of the System Test feature when certain aspects proved difficult. Ultimately the difficulties were resolved in a novel manner (Appendix 4).

## 7.4 THE USE OF ADA

Ada, as a language, has its origins in the disconcerting discovery in 1976 by the US Department of Defense (DoD) that more than 450 different languages were in

use in their computer systems. Even more disconcerting was the fact that none of them were considered suitable for adoption as a standard language. The outcome was a competition for a general purpose language which was won by a team led by Jean Ichbiah. The definition of the new language, Ada, was completed in 1979. It is well suited to concurrent and real-time programming, and became the mandatory language for DoD. In the UK it is not mandatory but is 'highly recommended' by the Ministry of Defence (MOD) so it was the natural, and the most politically acceptable, choice for the ACT Lynx System simulation. Despite the weight of authority behind Ada, it has not become universally accepted. Partly, this may be because it is not a suitable language to learn *ab initio*, or possibly it is C and its object oriented derivatives which have usurped its position in the systems area. Nevertheless, the extensive validation tests which are required by the DoD have resulted in a number of very high quality compilers being available. For any work closely related to flight critical applications the quality of the compiler is a relevant factor. Since the quality and precision of the specification is important in the present application, the comprehensive data-typing of Ada is a useful feature. In addition, packages and tasks are language features which have an important role. The are several possible mapping schemes between JSD and Ada [50,51]. The mapping used for the ACT Lynx project relies very heavily on packages, and is based on that described in Reference [51]. Each package corresponds to only one specification object, such as a process or a data-stream. This correspondence is particularly effective in the present application since it enhances traceability between the specification and the Ada. Finally, code generation tools were either available at the start of the ACT Lynx project or were under development.

## 7.5 CODE GENERATION

It is useful to begin this section by presenting some evidence of the need for code generation. If one assumes a particular implementation strategy then the use of code generation imposes little overhead on the number of lines of code actually produced; so one may conclude that the code produced by automatic generation is reasonably representative of that which would be produced by other means. The Alsys compiler for the PC is limited to 1000 compilation units in each of 7 library families, and this limit was effectively reached during increment 6 (see Section 7.3 above). One can conclude that the simulation involves something of the order of 7000 compilation units - a compilation unit is a separately compiled declaration or body of a package or subprogram, or subunit. The time taken to construct, *ab initio,* the ACT Lynx simulation at the stage of increment number 5, that is, generate the code, compile it and build it into an executable module, was of the order of five days. These figures give some idea of the size and complexity of the operation of producing a simulation. To

use the simulation in order to investigate alternative system architectures and redundancy management techniques by manually coding the changes and rebuilding the system would be an enormous task. In fact such a task could well be so prone to error as to make it impractical.

Software tools for code generation operate on a database of process descriptions and network connections. In this section CASE tools are treated with sufficient detail to clarify the production of a simulation from a JSD specification; they are discussed more fully and more generally in Appendix 1. Therefore the starting point for the discussion here is the existence of a database, such as Speedbuilder [37], which can hold all the information relating to a system network. The information includes a definition of all data-streams, state vector inspections, and processes. The process descriptions, in terms of structure diagrams together with a list of operations and conditions reside in data files of the Program Design Facility (PDF) [52]. If all of the operations and conditions are couched in the selected programming language, and the declaration material is included then there is sufficient information available from which to generate code. Some standard implementation strategy must be adopted, as discussed in Chapter 2, but when this is done the procedure can be automated. At the beginning of the ACT Lynx project LBMS Plc had such a tool, Adacode, under development. From the Speedbuilder description of a system it would generate code for it as a single Ada task. Since it was being developed for a IBM PC platform, the tool was ideal for the project. Indeed, in retrospect, its availability was crucial for subsequent developments.

### 7.5.1 Implementing Fault Tolerance.

The connections between units may be one-to-one or full cross connections, that is *broadcast*. Figure 6.3 illustrates both types of connection. For a broadcast connection there is a need for a voting mechanism as described in Chapter 6 in order to maintain the tolerance of the system to faults. In addition, for discrete information, consolidation must be incorporated in order to allow for the polling of data by asynchronous units. Thirdly, monitoring must be included in order to detect ,and report, errors that occur in the system. There are advantages in dealing with these three aspects of fault management in a standardised manner, rather than approach each connection as a special case. Figure 7.5 shows the architecture of the software associated with fault management. The connection of the Inceptor Element to the Control Law Input Support Element is used to illustrate a typical situation. In each of the units of the CLISE there is a voter process which provides the fault tolerance. A downstream monitor process detects errors and reports faults. Faults reported to the

monitor cause isolation of the faulty upstream unit. If there is a need for consolidation of discrete information - as in Figure 7.5 - then consolidation and sibling monitoring processes also need to be included. In Ada this fault management infrastructure is implemented, wherever it is required in the system, by instantiating a number of generic packages.

7.5.2 Unit and Connection Descriptions.

When a system contains replicated units then the corresponding JSD Units will consist of identical JSD networks. As a consequence there is an obvious invitation to generate copies of a single network by automatic means. That is, specify a single JSD Unit then from a description of the replication of the unit generate an appropriate number of copies. For the ACT Lynx System, such an approach is described in Reference 53. The unit description should be held on a database to allow processing by the code generation tool so that code for the whole system including replicated units can be generated. Figure 7.6 shows examples of unit descriptions as they are held on the Speedbuilder database for the ACT Lynx simulation. The first field is the unit name and after a few lines of standard information (STD-INFO) the specification begins (MAIN-PART). The type field specifies the unit as being either analogue or digital. Digital indicates a regular update frequency whereas analogue units are updated continually. The next field is the base redundancy - simplex or duplex for example, followed by the replication. A triplex element comprising three identical units would therefore have base redundancy one (SIMPLEX) and replication three. A dual duplex element would have duplex base redundancy and replication two. The units of a digital type of element may be run in synchrony or asynchronously with a specified frame lag. There is provision in the field labelled INTRA-UNIT-CONNECTIONS to list the state vector inspections between the units of the element. The UNIT-SID is a pointer to the description of the unit network, the name defaulting to that of the unit if it is omitted.

To complement the unit descriptions a specification of the connections between the units is required. Figure 6.3 shows, for example, that it is possible to have one-to-one connections or full cross connections (broadcast). In addition, since the nature of the inter-element connection is related to the monitoring and consolidation processes, the specification of the connection can include a statement of the monitoring and consolidation to be imposed. An example, of the connection object used in the ACT Lynx simulation is shown in Figure 7.7 for the connection between the Control Law Input Support Element and the Inceptor Element. The first field of the description contains the connection name, which is followed by the standard information of a Speedbuilder object. In the MAIN-PART the characteristics of the connection are

specified. The SOURCE and DESTINATION fields hold the names of the elements which are being connected. The nature of the connection, broadcast or one-to-one, is contained in the next field. If the connection is broadcast then it is possible to supply YES for the special interface field, in which case there is access to the outputs of a particular unit of the source element. Otherwise if the entry is NO then parameters for the automatically generated consolidation and monitoring must be supplied in the subsequent fields. If the entry for any of CONSOLIDATION, SOURCE_ERROR_MONITORING, or SIBLING_ERROR_MONITORING fields is YES then a HISTORY_LENGTH must be specified. The value must be greater than two samples to make sense. It is, of course possible to have monitoring without consolidation, the latter being relevant to data from a discrete set such as control mode values.

The final step is to integrate unit and connection descriptions within the code generation tool, and to reconsider the possibilities for implementation. Of particular interest is the approach taken in the Adacode development where each unit is implemented as an Ada task. Therefore to that extent the element simulations run independently exchanging data by sampling operations. This implementation is in the spirit of composite simulation as described in Chapter 5, and in fact the simulation becomes non-deterministic. Taken together the set of unit and connection specifications provide a formal high level network description of a system. Although, there currently exists no tool for generating such networks from the database, it is readily done manually and examples are shown in Figures 7.9 - 7.11 for various aspects of the ACT Lynx simulation. Further discussion concerning these figures is postponed until system dismemberment is reconsidered in the next section.

## 7.6 DECOMPOSITION REVISITED

Chapter 6 described a decomposition into hardware elements to which the principles of composite simulation, outlined in Chapter 5, could be applied. Earlier sections of the present Chapter have discussed an approach to fault management which used standardised methods and automatic code generation through a database of unit and connection descriptions. With the knowledge that the unit descriptions can be associated with the architecture of the redundancy and its management then it is appropriate to reconsider the original decomposition. The criteria to be applied in the review include not only the basic hardware but the implicit redundancy. The difference can be explained by examination of Figure 7.11 which shows the unit network principally from the system engagement and disengagement point of view. The decomposition into CLISE, CLE, and CLOSE is not necessary from any aspect of

redundancy - the triplex requirement could be achieved by a single element. Rather it is the need to isolate the user software within an easily replaceable hardware unit that has motivated this particular decomposition. The other criterion can be observed in the decomposition of the Pilot Control Panel into a simplex display and a triplex button - PCP Display and PCP Button respectively - so that the redundancy of the particular components of a single piece of hardware determine its decomposition. The same principle may be observed in Figure 7.9 where the Menu Panel is decomposed into four components each specified as a separate unit. For the buttons there is a triplex MP_Button unit; for the menu displays there is a simplex MP_Display; for the diagnostic messages from the CLISE and the ADME there are the triplex MP_CLISE and quadruplex MP_ADME respectively. Figure 7.9 shows the unit network at the end of increment 5 prior to the development of the dual-duplex ADME simulation. In it there is an interim representation of the ADME which is contained in the hardware unit named THE_REST. This unit contains all of the supplementary modules which are needed to support the simulation at this particular stage of its development. In the final network there is a one-to-one connection between the ADME and the MP_ADME similar to the one-to-one connection from the CLISE to the MP_CLISE.

The ADME is a complicated element and it is interesting to discuss the structure of its specification and simulation in some detail. It includes, for example, the processes which drive the parallel actuator in response to demands from the CLISE, the management of the engagement and disengagement of the ACT, and a major part of the monitoring and reconfiguration aspects of the fault management. The Series Actuator Drive and Monitoring Element (SADME) which produces the drive signals for the series actuator is also included within the ADME specification, but is treated separately because it must function to provide normal autostabilisation of the helicopter when the ACT system is not engaged. Before examining the ultimate unit/connection network of the ADME part of the simulation, it is beneficial to examine Figure 7.12 which shows the perceived internal architecture of the ADME during the preparation of Version 3 of the specification. The dual duplex architecture together with the requirements of the specification impose a distinctive architecture on the element. The sub-lanes of a lane in the dual duplex architecture are required to monitor each other and if either detects a discrepancy between the signals of its partner and itself then it must disconnect both sub-lanes. That is, the drive signals to the actuator are disconnected and a caution is issued to the pilot signalling a single fault. The other lane functions normally and hence provides the required fault tolerance. Should, however, the second lane also become disconnected because of the occurrence of a further fault then there will be no drive to the actuators and the requirement is to cease ACT operation and

disconnect the system, simultaneously issuing a disengage warning. It is clear that although the outline concept is for four similar units to be connected into a dual duplex arrangement, the reality must be more complex. It is necessary to provide the interconnections shown in Figure 7.12. Essentially, the MAIN unit provides the functionality, the COMPARATOR unit provides the cross sub-lane monitoring, and the CAUTIONS and WARNINGS unit operates the reconfiguration. Consequently there must be cross sub-lane connections between the MAIN and COMPARATOR units and in order to detect second faults there must full cross connections between the COMPARATOR sub-lanes and lanes. These features may be identified in the final unit / connection network, Figure 7.13. The ADME_MAIN unit, ADME_SV_COMPARATOR and ADME_DISENGAGER provide the same architecture. For dual duplex units the one-to-one connection also provides sub-lane cross connections, as required for the ADME_MAIN and ADME_SV_COMPARATOR. The connection to the ADME_DISENGAGER is defined as one-to-one but in fact uses information from all of the units to achieve a full cross connection. A significant property of the dual duplex architecture is that through its sub-lane cross connections it can provide a validity signal to accompany the functional data. For example, in Figure 7.13, the system state information being made available to the CLE and CLOSE from the ADME_MAIN is accompanied by validity information from the ADME_SV_COMPARATOR. The SADME has a similar MAIN and SV_COMPARATOR part but uses the same DISENGAGER.

Figure 7.13 reveals another interesting feature. For convenience, the supporting modules for the simulation, described in section 7.7 below, are contained in a unit called OUTSIDE_SIMULATION so that units such as PARALLEL_ACTUATOR and AFCS are merely acting as buffers to exchange information between the supporting modules and the rest of the simulation. These units contain rudimentary data entities of the type discussed in Chapter 5, and, in fact, that is their sole purpose.

It should be clear from the discussion above that the development of a database of unit and connection descriptions with facilities for automatic code generation invites a refinement of the original element decomposition. The coarse original decomposition was based simply on the nature of the hardware. It is illustrated in Figure 6.1 and was used in the production of the written specification. The refinement mainly draws on the redundancy of the components of the individual elements and therefore reflects more completely the redundancies within the design concept of the fault management. The final decomposition into units , therefore, reflects all of the redundancy of the system

and all of the basic separation into separate hardware elements. In the application of composite simulation each of the units is specified as a JSD unit, so that the simplicity and flexibility of composite simulation, as described in Chapter 5, is applicable to the refined decomposition. In a complex application it is an advantage to have a simple methodology [54].

## 7.7 SUPPORTING MODULES

The earlier sections of this work have described the features of an Ada simulation which is an implementation of the JSD specification of a complex system. In order to give a system simulation a representative environment in which to operate it will, in general, be necessary to supply either real hardware, or additional simulations with which the system simulation can interact. The use of real hardware usually requires real-time performance from the system simulation, so even in this case, an initial phase of additional simulation support is likely. It should be borne in mind that the features of the supporting simulation need to be tailored to the needs of the system simulation and not the overall application - as commented upon in Chapter 5. These principles are enlarged upon in the remainder of this section, where, as an illustration, the supporting modules developed for the ACT Lynx System simulation are discussed. Where Ada packages have been developed, their specification is listed in Appendix 5. The full Ada code is documented elsewhere [55].

### 7.7.1 Helicopter Model

The loop from the system's actuators back to the Sensor Element is closed by the provision of a helicopter simulation. The input to the helicopter simulation is the set of four PFCU displacements, as a percentage of total travel, and its output is a set of data for the AMSE and the ADSE. The AMSE requires the three body axes components of acceleration, the three body axes components of angular velocity and the three attitude angles. The ADSE requires the three body axes components of velocity and the altitude. The simulation was supplied as an Ada Package Linear_Heli. It is based on a simple linear model of the dynamics of the Lynx helicopter. There is no need, at least initially, to employ a full non-linear model since the purpose of the simulation is not to mirror with any precision the vehicle's flight mechanics. Also a linear model consumes a relatively small fraction of the available processor power. A more sophisticated model is easily incorporated at a later stage simply by specifying a different package body and compiling it. Nevertheless it is important that however simple the model incorporated into the simulation it should be adequately verified. The integration algorithm (Runge Kutta order 4) was checked with a test problem and the helicopter model verified against an independant simulation. Further, the eigenvalues of the system matrix were

checked against the values from the HELISTAB package [56], from where the trim values and matrix coefficients were obtained. Finally, the simulation was perturbed and the dominant eigenvalues were extracted from the sensor data and compared to the value obtained directly from the matrix. The extraction of the dominant eigenvalues is easily done by adapting the Power Method Iteration [57].

The state vector $\mathbf{x}_n$ after n steps of steplength h may be written:

$$\mathbf{x}_n = \sum_{i=1}^{r} c_i \mathbf{u}_i e^{\lambda_i n h}$$

where $\lambda_i$ are the distinct eigenvalues of the linear system, of dimension r, with corresponding eigenvectors $\mathbf{u}_i$. After a sufficiently large number of steps only the dominant mode, corresponding to i=1 and 2 without loss of generality, is significant so that:

$$\mathbf{x}_n = c_1 \mathbf{u}_1 e^{\lambda 1 nh} + c_2 \mathbf{u}_2 e^{\lambda 2 nh}.$$

Now put

$$\mathbf{v}_1 = c_1 \mathbf{u}_1 e^{\lambda 1 nh} \text{ and } \mathbf{v}_2 = c_2 \mathbf{u}_2 e^{\lambda 2 nh}$$

so that

$$\mathbf{x}_n = \mathbf{v}_1 + \mathbf{v}_2,$$

$$\mathbf{x}_{n+1} = \mathbf{v}_1 e^{\lambda 1 h} + \mathbf{v}_2 e^{\lambda 2 h},$$

$$\mathbf{x}_{n+2} = \mathbf{v}_1 e^{\lambda 1 2h} + \mathbf{v}_2 e^{\lambda 2 2h}.$$

If $e^{\lambda 1 h}$ and $e^{\lambda 2 h}$ are the solutions of the quadratic $a+bm+m^2 = 0$ then it follows that

$$a\mathbf{x}_n + b\mathbf{x}_{n+1} + \mathbf{x}_{n+2} = 0,$$

from which any two convenient components may be used to find values for a and b. The quadratic $a+bm+m^2 = 0$ may be solved for solutions $m_1$ and $m_2$. Finally, for i=1 and i=2, $\lambda_i$ is calculated from $m_i = e^{\lambda i h}$. For complex conjugates: $R[\lambda_i] = \log|m_i|/h$ and $I[\lambda_i] = \arg(m_i)/h$.

### 7.7.2 The AFCS

When the ACT system is not engaged the helicopter is stabilised by the normal Lynx autostabilisation equipment. The Automatic Flight Control System (AFCS) uses sensor signals to provide feedback via the series actuator of the existing Lynx PFCU. A single Ada package was supplied to simulate both lanes of the AFCS function. The package was subjected to the same stringent checks as the original Linear_Heli package to ensure that the AFCS correctly modified the helicopter's dynamics.

### 7.7.3 The PFCU

The existing Primary Flight Control Units (PFCUs) are to be retained in the ACT Lynx. The simulation module supplied for the ACT system simulation incorporates simple, linear lag dynamics. Initially, as discussed above, this simple model is acceptable and when a more realistic representation is required provision has been made in the existing Ada package to include non-linearities in port shapes and flow-rates.

### 7.7.4 The Series Actuator

In reality, the Lynx series actuator is embedded in the PFCU hardware. Its input is directly added to the pilot's control run by a movement of the fulcrum of the control run's lever. The PFCU valve is attached to the centre of the lever. The simulation module was supplied as a separate Ada package since the series actuator is closely integrated into the ACT system: it will be recalled that there is provision for the high frequency component of the demand to be diverted through the series actuator. Again, as initially supplied, the dynamics are represented by a simple linear relationship between valve displacement and actuator velocity with provision for subsequent inclusion of non-linearities. Such aspects are not crucial to the system simulation - but what needs to be included are the features necessary to support the ACT functions. Therefore, for each control axis, the package imports drive signals from each of the four ADME sub-lanes and consolidates them within lanes for driving the duplex series actuator. As output, the package exports a consolidated fulcrum position for use by the PFCU and four separate position pick-off signals for feedback to the ADME sub-lanes. In addition four connection signals are imported from the ADME sub-lanes since the redundancy management may need to disconnect a faulty series actuator lane. The performance which needs to be verified by suitable testing has, as a major part, those aspects which are concerned with the validity of the response under different connection and feedback arrangements.

### 7.7.5 The Parallel Actuator

The parallel actuator, it will be recalled, is a vital part of the ACT system. It is connected to the safety pilot's control run so that when the system is engaged - or equivalently, when the actuator is energised - the parallel actuator back drives the control run. When the system is disengaged the control run drives the parallel actuator. Therefore the engage/disengage information is part of the simulation of the parallel actuator. As with the series actuator, the Ada parallel actuator package incorporates linear dynamics with provision for including certain non-linearities. To support the actuation function, for each axis, the package imports drive signals from each of the ADME sub-lanes together with corresponding connection information. It exports the control run position, and four separate actuator position signals for feedback to the ADME sub-lanes. The engagement / disengagement function requires the importing of engage and disengage signals form the ADME and PCP respectively. These are detected by the simulation, and hydraulic bypass valves are appropriately positioned. The positions are monitored by microswitches and it is the positions of the microswitches which are exported to each sub-lane of the ADME to convey the actual engagement state. It is clear that it is the performance of the parallel actuator under the wide variety of sub-lane connection and engagement states which has to be thoroughly verified from the system point of view and not for example the authenticity of the port orifice characteristics.

### 7.7.6 Other Modules

It is convenient at this point to list the additional modules which were specified in algorithmic form for the ACT Lynx System simulation. They were incorporated within the JSD networks for the appropriate units.

(a) A basic control law for inclusion in the CLE. The AFCS algorithm was employed for this purpose.

(b) A frequency splitter algorithm for inclusion in the CLISE. The low frequency component was obtained by a low pass filter (simple lag) and the high frequency component calculated as the difference between the low frequency component and the original value.

(c) A curtain limiter algorithm for inclusion in the CLOSE. This type of limiter constrains the demand to a particular region of the phase-plane. The original version supplied by RAE was found to be incorrect. A correct version is derived in Appendix 3.

## 7.8 OPERATING THE SIMULATION

Figures 7.1 to 7.4 show some photographs of the screen of the Dell 310 during the operation of the simulation. Figure 7.1 shows the Repeater Panel when the ACT system has been engaged. In Figure 7.1(a) in addition to the the power-up lamps the green engage lamp is lit. After disengagement the red warning lamp is lit, as shown in Figure 7.1(b). To the right of the Repeater Panel there is a display of the parallel and series actuator displacements, which is essential for monitoring the effect of inceptor movements - or their emulated equivalent. Figure 7.2 shows the more complex layout of the Pilots Control Panel; there is little difference between the two displays in the simulation but in reality the PCP would incorporate the buttons for engagement, disengagement, and operating the system test. The button positions are shown on the display but naturally have no effect. Figure 7.3 shows the mode control panel display for the selection of control modes. A press of the mode scan button (or keypad equivalent) moves the scan position along the top of the display. When the desired position is reached then a press of the select button invokes the ARM and CAPTURE sequence for that mode, lighting the corresponding lamps. Select also turns off a mode if it is currently selected. Finally in Figure 7.4 the Menu Panel is shown. On the right are the display areas for diagnostic and interaction messages from the CLISE and ADME, while on the left are the displays and buttons for selecting control laws and parameter sets. Also on the left are the buttons and displays for the selection and injection of preprogrammed control inputs as part of the aeromechanics research facility.

## 7.9 REVIEW OF THE SIMULATION

The implications of using JSD techniques to specify a complex system were fully appreciated at the beginning of the ACT Lynx specification exercise. It was realised that such a specification would enable a simulation of the system to be implemented in a disciplined but straightforward manner. So that while the JSD specification would verify the design, the simulation would help to validate it.

As reported earlier, there were initial difficulties with using JSD but once these were overcome, and the principles of composite simulation understood, it was anticipated that generating the code would be a long, routine exercise largely devoid of intellectual content. In fact, as has been described above, the very magnitude of the task stimulated the unit description, and unit/connection network approach and the whole idea blended well with composite simulation. Apart from the practical utility of unit / connection descriptions in relation to code generation, there are two additional by-

products. First they focussed attention on the inherent architecture of the redundancy, and invited a refined decomposition. The application of this further decomposition necessitated a careful interpretation of exactly what constitutes duplex or triplex. These matters are considered again in Chapter 9, but may be simply illustrated here by taking the example of the Aircraft Motion Sensing Element AMSE. Each AMSU consists of identical (within practical manufacturing tolerances) rate and attitude gyroscopes so that it is not unreasonable to consider the AMSE as triplex. However, in practice, the AMSUs could not be all mounted in an identical position so that the data that they process differs from unit to unit. Therefore their outputs will differ in a way that is not attributable to noise nor to manufacturing tolerances and so in that respect the processing in the AMSE is not triplex. The second by-product is that the unit / connection networks shown in Figures 7.9 -7.11 to a large extent usurp much of the value of Figure 6.1 which shows the decomposition into logical elements. The latter has value as indicating the general framework and context of the specification, but the former, having been derived from the same database that is used for code generation, have an unchallengeable authenticity. The concepts and practice of the unit / connection descriptions and their corresponding networks are generally applicable and are a useful supplement to conventional JSD. When integrated with a code generation tool, the unit / connection description becomes a demonstrably powerful technique.

The value of the simulation as a tool to validate the written specification and to evaluate alternative development paths depends on the ability of the user to know what is happening in the simulation. Much of the information relating to engagement and disengagement of the ACT Lynx system is conveyed to the pilot by the PCP and RP. In addition, the system diagnostics are displayed on the Menu Panel. The remainder of the information about the state of the ACT Lynx system is managed by the External System Support Element (ESSE), and so this application inherently contains full instrumentation of its operation. In a general case this may not be so, and consideration must be given to providing "eyes" into the simulation by supplementing the system specification with a specification of adequate instrumentation.

One of the aims of the ACT Lynx system simulation was to assess the computational load so that realistic hardware could be specified for the airborne system. It has been a salutary experience to discover the actual size of the computational load and the need for close to real time performance even in a simulation to make it useful for validation and evaluation studies. Naturally one would expect the emulation of triplex and dual duplex systems on a single processor to provide some degradation but the extent of the degradation caused by the monitoring and consolidation is serious. At

the end of Increment 5, for example, a problem frame time of 20 ms took 14 seconds of computer time. The result was that the response to key presses was so sluggish as to make the simulation virtually useless for extended validation. The implications for a general situation are fairly obvious: the simulation may need to migrate to higher performance platforms in order to be able to mount useful simulation studies. There are two factors which, while not redeeming the situation, ameliorate it to a certain extent. First, deficiencies in computational power are much easier to remedy at the specification stage than at construction - so to that extent the simulation has done its job. Secondly, one of the features of a JSD design is its relative ease of transfer between different implementations. For a large simulation it would not be a trivial exercise, of course, and the availability of code generation tools would probably be essential, but the method caters for it.

# CHAPTER 8

# SUMMARY


Summary

This brief chapter summarises the achievements of the research described in this thesis in relation to the original aims (In Chapter 9, these achievements are examined more critically). First the application of JSD to complex systems is considered, followed by a discussion of the evolution of the living specification. Next the role of decompositions and the value of unit networks are emphasised. Finally the overall strategy regarding fault management is described.


## 8.1 JSD FOR COMPLEX SYSTEMS

The starting point for the research was the need for a specification method for complex systems. The need for precision and verifiability invited the use of software engineering techniques and established the task of applying Jackson techniques to other than digital systems. The resolution of the difficulties associated with the use of JSD came from a generalisation of component simulation into the concept of composite simulation, where a system is divided into independently evolving units which inter-communicate solely by state vector inspections or polling operations. The units are conveniently conceived as being based on the fundamental hardware architecture, but other criteria, such as integral redundancy, may be used. The simplicity of the basic technique suggests that it is adaptable to a variety of decompositional criteria. The method's simplicity should also make the JSD aspect more accessible to the typical systems engineer since it focuses on data entities initially and postpones the more esoteric modelling until a later stage. The method is described in Chapter 5, and was used to develop a JSD specification for the ACT Lynx system. The ACT Lynx system is sufficiently complex and comprises sufficiently diverse components as to make its specification a substantial task and a demanding test of the proposed method. The success of the extended JSD method in this application demonstrates its suitability for a wide range of complex applications.

## 8.2 THE LIVING SPECIFICATION

The opportunity to derive a simulation from the JSD specification was appreciated at an early stage of the work, but the realisation of its relevance as a integral part of the specification came later. It matured into the concept of a living specification; one that was derived in a formal manner from the written specification

and could be exercised to give a 'hands on' validation of the specification or evolve in parallel with developments in the system design. The use of simulation to validate the specification of large software systems is gaining acceptance, but the application to the specification of a complex system is believed to be new. It is important to distinguish between unstructured prototyping and the discipline of the living specification. Prototyping as a way of designing and specifying systems was earlier referred to as high level 'hacking'. Such abuse may be uncalled for in many applications but it does highlight the dangers of an unstructured approach to system development. The terminology 'living specification' was adopted to emphasise the distinctive nature of the disciplines adopted in the current work. It has four components:

(1) Text specification.
(2) JSD design
(3) Ada code
(4) Simulation

Any modification - correction or enhancement - progresses in the order shown, from text through JSD and Ada to simulation. The simulation is modified only by this route. Indeed, components (3) and (4) are generated automatically. The crucial aspect of the living specification is that, through the simulation, it makes the specification accessible to systems engineers and other persons who are not expert in JSD. Thus the range of persons who can contribute to the validation of the specification is increased significantly.

## 8.3 DECOMPOSITIONS

The successful use of composite simulation with JSD Units relies on a sensible decomposition of the original system. This additional factor departs from conventional JSD since the decomposition operates at a high level and JSD is not a top down design method. Therefore there is some danger of making a decomposition that is inappropriate. The result would be a JSD design where process (modules) would exhibit a high degree of cross Unit coupling and low cohesiveness. Conventional JSD relies on its careful modelling phase to guarantee high cohesiveness. The present work has demonstrated that a decompositions based on the underlying hardware and on the replication characteristics of the components of the system can both be successfully treated by composite simulation. In addition to breaking down the system into manageable units from a specifier's point of view, decomposition assists by providing a rapidly assimilated view of the total design concept which can be useful in

communication between members of the design team and in project management in general.

## 8.4 UNIT NETWORKS

The need for a graphical representation of a system is well illustrated by their evolution within the ACT Lynx application. The first attempt is shown in Figures 3.1-3.3 which brought together all of the information of the preliminary studies and was compositional in nature. The next overall representation was the decomposition into logical elements as depicted in Figure 6.1, where the compositional aspects are hidden within the element components. Finally the unit / connection networks of Figures 7.9-7.11 present the results of a decomposition based on a practical allocation of hardware and on the redundancy of components within that hardware. However valuable the representations prior to the unit /connection networks, it is these final diagrams than possess the ultimate integrity. Their status is unique in the project: they offer a high level view of the system from a variety of standpoints yet they are also the foundation of the code generation database. The ACT Lynx experience is convincing evidence that unit / connection networks must form part of any JSD specification of complex systems: they have an authority which less formal representations can never possess and yet they are just as accessible to the non-JSD specialist.

## 8.5 FAULT MANAGEMENT

Redundancy is introduced into a complex system in order to comply with requirements concerning reliability. As such, it is not usually the prime function of the system or a concern that first drives the design concept. When it is introduced, one must guard against the neglect of a proper consideration of how that redundancy is to be managed in the context of a fault management strategy. There are benefits in adopting a uniform policy throughout a system: not only because of the conceptual simplification that occurs but also because of the practical benefits of reuse of standard modules (instantiations in Ada) in the specification and simulation. In the ACT Lynx specification the fault management is built on three layers. The first layer provides the robustness to invalid data by a voting or median-select algorithm. The second layer provides a monitoring of the data in order to identify invalid data - errors - which are symptoms of a faulty source. The second layer generates diagnostic information and also sends messages to the third layer, which provides the reconfiguration necessary to isolate the faulty source. The success of this approach including its suitability for code generation shows a potential for general applicability.

## 8.6 CONCLUSIONS

All of the original objectives of the research have been achieved. Principally:-

**1. The way to use JSD for complex systems has been established.** The absence of such a technique was a considerable stumbling block in the early versions of the ACT Lynx specification.

**2.The concept of a living specification has been developed.** It has resulted in a specification which is accessible to non-specialists and therefore widens the scope for validation.

**3. The unit / connection network has been developed from the initial JSD Unit concept.** This network is a valuable contribution to the documentation of the specification since it provides a high level view of the system which, again, is accessible to non-specialists but at the same time contains essential information, which when viewed from beneath, controls the generation of the simulation.

**4. An overall fault management strategy has been developed.** The uniform application of the strategy within a system confers benefits in terms of the clarity of the specification and standardisation of software.

The techniques described above have been applied in full to the ACT Lynx system with considerable success. One can be confident that this application is sufficiently representative as to ensure that the methods have a wide applicability.

# CHAPTER 9

# REVIEW


## Summary

This chapter contains a critical review of the achievements of the research. The purpose of the review is to identify recognised or potential weaknesses in the specification method. They are then be analysed in order to suggest future work for strengthening the method's philosophy and techniques. First, the role of JSD is examined against some possible alternative methods of producing a specification. Next, the contribution of JSD Units to the method, and a possible area of generalisation, are considered. Following this, some aspects of the treatment of redundancy are elaborated and the requirements for enhanced CASE support are discussed. Finally the need for a tighter integration of the text - to - JSD step of the living specification is argued.


## 9.1 ROLE OF JSD

Although in the ACT Lynx specification the use of JSD was essentially preordained, the initial difficulties experienced with the method caused a certain amount of casting around for alternatives. In the preparation of Version 2 De Marco methods and associated CASE support (Structured Architect) were employed. The resulting hierarchical description of the system ended its decompositions before many of the significant design features had been encountered - thus providing a good example of the situation JSD is guaranteed to avoid. Of course if that approach had been pursued down to sufficient detail then it could have been successful. An important feature of JSD is that it asks the designer or specifier to tackle the difficult areas first. In general, design needs to be done in a compositional way [54], while analysis may be done in a hierarchical way. Therefore the hierarchical approach does not help the designer / specifier to solve the design problems, but it can help him write down the design once he has solved them. Difficulties can arise, as in the early versions of the Lynx work, when the hierarchical description is partially written down, because it is not clear whether or not the designer had solved the problems when the description was terminated. One may draw the conclusion that hierarchical methods are valid provided they are worked down to sufficient depth.

The same general conclusions may be applied to SADT [19]. The specification methods CORE [12] and MASCOT [13,14] have an established reputation for the

rigour of expressing a specification but in essence depend on the explicit or implicit existence of a valid system definition. Petri nets [11] were considered for expressing certain of the synchronisation requirements at one stage of the ACT Lynx work but in the absence of a formal transformation into code were not pursued.

Formal methods, such as Z [58], were only briefly considered. The aspect that was of particular importance to the present work was the validation task. It is possible to pose questions to a formal specification and to be able to derive its the response, so that in principle thorough validation may be performed. It did appear that to perform this task required skills in formal methods and predicate calculus that were beyond those of most of the persons who would need to be involved in the validation exercise. In this respect a simulation, as used in the Lynx study, has a major advantage. An additional consideration was that tools for generating code from a Z specification were not available. Moreover it was considered that the application of Z to such a large system would be a task of enormous magnitude. It is possible that such concerns are unwarranted misconceptions and that a representative subsystem should be selected for a future study.

It is interesting to note that Jackson methods provide complete coverage from system definition and scoping to the actual implementation in code. This wide coverage had relevance to its application in the living specification sequence: text - design - code - simulation, and in retrospect there are no obvious disadvantages to the preference of JSD over other methods.

## 9.2 JSD UNITS

JSD Units, as described in Chapter 5, are time evolving JSD networks that only interact through state vector inspection, that is, polled information. Their very simplicity is a cause to question whether or not they are too simple. It would be possible, formally, to allow datastream connections between Units. In a practical application this would correspond, in composite simulation, to a discrete message being sent, for example, from a digital unit to an analogue unit. Using only SVI the method of handling the message would need to be an explicit part of the design - which could indeed reflect the real situation with respect to the requirements of the analogue unit.

In fact, in the ACT Lynx application SVIs were retained as the sole means of communication between JSD Units. The simplicity of this approach and its flexibility when the options for decomposition are being considered convey substantial benefits,

which have been noted elsewhere [32]. Therefore the preferred method is to avoid data-stream communication in the early stages of design and specificaiton and, if necessary, include them later. As in the Lynx project, that may prove unnecessary. Nevertheless one can envisage the situation where the datastream is likely to be essential - keyboard input into part of the system, for example. In fact, asynchronous communications between digital units can only be accomplished by data-stream, and SVI must be implemented this way [59, 60] - although this may not encroach on the design representation. Incorporating data-streams into the hardware descriptions is not a difficult task so the principal criterion must be that arising from specification efficiency (that is, clarity and simplicity) this criterion added to the experience on the ACT Lynx work indicates that the primary emphasis must be on SVI as the preferred means of communication between JSD Units.

