AD-A241 948

UMENTATION PAGE

Form Approved
OMB No. 0704-0188

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
FINAL, 01 JAN 89 TO 30 APR 90

4. TITLE AND SUBTITLE
EVENT ORIENTED DESIGN AND ADAPTIVE MULTIPROCESSING

5. FUNDING NUMBERS
AFOSR-89-0157
61102F, 2304/A2

6. AUTHOR(S)
Dr David Lefkovitz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Dept of Computer and Information Sciences
Temple University
Philadelphia PA 19122

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 91 0795

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

10. SPONSORING MONITORING AGENCY REPORT NUMBER
AFOSR-89-0157

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The work performed under this grant relates to the performance design of real-time systems. Performance requirements specify time and reliability factors such as response time, throughput, precision, and fail safe/recovery characteristics.A design method, called Event Oriented Design and a corresponding tool, called the Embedded Software Design Simulator (ESDS), was developed.

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

EVENT ORIENTED DESIGN AND ADAPTIVE MULTIPROCESSING

FINAL REPORT

Principal Investigator
Dr. David Lefkovitz

Grant No. AFOSR-89-0157

August 31, 1991

Submitted To
Department of the Air Force
Air Force Office of Scientific Research
Bolling Air Force Base, DC 20332-6448

By

Department of Computer and Information Science
Temple University
Philadelphia, PA 19122

Research in Design Methodology:

Event Oriented Design

Abstract

The work performed under this contract relates to the performance design of real-time (RT) systems. Performance requirements specify time and reliability factors such as response time, throughput, precision, and fail safe/recovery characteristics. The most fundamental performance requirement of an RT system is *response time*, which is defined as the time elapsed between the appearance of a particular system input and the appearance of a specified output. In RT systems response time can be as critical as algorithmic or functional correctness.

The research performed under this contract had two major objectives. One was to analyze the current state of research in RT design, particularly for mixed asynchronous/synchronous systems, and to map it into a classification. The classification could then serve two purposes. One, to determine whether there exists a unifying concept in RT design; the other, to determine whether there are serious gaps in our knowledge about these systems.

The second objective of the contract was to develop a design technique to handle a part of the problem indicated as lacking by the classification.

The Classification views software development based upon the way software is actually structured. It encompasses all software types but then focuses more specifically on real-time software. It relates various methods of analysis and design that have appeared in the literature to respective classes, thus revealing how the facets of sofware development are covered by the various published methods.

What results from the first objective is that there does not appear to be a unified concept or theory of software, much less RT software development. Rather, there are distinct facets of the problem that are addressed by different researchers. Some facets are covered more throroughly than others, and some gaps exist. The research, under objective 2, addresses two areas in which these gaps appear to exist within the RT software subclassification. One relates to a class of systems called *communicating multitasking* systems. A design method, called *Event Oriented Design* and a corresponding tool, called the *Embedded Software Design Simulator (ESDS)*, was developed. The other gap is related to scheduling and allocation algorithms for a particular class called *multistate task activation* systems.

Research in Design Methodology:
Event Oriented Design

Table of Contents

Table of Contents (Cont'd)

EVENT ORIENTED DESIGN AND ADAPTIVE MULTIPROCESSING

Final Report

David Lefkovitz

Grant No. AFOSR-89-0157
Air Force Office of Scientific Research

1. Introduction

The work performed under this contract relates to the performance design of real-time (RT) systems. All systems can be said to have both functional and performance requirements. Functional requirements specify the content and format of inputs and outputs and specify the algorithmic relationship of each output to the inputs that produce it. That is, it is a mapping of inputs to outputs; hence, the analogy to a mathematical *function*. Performance requirements specify time and reliability factors such as response time, throughput, precision, and fail safe/recovery characteristics.

The most fundamental performance requirement of an RT system is *response time*, which is defined as the time elapsed between the appearance of a particular system input and the appearance of a specified output. The complete set of input/output pairs to which such a requirement can be attached comes from the functional requirement, though it need not be the case that every functional I/O pair have a corresponding performance requirement.

A major difference between real time and other systems is that the former have critical time constraints in the form of response time requirements. These are sometimes called "mission critical" to emphasize the fact that failure to satisfy these requirements can result in a failure that is as critical as an incorrect computation. However, the design process of real time systems does not differ significantly from that of other system types. We design for the functional requirements and then tune the system after initial implementation for performance. This tuning may occur at various levels. At the lowest, and least expensive, code can be tightened to run faster. At successively higher and more expensive levels, tuning can take the form of algorithm change, program architecture restructuring (eg recomposition of tasks), addition or change of processors, if this is an option, and, as a last resort, relaxation of the performance specifications themselves.

In RT systems response time can be as critical as algorithmic or functional correctness. Up until recently most RT systems have used the cyclic executive method as a technique for assuring response time. By this method tasks are assigned a particular time slot in a round robin executive process. It's a very effective design approach for systems with purely periodic tasks, but not for systems largely populated with asynchronous or sporadically initiated tasks [1]. Demands of high performance and highly interactive systems have now swung heavily toward the asynchronous, and Ada multitasking along with distributed processing system configurations are a direct response to this need.

The research performed under this contract had two major objectives. One was to analyze the current state of research in RT design, particularly for mixed asynchronous/synchronous systems, and to map it into a classification. The classification might then serve two purposes. One, to determine whether there exists a unifying concept in RT design; the other, to determine whether there are serious gaps in our knowledge about these systems.

The second objective of the contract was to develop a design technique to handle a part of the problem indicated as lacking by the classification.
Section 2 of this report presents the classification as a view of software development based upon the way software is actually structured. It encompasses all software types but then focuses more specifically on real-time software. It relates various methods of analysis and design that have appeared in the literature to respective classes, thus revealing how the facets of sofware development are covered by the various published methods.

What results is that there does not appear to be a unified concept or theory of RT development. Rather, there are distinct facets of the problem that are addressed by different researchers. Some facets are covered more throroughly than others, and some gaps exist. The research of this contract addresses two areas in which these gaps appear to exist. One relates to a class of systems called *communicating multitasking* systems. A design method, called *Event Oriented Design* and a corresponding tool, called the *Embedded Software Design Simulator (ESDS)*, was developed. The other gap related to scheduling and allocation algorithms for a particular class called *multistate task activation* systems.

Sections 3 and 4 present the event oriented design approach and the Embedded Software Design Simulator. Section 5 presents the results of the multistate task activation problem.

2. A Classification Approach to the Description of Software Development

Software Engineering has made significant progress in the past decade with respect to methods and tools for the development of software systems. There exists a large body of literature on the subject and many tool and documentation products that are based upon these methods. Researchers in the field are faced with the questions of how their work relates to that of others, where their work fits in the general scheme of software development, how unique their work is, and whether and where there might be gaps requiring new research effort. Another question that might be asked is whether there exists a single, comprehensive development methodology or whether one could be created by integration of existing methods that address different aspects of the problem.

In order to answer these questions, two steps were required. First was to create a perspective or view of the entire spectrum of software development processes by which one could classify the relevant literature. Second was to identify those approaches, techniques and methods in the literature that exemplified these classes.

Because the orientation of this research is real-time systems, the classification is deepest in this area. Others can elaborate by further subdivision of specific classes that may be of

particular interest or by adding or modifying branches based upon different perspective or even new technology and. What will also be shown is that there is an inherent structure to all software systems that is revealed by the higher levels of the classification, and that different software types are elucidated within the substructural elements or subclasses.

2.1 Approach to the Development of the Classification

Formally, a classification is a tree structure. Therefore, as with the development of a system, one can adopt a top down or a bottom up approach. A combination of the two was used in this case. First, it was decided that the major classes, or highest level breakdown, would represent a comprehensive set of cognitive elements existing in every software system. A decision had to be made as to whether the classification would primarily follow the life-cycle stages (analysis, design, implementation, etc.) or the way systems are inherently conceived and structured, independent of stage. The latter was chosen because methodologies are *structured thought processes*. As such they are best distinguished (which is the main purpose of the classification) by looking at the *object* of these thought processes, which in this case is an inherent conceptual element of the software, such as *data structure, algorithm, program structure, data flow, external interface, etc.* The life cycle stages are merely steps in the over-all development process of transforming concept to software implementation, and even though a given method may be designated for use in a particular stage, it was felt that the more significant distinction or relation between methods is to be found in how they deal with the above cited conceptual aspects of the system. This is not to prevent one from assigning a life cycle stage attribute to a method to further distinguish it. This led to the following top level breakdown.

1. Hardware-Software Architecture
2. External Interfaces
3. Operational modes
4. Data and Control Flow
5. Program Structure
6. Data Structures
7. Functional Logic

The order of these major classes is also significant. They form an approximate hierarchy, as shown in Figure 1, where the path from top to bottom represents a progression from the external to internal environment and from operational to functional concepts as well as to greater conceptual detail.

Viewing a system and its operational environment top down one can identify three principal units: (1) hardware-software relations, (2) external interface to devices or people that are served by the system, and (3) software. One could debate whether (1) or (2) is at a conceptually higher level, but regardless, these would form the top of the hierarchy. The first class, *Hardware-Software Architecture*, describes the architectural relationships between the hardware and software, such as single vs. multiprocessor configurations and the operating system. The second class, *External Interfaces*, will describe how devices or people interface to the system. The latter can be quite complex, involving what is referred to as the *user interface*.