## 9.3 CONSOLIDATION WITH VOTING

In Chapter 6 there is a description of the consolidation algorithm [48] which is needed to ensure that signals are 'seen' by all or none of the replicated units of an element. The modification which is necessary in order to achieve robustness - that is continued operation in the event of a single failure - is also described. However, it does appear that even with this consolidation strategy there is a danger that asynchronous operation of sibling units could give problems. As an example consider the operation of the CLISUs when conducting a system test. The system test is only started if the rotor brake is on. Therefore it is possible because of the different phasing of the asynchronous units for one CLISU to enter system test while another does not, because the rotor brake has changed its state in between the samplings of the CLISUs. There is no problem in this application because the system test process will abort when the rotor brake is subsequently detected as being off, but, in general, one must guard against the situation where although the signals are consolidated the internal logic may not be; particularly, of course, if there is any latching of discrete data.

## 9.4 UNIT DESCRIPTIONS AND REDUNDANCY

Where replicated units are employed to provide a specified redundancy, the unit / connection description allows a simple description of the system. At some parts of the design this can open up a debate about the nature of replication - or when triplex is actually triplex. In Figure 7.10, for example, it should be clear that the triplex CLISE is actually triplex because, within the tolerances of an asynchronous system there is identical processing of identical data. In reality, this is not the case for the AMSE since the triplex sensor pack contains attitude and rate gyros together with accelerometers,

and while these are identically constructed they physically cannot be placed at the same location relative to the centre of mass. Consequently the data that they produce cannot, in general, be identical. This effect was initially ignored in the ACT Lynx simulation but it is clear with hindsight how it should be treated. It is correct to describe the sensor element as triplex and the units should be identical. The information relating to the relative centre of mass position should not form part of the sensor element but should be obtained from the Helicopter Entity as part of its configurational information, since this data belongs to the helicopter itself and not the sensor. As a corollary, one can asssert that the connection to the CLISE would not then involve a conventional monitoring but would necessitate a special interface. The resulting view of the AMSE is much more satisfactory and is a useful prototype of how to treat identical units which process dissimilar data.

## 9.5 DUAL DUPLEX CONCERNS

A dual duplex system architecture is a simple concept but provides a number of puzzles should designer wish to incorporate a consistent philosophy in its specification. In the ACT Lynx, the dual duplex nature of the ADME was adopted at a time when the parallel actuators were envisaged as being totally duplex with duplex driven valves (as in the PFCU series actuator) in addition to a duplex hydraulic supply. The architecture persisted when, subsequently, an actuator with a simplex valve became the favoured solution. It is one result of the JSD design phase that it has revealed just how intimately the processing in the ADME is influenced by the architecture of the parallel actuator and any alternative actuator architectures need to be evaluated in detail.

There was also the consideration that a dual duplex implementation would allow an inexpensive protection against common mode failures. This would be achieved by a dissimilar implementation of the A and B sub-lanes. Clearly the argument is nonsense since any such discrepancy between the A and B sub-lanes would be repeated in both lanes and the whole system would fail. Nevertheless it is an argument that gains occasional currency.

The benefit of dual duplex architecture arises from its self monitoring capability, as shown schematically in Figure 7.12. Thus each sub-lane not only supplies its functional data but in addition a validity flag. Both sub-lane flags of a lane must be set to valid before any sub-lane functional data is to be accepted by a destination unit. Therefore any sub-lane of a lane can mark as invalid both sub-lanes of the lane it belongs to; this property is the strength of the dual duplex arrangement. It will be observed in Figures 7.12 and 7.13 that the Main part of the ADME produces the

functional data while the Comparator part produces the validity data. The question then arises as how to connect a dual duplex element to, for example, a triplex element and whether standard monitoring and voting is applicable. The connection which carries the system state information from the ADME to the CLISE and CLOSE in Figure 7.13 is a particular example. Since the validity information is already available then no monitoring of the connection is required and all that must be included is the appropriate reconfiguration response to the received validity signals. This is the essence of a connection issuing from a dual duplex source - no monitoring is needed since it has already been done. In fact the dual duplex arrangement is at its most useful when the destination element, by its nature, contains no monitoring: a situation which is precisely satisfied by the connection to the parallel actuator in the ACT Lynx system. The actuator is a standard electro-mechanical hydraulic device and can process (respond to) the drive data sent to it and even provide a measure of voting by means of a force-fight in the valve. In addition, it can reconfigure - simply by means of disconnection of the drive signals. However, it cannot monitor signals and relies on the validity signals of the ADME in order to effect a reconfiguration.

Consolidation of signals received by a dual duplex element is another difficulty. The consolidation algorithm [48] does not transfer directly to dual duplex. Consolidation across sub-lanes, followed by consolidation across lanes is possible but carries a burden of substantial complexity. Possibly the best solution is to treat dual duplex as quadruplex for the consolidation of incoming signals with a subsequent one to one connection to the dual duplex arrangement. The connected units would need to operate in synchrony as in Figure 6.8 so this architecture could not be cast in terms of separate hardware units. Therefore one is left with the rather untidy situation of an element composed of four sub-lanes viewed on input as quadruplex but on output as dual duplex. In the ACT Lynx simulation this issue was left unresolved and only consolidation between sub-lanes was applied to the connection between the CLOSE and the ADME.

## 9.6 TEXT INTEGRATION

The weak link in the progression from a written text specification to a working simulation as shown in Figure 8.1 is the step from written text to a formal JSD design. This step is a major design activity; in the case of the ACT Lynx the text was prepared first and the full JSD derived from it as a subsequent verification exercise. No major problems arose inthis instance because great care was taken in the preparation of the original text to solve all of the design problems and weave the solution into the operation narrative. An alternative approach would be to develop the narrative and the

JSD in parallel. Either way one arrives at the situation where both text and JSD specifications are available and one must then consider their consistent, mutual evolution as subsequent design changes are incorporated. Since the text to JSD step is not automated, maintenance of this step must be a manual activity and hence prone to the usual sources of error and omission. It is therefore appropriate to consider how it is possible to protect the integrity of this step. Ideally it should be automated but a less ambitious approach may be considered in the first instance which is to maintain the specification on the same database from which one can either generate a text narrative or a JSD design. An initial, simplistic implementation would be to hold within the unit and connection descriptions the block of narrative appropriate to that unit. At that level the integration is unlikely to be adequate yet to go beyond that to the process level is more problematical. This is a significant technical problem which requires further study. In any work in this area it should be borne in mind that the text narrative should be readable, natural language and not an inaccessible formalism.

## 9.7 LARGE SIMULATIONS

The focus of the work described in this thesis has been on the development of a specification method for complex systems - in the special sense of 'complex' meaning composed of dissimilar engineering devices. Ultimately as the final stage in the 'living specification' there is a simulation whose principal purpose is to provide validation of the specification and design.

It has become clear during the course of this work that the technique has implications for simulation studies too - and one may cite the avionics-system evaluation facility of Reference [32] as an indication of the relevance of this remark. Large scale simulations which often include elements of real hardware and imported software are notoriously difficult to manage. The principles associated with the living specification offer a way of managing the often dynamic environment of a large simulation in a way which protects its integrity. There may be a shift of emphasis in the purpose of the exercise - away from validation and towards performance prediction - but the principle of specifying the higher level and generating the detail holds good. Some simulation establishments have made progress in this direction [61]; the work described herein suggests that these developments are on a solid foundation.

## 9.8 CONCLUSIONS

This chapter has concentrated on examining critically the difficulties encountered in applying a specification method in a practical situation. The discussion should not detract from its success in that application since in many instances it is the

application of the method which has brought to light fundamental issues - particularly in regard to consolidation and the management of redundancy. The general approach of the method led to the unit / connection descriptions for which some uncertainties remain. This is mainly because they are not yet a mature concept and need to be employed in other applications before one can assert what are the most useful interfaces to incorporate within the unit and what should be left as special - or particular to the application. As a consequence of the discussions in this chapter, several important areas for future study have been identified. They include:-

1. The application of formal methods to a representative sub-system of the ACT Lynx.

2. Investigation of the potential scope and variety of JSD Units.

3. A full analysis of the ramifications of consolidation on asynchronous redundant systems.

4. The automation of the connection between the text and JSD parts of the specification.

The last of these being the area requiring most urgent attention.

# APPENDIX 1

# CASE TOOLS

Summary

The requirements of software tools to support the specification and simulation method described in the present work are discussed. The foundation for such tools is a database which is supplemented by a suite of object editors. The discussion is centred on the suitability of the Speedbuilder CASE tool [37].

## A 1.1 THE DATABASE

At the heart of any software tool for Computer Assisted Software Engineering (CASE) is a database of some kind. At is simplest it may take the form of a set of records of the type shown in Figure 3.6 which may be used to assemble a diagrammatic representation of the whole network diagram similar to Figures 3.1 - 3.3 . In this case there is only one kind of record - that of a process. A more useful representation of the network may be obtained by additionally storing a set of records relating to the communication between processes, such as data streams in JSD. Even at this elementary level the database serves two important functions. First, the consistency of the design may be confirmed, in as much as each message has both source and destination, and contains valid attributes. Second, a diagram can be assembled from the components defined on the database. This second factor - the visual representation of the design - contributes very little to the integrity of the design but provides a vital human interface for communication between a design team and for design elaboration and validation. The combination of a graphical representation for human interaction together with the imposition of consistency through an underlying database is the essence of a successful CASE tool and Speedbuilder, for example, suffered initially from the absence of network graphics. Entering network information in text form is not conducive to creative design work.

The layer below the network in JSD is the process layer and the database should also contain, or in practice point to, the details of the process internals (Other methods refer to the processing being done by activities). Since the processes are defined using JSP, the database contains the JSP structure together with information about conditions and operations. Once again there is a graphical representation [52] for the database information and JSP structures are displayed as tree diagrams.

The are two further areas where the database, through its Data Base Management System (DBMS) should contribute. One area generalises the maintenance of consistency referred to earlier and imposes the disciplines of the design method. For example, in a hierarchical method such as SADT [19] the database must impose the network hierarchy and constrain the number of branches at each level to the appropriate number (typically 6). In a JSP tree diagram if one component is a selection then all of its siblings are also selections; a component that is an iteration cannot have a sibling (Such considerations have given rise to the comment that the tool is the method - a statement that may serve as a warning to designers of CASE tools). To reinforce the point further, consider Figure 1.2. It has been drawn using a drawing package; its objects are boxes, lines and character strings and the drawing package stores the drawing merely as a set of graphical objects since it has no knowledge of the syntax required for JSP. A CASE tools would store it in terms of sequences, iterations and selections and implicitly verify the JSP syntax.

The second area is of particular importance in the development of the living specification and concerns the use of the database for the generation of code. Software for code generation requires complete, standardised information upon which to apply its templates and the database is required to support this activity.

A1.2 EDITORS

The graphical representations arising from the database objects described above offer the opportunity to populate the database by direct editing of the graphical representation rather than by entering text into database records. The Program Design Facility PDF [52] provides this facility for the JSP structure diagrams, and a network editor is available for use with Speedbuilder. Changes made to the graphical representation are then *written through* to the underlying database automatically updating the fundamental data. A disadvantage of this approach is that the system network may become so large as to be unmanageable on the screen. For Speedbuilder, the compromise has been to restrict the graphical window to the size of two A4 sheets, and display only a fragment of the total network. Therefore the database includes a set of network fragments which fit together like a jigsaw to comprise the whole system. Alternatively, they may be considered to represent different *views* of the database relating to various aspects of functionality. Figure 2.8 shows, for example, that part of the CLE network concerned with the control law algorithm. A hierarchically developed network is obviously much more straightforward to display on a screen.

Although it is, in principle, quite possible to specify a system solely by interacting directly with the database records the use of graphical editors gives a substantial increase in user acceptability and design visibility.

## A1.3 UNIT / CONNECTION DESCRIPTIONS

The discussion above is well illustrated by the introduction of JSD Units and their inclusion in the specification and code generation aspects of the ACT Lynx project.

Speedbuilder allows the user to define new objects - in this case Units and Connections - and to manipulate them within the Speedbuilder interface. Further, the object definitions are then available in a standard way to the code generator. It is clearly advantageous if a DBMS will allow the definition of new objects since the ability to explore new specification and simulation techniques was critical to the development of the living specification for the ACT Lynx. However, in the case of Unit and Connection descriptions there was no editor available to display or modify the unit networks. The absence of software to produce diagrams such as Figures 7.9 - 7.11 automatically from the descriptions held on the database was a distinct disadvantage. Note that for these diagrams too it is necessary to restrict the diagram to particular views of the system in order make the size acceptable.

## A1.4 CONCLUSIONS

From the experiences with the Jackson software in use on the ACT Lynx project it is clear that even with a well defined method such as JSD, it is important to have tools with a general capability which are then tailored to the various disciplines of the method. For JSD this would include a generalised tree diagram editor and a generalised network editor.

# APPENDIX 2

## MODELLING OF ACT LYNX SYSTEM CONTROL

Summary

The modelling of the ACT Lynx system control via a supervisor process is discussed for both Finite State Machine and Jackson approaches. A comparison is made between the two methods and the concept of an overall supervisor process is questioned.

### A 2.1 INTRODUCTION.

Two questions which dogged the early stages of the development of the specification of the ACT Lynx System were those of the suitability of JSD and the necessity for a *supervisor process* to oversee all of the pilot interactions. The supervisor is discussed in Chapter 4, and an initial state transition diagram for a FSM approach is shown in Figure 4.3. Within the design team there was some support for the basing future developments on the FSM since the initial state transition diagram was readily understood by non-specialists - which property is a considerable aid to validation. The initial misgiving were in relation to the difficulty of implementing a supervisor process with sufficient integrity since it was conceived as a separate component of hardware. Nevertheless there appeared to be value in continued development of the idea and a more detailed study was undertaken [40]. The study compared the FSM and Jackson designs for a more detailed analysis of the ACT Lynx requirements. The discussions and conclusions are pertinent to the general question of control of a complex system so they are summarised in the sections which follow.

### A 2.2 FINITE STATE MACHINE MODEL

Figure A2.1 shows the state transition diagram (STD) which resulted from an elaboration of Figure 4.3 . The terminology is that which prevailed at the time when Reference 40 was produced; for example, Built In Test Equipment (BITE) is used rather than System Test. There are two kinds of flight possible: either non-ACT or ACT which must be preceded by a successful BITE while the helicopter is on the ground. In the air an Airborne BITE can be selected (A/B BITE and N_BITE) and indeed must be successfully passed for the ACT system to be engaged. The distinction is necessary because the BITE on the ground produces movements of the rotor blade pitch which is, of course unacceptable, in flight. Control mode changes are possible at any time when the system is not engaged as is the reset of the monitoring system.

Therefore these actions appear the ground, the non-ACT and the ACT sections of the state transition diagram. A clamp action has been introduced which indicates the bake applied to a stationary rotor. As a consequence the diagram is a little simplified and has a single entry point. Many of the transitions caused by clamp or apply brake actions are shown by a token symbol since their inclusion in the conventional way results in a confusing maze of transition paths. The diagram becomes dominated by the interruptions to the normal functioning of the system which arise from the application of the brake. The diagram shows that once almost independent activities such as mode selection, reset selection, and brake application are accommodated on a single STD there is an almost combinatorial increase in complexity and the simplicity of the basic design is lost. In fact the complexity does not detract from the ease of verifying an implementation of an STD, since its simple structure allows automation of this task. This is not true of validation, where the complexity of the design process is reflected in the difficulty of validation. For example, while it is reasonable for BITE to be inhibited while the system is the rotor-slowing state, it is also appears that mode selection is inhibited too. The question is whether this is by default or by design.

## A 2.3 JACKSON MODEL

The structure diagram which models the same activities as the STD of Figure A2.1 is shown in Figures A2.2 - A2.7. The logical view taken is that a flight is either an ACT excursion or a non-ACT excursion. This is shown in the first figure - A2.2 and developed further in the remainder. In particular the BITE, ENGAGE, DISENGAGE ordering is shown in Figure A2.4. The are three points which can be usefully made about the diagram. First, although the transformation from STD to tree diagram can be done mechanically, constructing a structure diagram with sensible logical views was a difficult task [40]. Second, the number of leaves on the tree diagram is about the same as the number of states on the STD. There are more boxes on the tree diagram because of the intermediate boxes in the hierarchy, but the basic complexity is the same - as one would expect. Finally, the interrupts caused by clamp actions are neatly handled by QUITS from Posited structures which method, one might argue, properly models the real situation. The numbers attached to certain leaves of the tree identify the prevailing context so that a context filter process can filter out unwanted actions. Examples of allowed actions are:

| Context | Allowed actions |
|---|---|
| 1 | clamp |
| 2 | end_mode,release_brake |
| .. | .. |

| | |
|---|---|
| 35 | clamp |
| 36 | release_brake, select_mode, select_BITE, select_reset |

## A2.4 CONCLUSIONS

The conclusion of Reference 38 regarding the comparison of FSM and Jackson methods was that there was little technically to choose between the two methods. Each had its advantages and disadvantages and in any case the FSM design can be transformed automatically to a Jackson structure diagram by a relatively simple tool.

With the benefit of hindsight and subsequent modelling activities it is possible to add to this conclusion and state that the imposition of a monolithic supervisor process is unwise in all but the most simple applications. In a complex system, there are almost certainly a number of almost independent control activities taking place simultaneously, and a single STD or single thread structure diagram is bound to be so complex as to give doubtful validation. One can illustrate this by referring to Figure A2.1 and asking how it should be altered to accommodate control law selection, parameter set selection, and disturbance selection and injection. The better approach is to model these activities independently - by FSM if so desired - and them build in the appropriate inhibitions explicitly. The JSD modelling, with its emphasis on roles, captures this idea in a natural way. In short, a monolithic supervisor has advantages when it gives a simple overview of the whole activity. Once it loses this property it should be discarded.

# APPENDIX 3

## THE CURTAIN LIMITER

Summary

The need for a limitation on the demands and rates which can be applied to an actuator is discussed in this appendix. The properties of a curtain limiter are introduced and some examples of its effects are shown. A new algorithm to implement a curtain limiter is derived; the new algorithm corrects the deficiencies of a previously published algorithm.

### A3.1 LIMITERS

The output displacement from a hydraulic actuator is physically limited by the movement available to the piston within its cylinder. Similarly, the rate of the displacement is limited by the physical characteristics of the actuator such as the orifice size or hydraulic pressure supply. These limits are represented in a simplified form in the phase plane shown in Figure A3.1(a) by straight lines. The lines bound the region in which the actuator can move and since they arise from physical characteristics may be referred to as *hard* limits for the actuator. It is possible to process the demands which are to be applied to the actuator and further constrain the actuator movement. This may be done in order to protect the actuator hardware or in an experimental situation, for example, to give a measure of protection against system failures; the effect of a failure which results in a demand being stuck at its maximum value can be lessened by constraining both the demand and the rate of movement to those encountered in fairly modest manoeuvres. The pilot thus has a longer period of time in which to cope with the failure situation. As the system matures and confidence increases the constraints can be gradually eased to allow the use of the full performance of the actuators. Limits achieved by processing the demands are referred to as *soft* limits and a set of soft limits for demands and rates are also shown in Figure A3.1(a)

### A3.2 CURTAIN LIMITERS

The imposition of simple limits on demand and rate confines the output to a rectangle in the phase plane. There are benefits in considering a region of an alternative shape and in particular a region shaped so that close to the extremes of the demand's range the rates are limited more severely than in the central portion of the range. Such a situation is depicted in Figure A1.1(b) where the corners of the rectangular region are excluded from the allowed region. The shape of the corner in Figure A3.1(b) is one of

a set proposed by RAE for the ACT Lynx system and investigated by Broad [62]. The shape is characterised by a quadrant of the conic

$$X^2+eXY+Y^2=1$$

where typical values for the parameter are e=0, corresponding to a circle (scaling to an ellipse), and e=2 corresponding to a straight line; both are shown in Figure A3.1(b). The resulting shape shape in the phase plane resembles a curtained window, which resemblance gives the limiter its name. 'Closing the curtains' or bringing the corner quadrants closer into the centre of the region gives protection against immature control law software and the curtains can be opened as experience is accumulated and the software matures. The intercept fraction parameter determines the state of the curtain: a value of 0.0 fully closes the curtains while a value of 1.0 fully opens them.

## A3.3 PHASE PLANE DIAGRAMS

The test runs presented in Reference 62 did not include results in the phase plane. When the algorithm was used to generate phase plane information the results shown in Figure A3.2 were obtained. The input demand as a function of time, t, is

$$x = 50 + 60 \sin t$$

and the parameters of the algorithm are set to limit the output to the range 0 - 100 and the rate limited to +/- 50. The curtain parameters are: e = 0 and both intercept fractions are set to 0.5. The output can be seen to be limited correctly in demand but not in rate. Moreover, there is no evidence of the curtain intercepts. The correct output is shown in Figure A3.3 which is produced by the algorithm derived in the next section, A3.4. The correct rate and demand limits have been applied and the curtains can be seen to have the stipulated intercept fraction. It can be seen that there is a noticeable skew to the phase plane response; the reason for this skew is explained within the derivation of the algorithm and is a consequence of the discretisation. An example with the curtains in a further closed position is shown in Figure A3.4 . Here the input demand as a function of time, t, is

$$x = 50 + 50 \sin t$$

and the parameters of the algorithm are set to limit the output to the range 0 - 100 and the rate limited to +/- 50 as in the previous example. The curtain parameters are: e = 0 and both intercept fractions are now set to 0.25. The tighter constraints on the

movement are clearly shown. The influence of the curtains is marked in this case and leads one to question its effect on a high performance control law such as is anticipated for the ACT Lynx application. It is an additional aspect which would appear to call for careful investigation.

## A3.4 CURTAIN LIMITER ALGORITHM

Although no derivation of an algorithm is included in Reference A1.1 it would almost certainly be similar to the following:

Let the supplied input demand be x; its rate dx/dt is y and the relationship y= dx/dt may be approximated by the backward difference:

$$y_{new} = (1/d\tau)\, x_{new} - (1/d\tau) x_{old}$$

from which the current value of the rate ynew may be calculated from $x_{new}$, the current input demand, and $x_{old}$, the demand a time step $d\tau$ previously. Therefore the point R $(x_{new}, y_{new})$ lies on the line through Q $(x_{old}, 0)$ with slope $(1/d\tau)$ as shown in Figure A3.5(a). The line QR is the characteristic line of the algorithm.

If the point R is external to the allowed region of the phase plane, as shown in Figure A3.5(b), then the values of demand and rate to be provided as output from the limiter are the x and y values of the point S where the characteristic line meets the boundary of the allowed region. The use of the characteristic line to implement the curtain limiter gives rise to the skew previously noted. The skew is determined by the slope $1/d\tau$, and the skew is reduced as $d\tau \rightarrow 0$. Where S lies on a simple demand or rate limit the calculation of the coordinates of S is straightforward; when S lies on part of the curtain then the approximation illustrated in Figure A3.5(c) is employed. A is the point on the curtain with the same x value as R and B is the point on the line QR with the same y value as A. (The simple limits are first applied to R if they are initially exceeded.) The point B is used as the approximation to S; the approximation is valid if QR is steep ( $d\tau << 1$). In fact the method can be viewed as one step of a Fixed Point iteration for finding S. For a single step the configuration is symmetric in all quadrants, which simplifies the algorithm. If a more accurate solution were required then additional steps of the iteration would involve distinguishing between the quadrants.

It is interesting to note that an application of the trapezium rule leads to an alternative procedure based on

$$y_{new} = (2/d\tau)\, x_{new} - (2/d\tau)x_{old} - y_{old}$$

where additionally the previous value of the rate is required. In this case R' $(x_{new},y_{new})$ lies on the line through Q' $(x_{old},-y_{old})$ with slope $(2/d\tau)$, as shown in Figure A3.5(d).

## A3.5 THE ALGORITHM PROCEDURE

The algorithm described above was implemented in Acorn BASIC V procedure for extensive testing. The following is a commented listing of the procedure:

```
DEF PROCcurtain(oldem,indem,RETURN outdem, RETURN outrate)
REM ***************************************************
REM This curtain limiter procedure takes in the current value of the demand
REM  and outputs the limited demand and rate.
REM  indem - the current value of the input demand
REM  oldem - the value of the output demand at the previous frame
REM  outdem - the output demand from the limiter
REM  outrate - the output rate from the limiter (not normally used subsequently)
REM
REM Global variables accessed by the procedure
REM
REM actl,acth - minimum and maximum demands, that is, demand limits
REM actc - datum for limiter (not used )
REM maxrate - maximum value for magnitude of rate, that is, rate limit
REM dt - frame time interval
REM  ecc - eccentricity, curtain limiter parameter e
REM  dx,dy  - fractional intercepts, curtain limiter parameters
REM ***************************************************
LOCAL x1,y1,x2,y2,x3,y3,xmax,xc,hx,hy,inrate
REM
REM first limit actuator demand
REM
IF indem > acth THEN
        x1=acth
ELSE
        IF indem < actl THEN
                x1=actl
        ELSE
                x1=indem
        ENDIF
ENDIF
REM
REM actuator demand is now x1
REM
REM calculate rate
REM
```

```
inrate=(x1-oldem)/dt
REM
REM limit rate and find corresponding demand
REM
IF inrate > maxrate
THEN
        y1=maxrate
        x1=oldem+dt*y1
ELSE
        IF inrate< -maxrate
        THEN
                y1=-maxrate
                x1=oldem+dt*y1
        ELSE
                y1=inrate
        ENDIF
ENDIF
REM
REM actuator rate is now y1
REM
REM calculate half range and mid value of demand to use in scaling
REM
xmax=(acth-actl)/2
xc=(acth+actl)/2
ymax=yrate
REM
REM scale and use absolute value to confine to first quadrant
REM
x2=ABS(x1-xc)/xmax
y2=ABS(y1)/ymax
REM
REM scaled variables for demand and rate are now x2 and y2
REM
REM transform to coordinates at the centre of the conic curtain
REM
hx=1-dx
hy=1-dy
x3=(x2-1)/hx
y3=(y2-1)/hy
REM
REM x3, y3 are scaled demand and rate relative to the centre of the conic curtain
REM
REM check whether inside conic, if TRUE then place on boundary at x3
REM
IF (x3*x3+ecc*x3*y3+y3*y3)<1 THEN
        y3=-.5*(ecc*x3+SQR(x3*x3*(ecc*ecc-4)+4))
        y2=1+y3*hy
ENDIF
REM
```

```
REM transform back to original coordinates
REM
outrate=y2*ymax*SGN(y1)
outdem=oldem+outrate*dt
ENDPROC
```

It will be observed that the intercepts dx and dy must lie in the interval [0,1). The parameter ecc (e) is intended to lie in the interval [0,2] for the type of limiter being considered in the present work.

# APPENDIX 4

## THE SYSTEM TEST

Summary.

This appendix describes the development of testing procedures during the evolution of the ACT Lynx system specification. Some general conclusions pertinent to complex systems are drawn from the experience.

### A4.1 ORIGINS

In the early studies [8] of the feasibility of the ACT Lynx there was the notion of a comprehensive test of the ACT system prior to flight. Since the test was to be largely automatic it was referred to in convention terms as Built in Test Equipment (BITE). It was also accepted that the test would involve adjusting the pitch of the rotor blades, either by inceptor movements or by programmed positional changes injected into the actuators. Since such activities could not be performed in-flight a more modest pre-engagement test, Airborne (A/B) BITE, was envisaged once the aircraft had left the ground. In fact, apart from a button marked 'system test' there was little specification as to the detail of these tests - as can be judged from Figures 3.1 - 3.3 where the BITE process appears to do nothing but illuminate the lamp on the control panel. There was little detail added as the specification evolved through Versions 1 and 2 since the argument was that the test would be dependent on the particular implementation of the system and therefore could not be specified until the implementation plans were being prepared. An argument to avoid proper specification activity is always seductive; particularly so in this case where a poll of the design team to establish those parts of the system which should be exercised in the system test procedure produced a list so extensive and unstructured as to make its casting into a specification a substantial task. Nevertheless in the planning of Version 3 it was ordained that such vagueness was unacceptable and that the design team should specify the test with as much precision as was possible in the absence of implementation information.

### A 4.2 A STRUCTURED APPROACH

Much of the difficulty with the system test came from the diversity of the requirements that were suggested by the various members of the design team. Examples are:

- it should ensure that when the system is engaged that inceptor movement actually moves the blades,

- it should ensure that all the triplex lanes are working properly,

- it should make sure that the RAM in the computing systems is valid.

Now these represent three quite different types of functionality of the system. The first is concerned with the pilot's interface - in this instance the inceptors. The second is concerned with the functioning of the ACT system in terms of its redundancy and monitoring; while the third is related to the validity of the basic components of the system. Version 3 of the specification separated out these different concerns in a structured or layered approach. The top layer is the pilot's interface: inceptors, Pilots Control Panel, Menu Panel, etc. and it is this layer alone which is the subject of the system test. The intermediate layer, the internals of the ACT system itself, is handled by the integral fault management, and the lowest layer - that of component level - is actually ignored in Version 3 although, as commented on below, it can be accommodated in a straightforward manner in the three-layer frame-work. With such an approach the FUNCTION entry for the system test becomes:

*There shall be a facility for conducting a test of the ACT system. It should test the correct operation of the following:*

*(a) Control of the blade pitch by the inceptors.*
*(b) The series actuators.*
*(c) The parallel actuators.*
*(d) The arm, engage, disengage actions.*
*(e) The selection of Control Laws.*

*...etc.*

The corresponding content of the OPERATION entry is then, after a description of the interactions via the menu panel:

*(a) Control of the blade pitch by the inceptors.*
*The system shall be engaged in the normal way in response to menu panel messages and then inceptor movements prompted.*

*(b) The series actuators.*
*The inceptor management process shall inject suitable signals into the engaged system to drive the series actuators.*

*...etc.*

All of the pilot interfaces, including the fault management diagnostic messages, are part of this schedule. One advantage of the scheme is that it is quite clear which aspects of the system are to be included in this test - now called System Test rather than BITE. Correspondingly it is clear that the remaining aspects of the system validity must be dealt with by the fault monitoring so there is now some important guidance on the required scope of the fault management. The lowest level, though ignored in Version 3, is accommodated in a straightforward way by this approach. All that is needed is for the lowest level BITE to make a validity signal to be made available to the monitoring processes at the next level up. It will be observed that even this level of description of system test lacks total precision although it is a significant improvement on its predecessors. Another problem was that even focussing solely on the pilot interface resulted in a large elaborate structure for the system test process when the simulation was being prepared. This complexity could cause difficulties should the test need to be modified to accommodate evolution in the system architecture. Although not implemented some proposals have been made for overcoming these difficulties [63]

A4.3 DEVELOPMENTS

As a means of specifying in detail a flexible, yet comprehensive, system test Moore [63] suggests a tool for automatic generation of the process from a standardised description. After an initial statement of the conditions which must apply throughout the test, such as rotor brake on, there is a list of individual tests. Each test is described in four parts:

(a) A statement of the preconditions for the test.

(b) A list of test messages.
The messages are either instructions to the pilot or the initiation of automatic injection of test data.

(c) The interval for a time-out on the test is specified.

(d) The conditions which define a 'pass' for the test are listed.

For the preparation of the JSD specification and simulation code. The file containing this information is then subject to preprocessing to produce the system test process structure and operations automatically. In addition to the obvious benefits in reducing the work involved and in subsequently introducing modifications to the test process there is the less obvious advantage of forcing the specifier to confine the tests within the standard description. There is less chance of the tests being specified incorrectly or incompletely when they have to be entered into a pro forma. It is this pro

forma which makes the test description manageable and enables the automatic generation to be straightforward.

A remaining difficulty is to match the vocabulary of the written specification to the variable names in the simulation. For example suppose that the specifier wishes to introduce within the system test a further test that requires as a precondition that the rotor is not rotating (Ignoring, for simplicity, that this is actually a requirement for the whole system test). One of the preconditions will be:

'rotor speed must be zero'.

To express this in a form which can be incorporated directly into the simulation it must be written in terms of the appropriate variable name such as

ROTOR_SPEED = 0.0,

or

ROT_SPEED =0.0,

or even, if the rotor speed sensor needs some processing to detect its zero position:

ROTOR_SPEED_ZERO = TRUE.

It is clear that the specifier needs a detailed knowledge of the specification in order to express the conditions correctly. Another problem is that of ambiguity in the text. For example, whether 'rotor brake is on' is related directly to the rotor speed being zero is a question that can only be answered by the specifying engineer. The simplest way to deal with this situation is to have one, master, version of the specification written in normal text, which is then converted to its equivalent version using variable names. The conversion would occur as part of the preparation of the full JSD specification from the written text, so that the treatment of the system test description would be consistent with the treatment of the specification as a whole.

## A4.4 CONCLUSIONS

It is instructive to draw out from the work on the system test those lessons which have more general applicability than the ACT Lynx application. It is generally recognised that testing and fault management should be developed in parallel with the development of the principal design concept. Unfortunately, it is unlikely that such an integrated approach will ever become universal practice, particularly when the applications are prototypes or solely for research. Nevertheless, what can be reasonably done in this respect should be done. Specific conclusions are:

(a) The system test should not be a blank left to be completed only when the rest of the specification has been finalised.

(c) When a specification is being drawn up, it must be subject to the discipline of associating with every function the manner of its testing. The testing may involve some interactive procedures or rely on the embedded fault management.

(d) The automatic generation of a system test process has important potential particularly in the rapidly changing environment of a prototype application.

These conclusions are not revolutionary (except possibly (d)); but their adoption should ensure that the system-test aspects do not frustrate the completion of a specification.

# APPENDIX 5

## PACKAGE SPECIFICATIONS

Summary

This appendix contains listings of the package specifications of the modules provided for the ACT Lynx simulation. For convenience in the total simulation they were ultimately incorporated into the Outside Simulation hardware unit. The package bodies may be found in the simulation documentation [55] or the delivered software; they are omitted here solely because of the space that the listings would require.