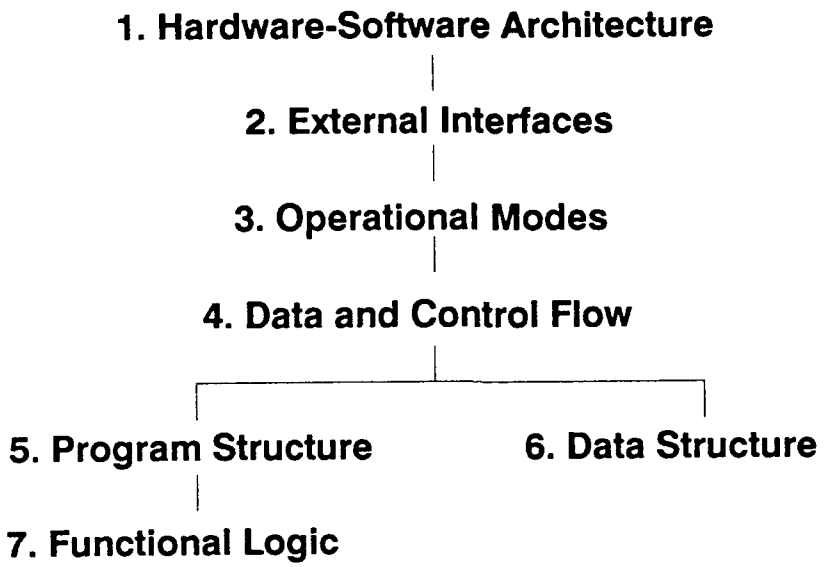


Fig. 1: Developmental Hierarchy of the Six Major Classes

The remaining classes characterize the software itself. The first is *Operational Modes*, which subdivides all software systems according to the way in which they operate, such as interactive, non interactive, real-time, etc. Having defined the system's environment in Classes 1 and 2, and having broadly classified software by operational type in Class 3, we turn to the next conceptual level, which is the block structure of major software components, usually called processes, and the flow of data and control throughout the system. This is Class 4, *Data and Control Flow*. Below this the picture divides, reflecting the basic duality of computing systems: *Data Structure* and *Program Structure*. Jackson, in his Data Structure Design method attempts to unify them [2], and Object Oriented Design attempts to create its own kind of symbiosis through the abstract data type, but the conceptual unification of these two characteristics still eludes us, because the two are inherently different. *Data* are essentially non active elements in the sense that, once declared, they are always acted upon either for interrogation or modification, whereas *programs* are the actors that perform upon the data; hence, both their roles and their essential structure are quite distinct. An important question is whether methodolgy should attempt to unify them. Having raised the question this paper does not offer an opinion on future development, but continues to recognize both their separate as well as parallel nature, as shown in the Figure.

In accordance with prevailing convention, program structure has been divided into two classes. The higher level, Class 5, characterizes the modular structure of the program, while the lower level, Class 7, characterizes the algorithms or functions employed within the modules.

Since the emphasis here is on real-time systems design, the next steps were (1) to develop the context for real-time systems within the classification and (2) to determine the most essential distinguishing characteristics of real-time systems, which would create the necessary depth in this part of the classification tree. This constituted the bottom up part of the process.

2.2 Definition of a Real-Time System

Participants at the Office of Naval Research Workshop on Real-Time Systems Research Issues in November, 1986, sought to define real-time computing. Despite initial differences, they seem to have agreed on the three major components [3]:

1. "Time" is the most important resource to manage in real-time systems. Tasks must be assigned and scheduled in such a way that they can be completed before their corresponding deadlines expire. Yet how to specify or determine deadlines is an unsolved problem. A deadline is said to be "hard" if its violation may lead to a catastrophe, as in an air traffic or flight control system. If satisfying a deadline too early or too late would cause inconvenience rather than catastrophe, as in transaction systems, the deadline is said to be "soft".

2. Reliability is crucial since failure of a real-time computer could cause loss of life, failure of a mission, and/or great economic loss.

3. The environment within which the computer operates is an active component of real-time systems. A real-time computer and its environment form a synergistic pair. For this reason such systems are sometimes called "embedded."

In summary, the fundamental characteristic of real-time systems is that system correctness depends as much on meeting time constraints as on producing a correct computation. By implication then, each software component of a real-time system must specifically take into account the response time required by applications using it [4].

This extended definition might be called *operational* as opposed to *structural*, where the former defines the required behavior of a system, while the latter would directly address the system's design features. This dichotomy is of particular interest here, because it is within Class 3, *Operational Modes*, that real-time was placed in order to distinguish it from other system types, but it is the latter that is more useful in the subclassification below that point. What structural characteristics, then, affect "time constraints" and what methods of analysis or design apply? Four such characteristics were identified: (1) centralized vs. distributed architecture, (2) intertask synchronization, (3) queuing, and (4) response time. The first of these was assigned to Class 1 and the others to Class 3.

2.3 The Classification

Table 1 presents the classification based upon the above criteria. The bold typeface identifies representative approaches or methods, along with their literature references.

TABLE 1
Classification of Analysis and Design Methods

1. Hardware-Software Architecture
   - 1.1 Processor Configuration
     - 1.1.1 Single
     - 1.1.2 Multiple
       - • Centralized
       - • Distributed
         - * **Program Partitioning [5]**
         - * **Language Constructs [6]**
         - * **Real-Time Task Dispatching [7]**
   - 1.2 Operating System

2. External Interfaces
   - 2.1 Devices (embedded)
     - * **Approach of the NRL Software Cost Reduction Project [8]**
   - 2.2 People
     - * **Person-Machine Interaction [9]**

3. Operational modes
   - 3.1 Interactive
     - 3.1.1 Real-time
       - 3.1.1.1 Program Structure - Concurrency
         - 3.1.1.1.1 Single task
           - * **Cyclic Task Executive [1]**
         - 3.1.1.1.2 Multitask
           - * **Concurrent Programming [10]**
           - * **Communicating Sequential Processes [11,12]**
           - * **Ada (Rationale) [13]**
             - 3.1.1.1.2.1 Coupling
               - • Tight
               - • Loose
                 - * **DARTS [14,15]**
                 - * **MASCOT [16]**
                 - * **Ada (Rationale) [13]**
       - 3.1.1.2 Performance (Response Time)
         - 3.1.1.2.1 Task Execution Time
         - 3.1.1.2.2 Task Communication (Same as 3.1.1.1.2.1)
         - 3.1.1.2.3 Task Activation Mode
           - 3.1.1.2.3.1 Temporal
             - • Periodic (synchronous)
             - • Sporadic (asynchronous)
           - 3.1.1.2.3.2 Control
             - • Unistate
             - • Multistate
               - * **Structured Development of RT Systems [17,18]**
         - 3.1.1.2.4 Task Scheduling
         - 3.1.1.2.5 Task Allocation
         - 3.1.1.2.2 Response Time
     - 3.1.2 Rapid Response
   - 3.2 Non Interactive

4. Data and Control Flow
   - * **Dataflow Diagrams [19,17,18]**
   - * **Actigrams [20]**
   - * **Activity Charts [21]**
   - * **HIPO [22]**

5. Program Structure
   5.1 Module Construction
      * **Functional Decomposition (Structured Design) [23]**
      * **Module Construction [23,24]**
      * **Object Oriented Design [25,26]**
   5.2 Control and Packaging
      * **Buhr Diagram [27]**
      * **Structure Chart (Call Tree) [23]**
      * **Program Structure Notation [See Appendix A]**

6. Data Structures
   * **E-R Diagrams [28]**
   * **CODASYL Diagrams [29]**
   * **Relational Databases [30]**
   * **Basic File Structures [31]**

7. Functional Logic
   * **Structured Flow Charts [32]**
   * **Decision Tables [33]**
   * **Warnier-Orr Diagrams [34]**
   * **JSD Structure Charts [2]**
   * **PDL, Structured English, Pseudocode [35]**
   * **Algebraic Specifications [36]**
   * **Non Procedural Languages [37,38]**

The Table is incomplete in the sense that (1) the methods cited represent a fraction of the total publications on the subject, and (2) the classification itself is subject to additional interpretation and development. Its purpose here is to focus on the Real-Time systems (subclass **3.1.1**), but it is hoped that others would continue its development for their own purpose. For example, there certainly exists a significant literature on subclass **2.2 External Interfaces: People**, though most of it describes rather informal approaches (usually described by example) to the design of screens and person/machine interactions.

To the extent that one might find a unifying conceptual structure, it may be seen in the way the seven major classes form an over-arching developmental hierarchy as shown in Figure 1; however, within the subclasses there is considerable overlap and diversity of method, which leads to the conclusion that a well defined, comprehensive approach to the analysis and design of software systems does not yet exist. What we have instead are specifically tailored techniques for particular elements and subelements of the classification, where individual practitioners may either adapt a particular method to their own working environment and project or may ignore all published work and develop an *ad hoc* approach.

2.4 Real-Time Systems

Class 3 divides all software systems into three types based upon their *Operational Mode*. First, there are two main subclasses, **Interactive (3.1)** and **Non Interactive (3.2)**. All computing systems can be characterized operationally by the activity pattern: *input-process-output*. The essential difference between interactive and non interactive systems lies in the interpretation and in the repetitive nature of this pattern. In the former the pattern itself is adhered to very strictly and repeats many times within the total run time of the program.

In addition, there is usually a reponse time requirement placed on the execution of each instance of the pattern, ie, the time between the presentation of an input to the system and the appearance of its corresponding output.

In the non interactive system the pattern can have variants and, depending on the variant, may have a high or low repetition rate within the run. For example, the pattern may be defined as: $input_1$, $input_2$, ..., $input_n$-*process-output*. Each $input_i$ is called a *transaction*. The output might be a report. This pattern would normally occur only once during the run. Alternatively, one could define a pattern: $input_i$-process-$output_i$, where $output_i$ represents the update of a database, and this pattern repeats n times throughout the run. There may also be a performance requirement, called throughput, that is basically the reciprocal of response time. It is defined as the number, *n*, of transactions processed per unit time.

Class 3.1: Interactive

Within the *Interactive* subclass there are **Real-Time (3.1.1)** and **Rapid Response (3.1.2)**. Here the term *mission critical* becomes the commonly understood distinction. Being *Interactive* they both have associated response times, but in the case of a *real-time* system, failure to meet this requirement is regarded as a system failure (just as failure to compute correctly), but in a *rapid response* system, some inconvenience may be caused to those involved in the interaction, but the system is not regarded as having *failed* when response time is exceeded, unless it is excessive. Real-Time systems are also referred to as "embedded" inasmuch as the software is embedded within and interfaces primarily with hardware devices that control some process. The rapid response system primarily interacts with people.

Class 3.1.1: Real-Time

Within **Real-time (3.1.1)** there are two subclasses that provide the most significant distinguishing characteristics of real-time software, *Program Structure* and *Performance (Timing)*.

Class 3.1.1.1: Program Structure - Concurrency

This subclass represents those aspects of software structure that enable it to support either real or pseudo concurrency. This includes the simulated concurrency of single task systems and the intertask communication methods of multitasking. The lowest level of this subclass (**3.1.1.1.2.1 Coupling**) distinguishes between tight and loose coupling, which is a design concept of fundamental importance, because queueing is most sensitive to this design feature. Tightly coupled tasks, as in Ada, require that a client task be suspended until the server has processed the call. Loosely coupled tasks permit the client task to proceed immediately after the call has been placed but not necessarily processed.

Class 3.1.1.2: Performance (Timing)

Performance relates to such characteristics as response time, throughput and reliability. The distinguishing characteristic of RT systems is response time. Determination of the response time of a system requires that atomic units of activity be identified that are contribute to the response time. One model for viewing these activities is the Petri Net

[39]. When translated to RT software this atomic unit of activity becomes the *task*. A task is a program unit that the operating system views as having a certain kind of independence in the time domain. That is, it can run in "parallel" with other tasks. If the hardware has mutliple processors then it can physically run in parallel; otherwise, the operating system simulates parallelism by time-sharing the single processor. Even in a multiprocessor configuration multiple task may still be assigned to a single processor, and hence scheduled. Tasks can also be transferred from one processor to another to balance load and to enable task to meet required deadlines. These deadlines are imposed by the need to satisfy a system response time. There are five task oriented design factors that relate directly to response time:

1. *Task Execution Time:* The execution time of tasks that form a pathway between the input and output.