### A 5.1 SYSTEM_MATRIX

This package contains the system and control matrices used in the linear helicopter model in the ACT Lynx simulation. The model takes the actuator displacements and generates the corresponding kinematic data for the sensors. The package also contains the trim state and control vectors together with the gain matrix which is used to convert from actuator displacements to blade pitch values. The values supplied in this package correspond to a Lynx helicopter in horizontal rectilinear flight at 80 knots and were obtained from the HELISTAB package [56]. For brevity the numeric values are omitted.

```
Package System_Matrix is
type Matrix is array (Integer range <>,Integer range <>) of Float;
type Vector is array (Integer range<>) of Float;
--*************************************************
--
-- System Matrix A with ordering
-- 1:u, 2:w, 3:q, 4: theta, 5:v, 6:p, 7:phi, 8:r
-- for trim at 80.0 KNOTS imperial units
--
*************************************************
A: Constant Matrix(1..8,1..8):=
(1=>(1=>...
                                              ...));
--*************************************************
--
-- Control Matrix B with ordering
-- 1:thetac, 2:theta1s, 3:theta1c, 4:thetatr
-- collective, long-cyclic, lat cyclic, tail collective
--
--*************************************************
B: Constant Matrix(1..8,1..4) :=
(1=>(1=>...
                                              ...));
--*************************************************
--
```

```
-- Trim state at 80 KNOTS
--
--*************************************************
x0: Constant Vector(A'Range(2)):=
(1=>...                                          ...);
--*************************************************
--
-- Trim controls at 80 KNOTS
--
--*************************************************
u0: Constant Vector (b'Range(2)):=
(1=>...                                          ...);
--*************************************************
--
-- Gain Matrix, percent to radians
--
--*************************************************
G: Constant Matrix (b'Range(2),1..2):=
(1=>(1=>...
                                                 ...));
end System_Matrix;
```

A5.2 LINEAR_HELI

The state vector is initialised and updated by this package. Sensor data for the triplex AMSE and the duplex ADSE are derived from the model. A Runge-Kutta integration routine is used.

```
with System_Matrix; use System_Matrix
Package Linear_Heli is
--*************************************************
--
--amsu components are ax,ay,az,p,q,r,theta,phi,psi
--adsu components are u,v,w,h (altitude)
--
--*************************************************
subtype Amsu is Vector(1..9);
subtype Adse is Vector(1..4);
amse: array (1..3) of Amsu;
adse: array(1..2) of Adsu;
x: Vector(A'Range(2));
u: Vector (B'Range(2));
procedure Initialise;
procedure RK4 Step(h: Float);
end Linear_Heli;
```

A 5.3 AFCS_PACKAGE

The standard Lynx autostabiliser and computer acceleration control equipment is modelled by this package. It calculates AFCS demands for the series actuator on the basis of kinematic information and pilot demands which are available from the packages Linear_Heli and Actuator_System respectively. The AFCS is duplex so the reference to Lanes is in fact 2; the Lane information is available from Actuator_System. The same

control law was supplied to form part of the CLE as one of those available in ACT mode.

```
with System_Matrix; use System_Matrix;
with Linear_Heli; use Linear_Heli;
with Actuator_System; use Actuator_System;
package AFCS_package is
type AFCS_demand_sets is array (Lanes) of Positions;
AFCS_demand_set: AFCS_demand_sets;
procedure Update_AFCS;
end AFCS_package;
```

## A 5.4 ACTUATOR_SYSTEM

The actuator system package provide types which are common to both series and parallel actuators. This includes the state of the hydraulic bypass valves in a dual duplex configuration and the pick off information for positional feedback and monitoring. The Axes type refers to the number, 4, of control lanes in order to distinguish it from the lanes and sublanes of the architecture.

```
package Actuator_System is
subtype Axes is Integer Range 1..4;
subtype Lanes is Integer Range 1..2;
subtype Sub_lanes is Integer Range 1..2;
type Positions is array (Axes) of Float;
type Bypass_pairs is array (Sub_lanes) of Boolean;
type Bypass_set type is array(Lanes) of Bypass pairs;
type Pickoff_pairs is array (Sub_lanes) of Position;
type Pickoff_set_type is array (Lanes) of Pickoff_pairs;
end Actuator_system;
```

## A 5.5 PARALLEL_ACTUATOR_SYSTEM

In the simulation for system evaluation, rather than as a flight-mechanics study, it is necessary to retain in the actuator model details relevant to the monitoring and reconfiguration. Consequently, the dual duplex engage and disengage switch position information is handled in detail even to the extent of modelling the microswitches attached to the bypass valve solenoids in order to detect engagement and disengagement. The individual drive signals and connexion state are also modelled, as are the hydraulic supply states of the actuator lanes. The update procedure contains a simple dynamic model of an actuator but there is a functional separation of drive-signal consolidation, valve dynamics, port characteristics and flow rate properties so that a more complex model may be conveniently incorporated once a data set is available.

```
with Actuator_system; use Actuator_system;
package Parallel_actuator_system is

type Engage_pairs is array (Sub_lanes) of Boolean;
type Disengage_pairs is array (Sub_lanes) of Boolean;
```

```
type Engage_sets is array (Lanes) of Engage_pairs;
type Disengage_sets is array (Lanes) of Disengage_pairs;
type Parallel_drive is array is array (Axes) of Float;
type Parallel_drive_pair is array (Sub_lanes) of Parallel_drive;
type Parallel_drive-sets is array(Lanes) of Parallel_drive_pair;

Parallel_actuator_position: Positions;
Bypass_set, Microswitch_set; Connexion_set: Bypass_set_type;
Pickoff_set: Pickoff_set_type;
Hydraulics: array(Lanes) of Boolean := (False; False);
Control_run_position:Positions;
Engage_set: Engage_sets;
Disengage_set: Disengage_sets;
Parallel_drive_set:Parallel-drive_sets;

procedure Initialise_parallel_actuator;
procedure Update_parallel_actuator(h: Float);

end Parallel_actuator_system;
```

### A 5.6 SERIES_ACTUATOR_SYSTEM

The package specification for the series actuator is similar to that of the parallel actuator with the exception of the engage and disengage information for the hydraulics. The series actuator must operate at all times when that part of the system is powered up even if the main ACT system is off. Consequently the assumption for the current simulation is that the main hydraulics are always on. Internally the package body has some differences since the real series actuator has separate valves for each lane and a pivot mechanism consolidates the actuator position. This variation is not visible at the specification level, of course.

```
with Actuator_system; use Actuator_system;
package series_actuator_system is
type Series_drive is array(Axes) of Float;
type Series_drive_pair is array(Sub_lanes) of Series_drive;
type Series_drive_sets is array(Lanes) of Series_drive_pair;

Pickoff_set: pickoff_set_type;
Series_drive_set: Series_drive_sets;
Series_actuator_position: positions;
Connexion_set: Bypass_set_type;

procedure Initialise_series_actuator;
procedure Update_series_actuator(h: Float);

end Series_actuator_system;
```

### A 5.7 P_F_C

The package to model the existing Lynx primary flight control unit is a similar actuator model to that incorporated in the series and parallel actuator packages. The inputs are simply the control run positions from the parallel actuator together with the

series actuator positions. The output is the PFCU actuator position for the blade pitch control of the helicopter model.

```
with Actuator_system; use Actuator_system;
with Series_actuator_system; use Series_actuator_system;
with Parallel_actuator_system; use Parallel actuator_system;
with Linear_heli; use linear_heli;
with System_matrix; use System_matrix;
package P_F_C is

procedure Initialise_P_F_C;
procedure Update_P_F_C(h: Float);

end P_F_C;
```

## A 5.8 CONCLUSIONS

There was no difficulty in incorporating these modules into the total specification and code generation exercise. In general, it is probable that any simulation of a complex system will need to import existing software modules, so as an enforced exercise it was not without its value.

# APPENDIX 6

# INJECTION OF ERRORS

Summary.

The need for a mechanism for injecting errors is discussed. The types of errors which must be considered are surveyed and a subset suitable for validation of a specification is selected. The relationship to the hardware description is defined.

## A 6.1 ERROR INJECTION

It is not sufficient merely to simulate a fault tolerant system over a period of time in order to demonstrate its tolerance. In a practical situation the reliability of the simulation environment is not sufficiently representative of the final implementation. Even if it were then it would be out of the question to investigate reliabilities of the order of 1 fault for every $10^9$ operating hours. Therefore it is necessary to modify the simulation environment and incorporate a facility for introducing errors of a specified kind at a predetermined time for specified interval. In simulation terms, of course, such a facility is not part of the simulation of the specification; it is an intrusion of the outside world into the simulation in order to evaluate certain aspects of the specification. The modelling of the outside world is a feature that needs careful consideration.

## A6.2 TYPES OF FAULTS

It is necessary to scope the types of fault that the system must be able to tolerate, that is, either withstand its effects through inherent robustness or detect it by monitoring in order to initiate a reconfiguration. The ACT Lynx design incorporated both of these features together as far as possible. Based on the work of Johnson [47] one can specify reasonable requirements for fault injection. An initial proposal was that for each process and data stream the following faults should be modelled:

(a) Cessation of operation.

(b) Output of rogue values.

(c) Output of an unchanging value.

It is clear that for a complex system the amount of software that such a treatment would introduce would be enormous. Fortunately it is possible to refine these

requirements by observing that if one process ceases operation then one would expect those processes which share the same processor to also cease. Therefore one can allocate cessation of operation at a unit level. It also makes sense to allocate communication faults at a unit level and specify error injection for connections. The output of an unchanging value (stuck value) is straightforward, but defining 'rogue' in the list (b) above is difficult. If by rogue it is meant that it is out of range for the Ada variable, then this would cause a run-time error if checking is switched on. If it is merely intended that the value is unusual compared to previous values of the same variable then one would expect the voting and monitoring to provide the correct interpretation. It is preferable to avoid the ambiguity of the word 'rogue'.

Johnson [47] defines an error as incorrect information which has arisen because of a particular fault. That is, errors are the evidence that a fault has occurred. Therefore one could pose the question whether the proposals above inject faults or inject errors. The semantics are satisfied if one views the modelling and inclusion of faults as the injection of faults, but the errors which these models produce as the injection of errors.

## A 6.3 HARDWARE DESCRIPTION

The development of a top level unit/connection description of the specification enables the error injection facility to be incorporated at this level. It is an appropriate level because it can, as in the ACT Lynx application, reflect both the hardware decomposition and the redundant architecture of the system. Therefore, it is reasonable to categorize faults as lying within these units or in the connections between them. For example in the ACT Lynx simulation [53] it is possible to specify that a particular unit will cease operation at a particular instant for specified interval. It is also possible to specify a connection failure where a particular connection between units is set to a constant value during a stipulated interval of time. The constant value on the connection is specified by a 32 bit binary number. Although these facilities may appear to be limited in scope, in practice they are surprisingly flexible, and more than adequate to accommodate a lengthy validation programme.

## A 6.4 CONCLUSIONS

It is necessary to model explicitly the occurrence of faults in order to validate a fault tolerant system. It is also necessary to scope carefully the types of fault that are to be modelled. Based on experience with the ACT Lynx system, it is recommended that the anticipated validation programme be drawn up in advance of - or at the same time as - the modelling of faults. Then the fault modelling can be specifically oriented towards the required validation.

## REFERENCES


1. Padfield, G. D., Bradley, R. & Moore, A. *The development of a requirement specification for an experimental active Ffight control system for a variable stability helicopter - an Ada Ssmulation in JSD,* Agard CP -503, Software for Guidance and Control, AGARD, 1991.

2. Padfield, G.D. and Bradley, R., *Creation of a living specification for an experimental helicopter active flight control system through incremental simulation,* Paper No. 91 - 74, Seventeenth European Rotorcraft Forum, Berlin, 1991.

3. Padfield, G.D., (Editor), *Helicopter Handling Qualities and Control,* Proceedings of the R.Ae.Soc Conference, London, 1988.

4. Winter, J.S. & Padfield, G.D., *A discussion paper on an ACT flight research programme using the RAE Bedford Lynx,* RAE Tech Mem FS(B) 523, 1984.

5. Padfield, G.D. & Winter, J.S., *Proposed programme of ACT research on the RAE Bedford Lynx,* RAE Tech Mem FS(B) 599, 1985.

6. Tomlinson, B.N., Padfield, G.D. & Smith, P.R., *Computer-aided control law research from concept to flight test,* Agard CP-473, Computer Aided System Design and Simulation, AGARD,1990.

7. Winter, J.S., Padfield, G.D. & Buckingham, S.L., *The evolution of active control systems for helicopters; conceptual simulation to preliminary design.* Proceedings of the AGARD FMP symposium on ACS, Toronto, AGARD, 1984.

8. Thomson, K., *The results of the WHL feasibility study in support of the RAE Bedford flight controls research programme,* Systems Technology Note STN 19/84, Westlands Helicopters, 1984.

9. Freeman, P. and Wasserman, A.J., *Software Development Methodologies and Ada, US Department of Defense - Ada Joint program Office,* 1982.

10. Department of Industry, *Ada-based system development methodologies study report,* Volume 1, DoI, London, 1981.

11. Birrel, N.D. & Ould, M.A., *A Practical Handbook for Software Development,* Cambridge University Press, 1985.

12. Mullery, G.P., *CORE - A method for controlled requirement specification,* In Proceedings of the 4th International Conference on Software Engineering, 1979.

13. Mascot Suppliers Association, *The official handbook of MASCOT,* MSA, Malvern, 1980.

14. Mascot Suppliers Association, *MASCOT 3,* MSA, Malvern, 1986.

15. Various, *Special issue on MASCOT 3,* Software Engineering Journal, Vol 1, No 3, May 1986.

16. Jackson, M., *System Development.* Prentice Hall, 1983.

17. Cameron, J.R., *JSP &JSD: The Jackson approach to system development,* IEEE Computer Society Press, 1983.

18. Thewlis, D.J., *A survey of available tools and methods for software requirements capture and design,* Agard CP-503: Software for Guidance and Control, AGARD, 1991.

19. SofTech, Inc., *An Introduction to SADT. Structured Analysis and Design Technique,* SofTech Report 9022-78R. SofTech, Inc. Mass., 1976.

20. Yourdon, E. and Constantine, L.L., *Structured Design,* Englewood Cliffs, N.J. Prentice Hall, 1978.

21. De Marco, T., *Structured Analysis and System Specification.* Yourdon Press, New York, 1978.

22. Ward, P.T. and Mellor, S.J., *Structured Development for Real Time Systems*, Yourdon Press, 1985.

23. European Space Agency, *HOOD Reference Manual*, ESA, 1989.

24. Stroustrup, B., *The C++ Programming Language*, Addison Wesley, 1987.

25. Cameron, J.R., *The evolution of JSD into a properly Object-Oriented method*, Jackson User Group Meeting, London, 1991.

26. Bradley, R. *A System for Parameter Estimation in Helicopter Dynamics and Data Management in a Flight Simulator Environment. PART 1*, Newcastle upon Tyne Polytechnic, School of Mathematics Statistics and Computing, Research Report, September, 1985.

27. Bradley, R. *A System for Parameter Estimation in Helicopter Dynamics and Data Management in a Flight Simulator Environment. PART 2*, Newcastle upon Tyne Polytechnic, School of Mathematics Statistics and Computing, Research Report, September, 1985.

28. Bradley, R. Minutes of ACT Lynx technical meeting meeting, Yeovil, November, 1987.

29. Fernandezde la Mora, G., Minguez, R., Khan, S. and Villa, J.R. *A Methodology for Software Specification and Development based on Simulation*, Agard CP-503, Software for Guidance and Control, AGARD,1991.

30. Wright, B.P., *RAE ACT Lynx - Airborne system requirement specification, Issue 1.* WHL Flight Control Department Note FCDN 88/05, 1988.

31. Wright, B.P., *RAE ACT Lynx - Airborne system requirement specification, Issue 2.* WHL Flight Control Department Note FCDN 88/05, 1988.

32. Corbin, M.J. and Birkett, P.R., *Design Considerations for Systems Modelling*, FS-91-WP-614, DRA Farnborough, December 1991.

33. Jackson, M.A., *Principles of Programme Design*, Academic Press, London, 1975.

34. King, M.J. and Pardoe, J.P., *Program Design Using JSP: A Practical Introduction*, Macmillan, London, 1985.

35. Cameron J.R. *JSD course documentation.* Michael Jackson Systems Ltd, 1986.

36. Sutcliffe A., *Jackson System Development*, Prentice Hall, London,1988.

37. Michael Jackson Systems Ltd., *Version 3 of Speedbuilder for IBM PC/Compatibles: Installation Guide*, MJSL, 1989.

38. LBMS Plc., *Jackson Work Bench User Guide* (In preparation), 1992.

39. Flower C.R., *ACT SYSTEM Finite State Machine*, ACT Lynx Technical Note, RAE, January 1991.

40. Bradley R., *ACT Lynx Control panel/ Supervisor Design Study: Version 3.1*, ACT Lynx Technical Note, Department of Aeronautics and Fluid Mechanics, Glasgow University, May, 1988.

41. RAE, ACT Lynx Airborne System Requirements Specification Issue 3.A, RAE, 1989.

42. Cameron, J. R., *The Use of JSD for Flight Control Software*, RAE, 1986.

43. Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.

44. Bradley, R., *The Use of Jackson System Development in the ACT Lynx Specification,* ACT Lynx Technical Note, RAE, 1989.

45. Jewel, C., *MODAS analysis system - system overview.* Prosig Computer Consultants, 1986.

46. D.T.I./N.C.C., STARTS Purchasers' Handbook: *Procuring software-based systems,* NCC Publications Second Edition, 1989.

47. Johnson, B.W., *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.

48. Silva, A., *Mode Synchronisation Algorithm for Asynchronous Autopilot*, Paper No. 38, Fourteenth European Rotorcraft Forum, Milan, 1988.

49. Bradley, R., *Simulation of the RAE Lynx AACTS*, ACT Technical Note 020889, Department of Aerospace Engineering, Glasgow University, August 1989.

50. Cameron, J.R., *Mapping JSD specifications into Ada*, Proceedings of the 6th Ada (UK) Conference, 1987.

51. Lawton, J.R. and France, N., *The transformations of JSD specifications into Ada*, Ada User, January, 1988.

52. Michael Jackson Systems Ltd., *Program Development Facility: Installation Guide (Version 2.1) IBM PC Version*, UKAEA, 1989.

53. LBMS Plc., *A Users' Guide to the Hardware Simulation Addition to Adacode*, LBMS, December 1990.

54. Mc Dermid, J., *Skills and Technologies for the Development and Evaluation of Safety Critical Systems*, In Safety of Computer Control Systems 1990 Ed. Daniels, B.K., IFAC Symposia Series, Number 17, Pergamon, 1990.

55. Friedland, B., *Increment 6 Documentation*, Contract AWL12A/141, LBMS , 1990.

56. Smith, J., *An Analysis of Helicopter Flight Mechanics Part 1- Users Guide to the Software Package HELISTAB*, RAE TM FS(B) 569, October 1984.

57. Gourlay, A.R. and Watson, G.A., *Computational Methods for Matrix Eigenproblems*, Wiley, 1973.

58. Ince, D.C., *An Introduction to Discrete Mathematics and Formal Specification*, Clarendon Press, 1988.

59. Bradley, R., *A design for the enhancement of the reliability of a Control Input Device by means of a multiprocessor system,* Technical note, Dept. Aeronautics and Fluid Mechanics, Glasgow, 1986.

60. Cameron J.R. "An overview of JSD", IEEE Trans. Software Eng., Vol. SE-12, No. 2, 1986.

61. Tomlinson, B.N., Bradley, R. Flower, C: *The use of a relational database in the management and operation of a research flight simulator.* AIAA Flight Simulation Conference, Monterey, California, August, 1987.

62. Broad, T., *The ACT Lynx Actuation Study Part 3: The Curtain Limiter,* WP FM(89) 041, RAE, July, 1989.

63. Moore, A., *A proposal for further support to the development of the ACT Lynx system requirements specification,* RT/JSW/9110001, LBMS, October, 1991.

## Figures

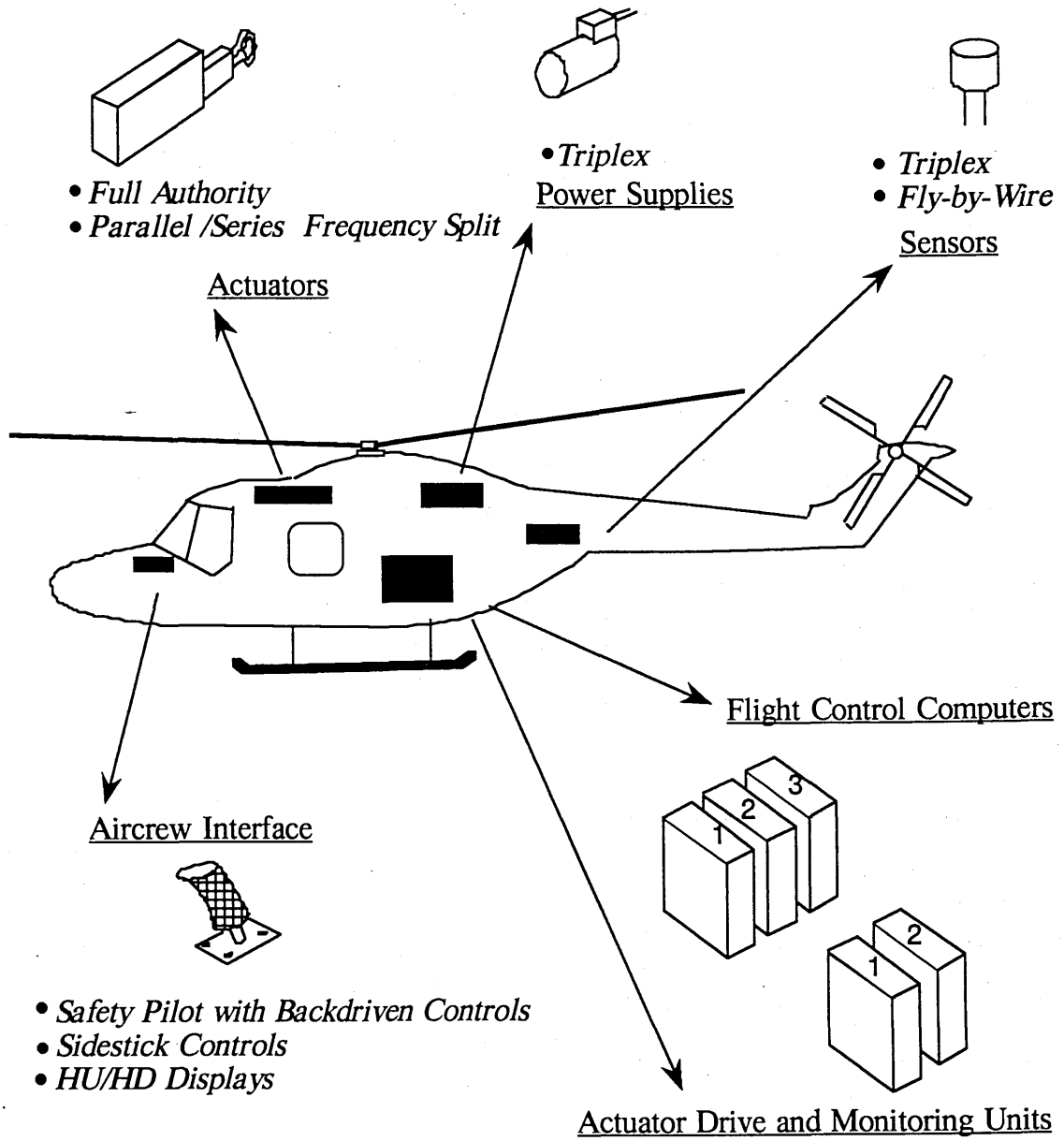



Figure 1.1 Proposed ACT Lynx System Architecture

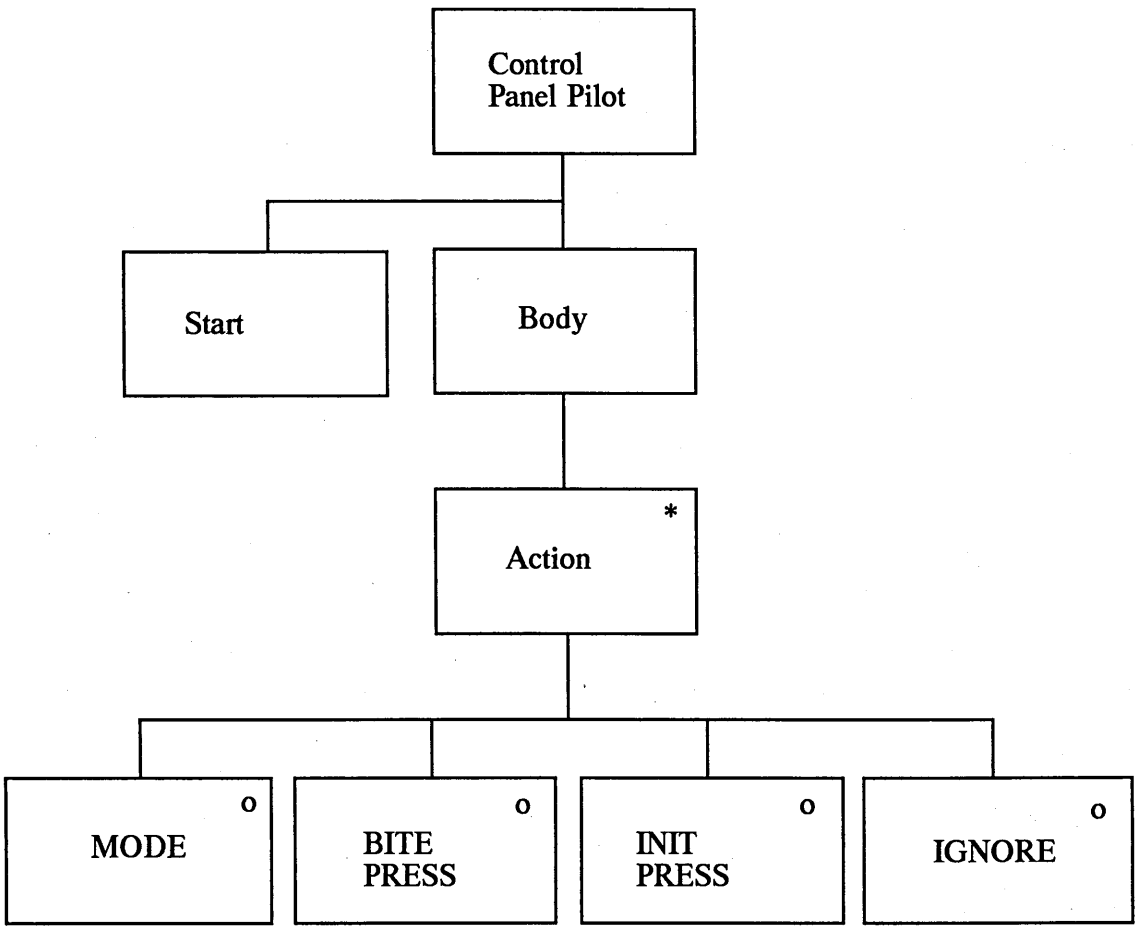
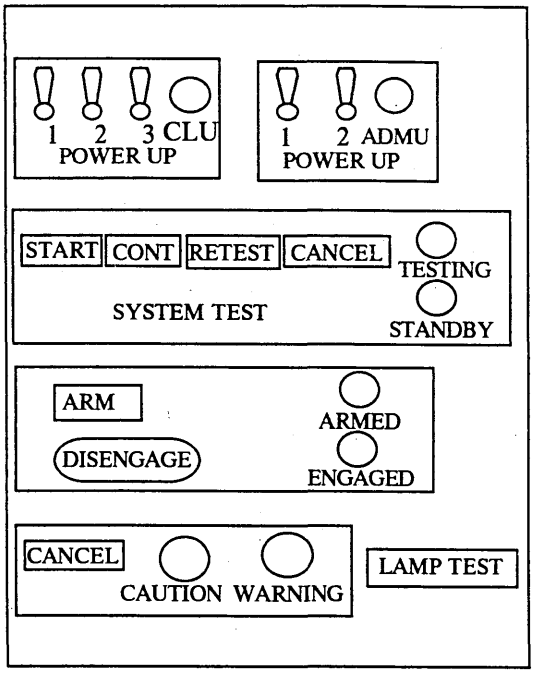
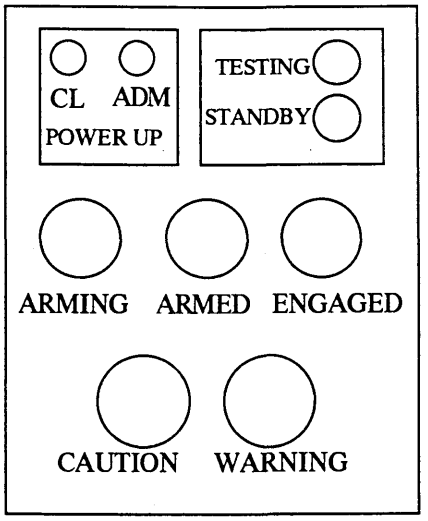


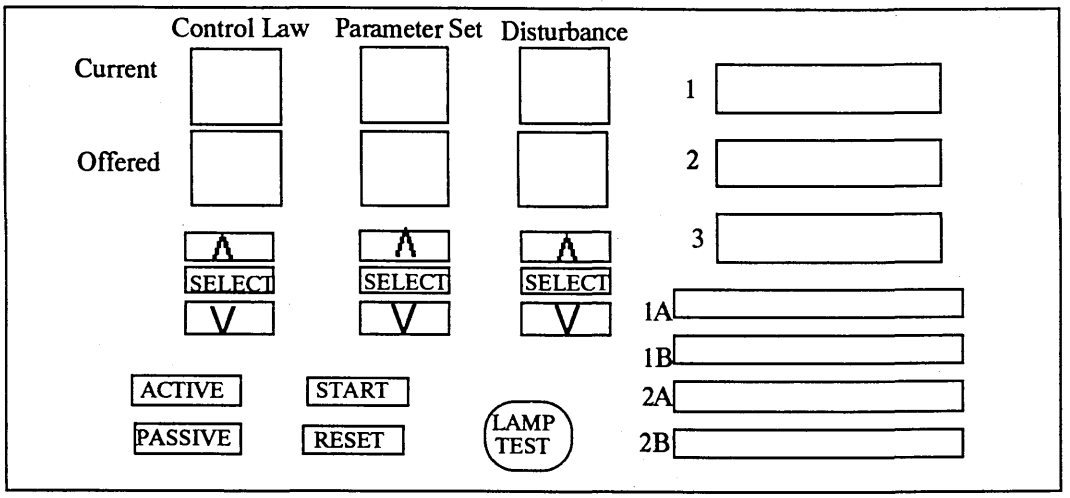

Figure 1.2 Early Pilot Model




Pilots Control Panel




Repeater Panel




Menu Panel

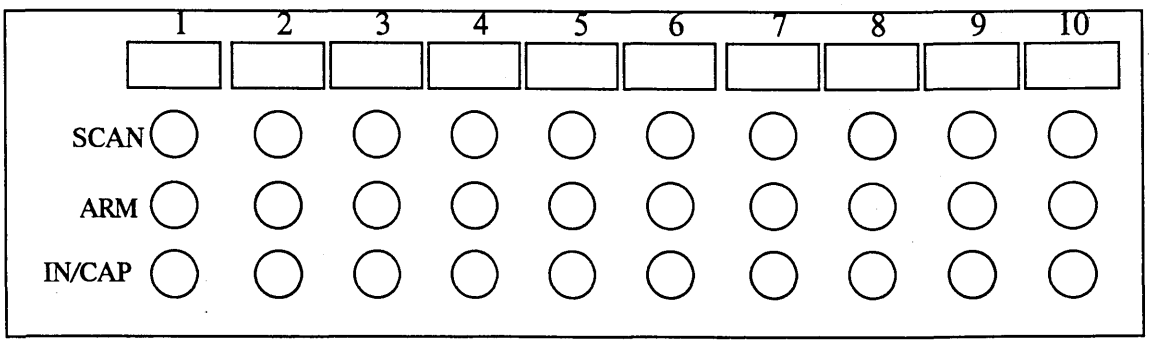



Figure 1.3. Act Lynx Schematic Control Panels

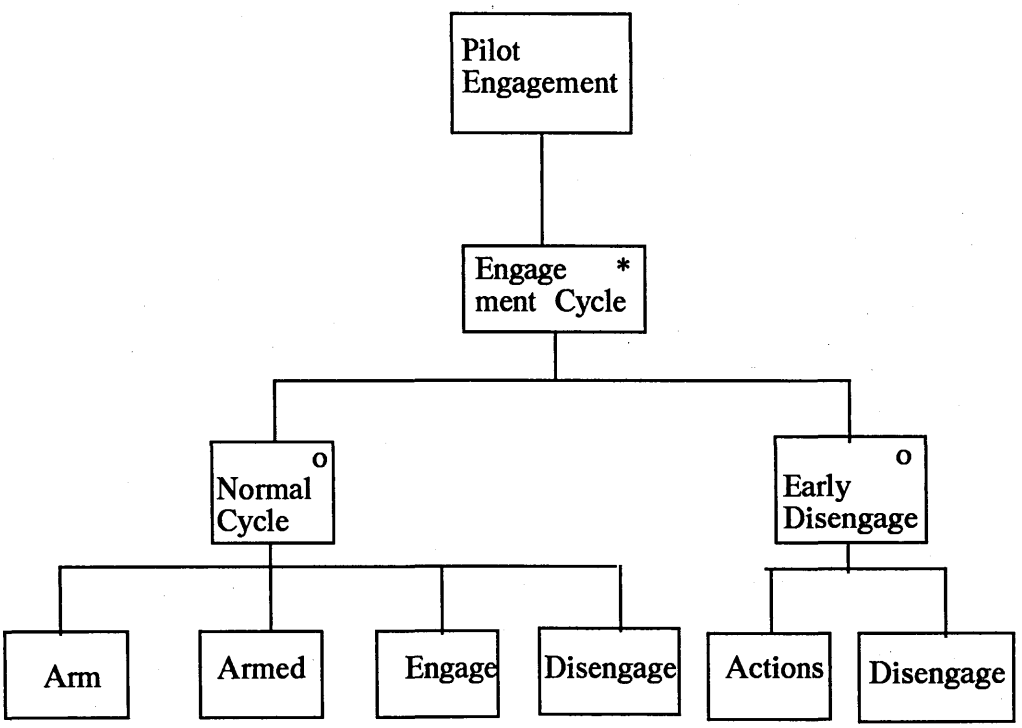
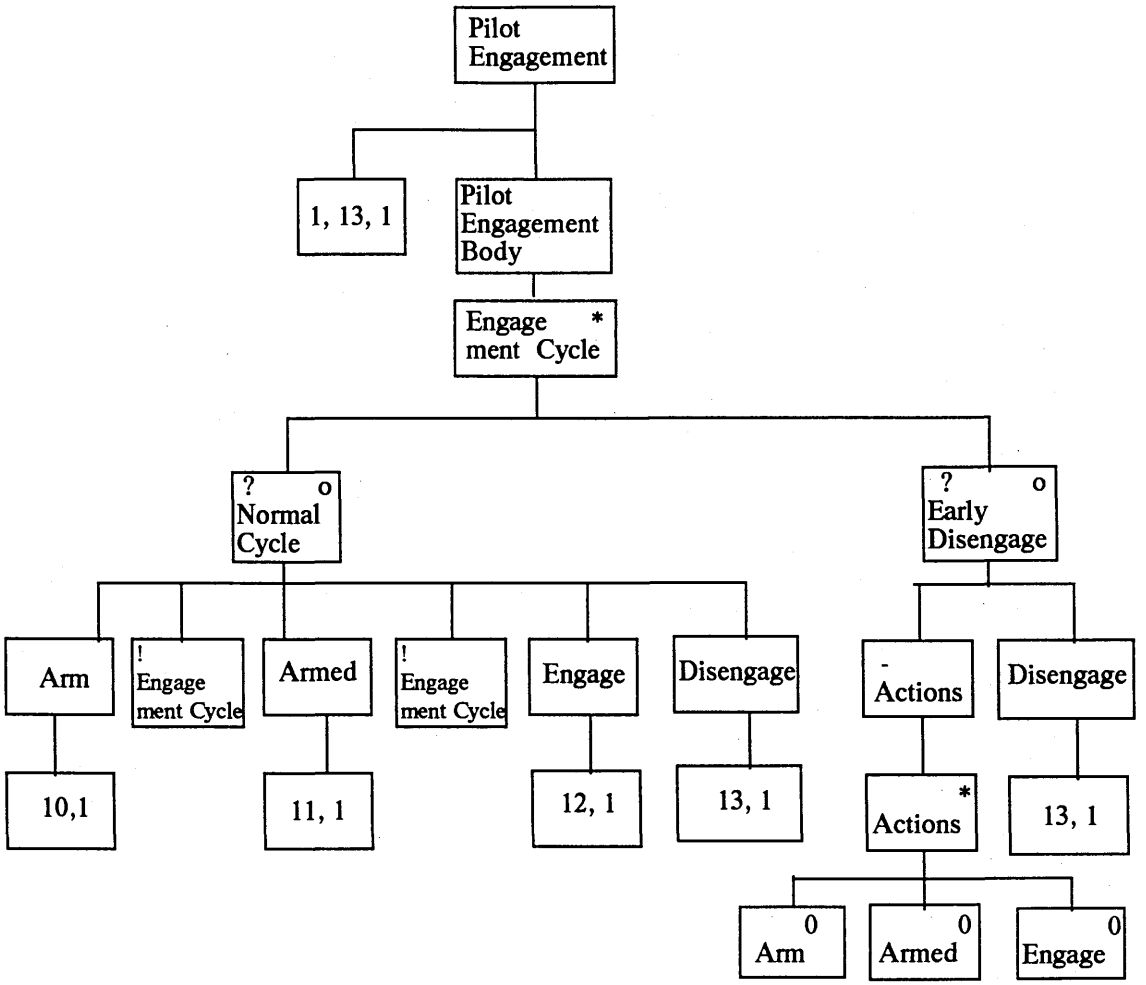