2. *Task Communication:* The mode of communication between tasks is sometimes referred to as *coupling*. There are two basic modes of coupling, tight and loose. Communicating tasks are designated as consumer (calling task) and server (called task). Under *tight* coupling the consumer is suspended until its call has been serviced. This means that the consumer cannot continue execution while its call is both queued and being executed for service. Under *loose* coupling the consumer task can continue to execute after the call and while it is queued and being executed by the server.

3. *Task Activation Mode:* A task is said to *activated* when it is called upon to perform service. At the time of activation it requires service of the CPU. During the course of service it may voluntarily or involuntarily give up the CPU. Such service lapses are called *suspensions*. These may occur if the task delays itself or if it calls upon the service of another application or operating system task such I/O.

   The classification subdivides activation into two types: *temporal* and *control*. *Temporal* activation refers to whether the task is activated periodically (sometimes referred to as synchronous) or sporadically (sometimes referred to as asynchronous).

   Control activation refers to a certain aspect of how the activation of a task is controlled. There are two types of control activation: *unistate* and *multistate*. A *unistate* system is one in which all tasks are eligible for activation at all times (during the course of system operation), and the periodic tasks run for all time with their assigned periodicity. A *multistate* system is one in which two or more states are identified, where each state consists of a subset of all tasks (periodic and sporadic). State changes are triggered by an external event (ie, a system input) or by an internal event (ie, the output of some currently activated task). When the system changes from state A to B, all tasks in A are terminated, and the tasks of B are then eligible for initiation. This includes both periodic and sporadic tasks. Thus, periodic tasks do not cycle and run indefinitely, as in the unistate case.

   The standard method for describing intertask communication of a unistate system

is the Dataflow Diagram (DFD) or the Buhr diagram [19,27]. The latter provides some more specific detail relating Ada constructs. Extensions to the DFD have been made by Ward/Mellor and by Hatley for describing multistate systems [17,18].

4. *Task Scheduling.* Given that two are more tasks in a single processor (CPU) are activated concurrently, a schedule must be created for assigning the processor to the tasks in such a way that each task will meet its completion deadline. These deadlines are imposed by the over-all response time required of an I/O pair that contains the given task within a path from input to output.

5. *Task Allocation.* In a multiprocessor system tasks are allocated to processors either statically, before execution begins, or dynamically during execution, or both.

2.5 Non Real-Time Systems

From the perspective of this classification the software world is divided into three types as revealed by the subclassification:

3.1 Interactive
    3.1.1 Real-time
    3.1.2 Rapid Response
3.2 Non Interactive

As described above, the distinction is based upon the repetitive nature of the activity sequence *Input-Process-Output* and the criticality of the Input to Output response time. Table 1 shows no methodology and literature citation within subclasses 3.1.2 and 3.2, because these are not the focus of this research, though it is also possible that there is no distinguishing methodolgy that is relevant to these system types within the *Operational Mode* context. Other parts of the classification would carry the relevant methods. For example, Class 2.2, **External Interfaces - People**, is of primary importance to Class 3.1.2, **Rapid Response**. It is here that the interactive person/machine dialogue would be developed, which is the important distinguishing characterization of **Interactive -Rapid Response** systems, just as the **Operational Mode** breakdown shown in Table 1 is the important distinguishing characterization of **Interactive - Real-time** systems.

2.6 Common Characterizations of all Software Systems

All software systems are based upon a set of functional requirements, which, broadly speaking, are instructions for transforming inputs to outputs. Implicit also is an existing state of the system, characterized by a set of internally and/or externally stored data structures, that can condition the transformation and can also be modified by the inputs or intermediates as part of the process. The software system can be viewed as a finite state machine of enormous proportion. An input moves the system from one state to another with the possible production of an output. Two mechanisms are thus required to effect this.

One is an *algorithm* that processes the input along with other state data to determine the next state. The other is memory in the form of *data structure* that records the current state. Figure 1 illustrates these dual characteristics as Classes 4 through 7.

This final part of the classification follows a fairly well-accepted paradigm. The higher level (Class 4) is a decomposition of the total system into processes and a specification of the lines of data flow among them. In some cases control flow, which establishes process sequences, may also be specified. The process is a somewhat generic unit, having no formal programmatic definition, but at some level of the decomposition it becomes a *program* or *task* (in the case of multitasking) and then, at the lowest level of decomposition, a *subprogram* (procedure, subroutine, etc.).

Program Structure (Class 5) is the *modular* decomposition of a program or task, which in current practice can assume two forms. One is the older and more conventional functional decomposition approach in which the modules are subprograms (also called procedures or subroutines). The decomposition can be represented as a tree that is referred to either as a Call Tree or a Structure Diagram. The other, newer type of decomposition is the Object Oriented approach, in which the modules are called *objects* or *data abstractions*. These are encapsulations of data structure and procedures that operate only on these data structures. The objects communicate with one another via *messages* (analogous to the subprogram calls), but the program decomposition in this case would appear as a network, with certain other graphical notations to designate creation of multiple objects from the same object type or data abstraction.

There arises here a problem that has yet to be fully addressed in the literature. The result of the Class 4 and 5 characterizations is a rather complex set of control and other structural relationships among the various "modular" components. **Control and Packaging (5.2)**, is the designated subclass for these characterizations. *Control* refers to the transfer of program control within a processor, while *packaging* refers to structural groups with a developmental or design rationale but do not directly imply transfer of control. For example, a package may contain a set of I/O procedures or a queue and its associated maintenance procedures.

These characterizations pertain primarily to the design stage of the life cycle. It would be of interest to represent in a single, easy to read and maintain document the following design elements that are directly related to program and packaging control within the context of Class 5:

- Subprogram call structure
  - Unconditional
  - Exclusive OR
  - In a sequence
  - In an iterative cycle
- Task entry structure (In Ada terms)
  - Calling subprogram
  - Called entry
  - Calling subprogram *select* structure
  - Called task *select* structure
- Task initiate, delay, terminate

- Exception block
- Package structure

At present two tools cover many but not all of these relationships: The Buhr Diagrams, which apply to Classes 4 and 5, and the Structure Diagrams or Call Tree (See Table 1). However, separately, they do not provide the designer with a sufficient grasp of all this information, which is necessary to comprehend the entire control and packaging picture of the design. The answer may simply be to combine the two types of graphics hierarchically, with the Call Tree nested within the Buhr diagram. An alternative, referred to in Table 1 as the Program Structure Notation, is a tabular format and is described in Appendix A. Its major advantage is that it can be manipulated with a word processor, contains all of the control and packaing information in one format, and can be expanded into and retained as documentation as one moves into a PDL (Class 7) and finally the code.

The **Data Structures** (Class 6) can be introduced at any level of the decomposition. They can interface, at the highest level, to the external environment; they can interface processes, programs, tasks or subprograms and can be internal to subprograms.

Finally, the subprogram of Class 5 must be algorithmically specified in a form that is here called **Functional Logic.** The conventional and predominant approach is **Procedural (7.1)**, which expresses the program logic in the form of *sequence statements* (expression computation and assignment, I/O, and subprogram calls), *branching statements* (ifthenelse, case) and *looping statements* (dowhile, dountil). The various of methods for expressing this logic are cited in Table 1.

The **Non Procedural (7.2)** logic is more recent and neither well understood nor widely used. Two such methods are cited in the Table. One of these, MODEL, is fully implemented in the sense that the MODEL Compiler translates the non procedural language (called by its developers a *specification)* into source code in one of three language: PL/1, C and Ada. MODEL has been demonstated on a wide range of applications, including database transaction systems, economic modelling, scientific calculation and real-time systems.

2.7 Software Metrics

Another possible application of the classification may be in the development of metrics for software cost estimation. For example, Albrecht in his Function Point method divides the process into two parts. In the first he creates his unadjusted function point count based upon the five processing activity characteristics of user inputs, outputs, internal files, interfacing files and inquiries. In the second part he adjusts this count via fourteen factors related to Technical Processing Complexity (TPC), such as data communications, distributed processing. on-line data entry, on-line update, reusability, etc. [40]. Symons has written a critique in which he states that the individual elements in Albrecht's TPC list should be more open ended, and he has added six more. He also views the processing activity as three elements rather than five [41]. Kemerer has performed validity tests on four methods, including Albrecht's Function Point approach and concludes that all of the systems tested would require extensive calibration within the actual environment that wanted to use it [42]. He also suprisingly found that the use of function points with and without the 14 TPC

factors did not make much difference. The Classification of Table 1 may help to identify those characteristics of software that relate most specifically to cost for a particular project in a given environment in either the processing activity or the TCP sense.

3. Performance Specification and Design of RT Systems

As presented in the Classification of Section 2.3 there are five factors that relate directly to response time.

(1) **Task execution time:** The execution time of tasks that form a pathway between the input and output.

(2) **Intertask communication:** The mode of communication between tasks. This is usually referred to as *coupling*. There are two basic modes of coupling, tight and loose. Communicating tasks are designated as consumer (calling task) and server (called task). Under *tight* coupling the consumer is suspended until its call has been serviced. This means that the consumer cannot continue execution while its call is both queued and being executed for service. Under *loose* coupling the consumer task can continue to execute after the call and while it is queued and being executed by the server.

(3) **Task activation:** There are two modes of task activation. First is whether the task occurs periodically or sporadically. Second is whether the system containing the tasks operates in a single or multiple state mode. A *unistate* system is one in which all tasks are eligible for initiation at all times (during the course of system operation), and the periodic tasks run for all time with their assigned periodicity. A *multistate* system is one in which two or more states are identified, where each state consists of a subset of all tasks (periodic and sporadic). State changes are triggered by an external event (ie, a system input) or by an internal event (ie, the output of some running task. When the system changes from state A to B, all tasks in A are terminated, and the tasks of B are then eligible for initiation. This includes both periodic and sporadic tasks. Thus, periodic tasks do not cycle and run indefinitely, as in the unistate case.

(4) **Task scheduling:** If two or more tasks are assigned to run in a single processor then the tasks must scheduled.

(5) **Task Allocation:** The allocation of tasks to processors in a multiprocessor environment. Normally, an initial allocation is made, but tasks can also be reallocated dynamically during the run.