Figure 2.1 Pilot Engagement Structure Diagram




Operations:
1. READ MESSAGE
10. SYSTEM_STATE:=ARMING;
11. SYSTEM_STATE:=ARMED;
12. SYSTEM_STATE:=ENGAGED;
13. SYSTEM_STATE:=DISENGAGED;

Figure 2.2 Pilot Engagement Program Structure

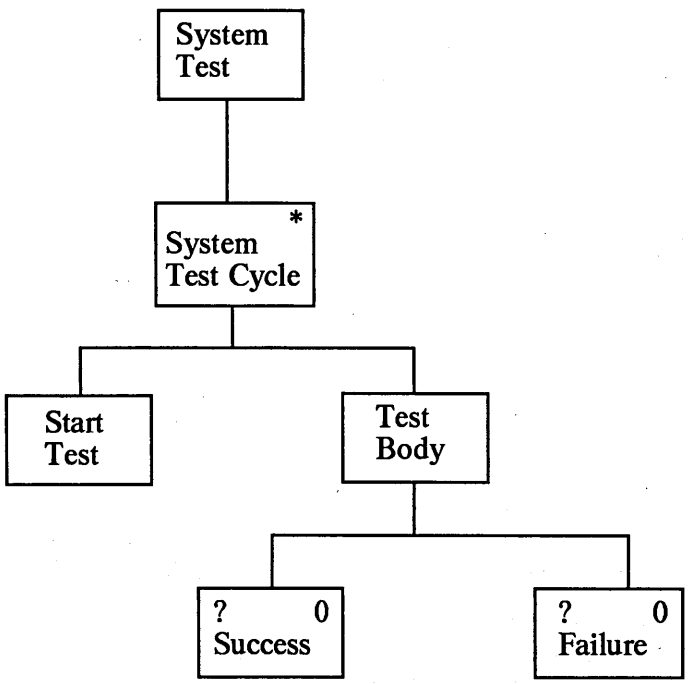
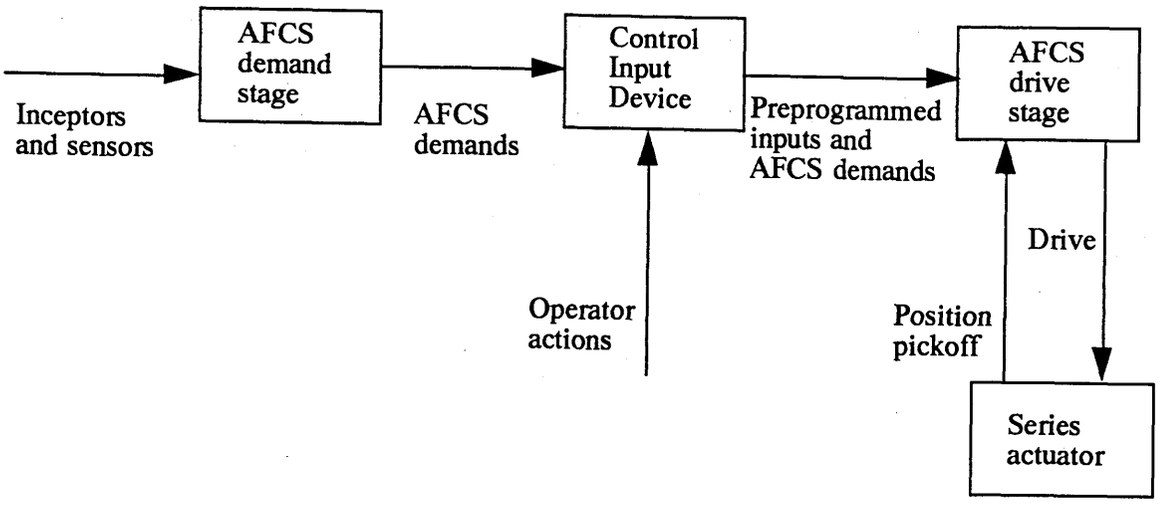


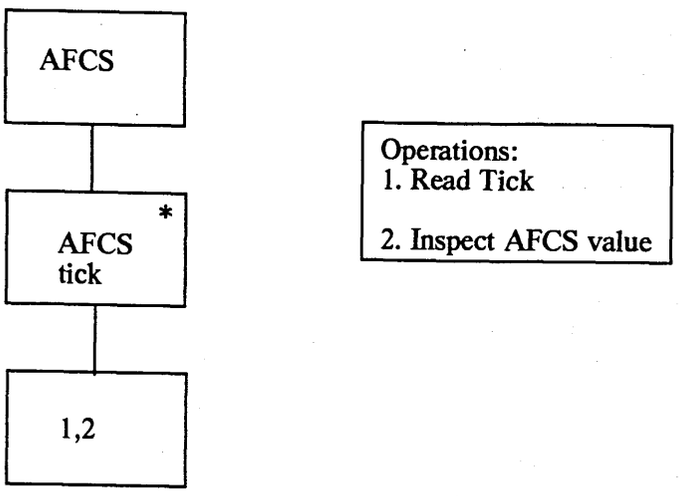

Figure 2.3 System Test Structure Diagram

| Action | Summary | Attributes |
|---|---|---|
| ARM | The pilot requests that the system be armed. | |
| ARMED | The actuator positions and the control law demands are in harmony | |
| ARM_DEFAULT_MODE | The initial arming of a default control mode. | ID: MODE_ID_TYPE |
| CANCEL_SYSTEM _TEST | A request to cancel the system test. | |
| CAPTURE | This is the signal to mode to go from ARM to ARM_AND_IN_CAP | ID: MODE_ID_TYPE |
| COMPLETED_SYSTEM _TEST | All tests of the system test have been successfuly completed | |
| CONTINUE_SYSTEM _TEST | Indication that the current test of the system test has been successfully completed. | |
| DISENGAGE | The system has been disengaged. This may happen before engagement (1) by the pilot pressing the disengage button or (2) by the system failing to get into the ARMED or ENGAGED state. It may happen whilst ENGAGED on receipt of a signal from an actuator relaying the fact that it has become disengaged | |
| DOWN_DISTURBANCE _REQUEST | The pilot wishes to be offered the previous valid disturbance, that is the first disturbance with a lower index number (ID) This is equivalent to the pilot presssing the DOWN button | |
| ENGAGE | The pilot requests (successfully) that the system be engaged. | |
| FAIL_TEST_STAGE | The current 'automatic' stage of the system test has not been successfully completed. | |
| INCEPTOR_VALUE | A new value representing the current position of an inceptor arrives | |

Figure 2.4 Typical List of Actions

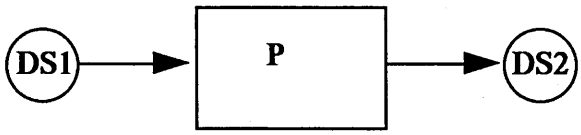



(a) Schematic Diagram

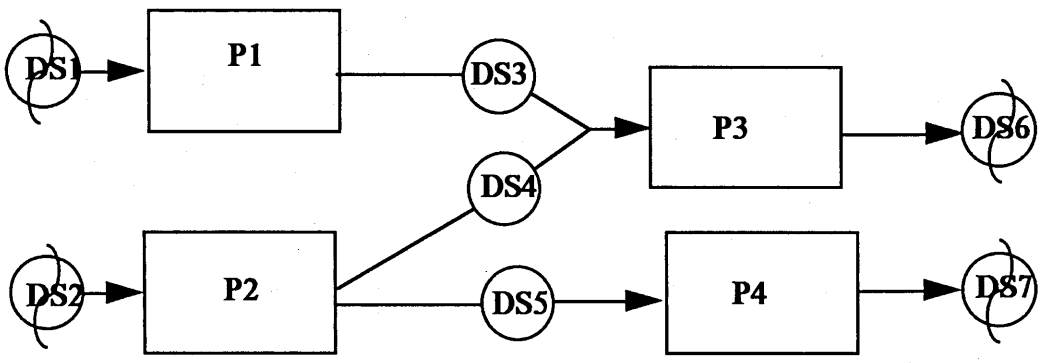



(b) Structure Diagram for AFCS Model

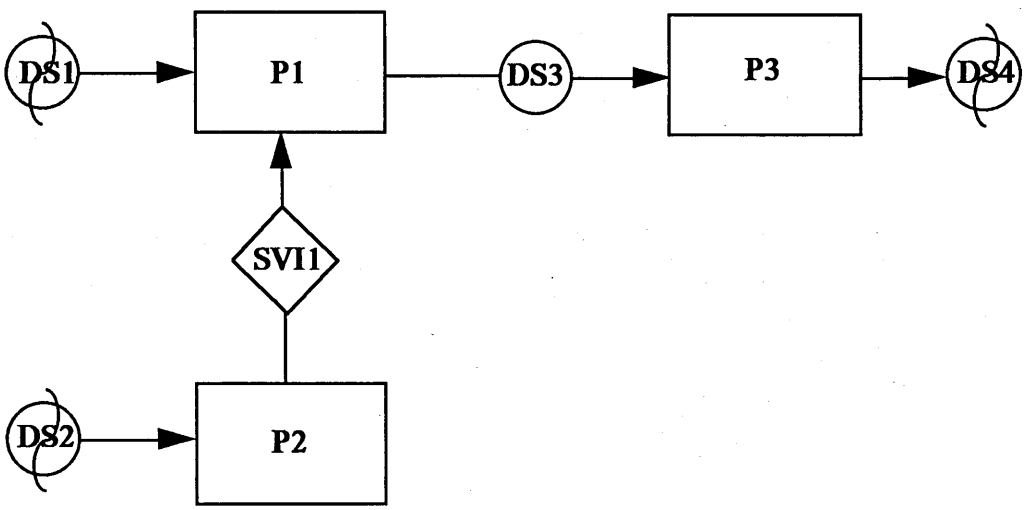
Figure 2.5 Control Input Device




(a) Elementary, single-process network




(b) Multi-process network




(c) Network with State Vector Inspection

Figure 2.6 System Network Diagrams (SND)

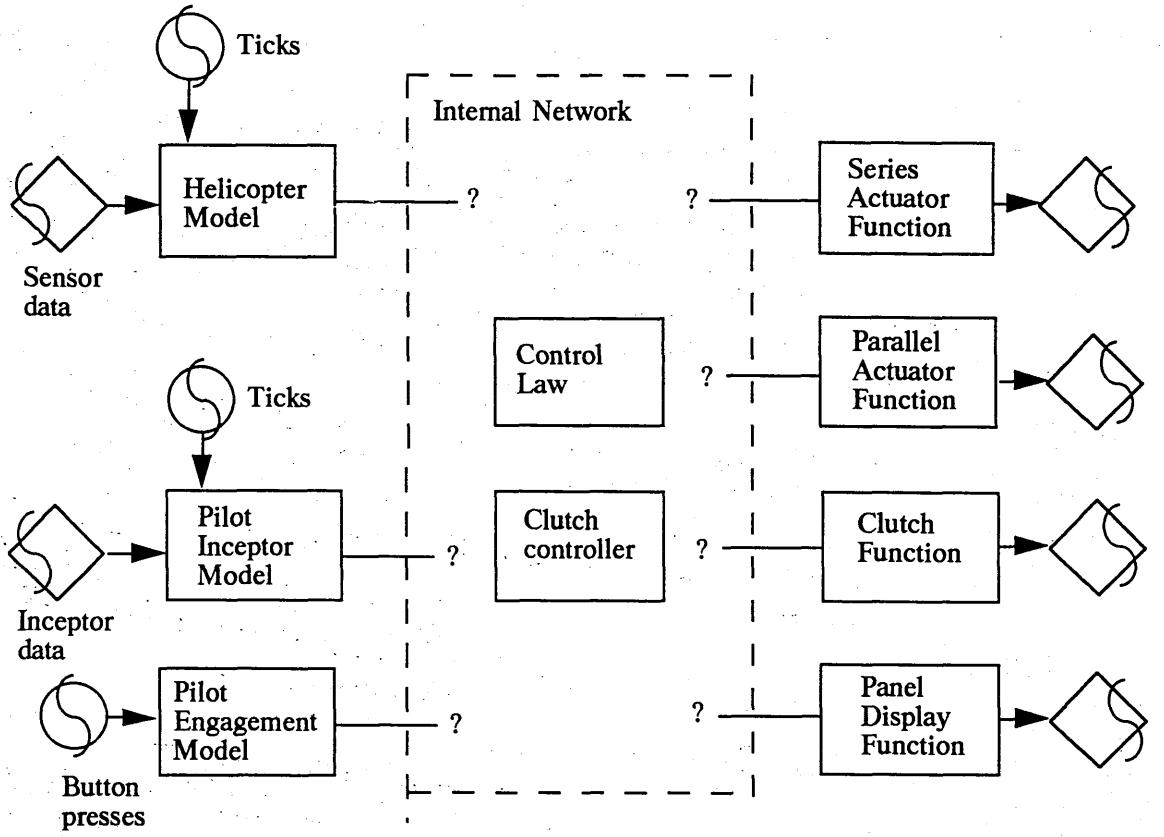



Figure 2.7 Example of Network Problem

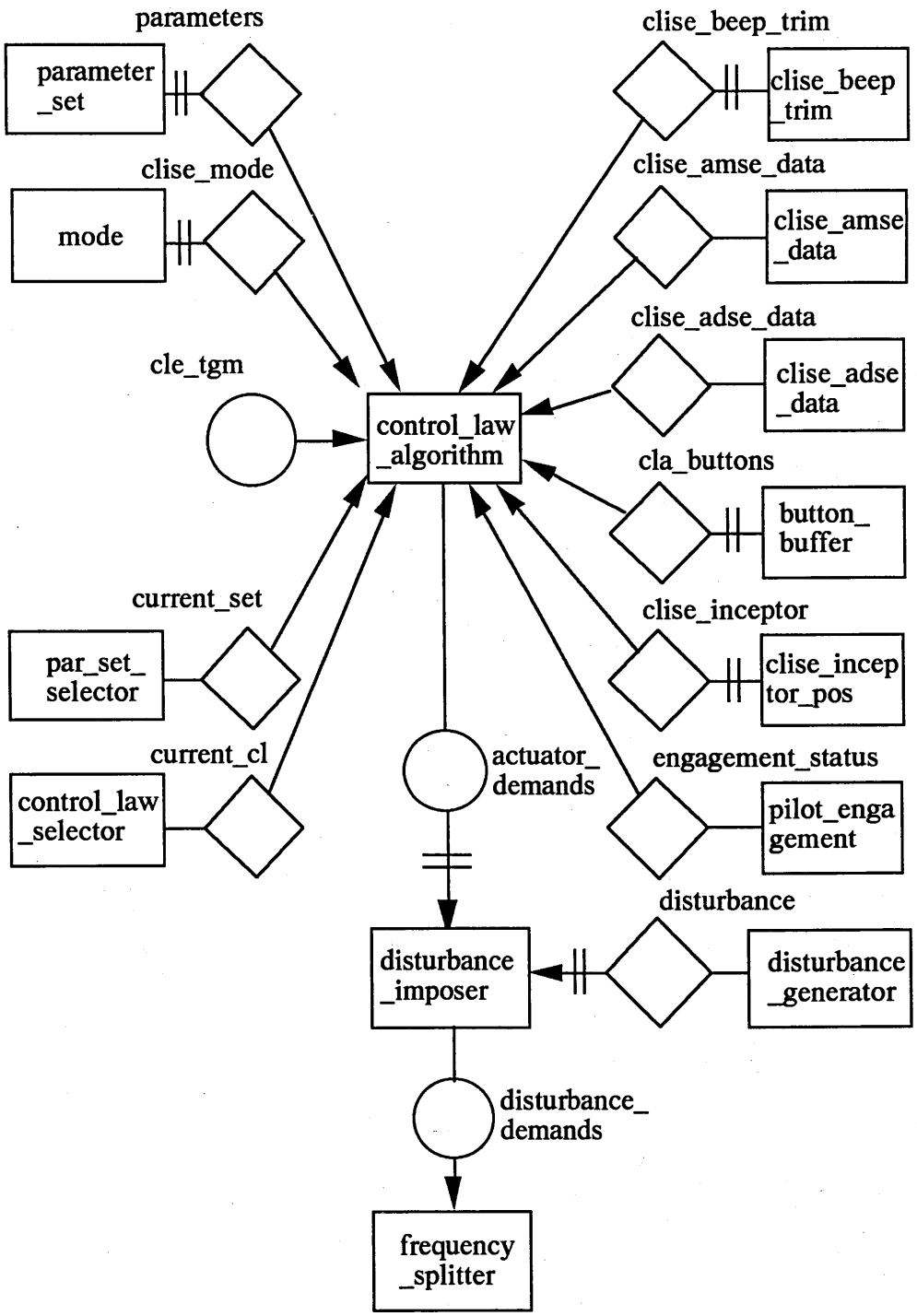
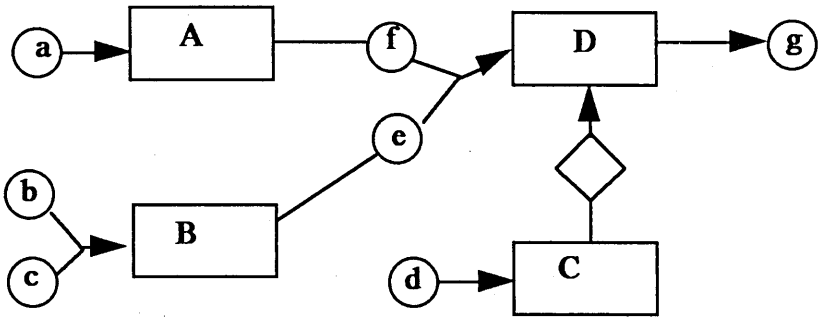
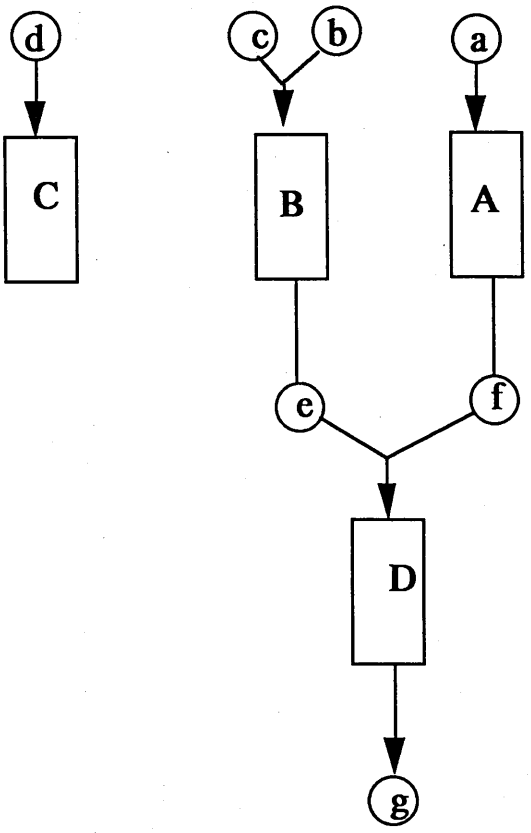



Figure 2.8 The ACT System Control Law Network

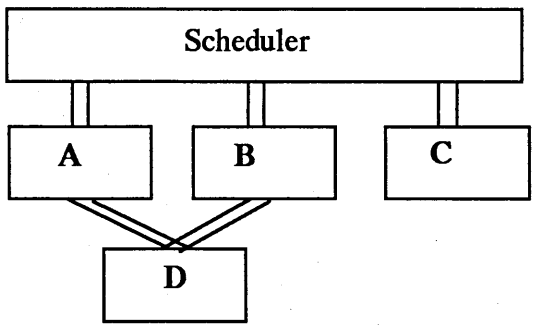



(a) The Example System Network

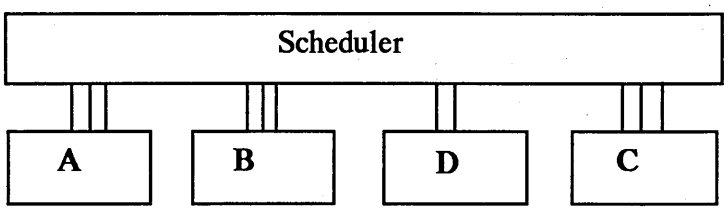



(b) Prepared Network

Figure 2.9 Implementation Schemes

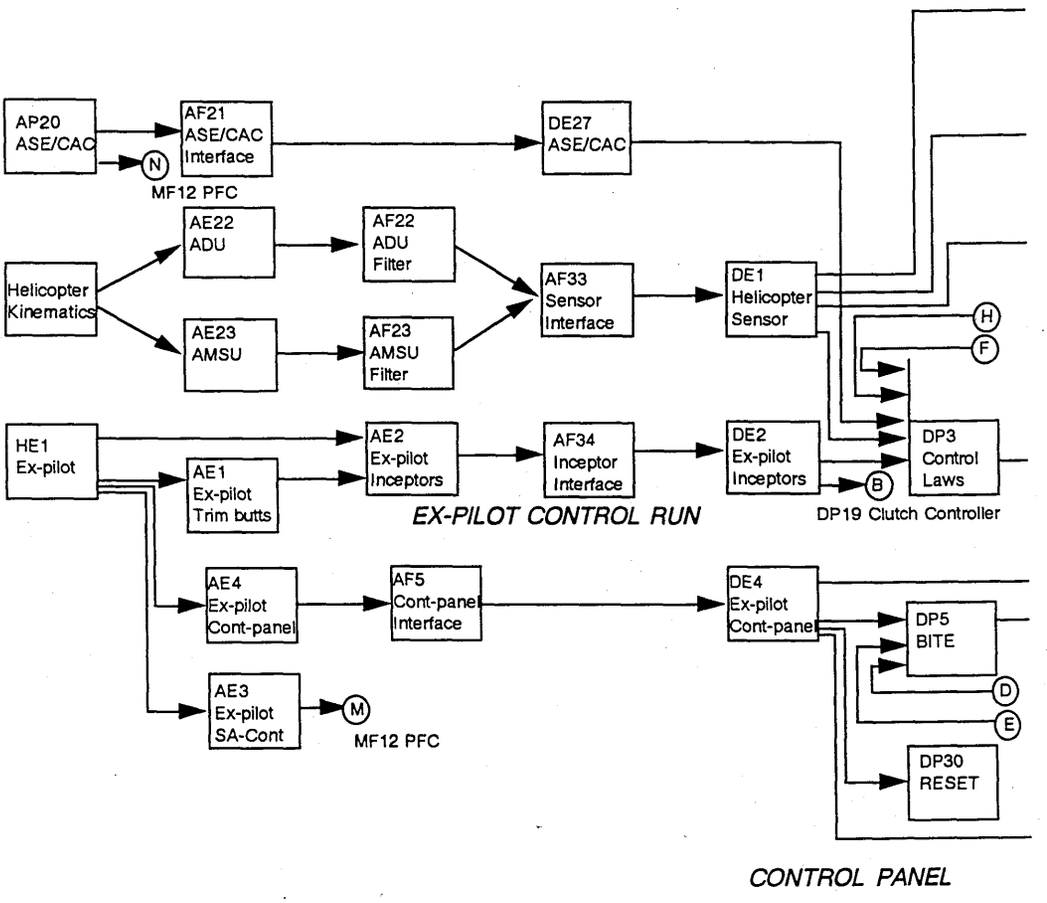



(a) System Implementation Diagram. I




(b) System Implementation Diagram. II

Figure 2.10 Implementation Alternatives



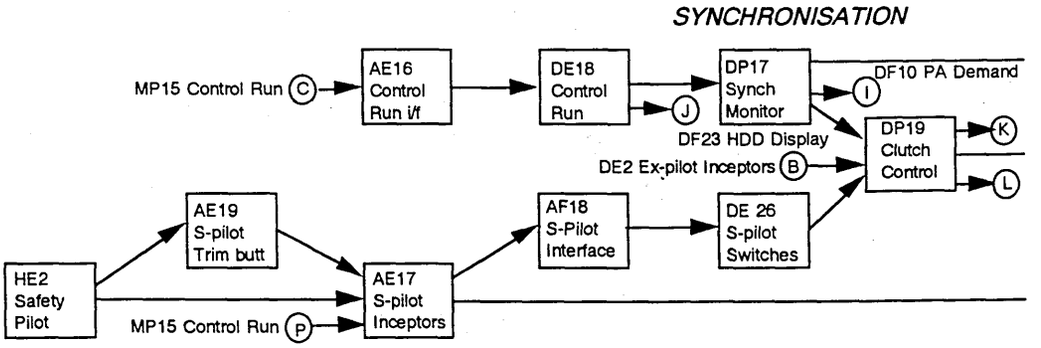




Figure 3.1 Initial System Network Diagram I

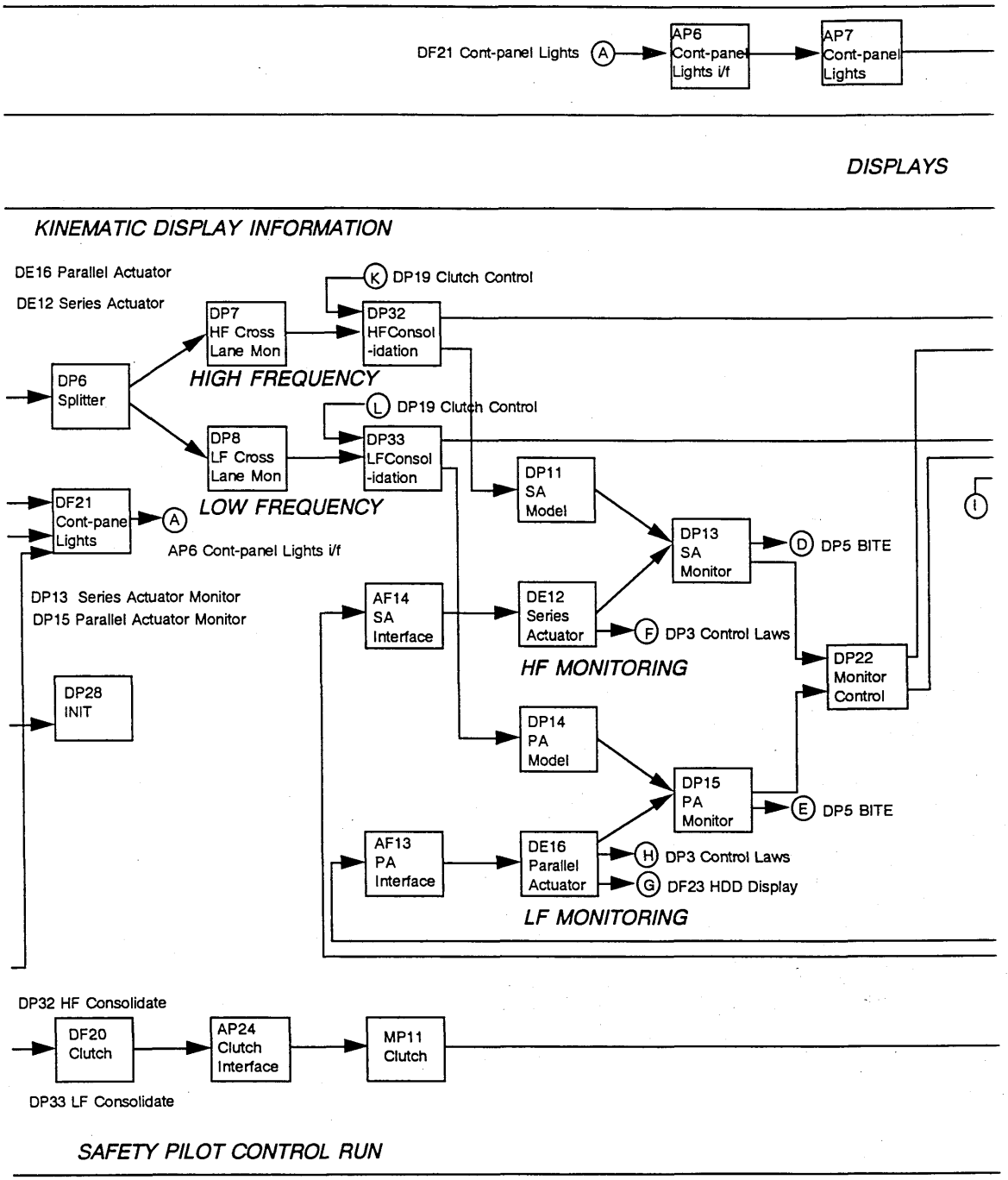



Figure 3.2 Initial System Network Diagram II

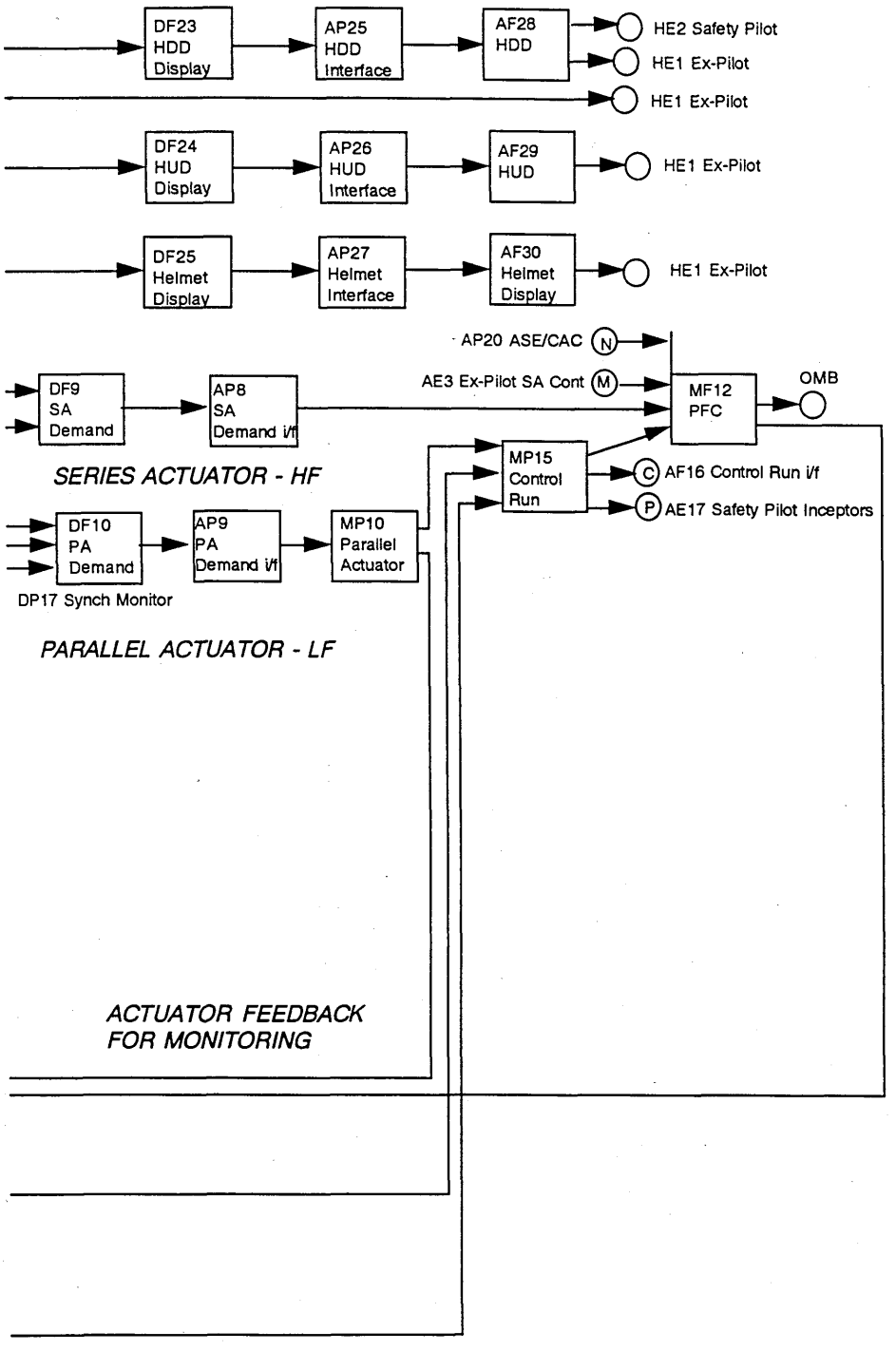



Figure 3.3 Initial System Network Diagram III

| IDENTIFIER | NAME |
|---|---|
| | |
| AE1 | Analogue entity: Experimental pilot trim buttons |
| AE2 | Analogue entity: Experimental pilot inceptors. |
| AE3 | Analogue entity: Experimental pilot series actuator controls. |
| AE4 | Analogue entity: Experimental pilot control panel. |
| AE16 | Analogue entity: Control run interface. |
| AE17 | Analogue entity: Safety pilot inceptors. |
| AE19 | Analogue entity: Safety pilot trim buttons. |
| AE22 | Analogue entity: Air Data Unit. |
| AE23 | Analogue entity: Aircraft Motion Sensing Unit |
| AF5 | Analogue function: Control panel interface. |
| AF7 | Analogue function: Control panel lights. |
| AF13 | Analogue function: Parallel actuator interface. |
| AF14 | Analogue function: Series actuator interface. |
| AF18 | Analogue function: Safety pilot interface. |
| AF21 | Analogue function: Autostabiliser Equipment/Computer Acceleration Control. |
| AF28 | Analogue function: Head down display. |
| AF29 | Analogue function: Head up display. |
| AF30 | Analogue function: Helmet mounted display. |
| AF33 | Analogue function: Sensor interface. |
| AF34 | Analogue function: Inceptor interface. |
| AP6 | Analogue process: Control panel lights inteface. |
| AP8 | Analogue process: Series actuator demand interface. |
| AP9 | Analogue process: Parallel actuator demand interface. |
| AP20 | Analogue process: Autostabiliser Equipment/Computer Acceleration Control Interface. |
| AP22 | Analogue process: Air Data Unit filter. |
| AP23 | Analogue process: Aircraft Motion Sensing Unit filter. |
| AP24 | Analogue process: Clutch interface. |
| AP25 | Analogue process: Head down display interface. |
| AP26 | Analogue process: Head up display interface. |
| AP27 | Analogue process: Helmet mounted display interface. |

Figure 3.4 Network Glossary I

| IDENTIFIER | NAME |
|---|---|
| | |
| DE1 | Digital entity: Helicopter sensor. |
| DE2 | Digital entity: Experimental pilot inceptors. |
| DE4 | Digital entity: Experimental pilot control panel. |
| DE12 | Digital entity: Series actuator. |
| DE14 | Digital entity: Parallel actuator. |
| DE18 | Digital entity: Control run. |
| DE26 | Digital entity: Safety pilot switches. |
| DE27 | Digital entity: Autostabiliser Equipment/Computer Acceleration Control. |
| DF9 | Digital function: Series actuator demand. |
| DF10 | Digital function: Parallel actuator demand. |
| DF20 | Digital function: Clutch. |
| DF21 | Digital function: Control panel lights. |
| DF23 | Digital function: Head down display. |
| DF24 | Digital function: Head up display. |
| DF25 | Digital function: Helmet mounted display. |
| DP3 | Digital process: Control laws. |
| DP5 | Digital process: Built in test equipment. |
| DP6 | Digital process: Frequency splitter. |
| DP7 | Digital process: High frequency cross lane monitoring. |
| DP8 | Digital process: Low frequency cross lane monitoring. |
| DP11 | Digital process: Series actuator model. |
| DP13 | Digital process: Series actuator monitor |
| DP14 | Digital process: Parallel actuator model. |
| DP15 | Digital process: Parallel actuator monitor. |
| DP17 | Digital process: Sychronisation monitor. |
| DP19 | Digital process: Clutch control. |
| DP22 | Digital process: Monitor control. |
| DP28 | Digital process: Initialisation. |
| DP30 | Digital process: Reset. |
| DP32 | Digital process: High frequency consolidation. |
| DP33 | Digital process: Low frequency consolidation. |
| HE1 | Human entity: Experimental pilot. |
| HE2 | Human entity: Safety pilot. |
| MF12 | Mechanical function: Primary flight control unit. |
| MP10 | Mechanical process: Parallel Actuator. |
| MP11 | Mechanical process: Clutch. |
| MP15 | Mechanical process: Control run. |
| OMB | Outside model boundary. |

Figure 3.5 Network Glossary II

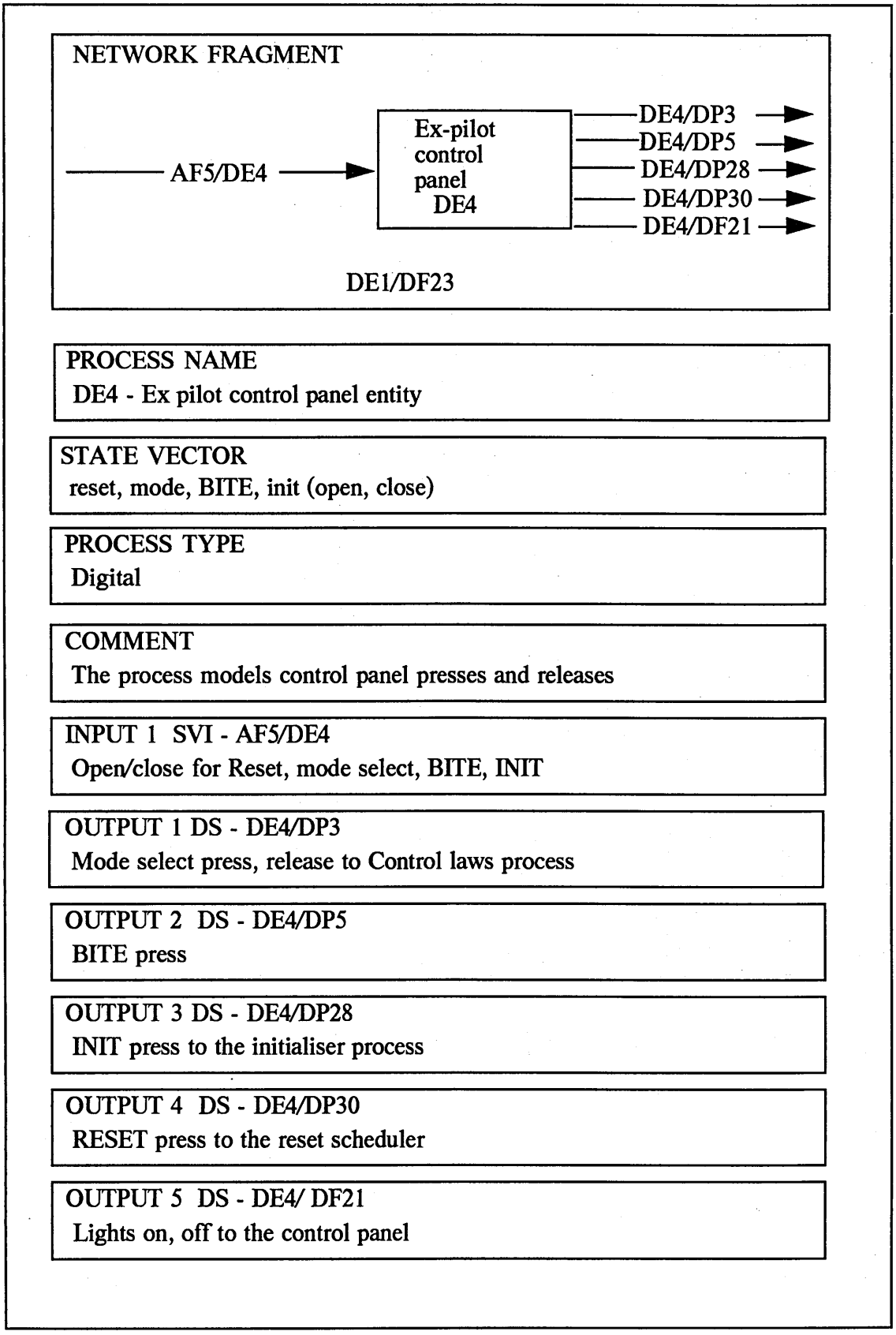



Figure 3.6 A record of the Version 2 Network Database

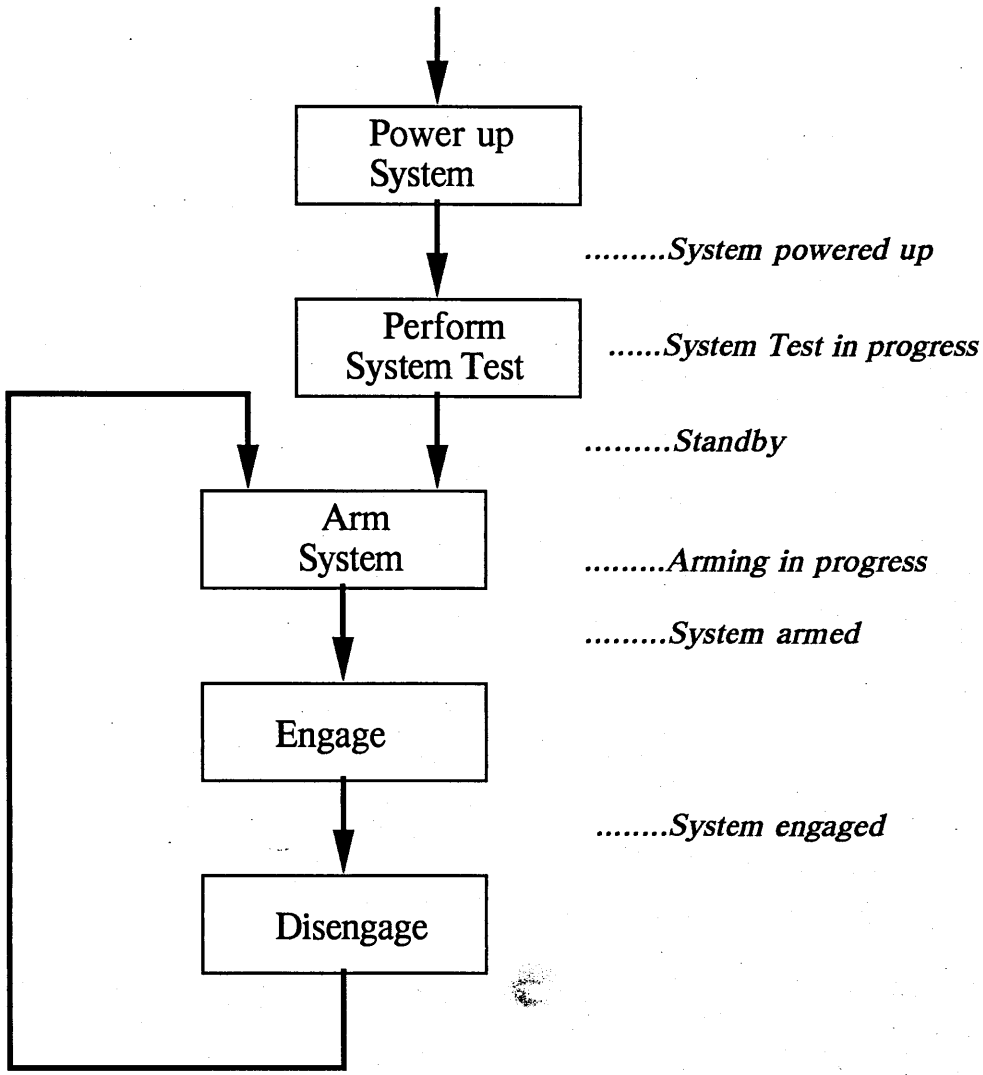



Figure 4.1. Possible System Control Flowchart

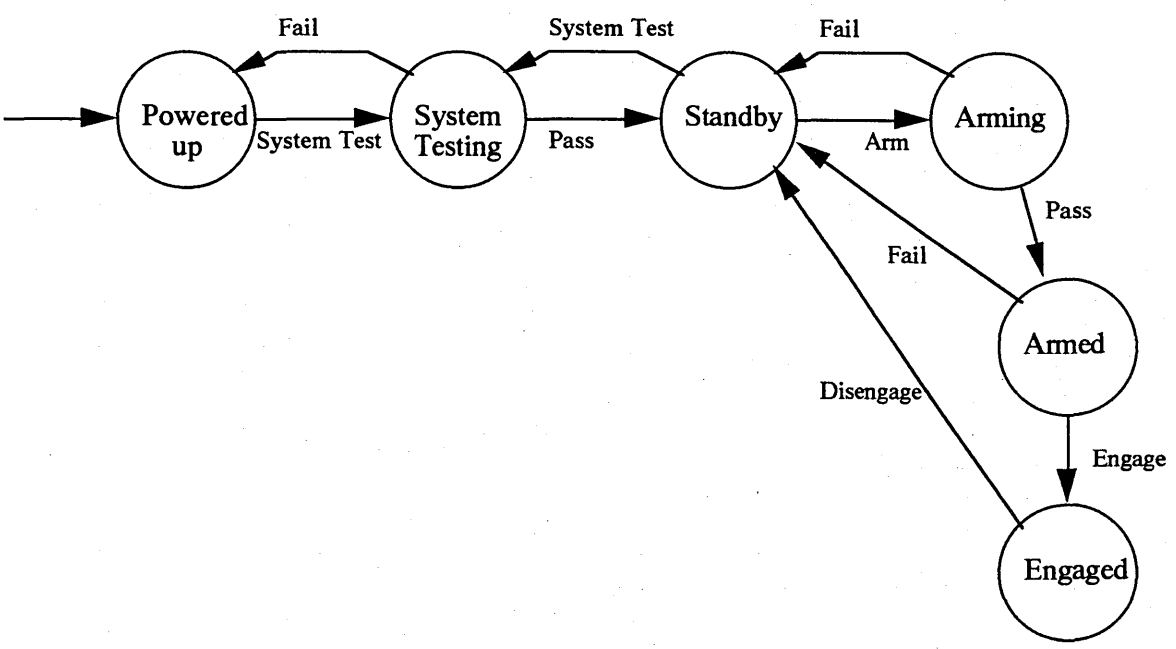



Figure 4.2. Possible FSM for System Control

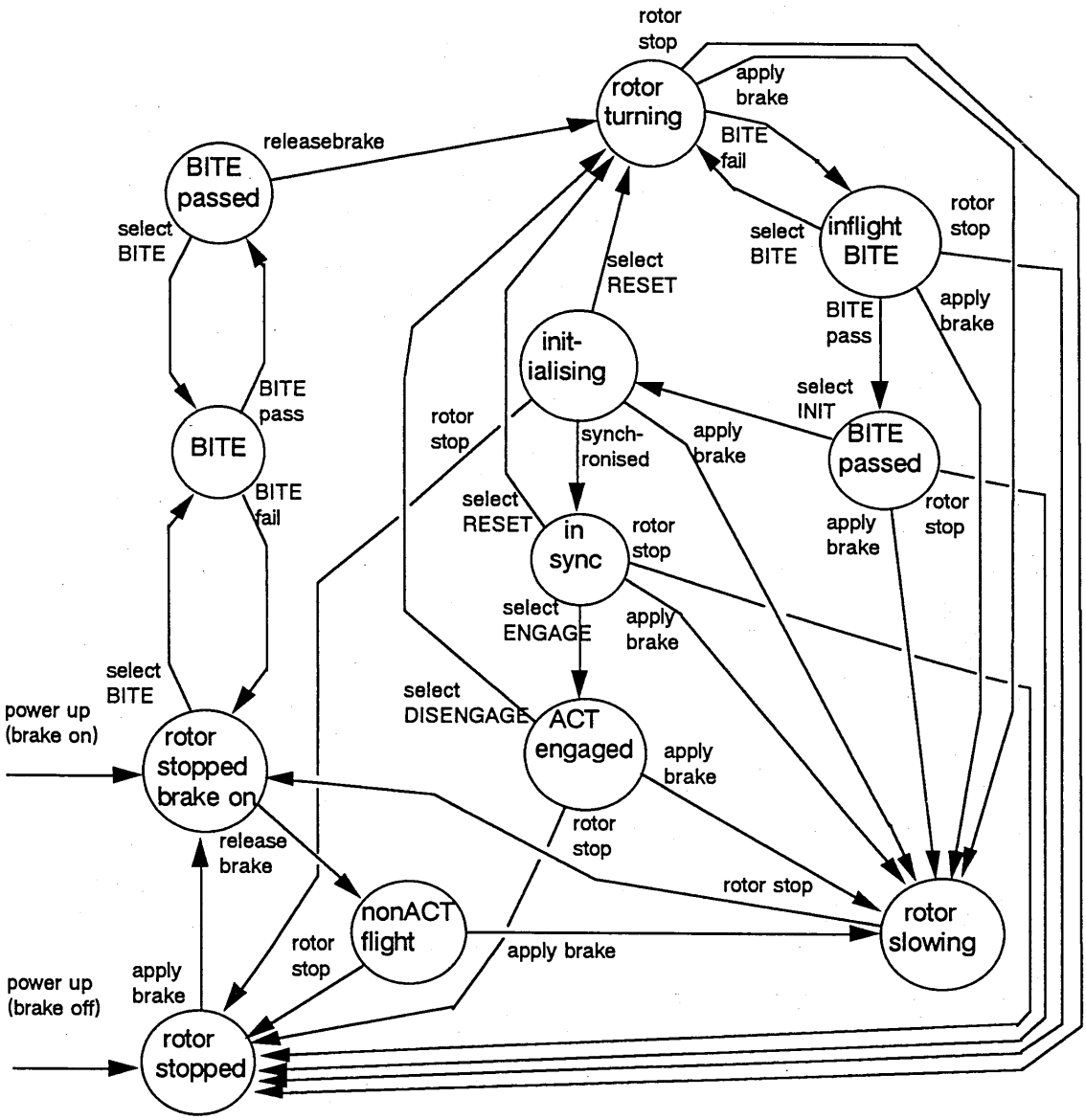



Figure 4.3 State Transition Diagram for ACT Modes

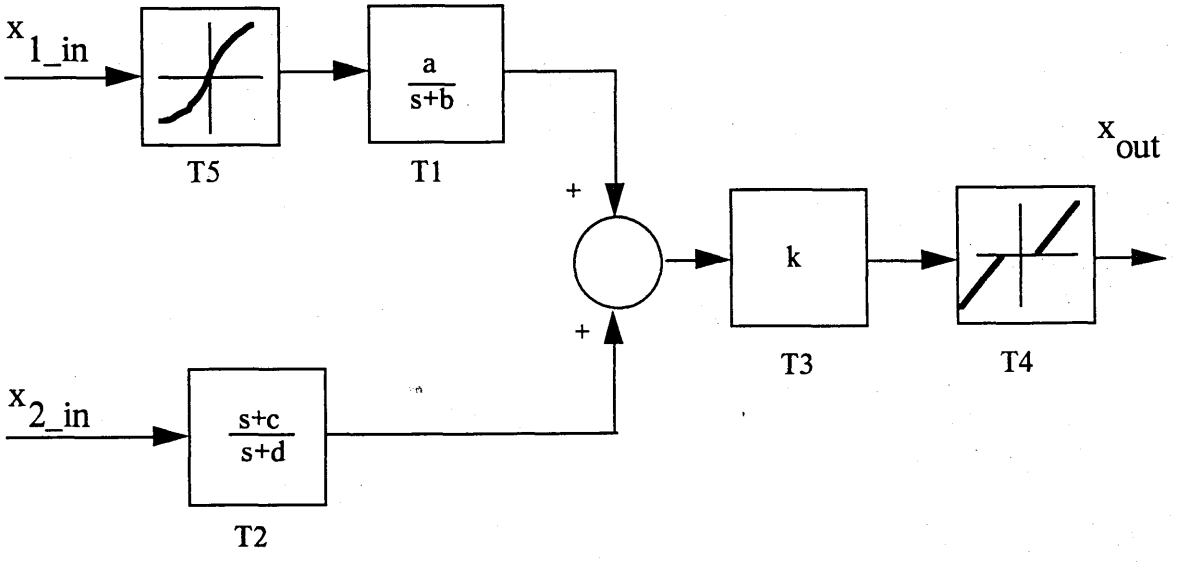



Figure 5.1 Transfer Function Block Diagram

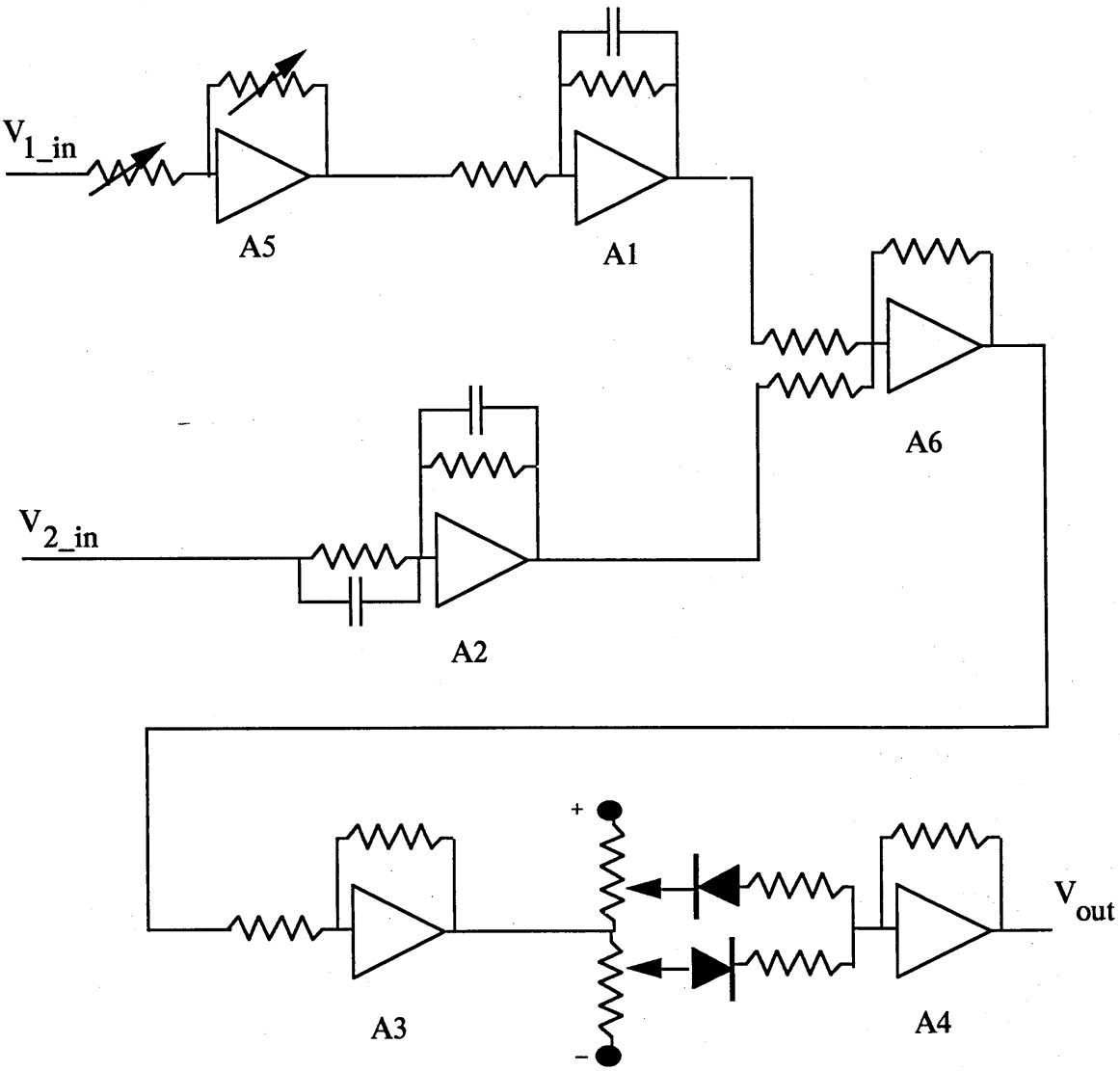



Figure 5.2 Analogue Component Diagram

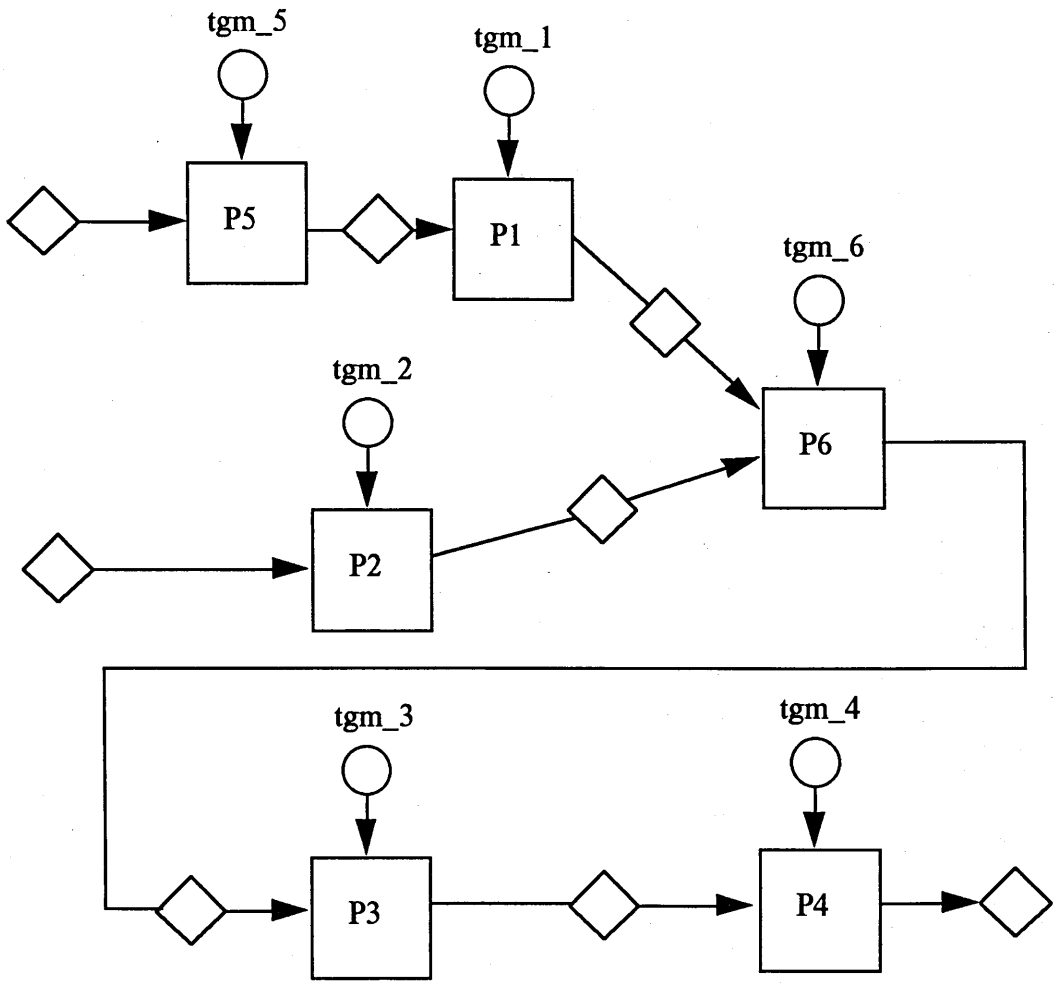
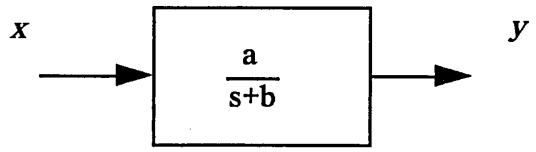
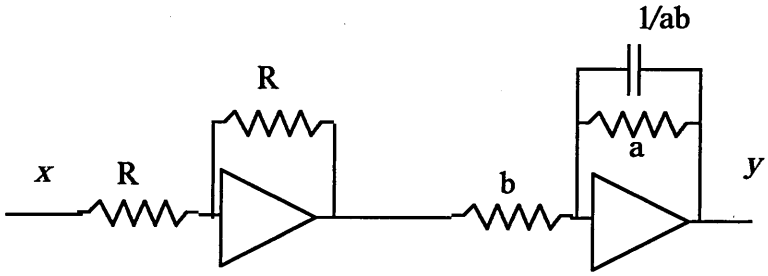