A question arises as to whether it is useful, or even possible, to specify any of these factors as a requirement or whether they should be left entirely to the design, leaving only the response time to be specified. To answer this question one must look first at the most common method of specifying the architectural structure of the system, which is some form of dataflow diagram (DFD). By means of this diagram the required system is hierarchically decomposed into processes. Left open is the terminating level of this decomposition. Some specifiers will go deeper than others. The designer will consider each process of the DFD and determine which of five software design elements it is: System, subsystem, program, task and subprogram. The specification typically terminates the decomposition of the DFD with what the designer would classify as a

subsystem, program or task. The designer will further decompose these to the ultimate level of subprogram. Most real-time systems are characterized by multitasking. This means that at some level of the DFD decomposition, every process is mapped into at least one task, including the main program. A task is defined as that unit of processing activity that can run independently of and, if desired, in parallel with other tasks. In some cases the tasks communicate and in others they do not.
All five of the response time related factors cited above are associated with the task. Two questions then arise: (1) Should the tasks be defined in the requirements specification or the design, and (2) if in the specification, which of the five task related response time factors, cited above, would be allocated to specification and which to design.

In response to question 1, parallel operation may certainly be included in the requirement, and hence specification would include tasks in the DFD. However, the five factors represent quite different levels of technical specification. Factor 1, *task execution time*, may be determined in specification, but normally it is either done in design or not even then.

Factor 2, *task communication*, is partly determined in specification, in terms of which tasks communicate with which others and what data are communicated. The question of loose vs. tight coupling may be left to design.

Factor 3, *task activation*, is normally determined in specification. This includes both unistate vs. multistate and periodic vs. sporadic activation. In particular, unistate activation can be handled by the DFD, while multistate activation is handled by the Ward and Mellor [17] and the Hatley [18] DFD extensions.

Factors 4 and 5, *task scheduling and allocation*, are always determined in design. However, all of the design techniques and algorithms for scheduling and allocation (factors 4 and 5) relate to unistate activation. [43-57]. The most notable of this work is the Earliest Deadline (ED) strategy of Mok [52] and the guarantee test algorithms of Chetto under the ED assumption [46].

The research on this project has been directed at Factors 2 through 5 in the design phase. It has been divided into two major tasks. The first treated primarily factor 2 and secondarily factors 4 and 5. It led to the developemnt of a design technique called Event Oriented Design, in which the RT system design with communicating tasks and I/O response time specifications formally introduces the timing requirement, and the design process attempts to produce a communicating task architecture that optimally satisfies that requirement. Event Oriented Design and its associated software tool, the Embedded Software Design Simulator (ESDS) are presented in Sections 4 and 5.

The second major task treated factors 3 through 5 as applied to the multistate case. As mentioned above, these factors have been covered for the unistate case in design, and specification techniques for the multistate case are well developed, but there is no design counterpart for the unistate techniques. Sections 6 and 7 will present an approach to the design of RT systems based on factors 3 to 5 for the case of multistate systems.

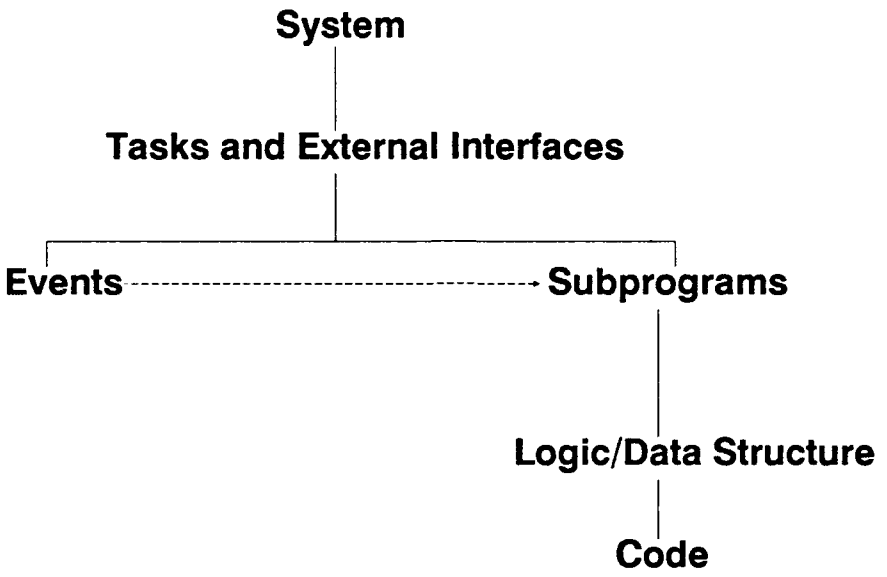
4. Design for Factor 2: Intertask Communication

In Chapter 5 of *Software Engineering with Ada* [58], Booch presents a diagram in which he depicts a *problem space* and a *solution space*. The former is the real world, with objects and operations, where objects may be velocity and frictional forces on an aircraft, and operations may be the functional relationship between the frictional force and the velocity. The solution space also contains objects and operations that map onto those of the real world, but which must be represented in a programming language. From this picture, Booch defines a programming methodolgy that he calls object oriented design. He contrasts this with other well known design methods such as top-down and data structure design. In brief, he says that top-down design concentrates on the operations and data structure design on the data objects, while object oriented design provides a more balanced treatment of the two. Booch also refers to a lack in both top-down and data structure design inasmuch as they do not take time and parallelism sufficiently into account. However, an examination of his object oriented design method also does not reveal an explicit accounting for these factors.

Current design methods and tools for the development of embedded systems in Ada follow one or a combination of the methodologies Booch refers to, such as Object Oriented, Structured, Top-Down and Data Structure design [26, 23, 59, 2]. However, these methods manipulate only two of three important factors in the design of an embedded software system, namely *objects* and *operations* on the objects. The third factor, that is not explicitly accounted for in these methods, is *time*. The reason for this omission may be historical. Conventional transaction oriented and scientific computational applications do not require that time (ie real time in the problem environment) become an explicit design factor, except in a very gross way that relates to economy or efficiency of performance. Even in the case of an interactive information system, where a desirable response time may be specified, failure to meet this specification does not usually mean system failure, only some elevated level of inconvenience to the users. In embedded systems, however, response time can be as critical as algorithmic or functional correctness, but even here other methods have sufficed up till now. Most systems in the past have used the cyclic executive method in which tasks are assigned a particular time slot in a round robin executive process. This method is relatively efficient for synchronous tasks but not for asynchronous or sporadic, multiple tasks [1]. Demands of embedded systems have now swung heavily toward the asynchronous, and Ada multitasking along with distributed processing system configurations are a direct response to this need.

Event oriented design formally introduces *time* into the design process, along with *objects* and *operations*, by means of a tool called the Embedded Software Design Simulator (ESDS). The ESDS is a software tool within a methodology called Event Oriented Design, which does not replace but rather augments one of the other, abovementioned object/operation design methods [61]. With ESDS many different designs can be quickly evaluated with respect to system response time and the location of queueing bottlenecks; in the process, timing specifications can be established for the subsequent writing of the procedural code.

Event oriented design views a system as having the hierarchic architecture shown in Figure 2.



Hierarchic View of System Architecture
By Event Oriented Design
Fig. 2

The *system* level decomposes into tasks and the input/output interfaces to external devices or other systems. Communication of data and control among the tasks is also included. The *task* level has two decompositions. On the one side each task is decomposed into *events* and their time relation. On the other side each task is decomposed into Ada *subprograms*. Probably the reason that the *event* was formerly overlooked as an important element in the system architecture is that it is not in line with the traditional hierarchy of decomposition; however, as will be discussed, it plays an important role. Also, as shown by the dashed line in the figure, it is formally incorporated into the mainline decomposition via the subprogram decomposition. The subprogram is a procedure or function. Its design consists of two parts, program logic and data structure. The program logic will become the procedural code at the next level. The data structure design specifies the individual data types and variables and the composites constructed from them in the form of arrays, records and lists.

Event Oriented Design assumes that the analysis stage of the development lifecycle has produced at least the following four types of documentation:

(1) Data Flow Diagram (DFD)

(2) Specification of algorithms required to transform DFD process inputs to outputs.

(3) Identification of all important response times of Input/Output data (signal) pairs associated with DFD processes.

(4) Specification of input loading, as signal arrival patterns. These may have a

fixed periodicity, a random occurence with some stated parametric or non parametric distribution, or may be a function of some other event, like another input or output.

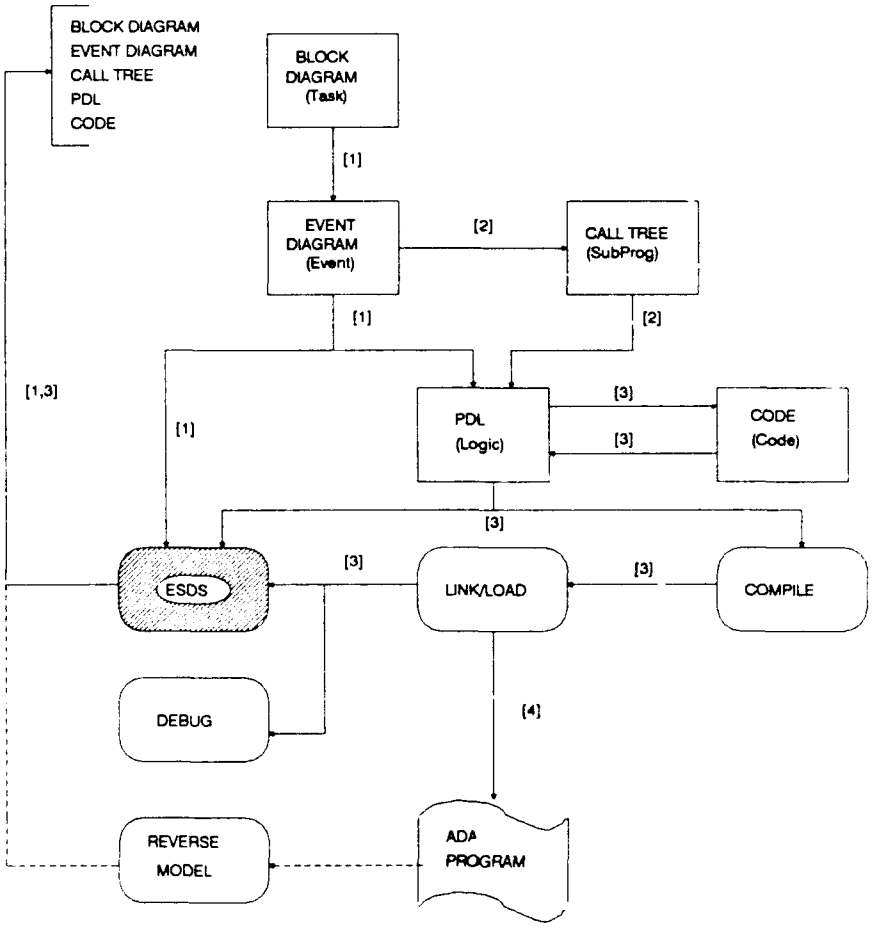
Documents (1) and (2) tend to be quite complete and accurate. Document (3) is less so, because the specifier include only critical or very important responses or may not know in advance the ultimate consequence of a given response within the system and hence may provide an estimate, to be adjusted at a later time. In other cases, a response time is arbitrarily selected and may turn out to be unnecessarily restrictive. Document (4) is even more problematic. In some cases there are standard sampling rates for sensors. In others, particularly the random inputs, one can only guess at what they might be.

Conventional design methods are addressed primarily to documents (1) and (2). Distributed processing is very much concerned with all four documents but there is as yet no systematic method for distributed design. Though event oriented design is not at present capable of serving as a general design methodology for distributed systems, it can help, because it focuses on the time constraint problem.

4.1 Event Oriented Design and the Embedded Software Design Simulator (ESDS)

Figure 3 presents a flow diagram of the methodology that supports this view. Rectangles represent diagrammatic or structured textual tools that can be implemented with graphics and word processing software. The processes associated with these boxes are intellectual and manual. Though systems are being developed that will automatically generate source code from a higher level design language that might be compared with a PDL [60,63], it is assumed here that Ada code is to be generated manually, based upon the PDL. Ellipses in the figure represent custom software for simulation, compilation and debugging. The name of the tool appears in capitals, and reference to the system architectural view of Figure 2 appears in parentheses below it.

Figure 3 presents a sequence of four steps or pathways, as indicated by the bracketed numbers. These constitute the methodology. In path [1] the system is designed at the task and event levels via BLOCK and EVENT Diagrams. This creates the ESDS or system architectural model, which involves decomposition of the system into tasks and within these, the task entries, external I/O interfaces to the entries, intertask communication via the entries, internal (non entry) processes within the task, and required response times from specified inputs to outputs. As will be described in more detail below, the Event Diagram also requires that the execution times of certain event types be specified. Thus, the combination of (1) specification of events, (2) event sequencing, (3) event execution times and (4) required input/output response times consitutes the formal introduction of *time* into the design process, as presented in the Introduction. All of these factors appear in the model produced in path [1] of Figure 3. In terms of the system architectural view of Figure 2, we are talking about the left hand branch, running from "System" to "Events". The right hand branch of Figure 2 appears in paths [2] to [4] of Figure 3. The Introduction stated that the Event Oriented design adds time to the other two design factors, objects and operations. Path [1] treats only time. Objects and operations are introduced in paths [2] and [3], and all three are combined in path [3].



Event Oriented Design with the ESDS
Fig. 3

Path [1] splits after the Event Diagram. One branch (to the left in the diagram) establishes the model as a database from which the ESDS simulation program can be run. The other automatically creates the skeleton of a PDL (Program Design Language), also as a database. The ESDS simulation provides the designer with expected response times and other quantitative performance information based on the architectural model, such as queue sizes and locations and task duty cycles. These enable the designer to adjust the model per the Block and Event diagrams until satisfactory performance in the simulator is achieved. The path [1] cycle may thus repeat a number of times.

When the architectural model is satisfactory, the design proceeds on path [2]. Tasks are decomposed into subprograms and their calling hierarchy, which are represented by the Call Tree and which is automatically added to the PDL database. The data interfaces between subprograms would also be specified at this time. This involves both the

specification of data structure and mode of allocation/transfer. The latter could be via parameter transfer or as global allocations.

The architecture of the system at this time can be characterized as follows: (1) The system has been decomposed into tasks. (2) All task intercommunication has been defined. This consists of the "call" from one task to another and the timing condition, if any, on the call. That is, the call entries are defined as is the select statement structure on either the calling or called side. It does not consist, at this point, of the data communication (See item 5). (3) The timing relationships between all inputs into and outputs from the system have been defined. (4) Each task has been decomposed into subprograms, and the calling relationshps among the subprograms have been defined. (5) The data that are passed between subprograms have been defined. This should also include the data passed with task calls.

State of the art editors and input graphics systems are capable of capturing the information contained in the architectural model defined by the above five items, and these can then be formally represented as a combination of textual and graphical information in the PDL, as implied by the path [2] entry into into the PDL in Figure 3.

As stated, event oriented design recognizes three basic design elements: *time*, *data objects* and *operations* on the objects. The design at this time contains the first of the three elements and a start on the second, namely the data objects that are communicated between tasks and subprogrograms. It has also defined "operations" in a broadly conceptual way as "tasks" and "major functions" (ie, the subprograms). What remains is to complete the definition of all data objects internal to the subprograms and the operations on them (as well as on the communicated objects) via the detailed program logic. These remaining design functions represent the conclusion of path [2]. They are purely intellectual and manual processes and are added to the PDL by means of a standard word processing text editor, though there are systems that establish conventions that result in valuable documentation reports [35]. Any of the accepted methods of design may apply here -- object oriented, structured or data structure design. Thus, event oriented design is not a replacement for these other design methods but an augmentation.

When the PDL is complete, the Ada code can be written in path [3]. This, again, is a manual process, until such time as it can be effectively automated, though, as mentioned, there will still be a higher level language in which the program must be written. The ESDS will accept a partial path [3] input on a subprogram basis. When the ESDS runs it will simulate the entire system based on the path [1] model but will actually execute any compiled and linked subprograms. The simulation run time used for these subprograms is still that in the original model. The execution of these subprogram stubs within the ESDS has two purposes. One is to obtain actual timing, which is logged into the model database for comparison with the original estimate or specification, and which can later be substituted, if the user desires. The second purpose is to begin to obtain actual data values and to track data flow along with timing. This, of course, requires that appropriate data values be available to the subprogram when it is called. The user must therefore develop a strategy via path [3] cycling that fills in the subprogram stubs and provides data input in a sequence that tests more critical paths first and less critical ones later. The pure time-related cycling of path [1] aids in identifying these critical paths. A standard debugger can also be linked to the run to find the ordinary

computational bugs in the newly linked subprograms.

The use of the ESDS in this way can be regarded as a new kind of debugging tool, which simultaneously provides the system developers (designers and programmers) with both data value and system timing information, in either an interactive or complete run mode. As each new set of stubs is added the entire simulation can be run with ESDS reports fed back to the user. As shown in Figure 3, this feedback may necessitate changes at any of five levels of the system development. Normally they are hierarchic, in that CODE corrections are made first, then PDL, then CALL TREE, etc. in order to limit the impact of these changes on the total design. For example, if a certain response time cannot be achieved, and through the simulator the problem is traced to the execution time of a particular subprogram, then the most desirable correction would be to increase the speed of the subprogram through recoding. If this cannot be achieved, then one could look to a higher design level change.

When all of the path [1] subprogram stubs have been coded via path [3], and the development team is satisfied, based upon ESDS reports, that the requirements have been satisfied, an operational Ada Program can be produced as step [4] that should be subjected to system tests, as would normally be the case in a standard life cycle sequence.

In terms of the design-implementation life cycle for a sizable system, one might find that time spent in path [1] would be measured in weeks to months, and time spent in paths [2] through [4] would be in months to a year or more.

The main purpose of path [1] is to provide the designer with a sense of assurance that required response times can be achieved by the architecture thus far designed at the task and event levels, through the specification of required execution times for various subprograms or subprogram legs to be implemented at the lowest level.

The purpose of path [3] is to combine the timing and performance information of path [1] with data flow and values and the recording of actual execution times of the executable subprograms. What the designer and programmer can see during path [3] iterations, that cannot be seen by normal debugging procedures, is the superimposition of the time or event related information onto data flow information. The event related information is characterized by queues, queuing statistics, derived efficiency numbers applied to tasks and I/O program paths, and task priorities. The data flow information is characterized by the identification of key data elements and their values as they move, in time, through the system. Their movement can be tracked in terms of points of input (tasks/entries), subprogram calls, intertask calls (and attendent queues), and outputs.

The main contribution to reliability, as well as cost control, is that the original performance and algorithmic specifications are continuously monitored throughout the design and implementation stages, rather than being determined after implementation in system test. This is not to say that the system test is no longer required. The simulator can only approximate the environment of the system, and execution times for the path [1] stubs are also approximate. Hence, the results of the ESDS runs and analysis must be regarded as indicative, not precise, and although one may have a greater expectation that the final system (after all path [3] iterations are complete) will meet original specifications, there must still follow a vigorous system test. Another advantage is that

new response specifications can be tested by simply replacing actual (path [3]) code with more tightly specified path [1] stubs and re-running the ESDS simulator. If the new specs are met, then the stubs can be recoded to meet the tighter specifications. If this is not possible, then other design modification options can be used.

Finally, there is a pathway shown at the bottom of Figure 3 that presents some interesting though speculative possibilities for future investigation. It is called "Reverse Modeling". In the figure it is represented by the dashed line path from the ADA PROGRAM block back to the BLOCK and EVENT Diagrams. Its purpose is to reconstruct, automatically, the Event Oriented model from the Ada source code. This would have two applications. For existing programs, that had not been developed under this method, the ESDS simulator could be used to analyze performance under many different I/O conditions. This would facilitate system testing by finding the potential bottlenecks via the ESDS first, which then enable system tests to be performed in a more directed and efficient way. The second application is for systems that have been designed under this method. In this case, a forward model will exist via path [1]. It would be possible, after all path [3] programming is complete, to compare the forward and reverse models, quantitatively (i.e., with respect to response times, queues, task efficiencies, etc.), to determine whether the coding has correctly implemented the original specifications, insofar as they are reflected in the forward model.

4.2 Steps in the Event Oriented Design Method

Step 1: Using the DFD and Algorithms specification documents, decompose the system or subsystem into tasks, where each task consists of processes that can be performed in parallel with those of another task.

Step 2: Design the task interfaces.

(1) Specify task entries and intertask call relations.

(2) Design the data structures that communicate data objects between tasks. Decide whether these data objects are to be communicated by parameter passage or by a commonly accessed package.

(3) Design the *select* structure on the called side. This includes (a) alternative entries, (b) *when* guards on entries, (c) *else* alternative processing, (d) *delay* alternative processing, (e) *terminate* alternative.

(4) Design the *select* structure on the calling side. This includes (a) unconditional call, (b) conditional call (untimed), (c) timed conditional call.