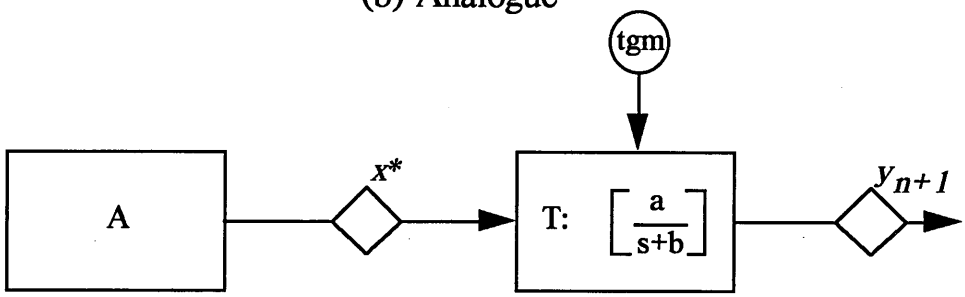

Figure 5.3 Component Simulation Network




(a) Transfer function




(b) Analogue




(c) JSD

Figure 5.4 Basic Component Simulation

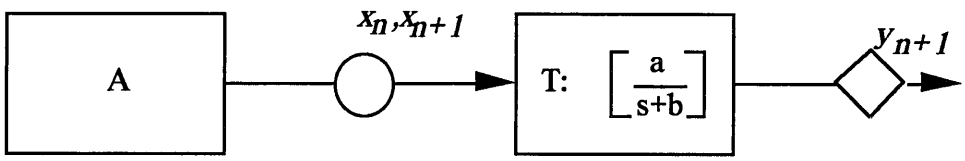



Figure 5.5 Tightly coupled component simulation

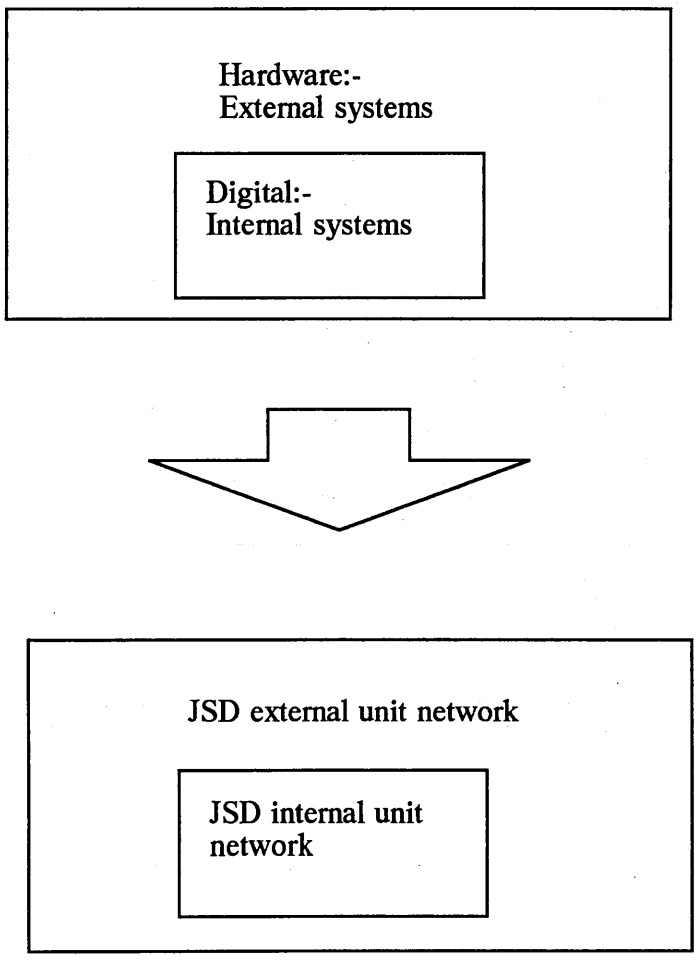



Figure 5.6 Total System Representation

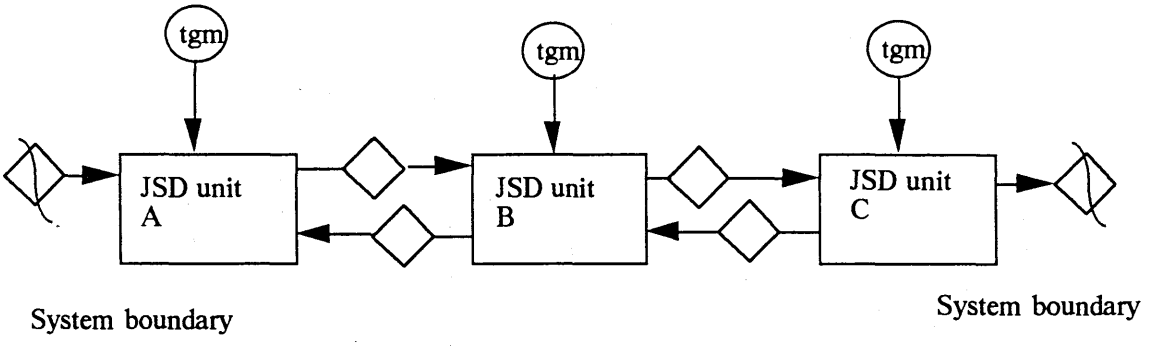



Figure 5.7 Unit network

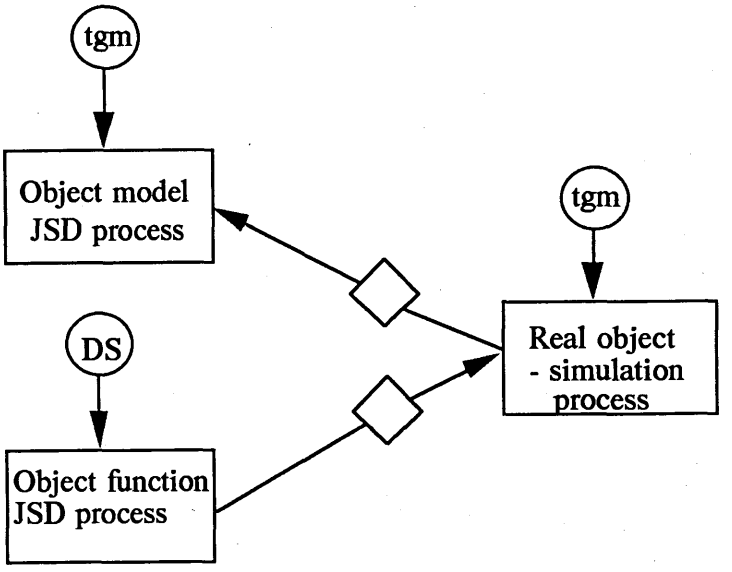



Figure 5.8 Configuration at Internal/External Boundary

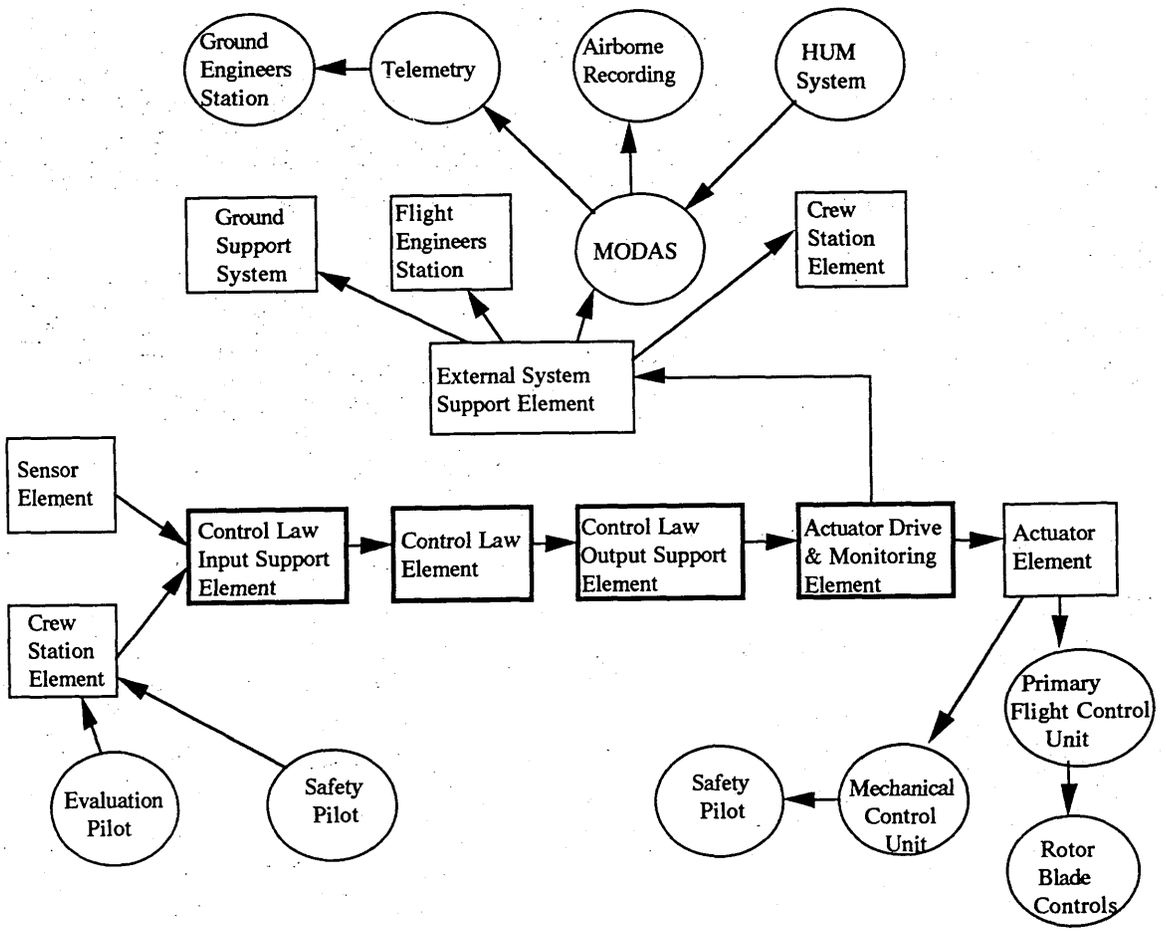



Figure 6.1: ACT System Logical Elements

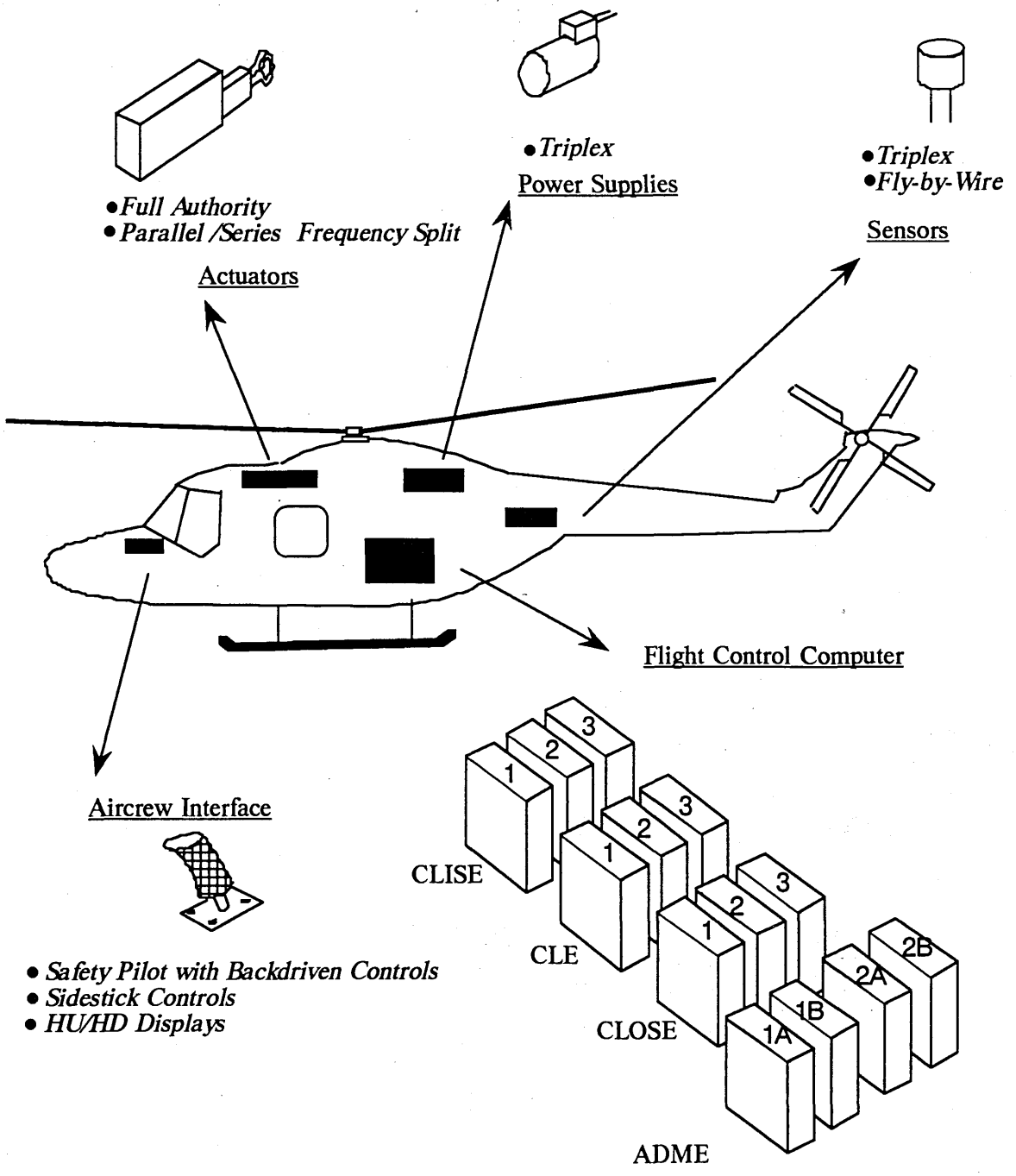



Figure 6.2 Revised ACT Lynx System Architecture

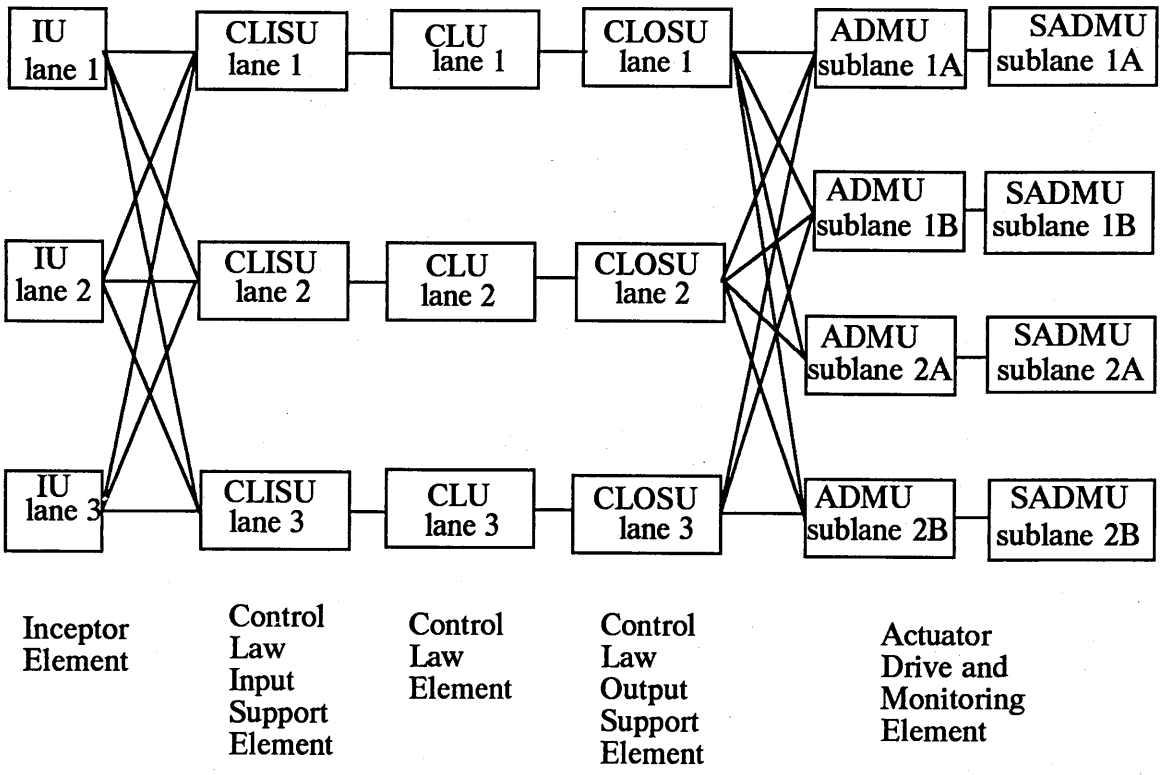



Figure 6.3 Connections Between Units

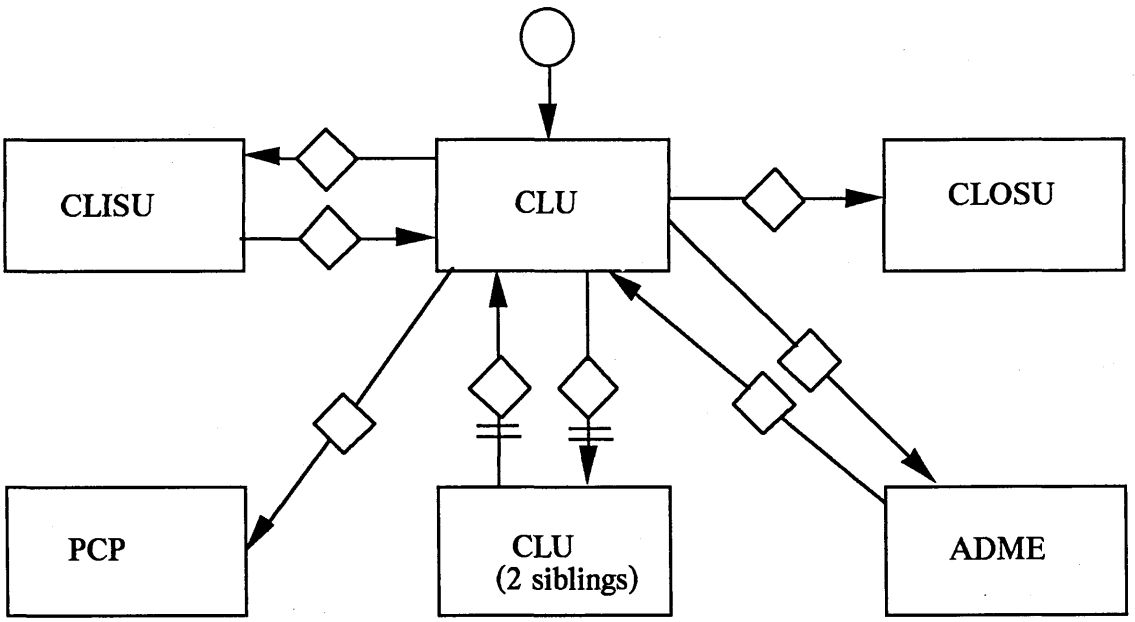
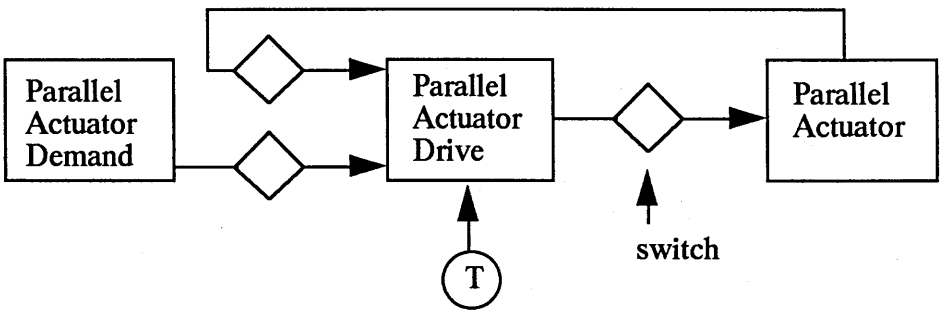
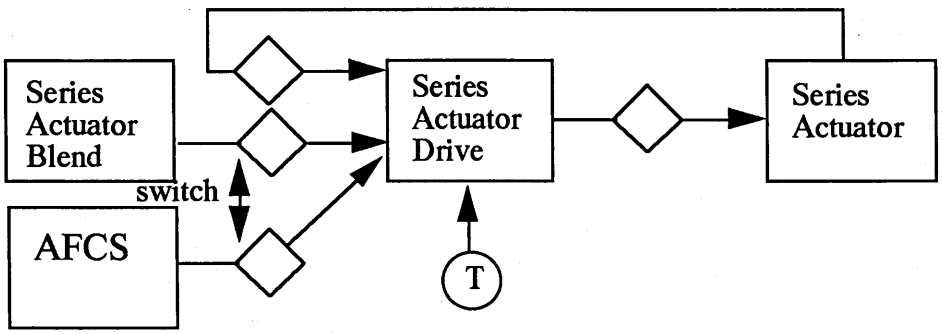


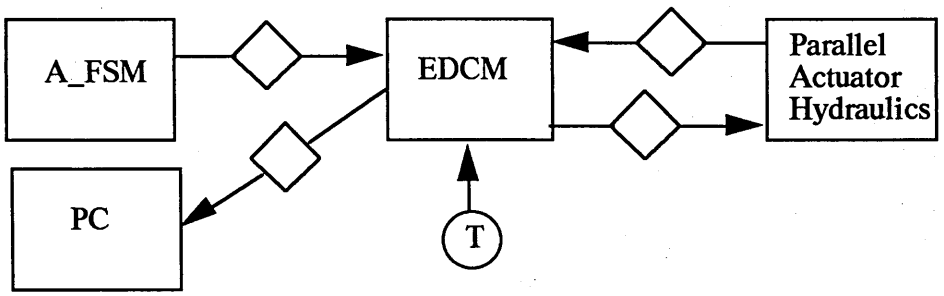

Figure 6.4 Connections to a CLU




(a) Parallel Actuator Drive

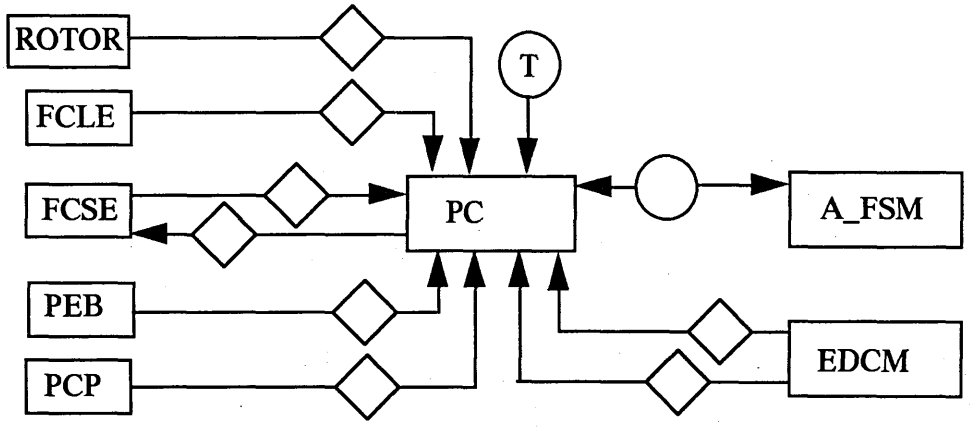



(b) Series Actuator Drive




(c) Engage/Disengage Control & Monitoring

Figure 6.5 Network Fragments



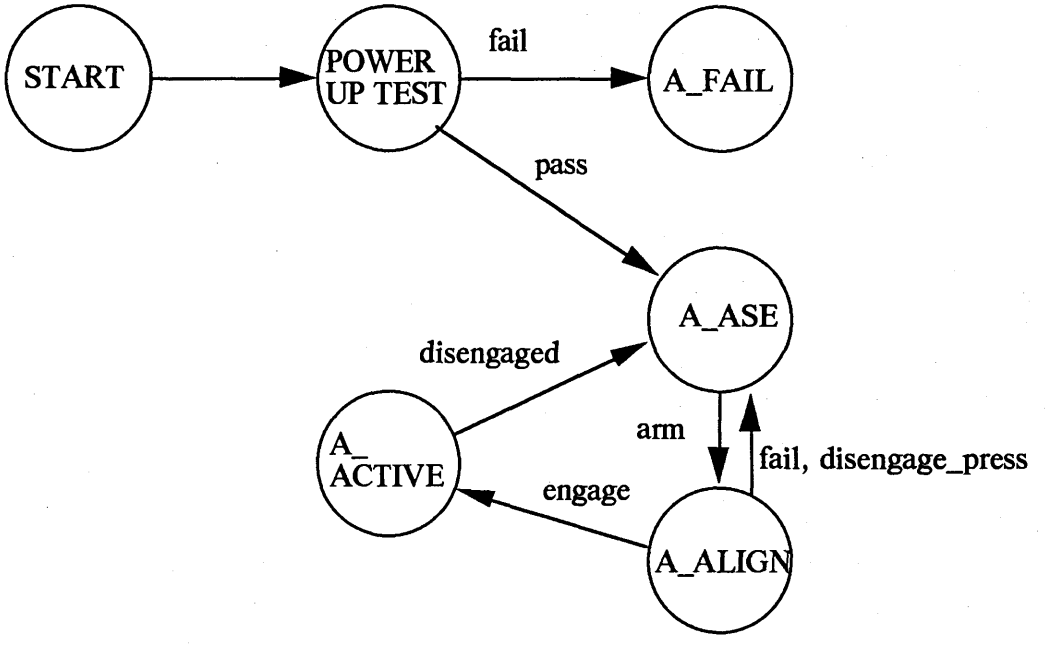




Figure 6.6 Pilot Control and ADMU State Control

| | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 6 | 7 | 7 | 7 | 8 | 8 | 8 |
| lane | A | B | C | A | B | C | A | B | C | A | B | C | A | B | C | A | B | C | A | B | C | A | B | C |
| sample | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| detect | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| consolidate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| vote | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

(a) Duration between 2 and 3 frames

| | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 6 | 7 | 7 | 7 | 8 | 8 | 8 |
| lane | A | B | C | A | B | C | A | B | C | A | B | C | A | B | C | A | B | C | A | B | C | A | B | C |
| sample | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| detect | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| consolidate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vote | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(b) Duration between 1 and 2 frames

Figure 6.7 Consolidation and Voting

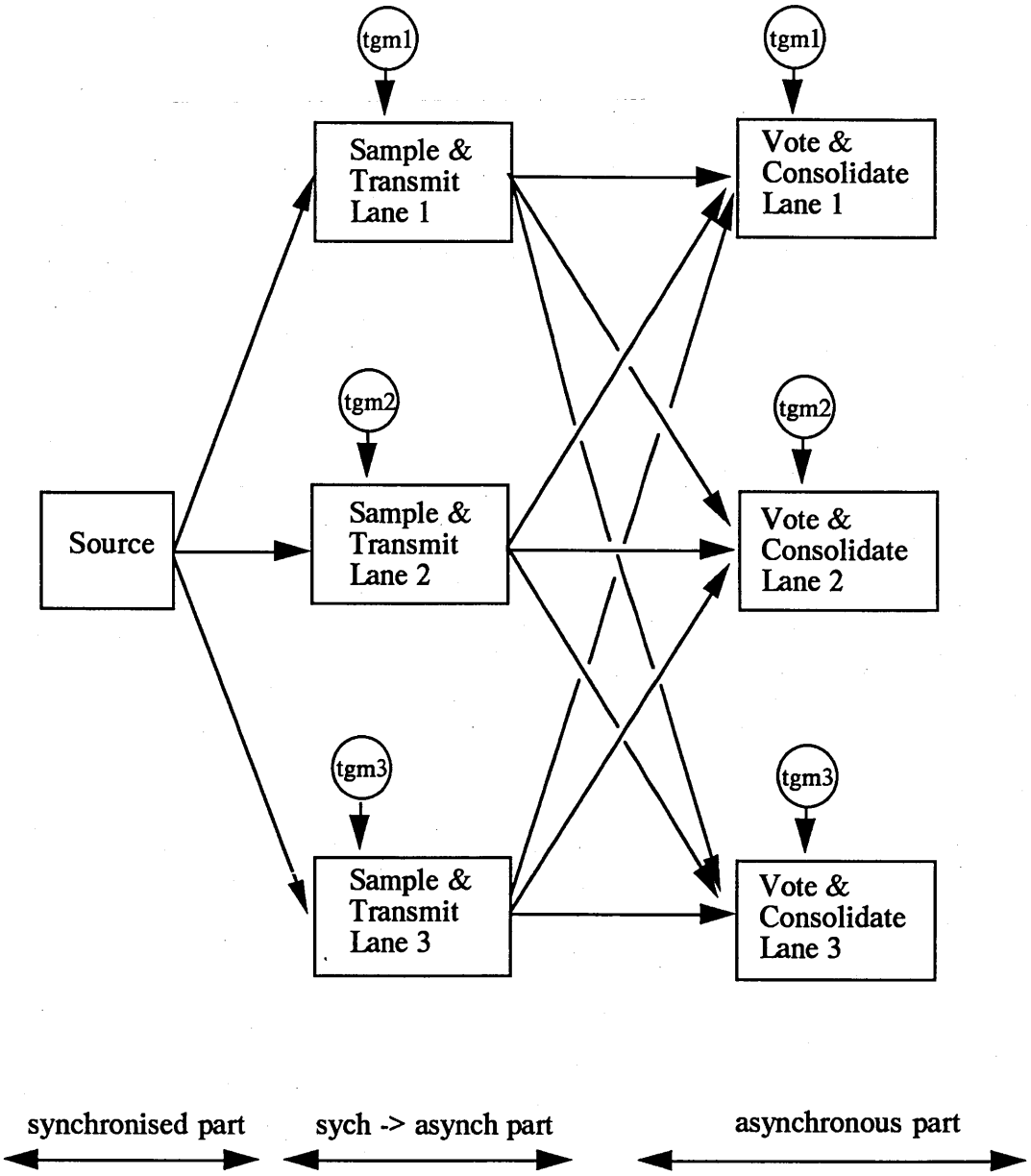
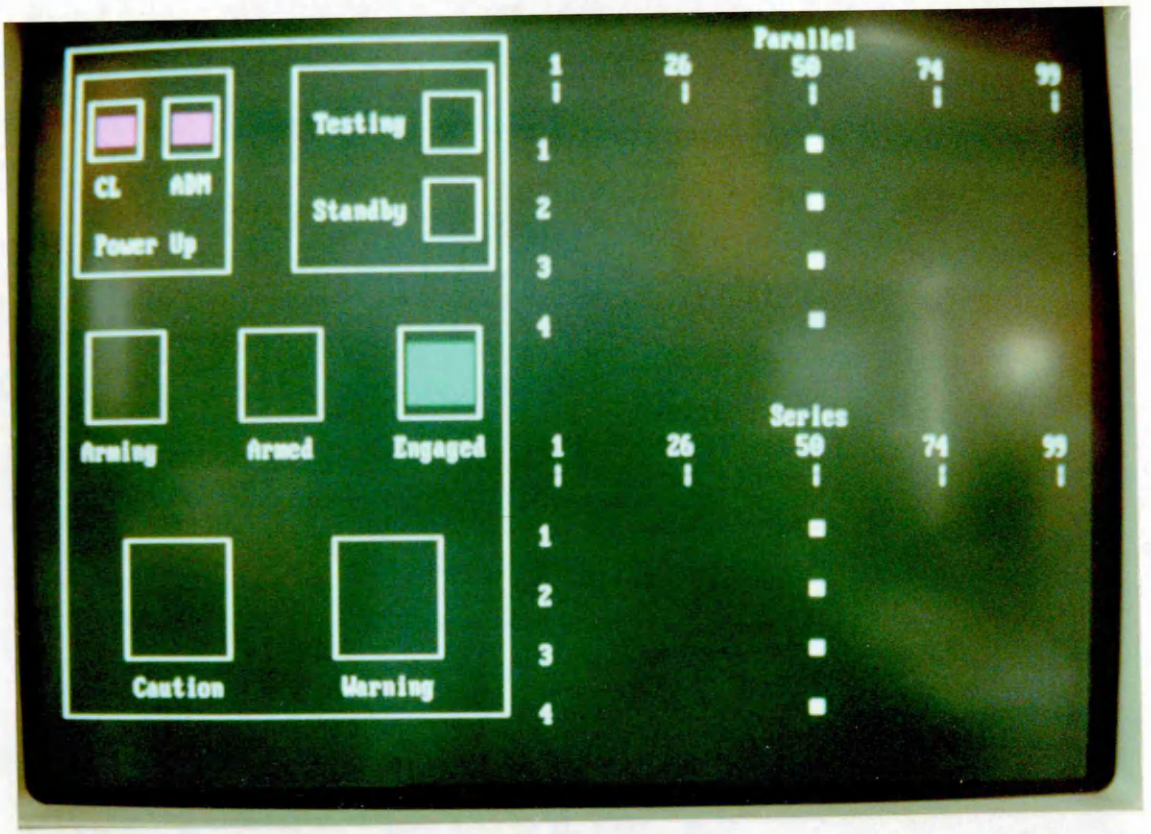
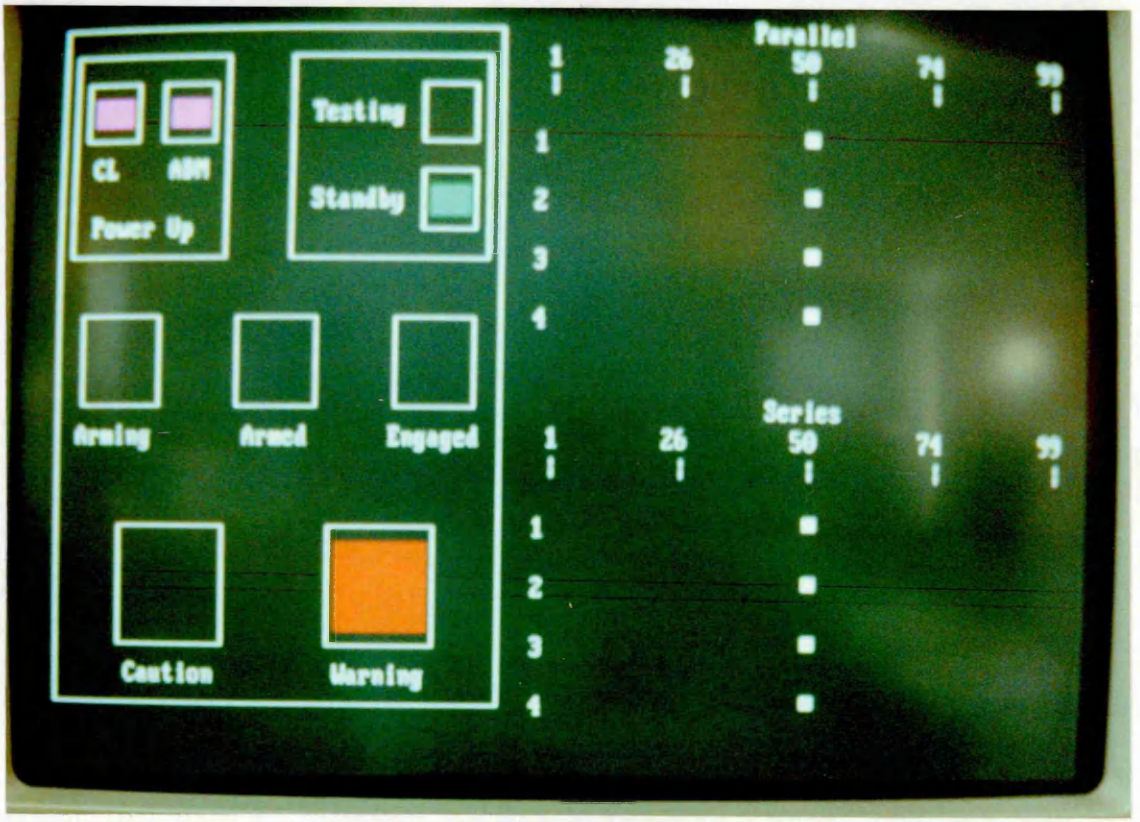



Figure 6.8 Consolidation Architecture




(a) Engaged




(b) Disengaged

Figure 7.1 Repeater Panel Simulation

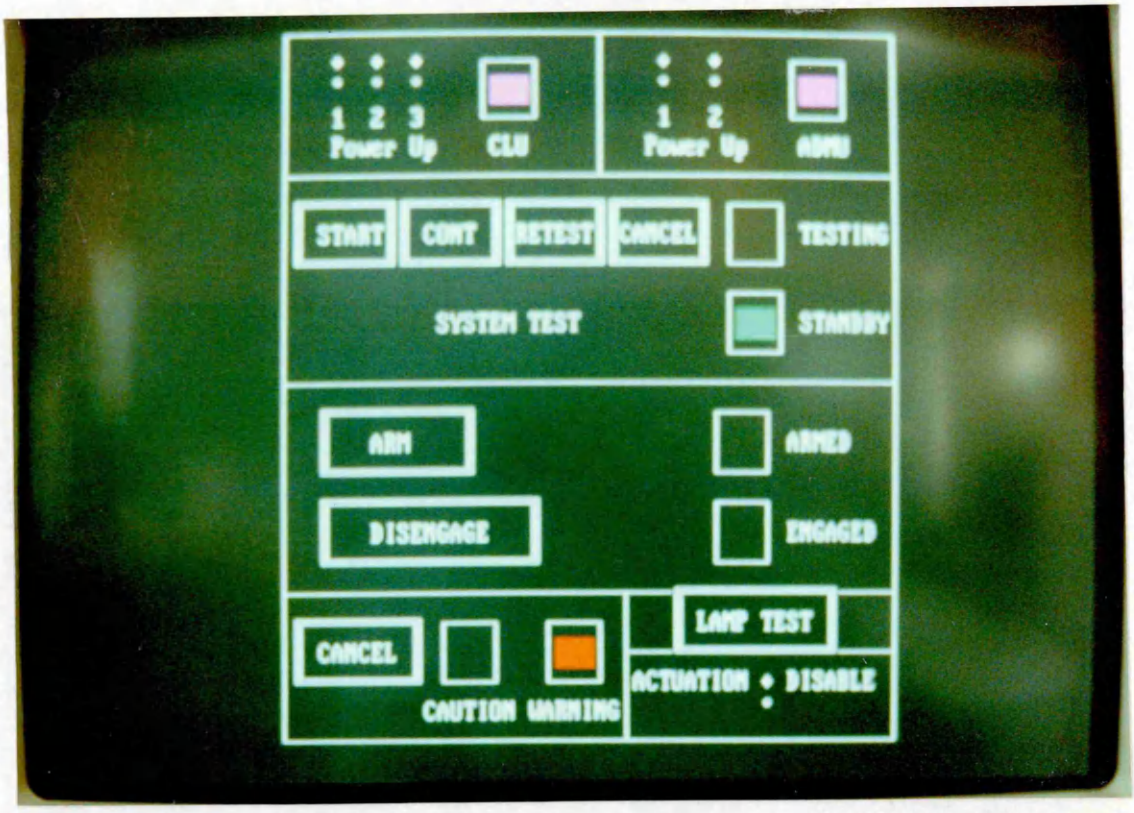



Figure 7.2 Pilots Control Panel Simulation


Figure 7.3 Mode Select Panel Simulation

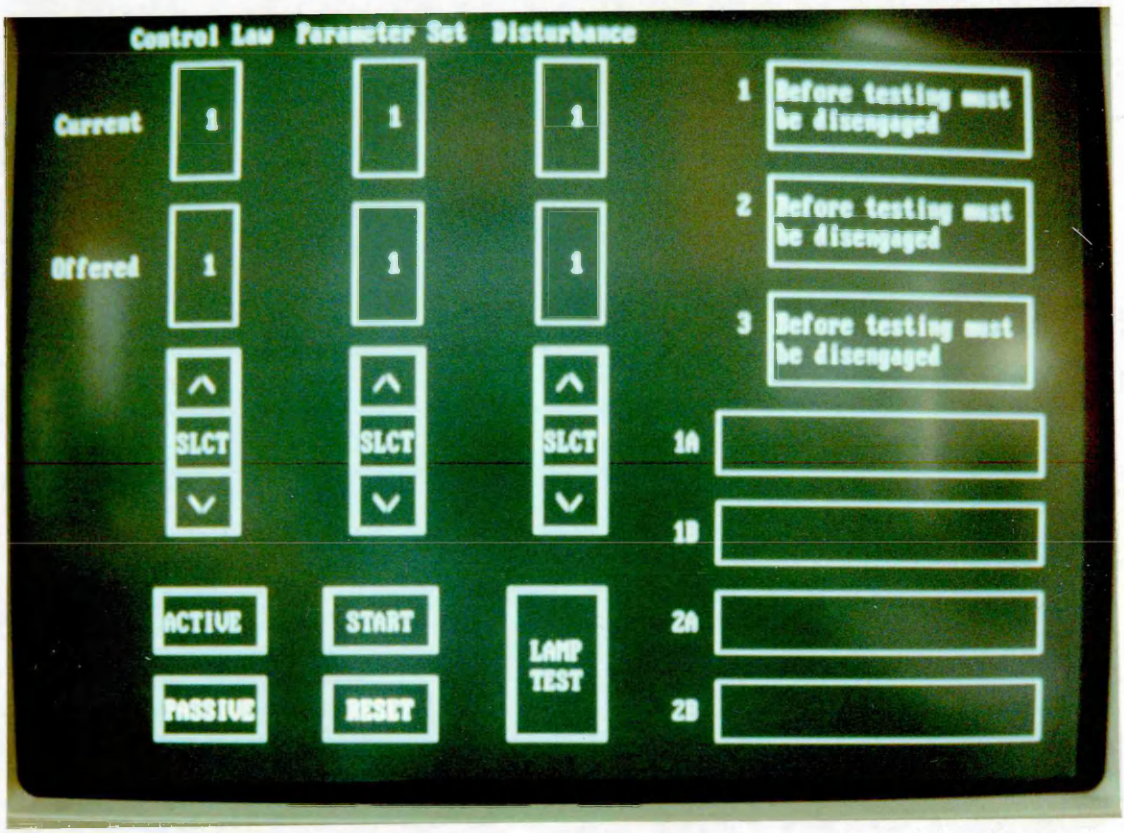



Figure 7.4 Menu Panel Simulation

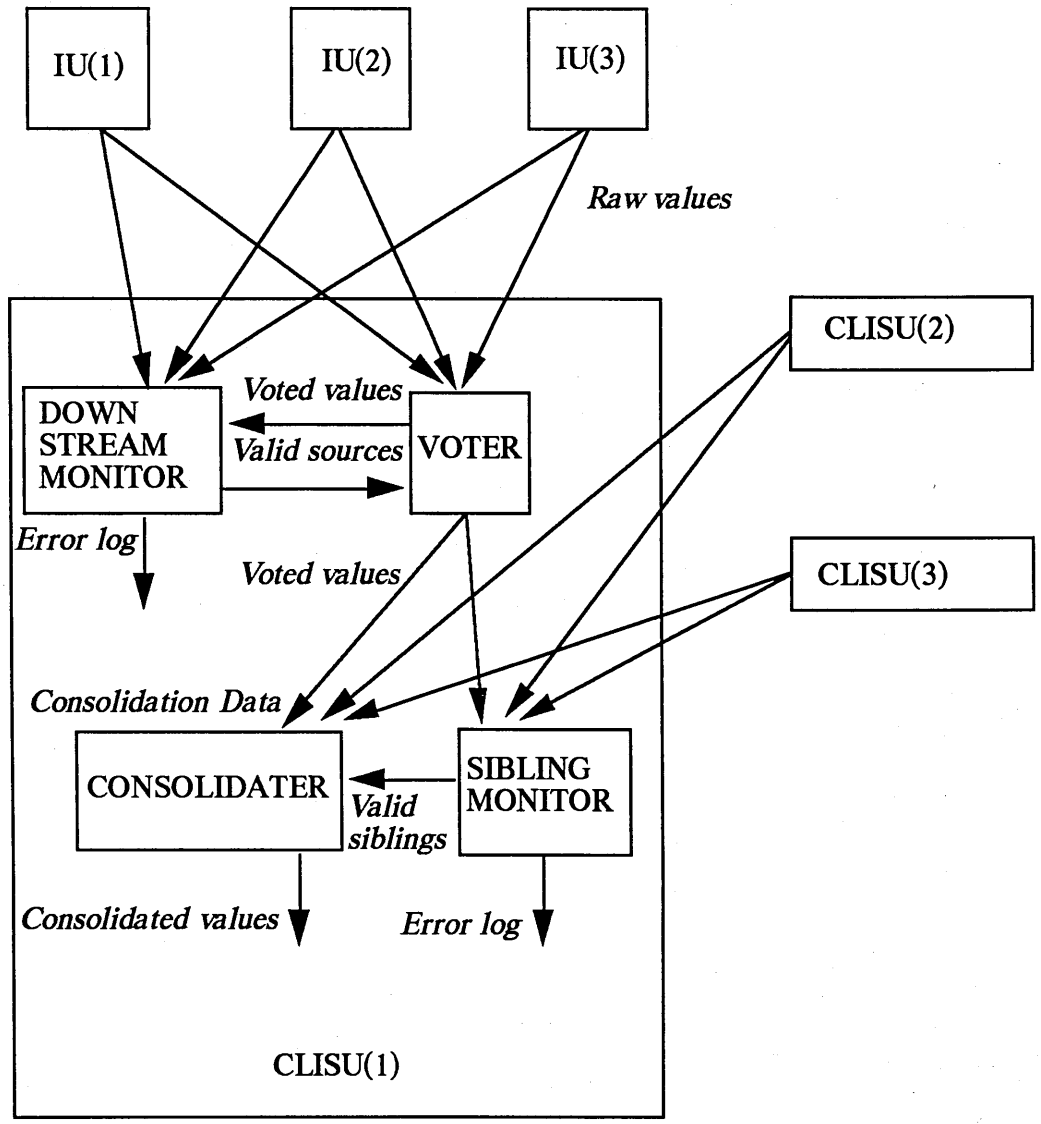



Figure 7.5 Schematic Diagram of Fault Processing

```
UNIT IE
    STD-INFO
        LONGNAME
        REFERENCE IE
        [*]CLASSIFICATION-SET
        [*]SUMMARY
        This unit is connected to the
        inceptors of the evaluation
        pilot.
        [o]NARRATIVE
            NO
MAIN-PART
    [o]TYPE
        ANALOGUE
    [o]BASE-REDUNDANCY
        SIMPLEX
    REPLICATION 3
    [o]UNIT-LVL-SYNCHRONISATION
        ASYNCHRONOUS
            FRAME-LAG
    [*]INTRA-UNIT-CONNECTIONS
UNIT-SID
```

```
UNIT CLE
    STD-INFO
        LONGNAME
        REFERENCE CLE
        [*]CLASSIFICATION-SET
        [*]SUMMARY
        This unit houses the control
        law algorithm and associated
        processing. It is the middle processor
        in a three processor "lane".
        [o]NARRATIVE
            NO
    MAIN-PART
        [o]TYPE
            DIGITAL
        [o]BASE-REDUNDANCY
            SIMPLEX
        REPLICATION 3
        [o]UNIT-LVL-SYNCHRONISATION
            ASYNCHRONOUS
                FRAME-LAG  10
        [*]INTRA-UNIT-CONNECTIONS
        UNIT-SID
```