Step 3: Represent Steps 1 and 2 as an Event Diagram.[1]

For every I/O pair in the *Response Times* specification document, there will exist a path in the Event Diagram. All type 1, 2 and 3 events must have an execution time assigned. Reference 2 contains a technical description of the ESDS and event types. Initial execution time assignments are made for these three event types such that the total I/O

response time for any pair is less than or equal to the required time specified in the *Response Times* document. These assignments are made with some understanding of the complexity of the processing involved in each event, but can only be approximate at first. In general, the more slack between the minimum and required times, the more loading (ie higher frequency of arrival for inputs) that is possible. Any response time that exceeds the required time is called an *I/O failure* by the ESDS.

Step 4: Run the ESDS with the inputs specified by the *Loading* (Input Arrivals) specification document. If the response times cannot be satisfied, then consider the following design and/or specification changes:

(1) Modify task priorities in Step 3.

(2) Modify event execution times in Step 3.

(3) Recompose tasks in Step 1 and then repeat Steps 2 and 3.

(4) Increase the number of processors and/or re-allocate tasks to processors.

(5) Increase selected I/O response time specifications.

(6) Decrease selected loading specifications. Options (1) to (3) lie with the software designer; option (4) requires hardware configuration design modification, while options (5) and (6) require concurrence of the specifier. There may also be hardware/software tradeoffs involved in option (2).

Step 4 is an iterative process that either converges on a solution or concludes that a single system cannot meet the requirements.

Step 5: Design the internal program architecture of each task using any of the conventional design methods. This involves the specification of each subprogram, the calling structure (usually represented as a tree) among the subprograms, and the packaging overlays for tasks, subprograms and data. If there arises a conflict between the program structure imposed by the Event Diagram of Step 3 and the conventional design method (which is only looking at the functional specification (documents (1) and (2)), then one must either adjust the design to suit the Event Diagram or repeat Step 4 if the Event Diagram is to be altered.

Step 6: Design the procedural logic and all data structures not designed in Step 2 for each subprogram of Step 5.

Step 7: Code and unit test system, using the event execution time specifications of Step 3 as a guide. If possible, obtain actual event execution times and to the extent possible, bring the actual code and the event timing specification into agreement.

Step 8: Perform integrated test. Use actual event execution times in model of Step 4. Re-run the ESDS and compare results with those from integrated test. If the actual system does not satisfy specification documents (3) and (4), then find the bottlenecks with the ESDS. Resolve the problem by one or a combination of the following options:

(1) Tighten the code of Step 7.

(2) Redesign via any of Steps 6 back through 1, starting with 6.

(3) Respecify via any of documents (4) back through (1), starting with (4). Note that it may be possible to respecify an algorithm in document (2) that will result in faster event execution times. The ESDS will pinpoint the exact events that are causing bottlenecks.

An example of the application of Event Oriented Design is presented in Appendix B.

5. The Embedded Software Design Simulator (ESDS)

5.1 Rationale for the ESDS

There are basically two ways to construct a system simulator. One is by a queueing theory analysis of the total system that then represents the system as a set of equations [63]. This is sometimes called the analytic method. The other way is to characterize the system as a set of events occuring in sequence or parallel, with specified durations and time occurence relations, one to another. This is usually called discrete event simulation [64]. The ESDS is the latter. The advantage of the former is that it usually runs faster. Its disadvantage is that it is a statistical view of the total system and is subject to whatever approximating factors are built into the particular formulas that are used. Hence, it tends to be less accurate in its result, and it's more difficult, if not impossible, to pinpoint problems, like bottlenecks, within the system, because it does not explicity identify discrete system components. The discrete event simulation, on the other hand is almost the converse. It generally runs slower but has the ability to control accuracy through the level of discrete detail that it chooses to use. It essentially creates an analog of the system by representing each component (task, task entry, subprogram, file, etc.) as a discrete element. Hence it can identify problem spots in the direct terms of this analog. Also, as will subsequently be explained, the user can interact directly with the simulation, because there is a simulation clock that "ticks" off the events as they occur. This cannot be done with analytic simulation, because all of the equations must be solved before anything is known about the system's performance.

The choice of discrete event simulation and its assumption of accuracy in the ESDS is essentially intuitive, being based upon the above considerations. The proof will reside in running tests and comparisons with real systems, which is ultimately a part of the plan. However, one always has the advantage that if sufficient accuracy is not achieved for a given model, then refinement of one or more of the discrete components of the simulation should improve the results.

Another accuracy problem faced by the user of the ESDS is how to assign the proper execution time to the various events. For example, assume that a particular event is a task entry that implements a signal processing algorithm. How is the designer to know the execution time of this algorithm? The answer is that he must make an estimate, based either upon experience with such an algorithm running on the intended target, or that he would establish a given time as a specification for the programmer to meet, where this value might be arrived at through an iterative process of actually running the model and tuning it for best overall performance. It should also be noted that the computer on which the ESDS runs need not be the actual target computer, because real problem time is in no way related to the clock speed of the ESDS computer or to the time that it takes the simulator to run. Real problem time is simply represented by a program variable in the ESDS. If the programmer cannot subsequently meet this specification, then the ESDS can be run again with the actual and then known time. If response time and performance for all other functions of the system are still intact, then the new time can be accepted. If not, then either design or specification must be changed, or the programmer might be asked to try harder.

The point here is that under existing design methodologies, system response times are

not determined until after all programming is complete and system tests are performed, when the cost of change is the highest. The ESDS reduces the chance of such problems occuring after coding is complete, or, at the very least, provides the programmers with execution time targets for their code to meet. Furthermore, it can quickly test the impact of an execution time variance in one part of the system on the operation and response times throughout the rest of the system. This requires time consuming, and usually incomplete combinatorial testing in a system test.

This is not to say that one can do away with thorough system testing. The ESDS is still only an approximate analog of the actual system. However, the hypothesis presented in this proposal is that use of the ESDS will reduce the number of response time problems found in the system test, and can even direct the system test to highly stressed areas of the system, which might be places where problems are more likely to develop.

As for reliability and cost control, these are based upon (1) the ability to obtain approximate response time information prior to coding, (2) developing the optimal system architecture with respect to intertask communication, functional task composition and subprogram structure based on this information and, again, prior to coding, and (3) implementing and testing the Ada procedural code incrementally so as to assure continued compliance with the performance and functional specifications as the work progresses rather than at the end. Maintenance is also enhanced, because the ESDS simulator can be applied to check system response times whenever new or modified modules are introduced.

5.2 The ESDS System

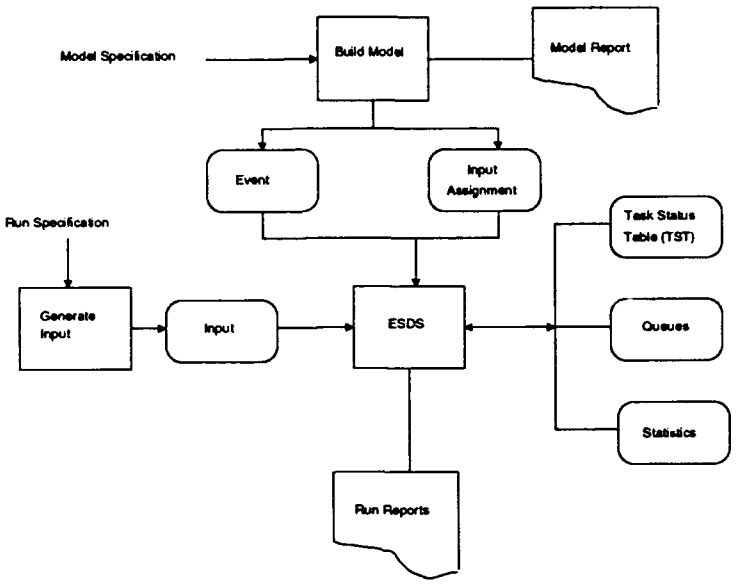
The ESDS is a system consisting of three separate programs that interface via files, as illustrated in Figure 4.

BUILD_MODEL builds the model by generating two files. The EVENT file contains the events and their sequence relationships. The INPUT_ASSIGNMENT file contains a description of each input to the program in terms of the event (or events) that it triggers and the output(s) that result. BUILD_MODEL also produces a report that documents the model.

GEN_INPUT creates an INPUT file containing a series of inputs and the time at which each is to occur during the run of the simulation. Inputs can be generated in two modes: *manual* and *automatic*. The *manual* mode represents a single input that the user enters with its time of occurence. The *automatic* mode enables the user to specify parameters to a generator (procedure) that will automatically create inputs according to a selected generator algorithm. Two algorithms currently exist; others can be added to satisfy special requirements. One is to create an input of a particular type periodically, where the type and *period* are entered as parameters. The second is to generate *n* inputs randomly, where a *type*, *period*, and *n* are given and where the n inputs are generated at random times within each interval.

Finally, the ESDS program runs the simulation. It uses three files internally: the Task Status Table tracks the status of each task; the Queue File maintains all of the necessary task queues, and the Statistics File maintains the simulation run time statistics for the

reports. The ESDS produces the various reports presented in Section 5.4.



The ESDS System
Fig. 4

5.3 ESDS Inputs

The input to the ESDS has been referred to as the architectural model of the system, in Section 2 above. In summary it contains:

(1) A decomposition of the system into tasks.

(2) All task entries.

(3) An enumeration and explicit sequencing of all events via the Event Diagram.

(4) The required or desired execution times of each event in the Event Diagram.

(5) Identification of each input signal to the simulated system, its type, the system outputs that it triggers, and the desired response time of each input-output pair. The types of input, as described above are manual and automatic.

As described in Section 3, an Event Diagram is produced for each task. Events relate directly to certain Ada commands, such as select, accept, delay, new, terminate and calls to subprograms or task entries. They can also relate to blocks of Ada code. There are nine event types associated with the method; they enable the designer to express parallel

operation and intertask synchronization and communication in terms that are congruent with Ada program structure. In this way the design principles inherent to the Ada language are introduced early in the design process. This has several benefits, while in no way restricting the flexibility or freedom of action of the designer. First, it improves communication between designer and programmer and enhances total system documentation by providing a smoother transition from Block Diagram through to code. Second, it creates an "Ada" design, as opposed to an independent design that is then recast into an Ada structure. One could argue against this approach if the implementation language were to be decided after the design, but such is not the case here. Third, the concept of concurrency and its effect on timimg and system performance is introduced explicitly into the design process. And fourth, because it bears a direct relation to the form in which systems are modeled by simulators, the Event Diagram can be used as a direct input to the Embedded Software Design Simulator (ESDS).

The nine event types are:

1. Task initiation
2. Task entry
3. Sequential processing
4. Call or interrupt to a task entry
5. End of entry
6. End of task
7. Rendezvous
8. Delay (non select)
9. Delay (select)

The above information specifies the model. Each run of the simulator, then expects an input file containing all of the manual inputs of the run, along with their arrival times. The automatic inputs are automatically generated as the run proceeds.