Figure 7.6 Unit Descriptions

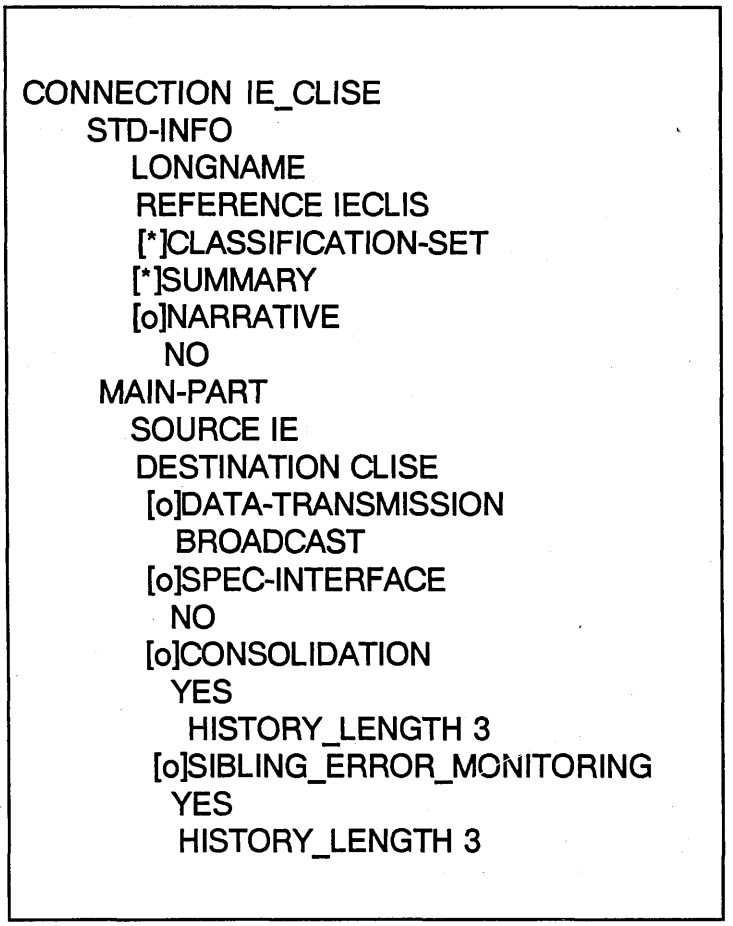



Figure 7.7 Connection Description

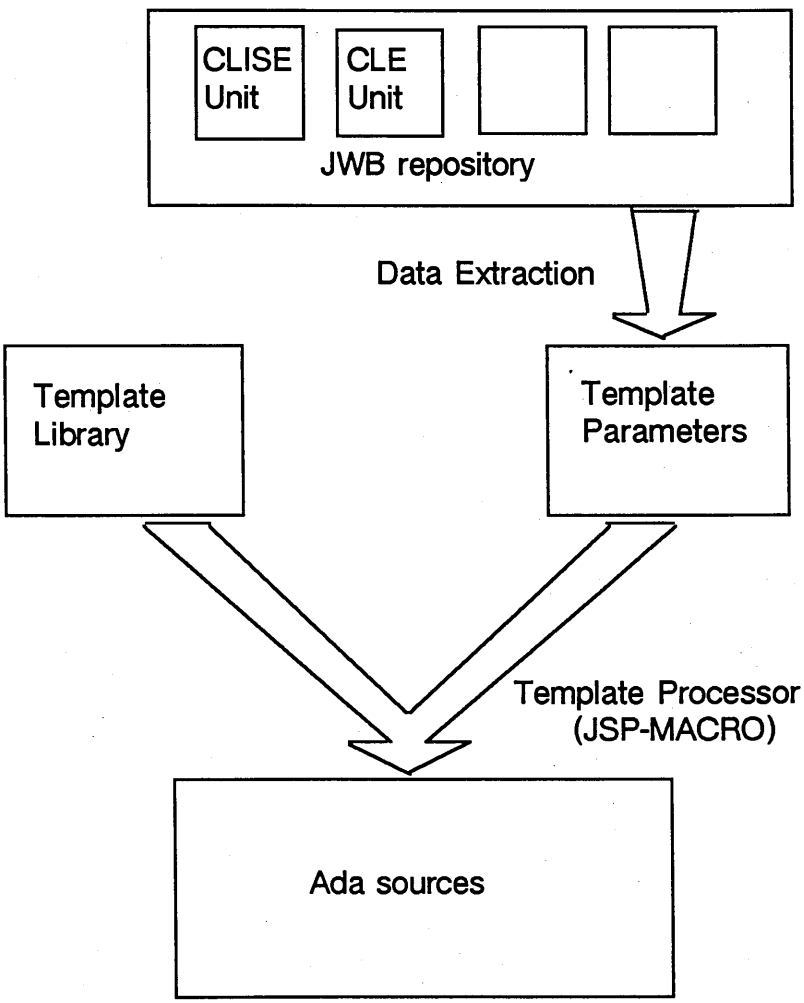



Figure 7.8 Operation of Code Generation Tool

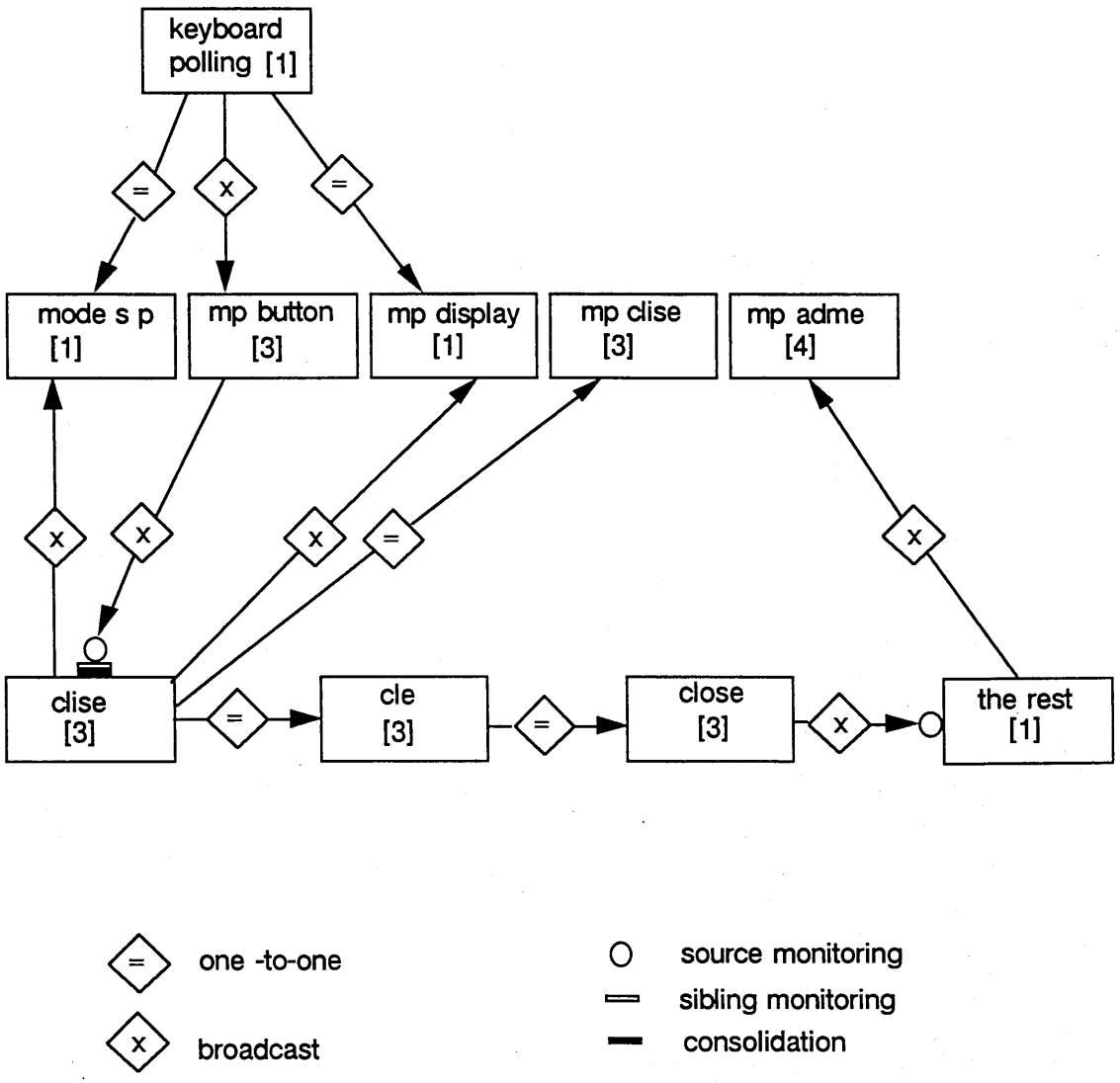



Figure 7.9 Menu Panel and Mode Select Panel Associated Unit Network

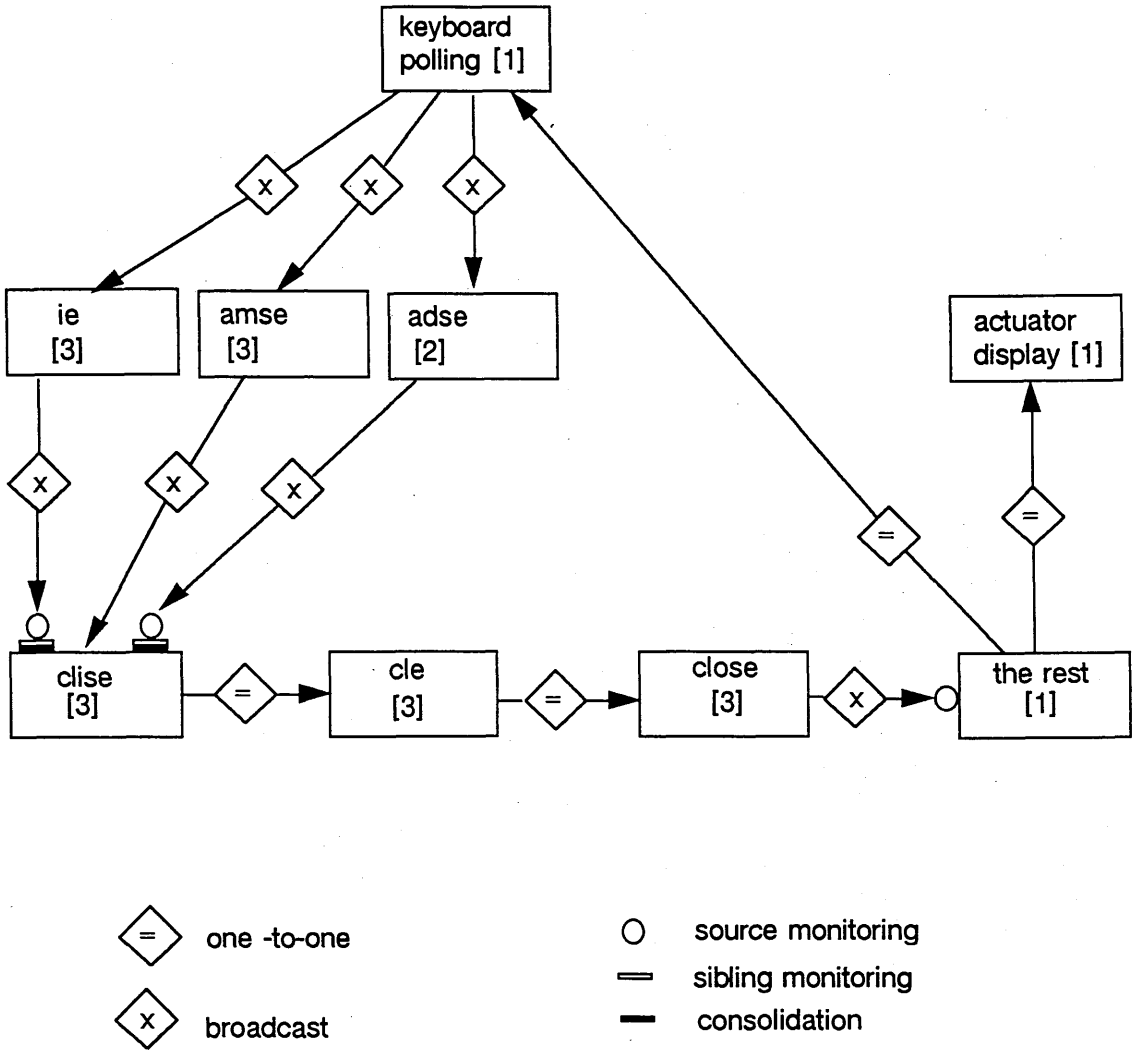



Figure 7.10 Helicopter Control Unit Network

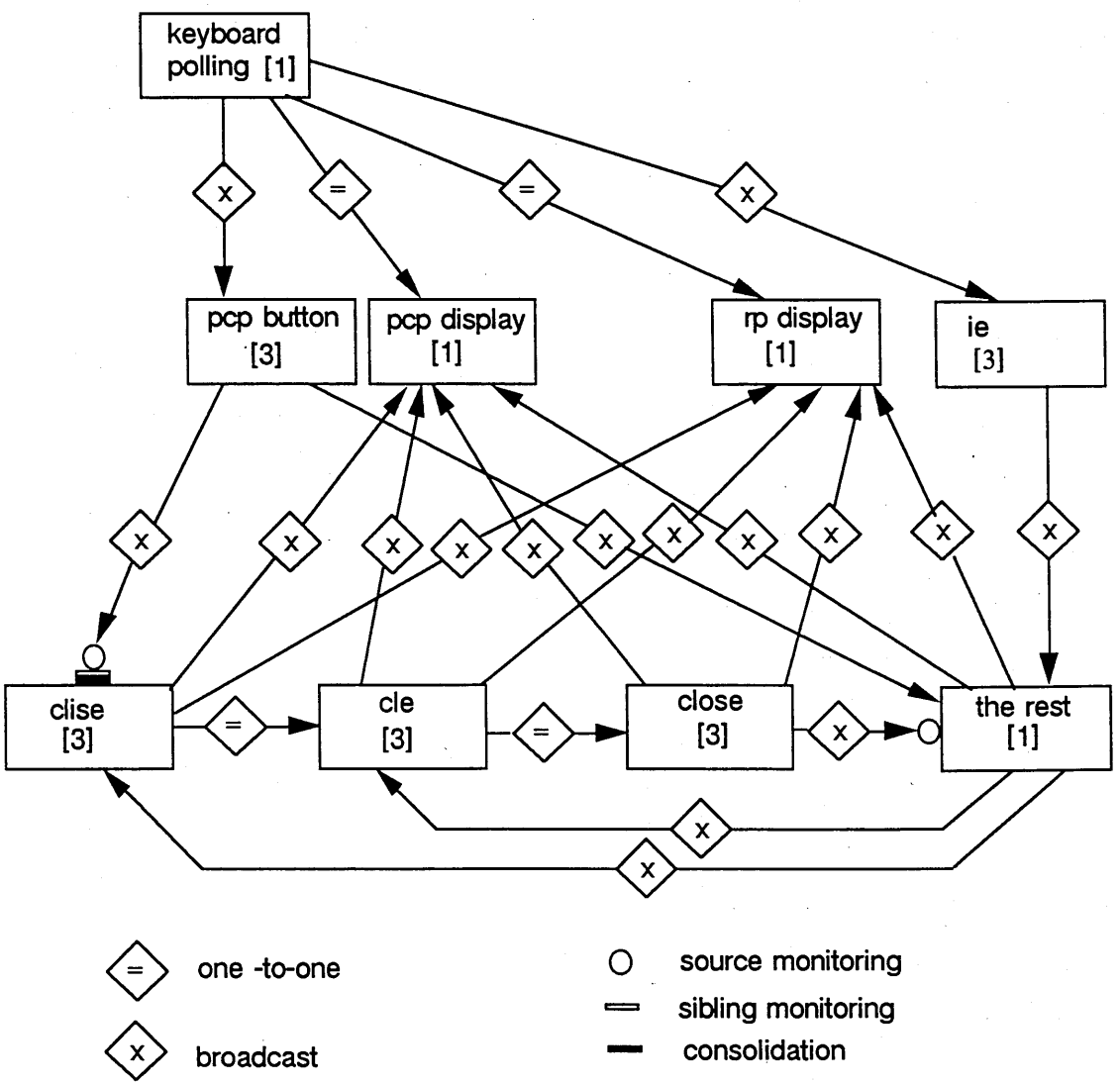



Figure 7.11 Pilots Control and Repeater Panels Associated Unit Network

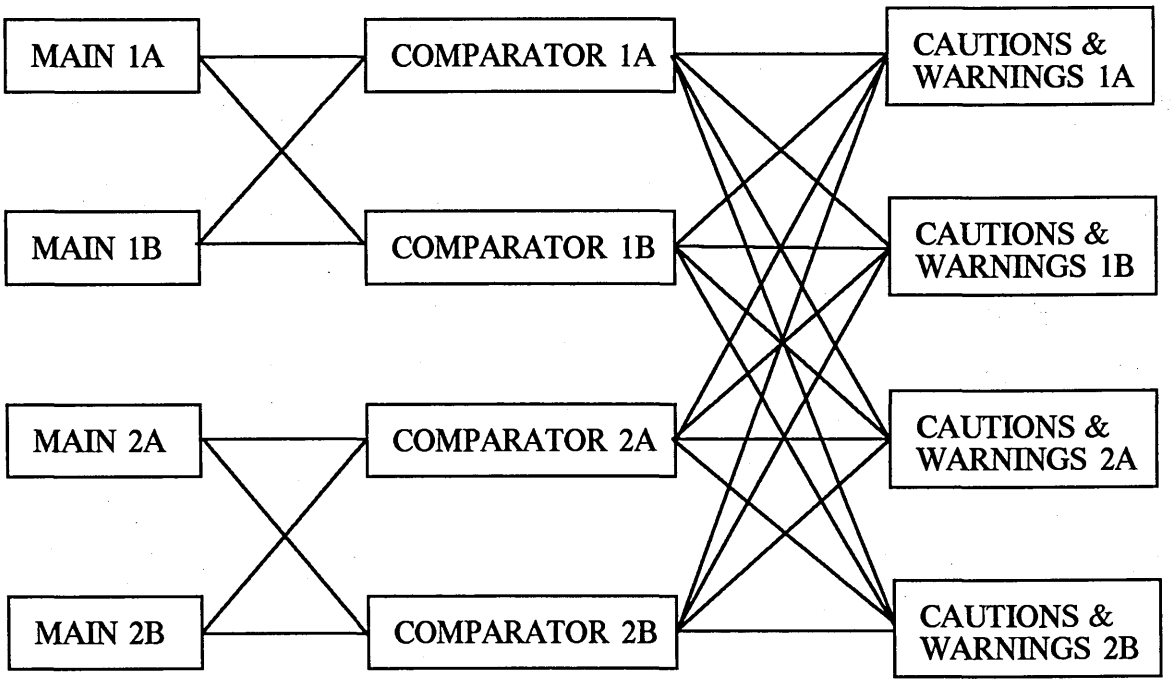



Figure 7.12 Internal Structure of the ADME

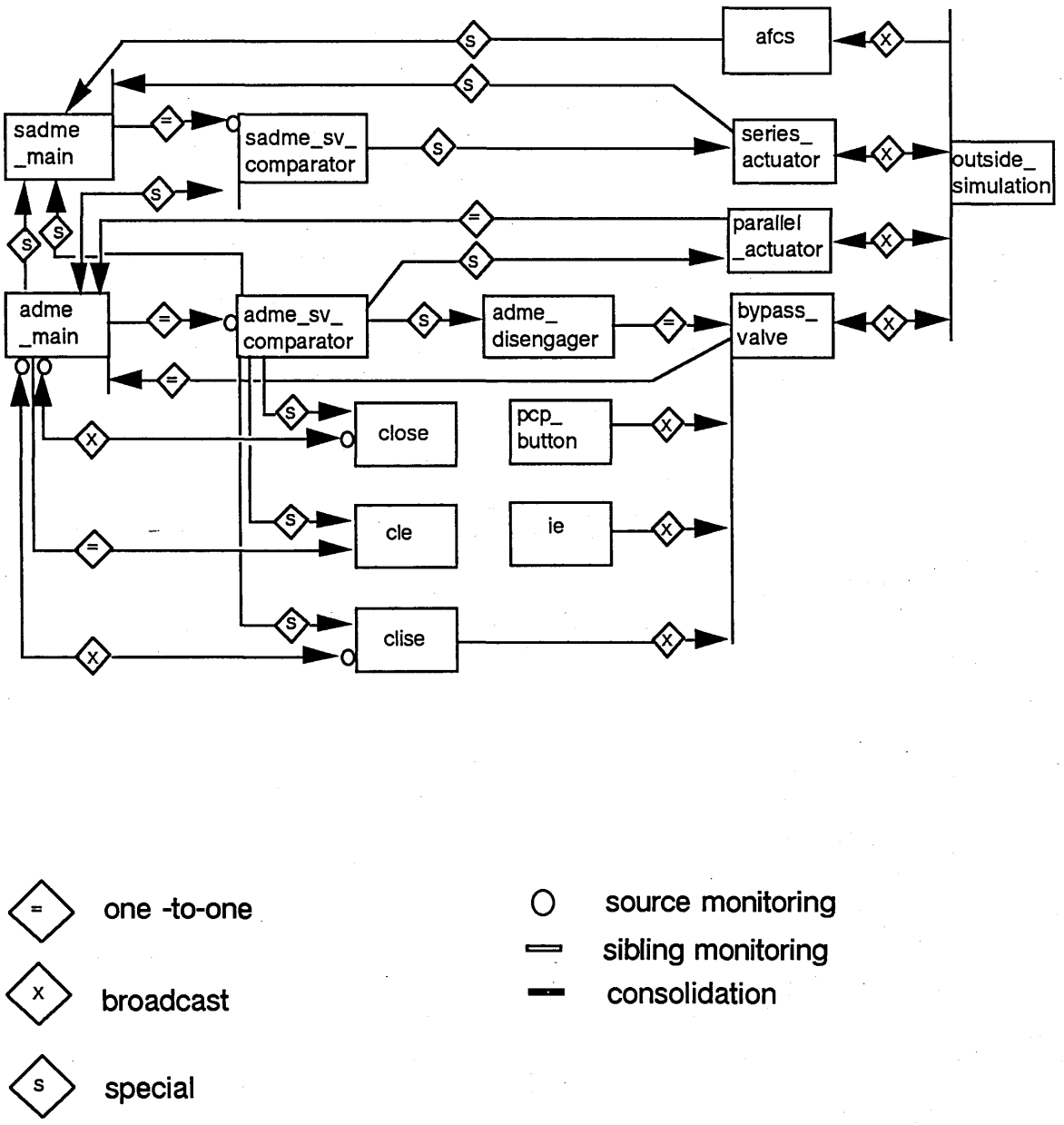



Figure 7.13 ADME Associated Unit Network

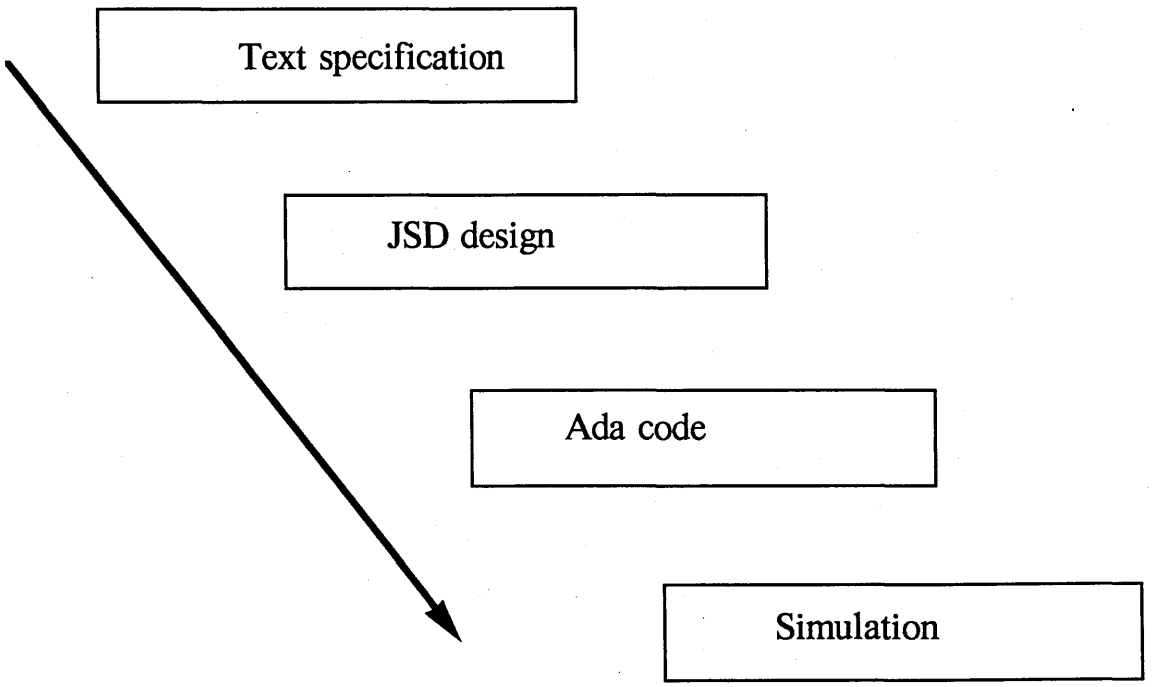



Figure 8.1 The Living Specification

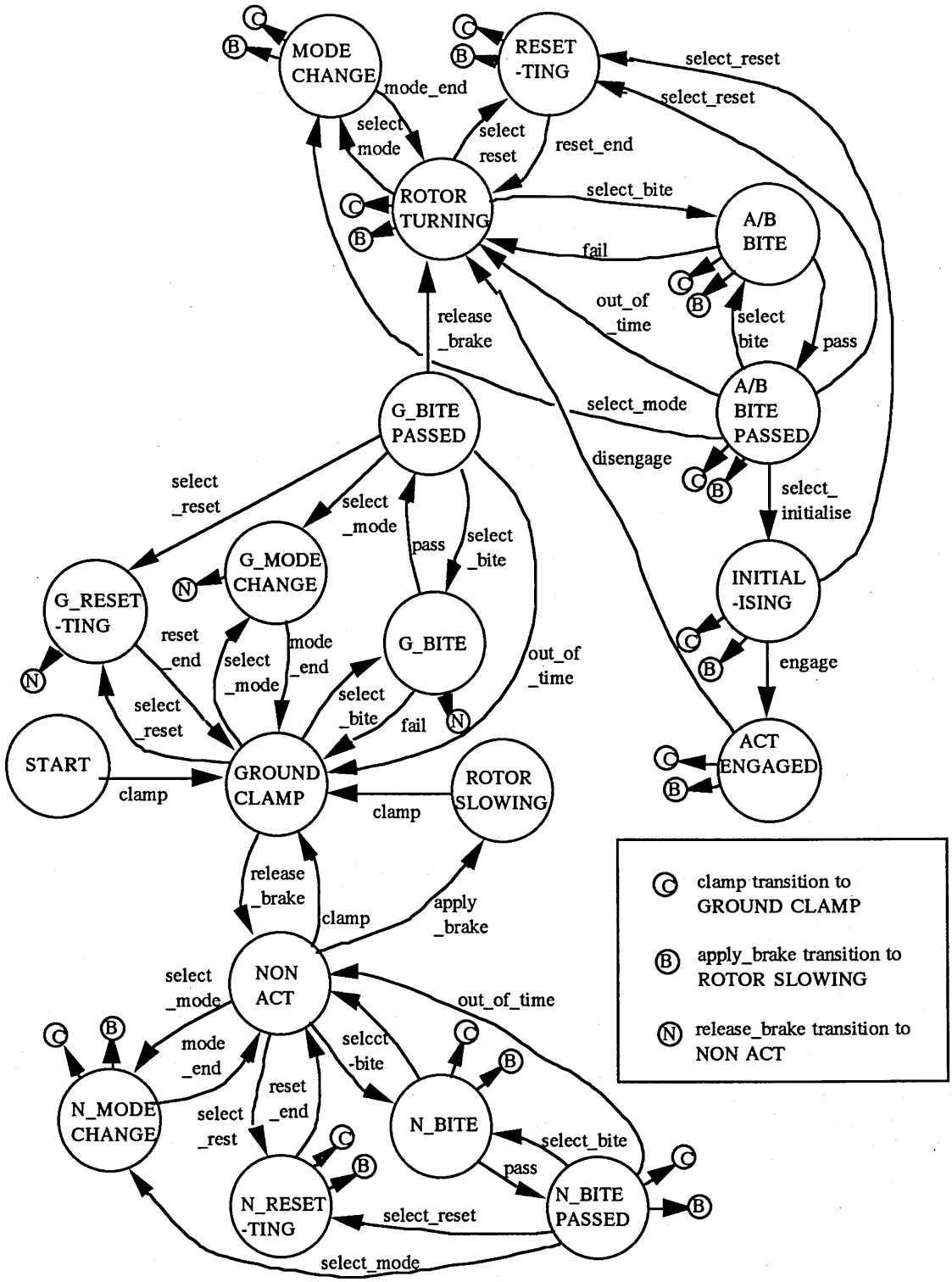



Figure A2.1 State Transition Diagram for Control Panel Supervisor

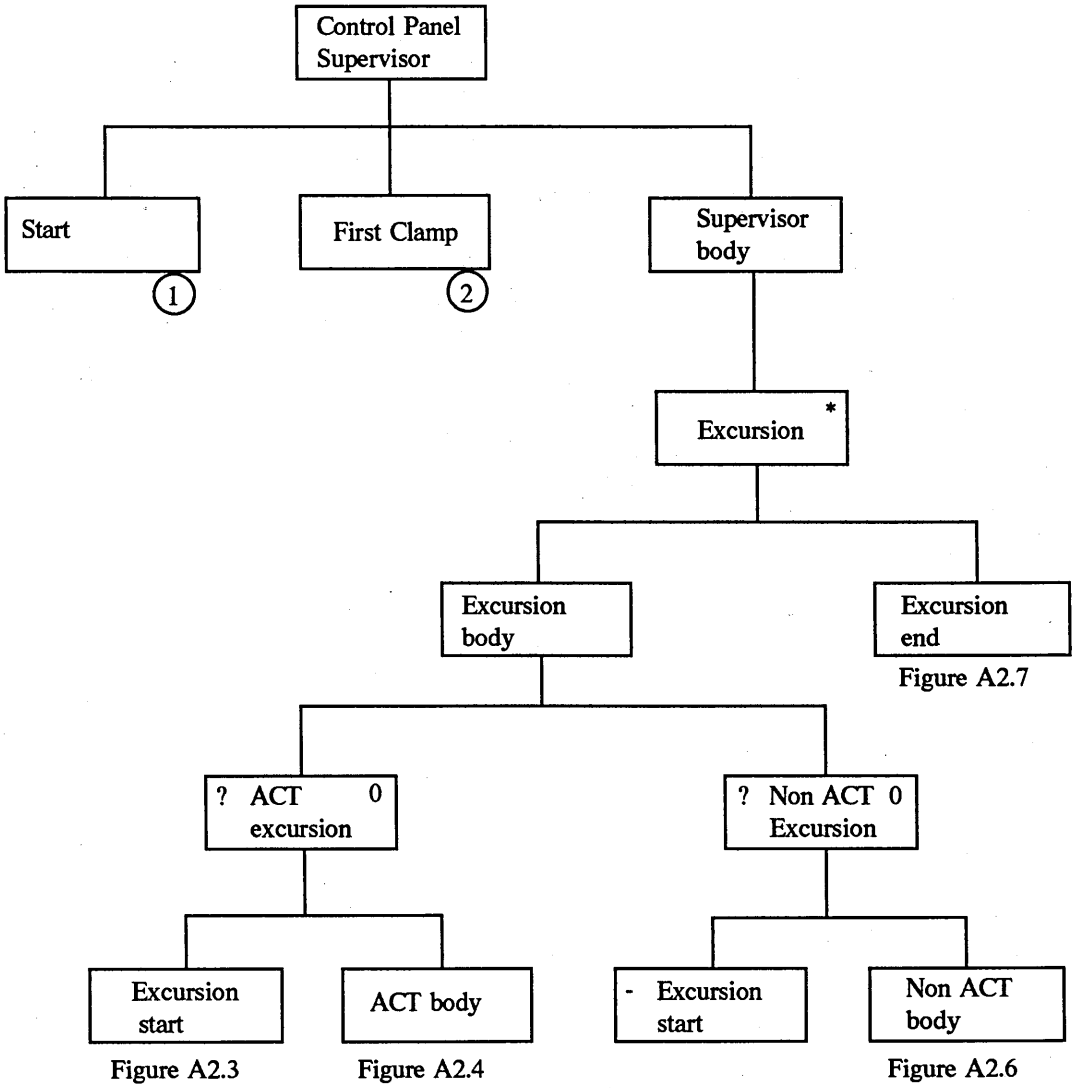



Figure A2.7

Figure A2.3

Figure A2.4

Figure A2.6

Figure A2.2 The Control Panel Supervisor

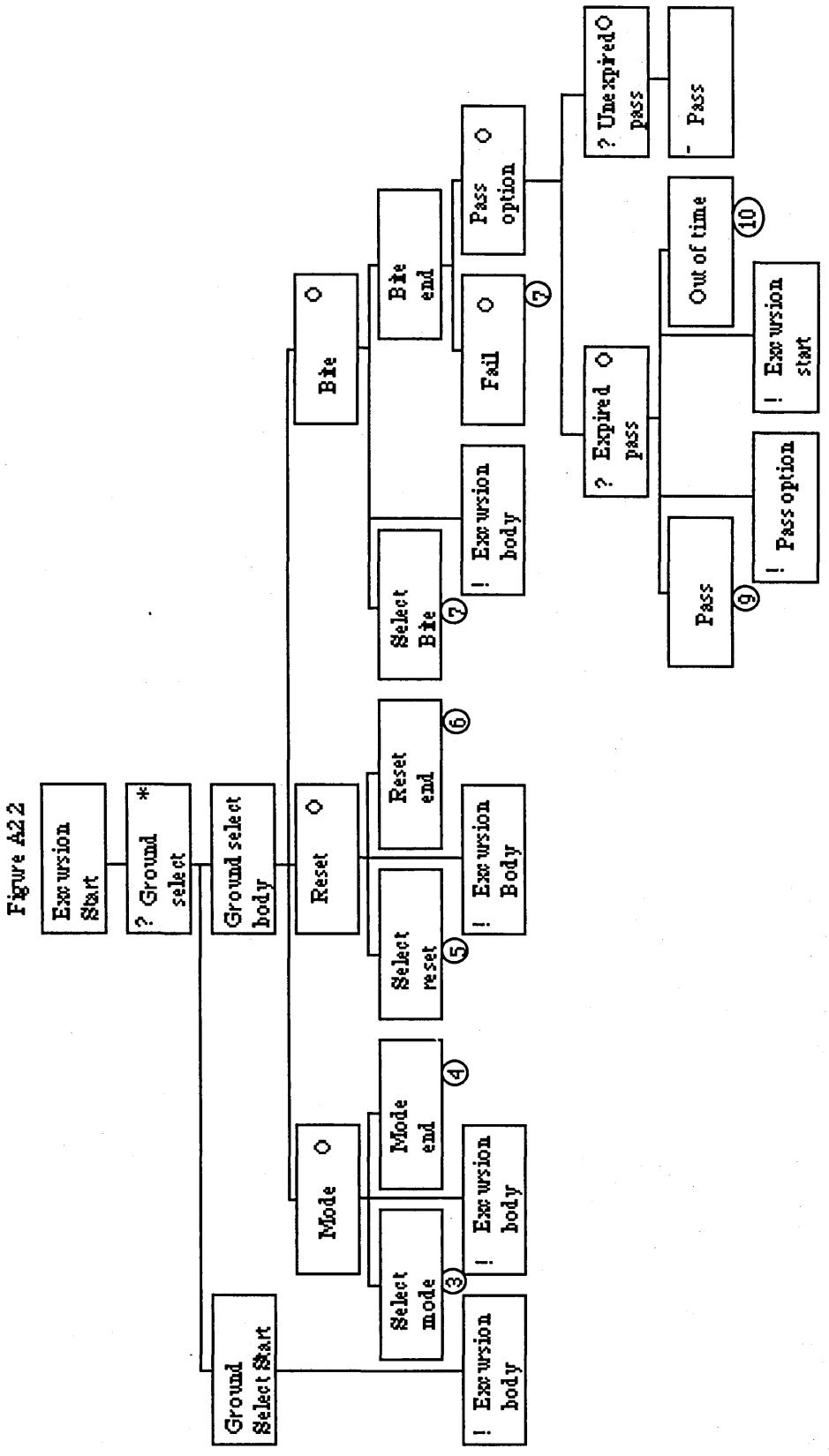



Figure A2.3 Excursion Start Component

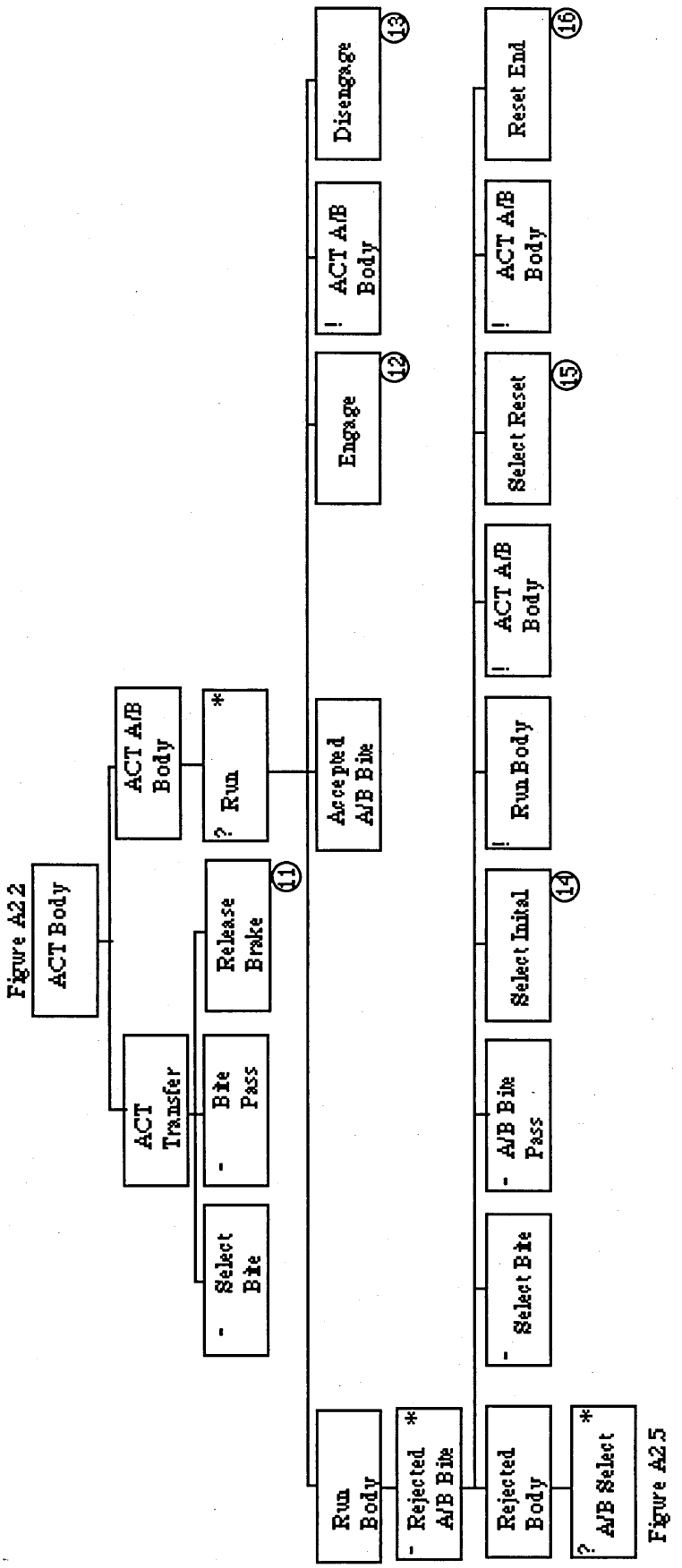



Figure A2.4 The ACT Body Component

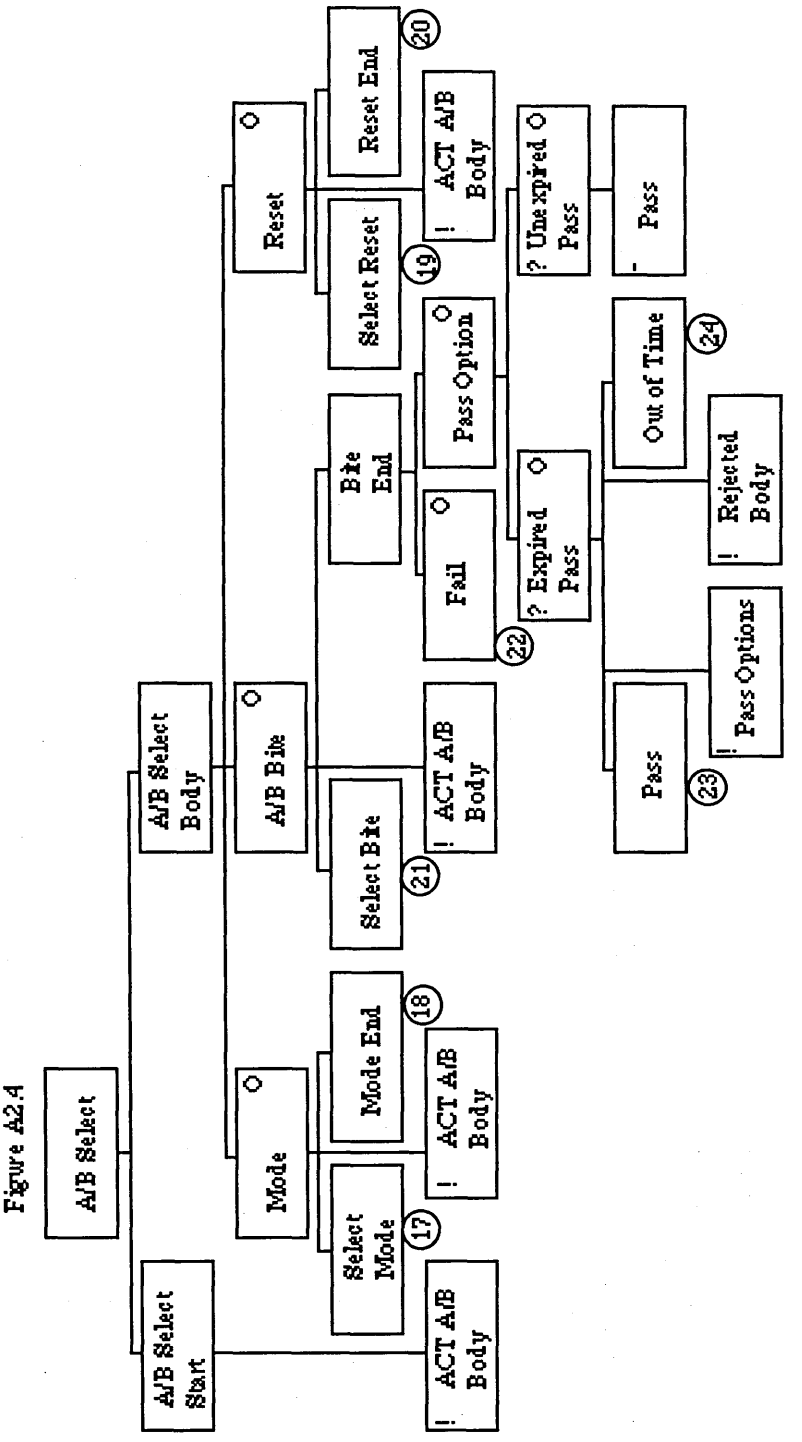



Figure A2.5 The A/B Select Component

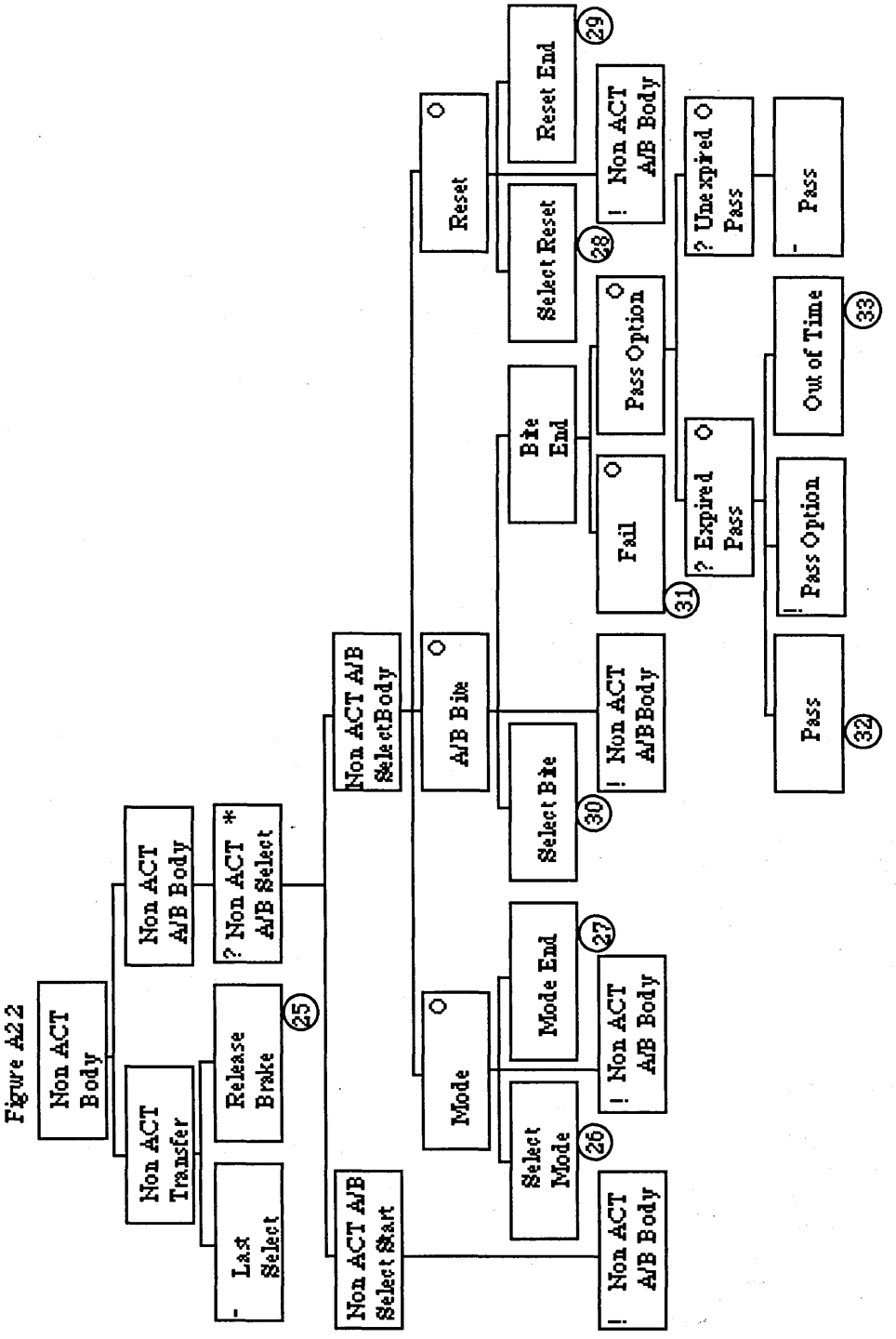