5.4 ESDS Outputs

The ESDS provides the designer with a high degree of precision in the operational analysis of the simulated program. First, he/she can determine which input-output pairs do not respond as required. Second, he/she can determine which task, task entry or event is causing the bottleneck. The reasons may be low task priority or excessive execution time of critical events. A variety of solutions may then be considered. One is to change relative task priorities. Another is to restructure tasks by separating non-critical (i.e. low priority) from critical (high priority) subprograms. Another is to reduce execution time of critical events. Another is to consider whether an unconditional task call can be changed to a conditional call. This would have the effect of not performing a certain task at a given time if the wait time were too long, which in turn has the effect of reducing queuing on that task. Alternatively, there may be hardware remedies to consider, such as additional processors in a multi or distributed processor configuration or a faster processor.

The ESDS also provides efficiency measures for each task or event. These are numbers between 0 and 1, where 1 is maximum efficiency and 0 would mean "locked out". These

numbers can be used as a general measure of design efficiency so that even if all response times were acceptable, one might still want to further optimize the system's performance by improving the efficiencies of selected tasks or events. The purpose of this would be to provide a greater safety margin in the event that future requirements change.

The ESDS provides these capabilities through a hierarchic set of reports that enable the designer to focus attention at the *system, task* and *event* levels. The report may be printed after the run or viewed interactively on the screen. The types of information contained in these reports are given in Table 2, followed by a description of how these reports are constructed so as to give the user a "zoom" lens effect down through the three levels of the hierarchy.

TABLE 2
Information in the ESDS Reports

DATA ITEM	MEANING
Task Efficiency	Tr = time during which the task can run but is not executing because processor is unavailable. Te = time during which task is executing. Task Efficiency = Te/(Tr + Te)
Task Duty Cycle Tr, Te, Td, Tz, Tc	Td = time during which task is suspended in a delay. Tz = time during which task is suspended at a rendezvous. Tc = time during which task is suspended in a call to another task. Task Duty Cycle = (Te + Tr)/(Te+Tr+Td+Tz+Tc)
I/O Response Time Avg I/O Response Time	I/O Response time is the time between a given input and one of its required outputs. The average is over all occurences.
I/O Efficiency Response Elasticity (Individual and Avg) A, M, R	A = Actual response time M = Minimum response time R = Required response time I/O Efficiency = M/A Response Elasticity = (R - A)/R
I/O Failures	Number of times that A > R
Task priority	Current task priority. The original priority can change temporarily if the calling task or interrupt has a higher priority.
Event Efficiency	Same as Task Efficiency, where Tr and Te are the times that the event is in the run and execute states, respectively.

TABLE 2 (Cont'd)

Queue location	Task/entry of the queue.
Queue size	Size of a queue at an entry.
Queue size/task	Total of queue sizes for all entries in task.
Time weighted queue size	Time weighted average queue size.
Time weighted queue size/task	Time weighted average queue size per task (over all entries in task).
Actual event time	The time from beginning to end of an event.
Minimum event time	Time of event if it were always in an execute state.
Variable value	The value of a given variable at a given time.

5.4.1 Report Generation

There are three types of reports:

1. Interactive reports during the course of a run.
2. Time plotted reports at the end of a run.
3. Multi-run comparisons.

5.4.1.1 Interactive Reports

The ESDS generates a series of interactive reports, each having greater detail. The values in the reports are updated every time the simulation clock changes. This clock will change with the completion of any event, and it advances by the time duration of the event. At such a time, the user can call for the display of any of the reports. At the end of the run, the user can obtain a hard copy "transcript" of any report, where all time change displays are printed.

The user also has two modes of control over the report displays as the simulation is running. One is called "stop frame", in which the simulation stops when the clock changes, and the selected report is displayed with current values. The other is called "continuous run", in which the simulation continues to run, at a wall clock speed set by the user. The report remains on the screen, and its values are changed with each clock change. In this mode the user can watch the values change on the screen and can interrupt the simulation at any time to change its wall clock speed, the display or the mode.

Table 3 presents the information content of reports at each of the three current system architectural levels plus the intended future procedural code level.

TABLE 3
ESDS Reports

LEVEL	REPORT CONTENT
System	1. Current time 2. Average task efficiency 3. Average task duty cycle 4. Per I/O pair: 4.1 I/O pair (identification of input and output) 4.2 Min response time 4.3 Avg actual response time 4.4 Required response time 4.5 No. of I/O failures 4.6 Avg I/O efficiency 4.7 Avg response elasticity 5. [Optional] Per selected I/O pairs: 5.1 I/O pair (identification of input and output) 5.2 Trace of intertask control path from input to output with current position of control, and Queue size/task
Task	1. Current time 2. Task number and name 3. Priority 4. Efficiency 5. Duty Cycle 6. Tr, Te, Tz, Td, Tc 7. Queue size/task 8. Time wtd. queue size/task
Event	
Event Report	1. Current time 2. Task number and name 3. Event number (On Event Diagram) 4. Event type 5. Minimum event time 6. Avg actual event time 7. Efficiency 8. Queue size 9. Time wtd. queue size
Queue Report	1. Current time 2. Queue location 3. Queue size 4. Time wtd. queue size

Procedural Code 1. Current time
2. Task No. and name
3. Event No.
4. Variable value
5. Output (Optional)
5.1 Flag
5.2 Input No.
5.3 Input value (Optional)
5.4 Actual response time
5.5 Required response time

5.4.1.2 Time Plot Reports

At the end of each run, the following data items can be tabulated or plotted by time.

1. Queue size per task/entry
2. Priority per task
3. Value of a program variable
4. I/O failures
5. I/O efficiency

5.4.1.3 Multi-Run Comparisons

Comparisons of different data items, in the form of the time plots shown above can be made for the same model with different input arrival patterns or for different models with the same arrival pattern.

5.5 Design Strategy using the ESDS

The designer can control the run and his method of analysis by means of the focusing mechanism of the reports and the two modes of interaction. For example, he could start in the continuous run mode at slow speed (i.e., the selected report will change on the screen continuously but slowly. The first report should be at the highest level, *system*. (See Table 3 above). As the run progresses, the designer will see the time change, the average efficiency and duty cycle of each task (through the current time), and an array of performance data on each of the I/O pairs, which basically will indicate whether the required response times are being met. The designer can note the times at which unusual or undesirable things happen. He can also stop the run at any time, change report and/or mode. That is, he could switch to stop frame mode and to a more detailed report. He can also restart the simulation and run at high speed up to a specified time and then continue with any desired mode and report. The situation is analogous to a movie projector with variable forward speed and stop frame motion, but, in addition, the user can change the viewing level of detail as well. If more detailed information were desired on the task, then the *task* report could be selected, in which case, for selected tasks, the designer would see the current time, priority at that time, efficiency, duty cycle, the amount of time spent thus far in the various states (running, executing, at rendezvous, etc.), current queues, and time weighted queue average through the current time.

If still more detailed information were desired for selected tasks, the third level, event, could be viewed via the *event* and *queue* reports.
The designer would be looking for the bottlenecks that are causing specific I/O responses to fail. These causes are traceable to particular tasks and events within the task. There are a number of design decisions that can be modified as a result of the analysis:

(1) The number of processors.
(2) Task composition and functions
    2.1 Number and function of entries
    2.2 Distributed vs. centralized processing
(3) Task priority
(4) Task allocation: static vs. dynamic
(5) Task call protocol: conditional vs. unconditional
(6) Estimated (or specified) event execution time. (This may also imply an algorithm.)
(7) Implication of hardware vs. software by items (2) and (6).
(8) Input pattern (arrival distribution and times)

For a total view of the run, the designer can print any of the four time plots which will correlate queues, task priorities, I/O failures and I/O efficiencies with time and with each other. The designer can also print the interactive reports as well, so that the detail can be examined along with the time plots.

Multi-run comparisons can then be made for different input patterns and different models. These will enable the same report data elements to be simultaneously displayed for different runs and models.

6. Design for Factors 3 and 4: Multistate Task Activation and Scheduling

The objective of this project task is to apply and extend existing scheduling and allocation algorithms to mutistate systems design. Section 6 extends Chetto's guarantee test algorithms to the multistate case. Section 7 describes a method of dynamic task migration (re-allocation) that is applicable to the multistate case and is used when the extended guarantee test algorithm fails for a particular task in its originally or currently assigned processor. Also explored is pre-emptive re-allocation, that would be applied before failure.

6.1 Problem Statement

6.1.1 Hardware Environment

The hardware environment under study is a distributed one. It consists of N nodes. Each node has one processor. Each node can communicate with any other node directly.

6.1.2 Software Environment

The software environment consists of a set of tasks. A task is the basic schedulable software unit. The following characteristics of a task are essential for the research.

6.1.2.1 Task Types

Tasks can be classified into two types, periodic and sporadic. Periodic tasks are defined by $T = \{T_i(s_i, c_i, r_i, p_i), i = 1,2,...,n\}$, where

- s_i : Initial request time of task T_i.
- c_i : Computation time for each request of task T_i.
- r_i : Required response time from time of request for task T_i. After a request is made ar_i is used to denote the absolute response deadline, which equals the time when the request is made plus r_i.
- p_i : Period of task T_i.

Sporadic tasks are defined by $S = \{S_i(c_i, d_i), i=1,...,m\}$, where

- c_i : Computation time for each request of task S_i.
- d_i : Required response time from time of request for task S_i. After a request is made ad_i is used to denote the absolute response deadline, which equals the time when the request is made plus d_i.

6.1.2.2. Task Allocation to Processors (Nodes)

Each task is assigned initially to a particular node, which will be called the *home node* of the task. If a task can only be executed at its home node then it is said that the task is assigned to the home node {\it statically}. If a task can also be executed at a remote node then it is said that the task is assigned to the home node dynamically.

(1) Every periodic task is assigned to a node statically.

(2) Every sporadic task is assigned to a node dynamically.

6.1.3. System States

At run time each node can be viewed as a finite automaton. Each node will have states, inputs and outputs.

(1) Each node has one or more *states*. It will assume an initial predefined state.

(2) The *inputs* of an automaton are the execution of sporadic tasks with this node as home node. Each periodic task has a status. It is *active* if it is allowed to be scheduled. Otherwise, it is *inactive*. A periodic task is *activated* if its status changes from inactive to active, or *deactivated* if its status changes from active to inactive.

(3) The *ouputs* of an automaton are the commands which change some periodic task's status within the same node; i.e., those tasks activated or deactivated by the output. It will be assumed that tasks are activated or deactivated only by the execution of a sporadic task. Note, in the automaton the execution of a sporadic task plays the input role while the output only affects periodic tasks.