Figure A2.6 The Non ACT Body Component

Figure A2.2

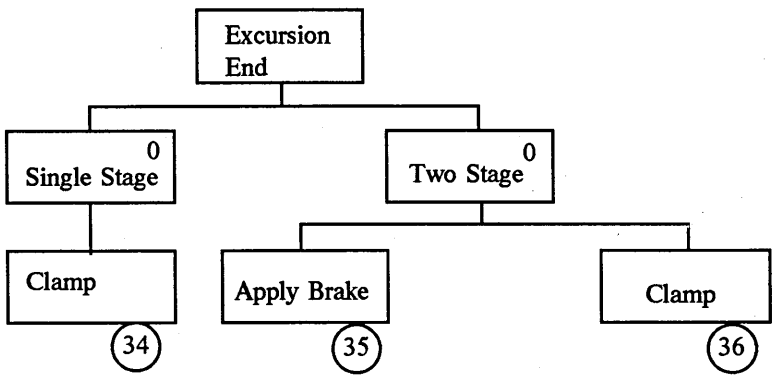
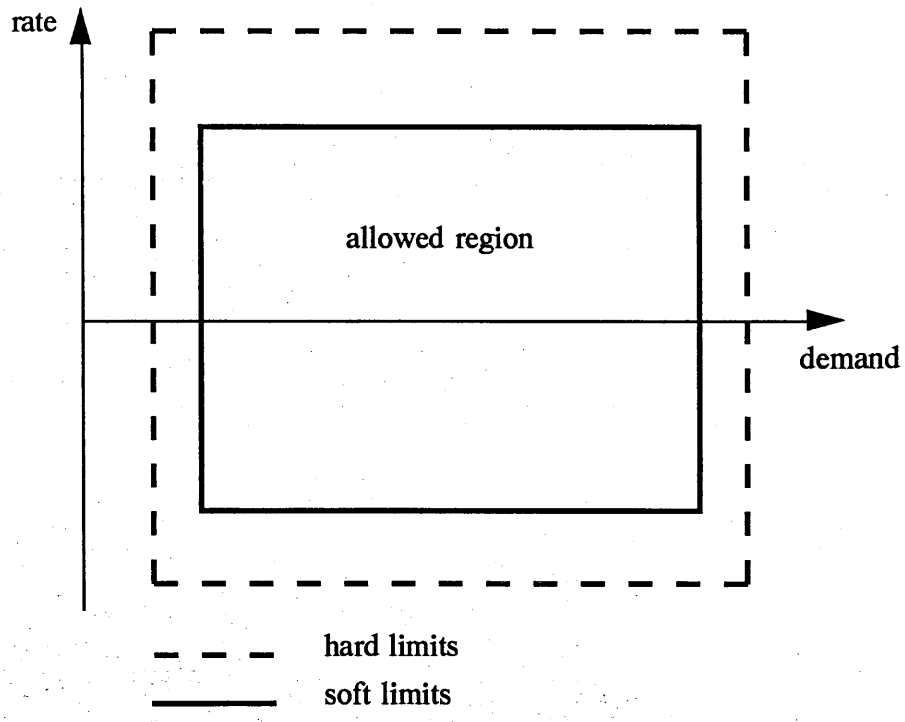
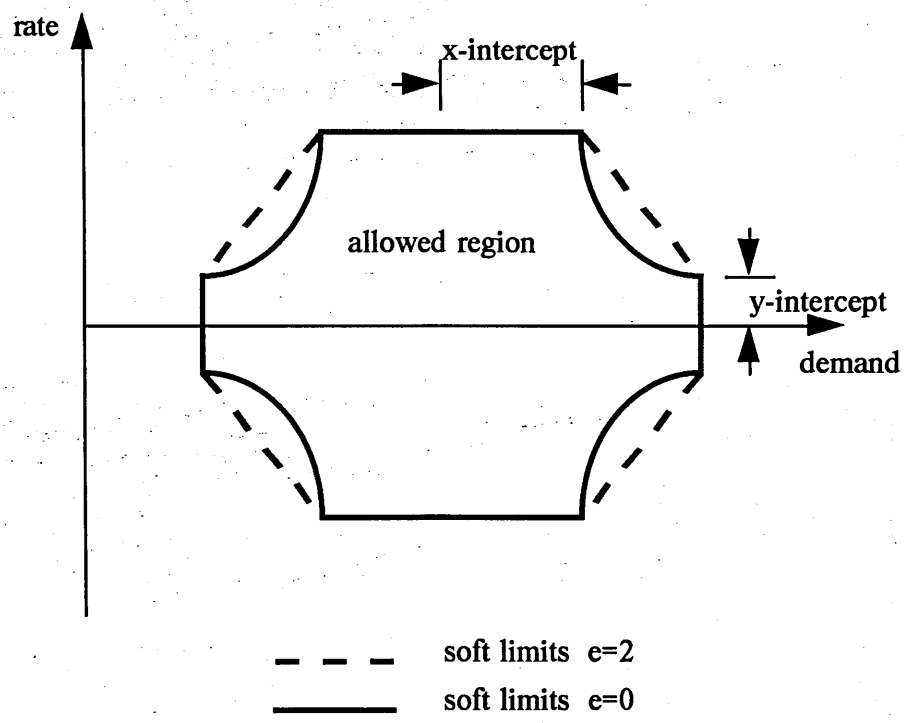



Figure A2.7 The Excursion End Component

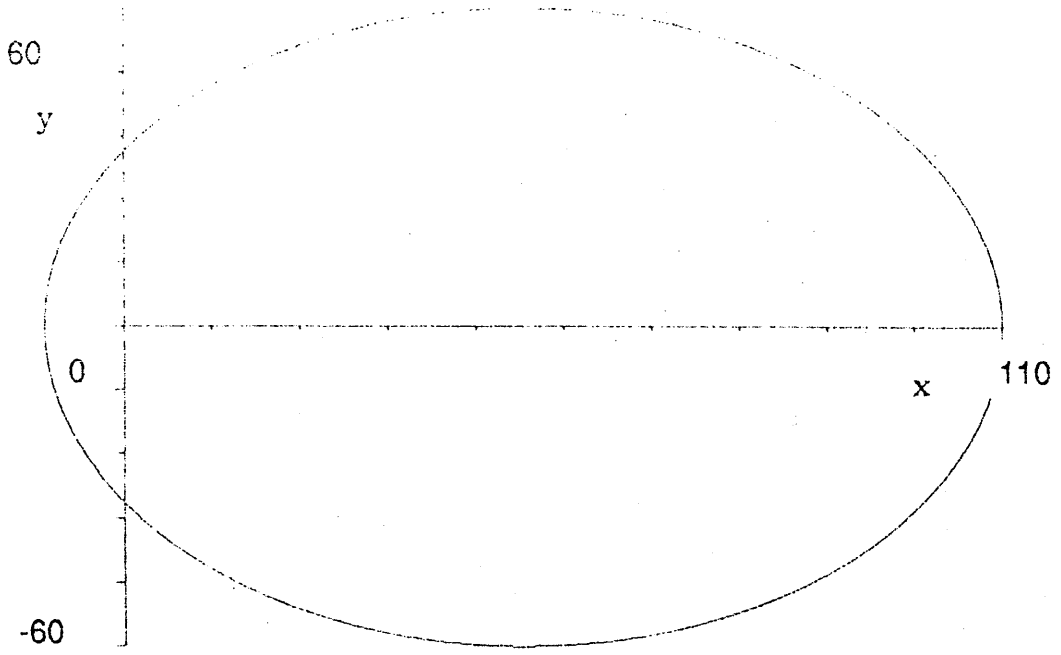



(a) Simple demand and rate limits

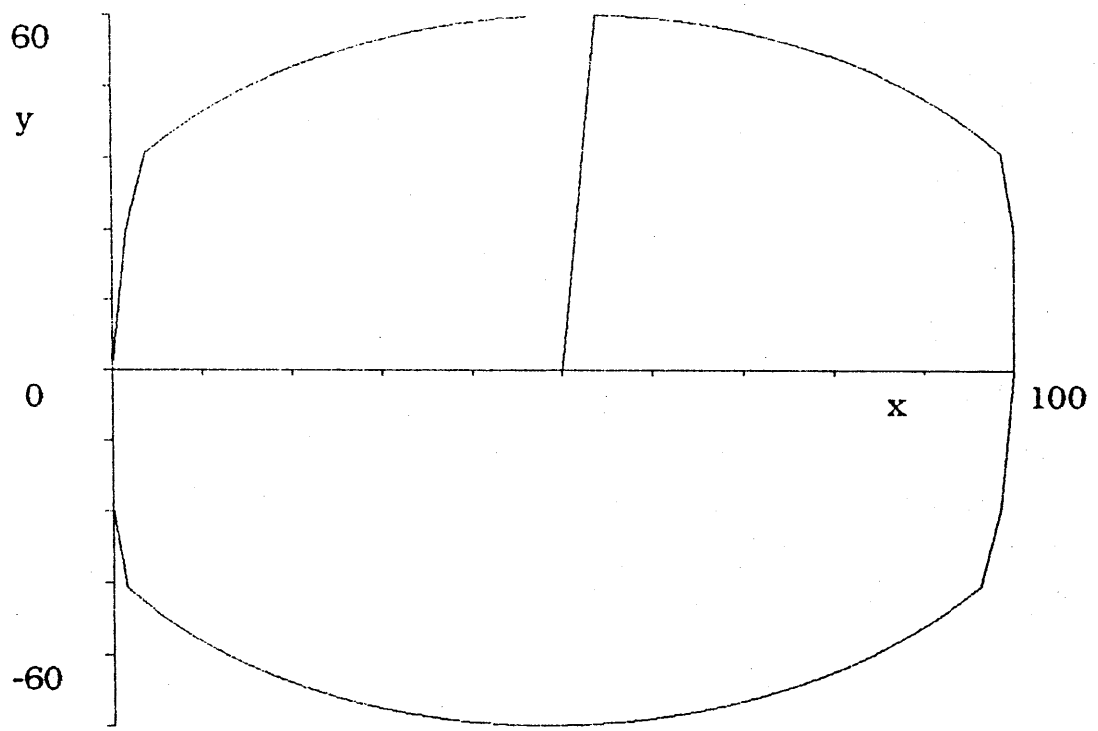



(b) Curtain limiter

Figure A3.1 Phase plane limiting

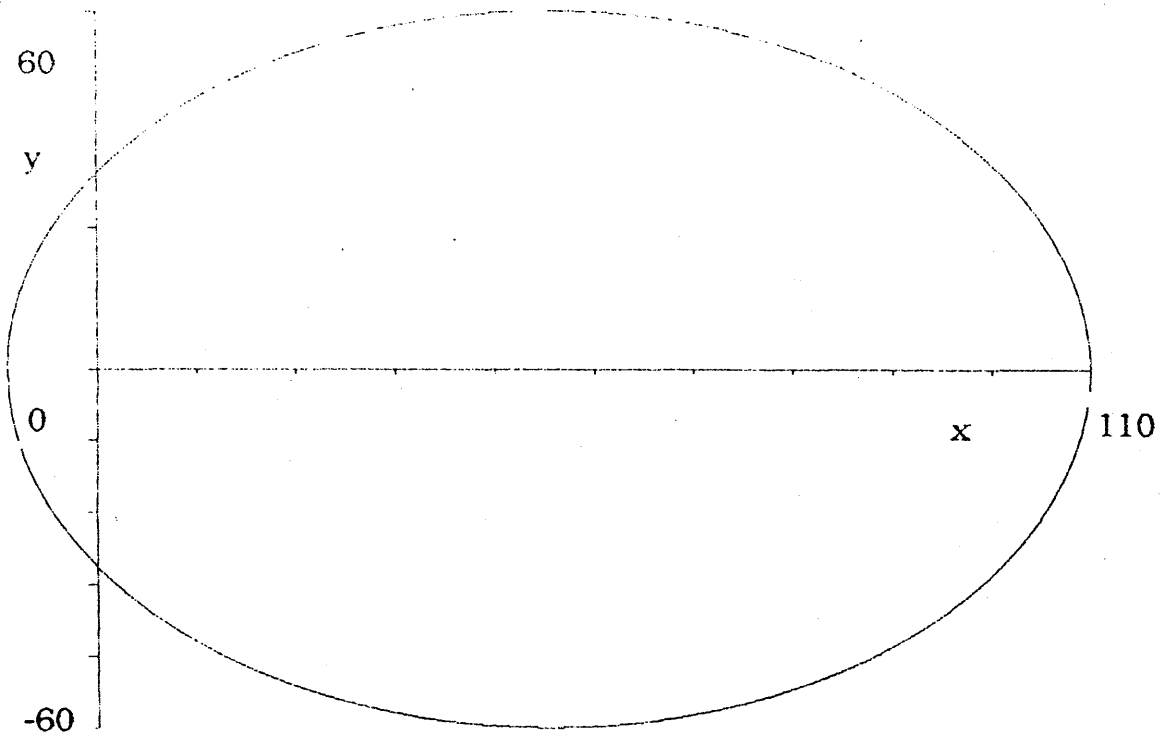



(a) Input demand x = 50 + 60 sin t

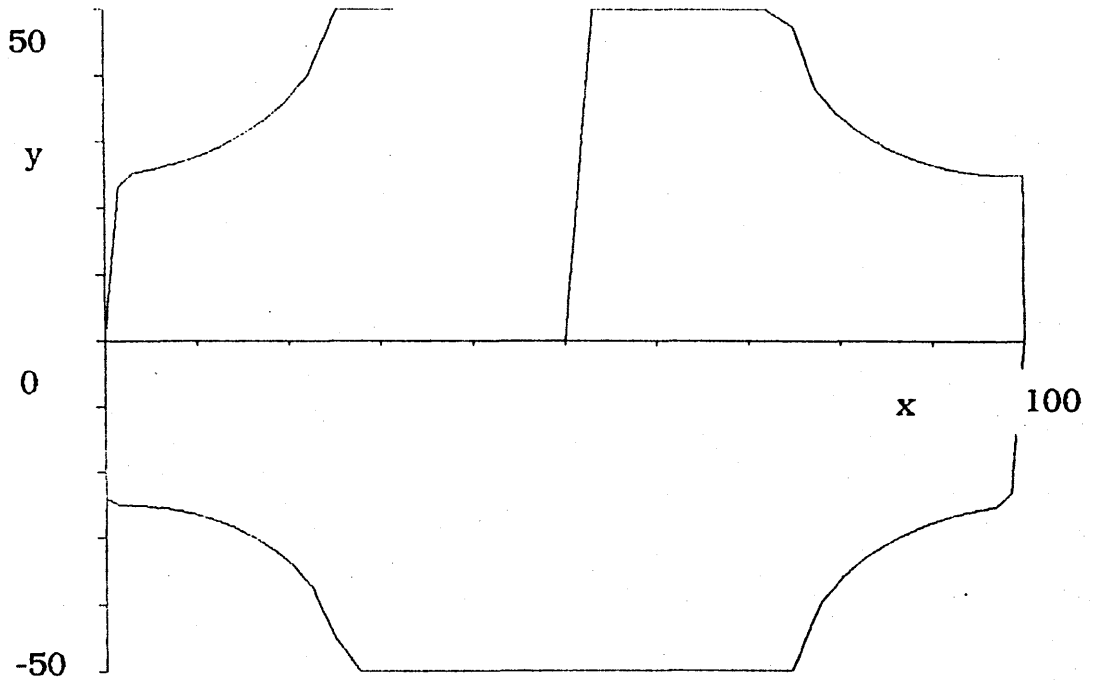



(b) Limited output with e=0, dx=0.5,dy=0.5

Figure A3.2 Original Curtain Limiter Algorithm

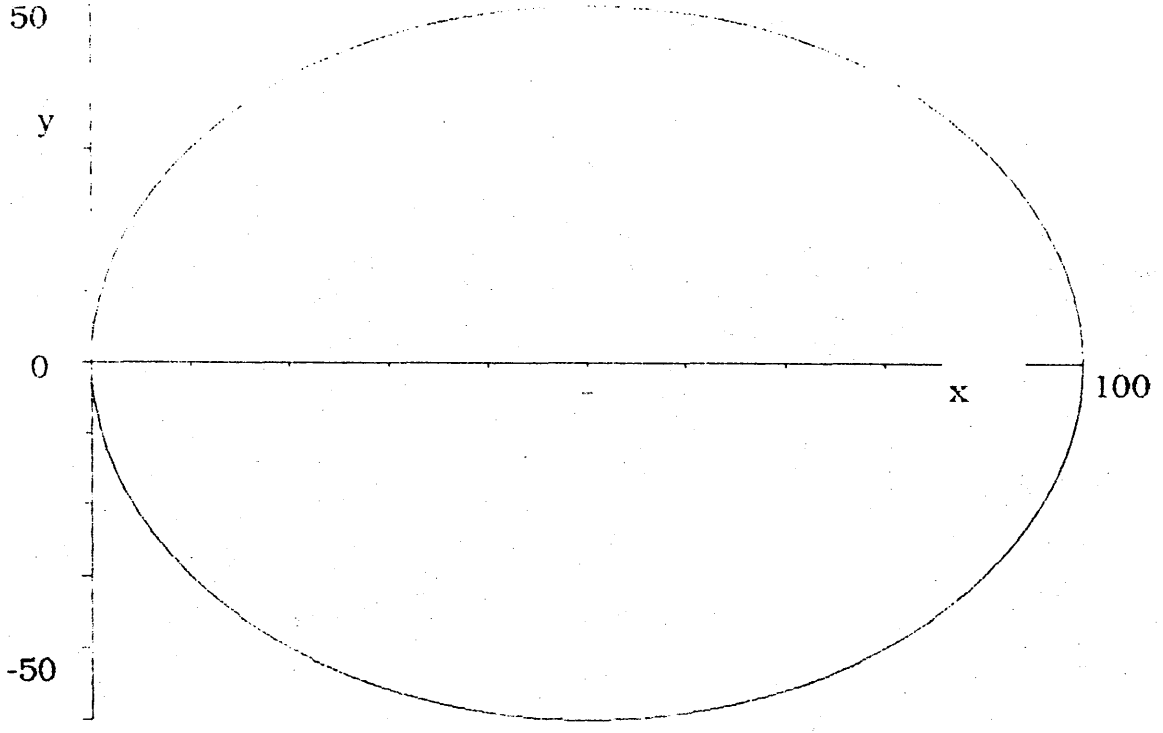



(a) Input demand x = 50 + 60 sin t

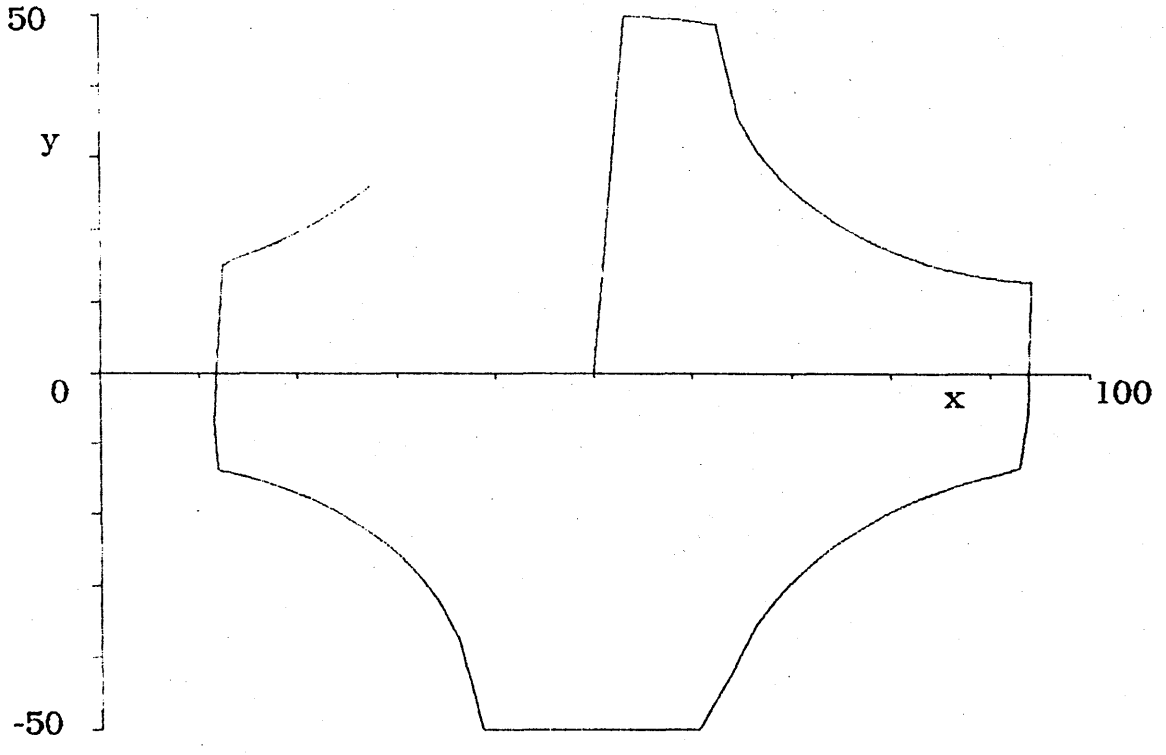



(b) Limited output with e=0, dx=0.5,dy=0.5

Figure A3.3 New Curtain Limiter Algorithm: case 1

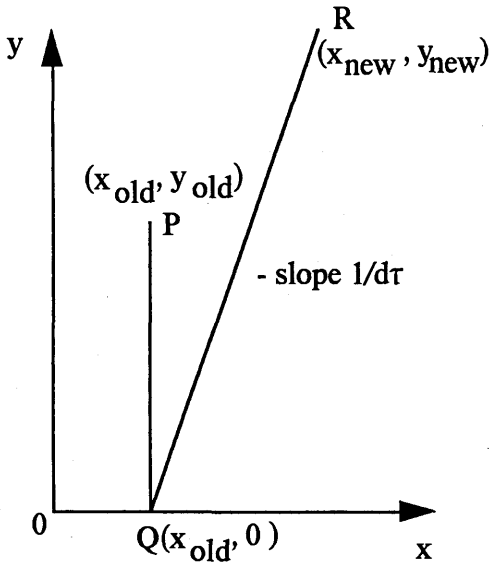



(a) Input demand x = 50 + 50 sin t

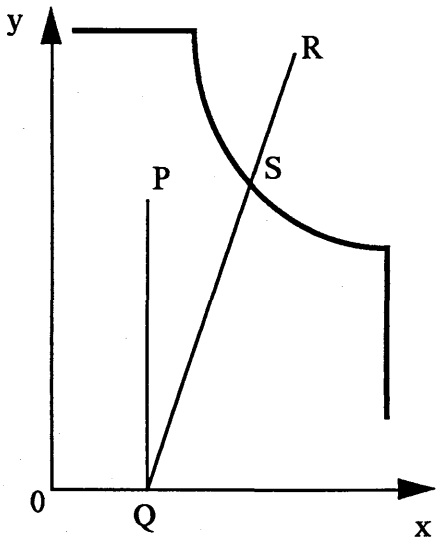



(b) Limited output with e=0, dx=0.25,dy=0.25

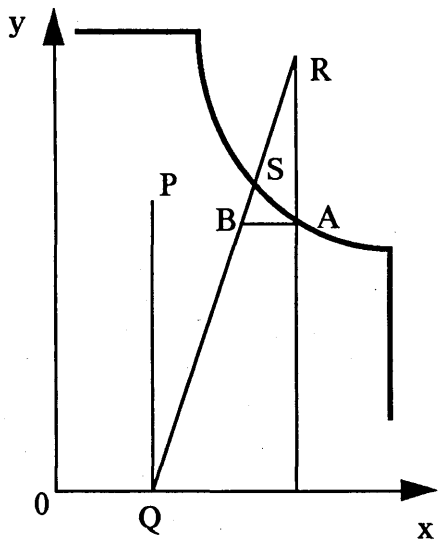
Figure A3.4 New Curtain Limiter Algorithm: case 2.



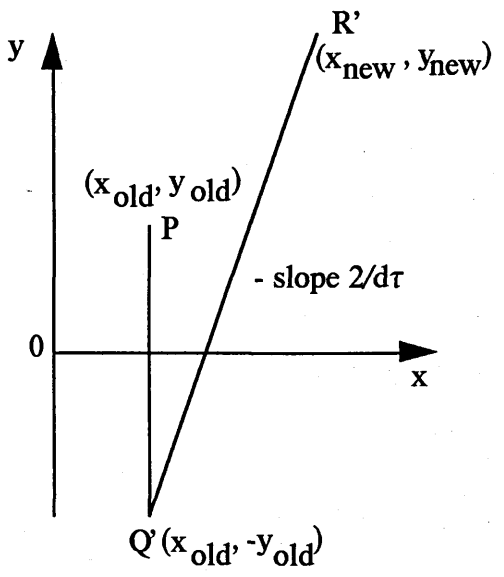

(a) The characteristic line




(b) Intersection with limit




(c) Approximation to limit




(d) Alternative method

Figure A3.5 Curtain limiter algorithm