6.2 Definitions

active
A task is said to be active if it is released and ready to be processed.

schedulable
A set of tasks is said to be schedulable if and only if there exists at least one valid schedule.

optimal
A scheduling strategy is said to be optimal if there exists at least one valid schedule for a set of tasks, and this scheduling strategy will find one of them.

request
The request of a particular sporadic task to be scheduled, or the automatic request implied by the beginning of the period of a periodic task.

ED
The *Earliest Deadline as soon as possible* scheduling strategy.

guarantee
The schedule for a set of tasks, which may include periodic tasks and/or sporadic tasks, is said to be guaranteed if and only if a valid schedule can be produced for all the tasks from current time t to infinity. When we say a task S is guaranteed it means that the task S is inserted into the current

active task set and a valid schedule produced for all the tasks from current time *t* to infinity.

ei(t)

A function $e_i(t)$ is defined for each periodic task *i*. It denotes the accumulated CPU time of the periodic task T_i up to time t, since its last request.

A *necessary condition* for T to be schedulable is

$$\Sigma\{i=1,n\}\ c_i/p_i \ leq\ 1.$$

It is obvious that total computation time requirements can not exceed CPU power.

A *sufficient condition* for T to be schedulable is

$$\Sigma\{i=1,n\}\ c_i/r_i \ leq\ 1.$$

The necessary condition is the bottom line to be satisfied. But the above sufficient condition is too conservative. In many cases even though this sufficient condition is not satisfied there are still opportunities to obtain a valid schedule. For example, let $T = \{T_i(s_i, c_i, r_i, p_i), i = 1, 2\}$, where $s_1 = 0$, $c_1 = 5$, $r_1 = 6$, $p_1 = 20$, $s_2 = 10$, $c_2 = 5$, $r_2 = 6$, $p_2 = 20$, and $\Sigma\{i=1,2\}\ c_i/p_i = 0.5$ *leq* 1, $\Sigma\{i=1,2\}\ c_i/r_i = 10/6$ *geq* 1. The necessary condition is satisfied. But the sufficient condition is not satisfied. A necessary and sufficient condition can also be obtained by constructing a schedule using certain strategy, such as ED strategy. Because the ED strategy is optimal it will certainly expose any valid schedule for the time-critical task set if there exists one. Because it is impossible to construct an infinite schedule, the remain issue is then to find the sufficient length of schedule to be constructed, which will imply the validity of infinite schedule, and how to construct it.

For a given periodic task set, T, it has been stated [55] that if {*forall* i,j} ($s_i = s_j$ *and* 1 *leq* i,j *leq* n) then it is sufficent to construct a schedule within [0, P], where P is the least common multiple of $\{p_1,...,p_n\}$, and if {*therexists* i,j} (s_i *ne* s_j *and* 1 *leq* i,j *leq* n) then it is sufficent to construct a schedule within [0, s+2P], where $s = max(s_1,...,s_n)$.

For a given node the CPU power is fixed. It is essential to determine how much CPU power is left when all supported periodic tasks are guaranteed. The term *idle time* is used to denote this unused CPU time. The *idle time* can then be assigned to sporadic tasks, which represents the maximum time available to sporadic tasks.

Let $fED^T(t)$ be the idle time function of a processor that supports the task set T using the ED scheduling strategy.

$fED^T(t)$ = *cases*

1, if t belongs to *idle time*.
0, otherwise.

Note: $fED^T(t)$ is a periodic function, $fED^T(t+p)$.

Let $\Omega(t_1, t_2)$ be a function of total processor idle time between time t_1 and t_2.
$\Omega(t_1, t_2) = \{integral(t_1,t_2)\}\ fED^T(t)dt$.

If there is no sporadic task in the system it is assumed that the CPU of the node has enough power to handle the periodic task set T.

6.3 Guarantee Test Theorems

The following subsections will discuss the guarantee test theorems which essentially will precalculate how much idle time is available and if the current active tasks, including periodic and sporadic ones, can be guaranteed.

6.3.1 A Non-Optimal Guarantee Test Theorem

At initialization time, assuming no sporadic tasks to be scheduled, the idle time within the time window [kp, (k+1)p], {*forall* k *geq* 0} can be computed by using the ED strategy. At any moment let $S = \{S_i(c_i, d_i), i=1,...,m\}$ be the current set of released sporadic tasks at the node. Let S be ordered such that i *lt* j implies ad_i *lt* ad_j. If S(c,d) is a newly occurant sporadic task and $ad = ad_h$, then the following theorem holds [56].

Theorem 1. Task S is guaranteed if {*forall* i *in* (h,...,m)}
$\Sigma\{j=1,i\}\ c_j$ *leq* $\Omega(t,ad_i)$.

Theorem 1 provides a sufficient acceptance condition. The schedule for T implied by the ED strategy at initialization
time will not be changed. Only sporadic tasks in S are tested to see if there is sufficient idle CPU time to meet their deadlines. As the predefined schedule for the periodic task set, T, will not be changed, the optimal scheduling strategy can not apply
globally to both T and S. The final schedule produced for all tasks, including both T and S, is not necessarily optimal.
A similar non-optimal guarantee test theorem has also been developed by Ramamritham and Stankovic [57].

In order to achieve an optimal schedule one should combine the two task sets and apply an optimal scheduling strategy to the combination. This is the approach that Chetto took.

6.3.2 Chetto's Optimal Guarantee Test Theorem

Let S_i denote the current sporadic task to be guaranteed and
- t : The current time.
- ad : The absolute response deadline of S_i.
- ad_m: The latest absolute response deadline of sporadic tasks in S.
- $e_i(t)$: The remaining required execution time of task i at time t.

The following lemma determines the length of the required schedule that guarantees S_i. [47]

Lemma 1. Task S is guaranteed if and only if there exists a valid schedule within [ad, ad_m+P].

The lemma reduces the issue of the guarantee of a sporadic task to the issue of the existence of a valid schedule within [ad, ad_m+P]. Consider all tasks within the time interval I = [t, ad_m+P], assuming that t *geq* s (s = max (s_i,i=1,...,n). Some of the request and computation time will be redefined for the purpose of formulating a new task set for testing. But all the absolute response deadlines of the tasks remain the same.

(1) All sporadic tasks that exist at time t are to be redefined as starting at time t. Their new computation time will be equal to their formerly remaining computation time, i.e. (r_i - e_i(t).

(2) All periodic task requests with deadline within [t, ad_m+P] can be classified into two cases. The first are those requesting earlier than t but not finished execution yet. They can be redefined as starting at time t. Their new computation time will be equal to their formerly remaining computation time, i.e. (r_i - e_i(t). The second case are those requesting later than t. They can be treated as released at their request time and their computation time remains the same.

By combining the above two cases and treating each request as a distinct task with the release time, execution time and absolute response deadline as described above, a new task can be formulated on the interval I. In order to test the existence of a valid schedule within [ad, ad_m+P] let G denote those tasks within I with deadlines greater or equal to *ad*.

$G = \{G_i(s_i, c_i, D_i), i = 1,...,N\}$
s_i : Release (request) time for each task G_i.
c_i : Computation time for each release of task G_i.
D_i : The absolute response deadline of task G_i. after the release.

By assumption, *forall* i *in* (1...,N), s_i *geq* t *and* D_i *leq* ad_m+P. *m* is the number of sporadic tasks, and N *leq* m+p, where

$$p = \Sigma\{j=1,n\}ceil((ad_m+P+ad)/p_j),$$

where n is the number of periodic tasks.

Let ad = ad_h = D_q. The optimal scheduling strategy ED can apply globally to periodic and sporadic tasks within G when constructing a schedule. The following optimal guarantee test theorem is developed based on the above formulation. It is the main result of Chetto's guarantee test theorem [46,47].

Theorem 2. Task S is guaranteed if and only if *forall* i *in* (q...N)

$$\Sigma\{j=1,i\}\ c_j\ leq\ D_i - t.$$

This theorem can derive a linear time complexity algorithm.

6.3.3 Multistate Case: An Extended Guarantee Test Theorem

6.3.3.1 Additional Assumption: Activation and Deactivation of Periodic Tasks.

The results discussed above assume that a periodic task runs for the duration of the program. In reality, one must consider systems in which periodic tasks can be activated and deactivated at arbitrary times. Ward and Mellor [17] present a method of describing systems in which tasks of any type are activated and deactivated based upon various state transitions of the system. Their method does not distinguish between periodic and sporadic tasks. For example, in a car cruise control system, when the brake is applied a sporadic task is triggered, and the fuel injection control, which is a periodic task should be deactivated. When the resume button is pressed another sporadic task is triggered, and the fuel injection task should be activated.
This section extends the guaranteed test theorems of the preceding section to this case, based on a series of additional assumptions.

Let the environment now be a distributed one. Each node has a local and identical scheduler. A set of periodic and sporadic tasks are initially preassigned to each node. With the absence of sporadic tasks, it is determined that each node has enough power to guarantee all active tasks. If a sporadic task occurs, a guarantee test algorithm will first try to guarantee it at the local node. If it cannot be so guaranteed, it will be migrated to other node where it can be guaranteed. Only when no other node in the distributed environment can guarantee is it determined that no system guarantee is possible.

Assumption 1:

The execution of a sporadic task may activate and/or deactivate some periodic tasks. The system has finite internal states. The execution of each sporadic task may cause the system to transit from one state to another. Each transition may activate and/or deactivate a subset of the periodic tasks. The current active periodic task set, those activated, will vary with the current system states.

Assumption 2:

If two or more tasks have the same absolute response deadlines the tie will be broken by a predefined rule. This results in a unique scheduling order for any pending task set.

Assumption 3:

Each periodic task will either be executed completely or not at all. Each periodic task has an initial request time s_i, and all subsequent requests will be made at $s_i + jp_i$, where $\$p_i$ is the period and j is the jth request. If a deactivation of the periodic task happens at time t, where $s_i + jp_i < t < s_i + (j+1)p_i$, then the request made in this period will be allowed to finish if the request has been partially served and will not be allowed to finish if the request has not yet been served.

Assumption 4:

Only sporadic tasks can migrate to another node for execution. If a sporadic task activates and/or deactivates a periodic task it will do so even though it is executed remotely (ie, in another node).

Assumption 5:

The activation and deactivation will happen at the absolute response deadline. This will simplify the task of scheduling construction. At run time when a task has been migrated, the local node will know when to activate and deactivate other periodic tasks by the migrated task.

6.3.3.2 Extended Guarantee Test Theorem

Because the occurence of a sporadic task can change the active periodic task set, the Chetto guarantee test theorem can not apply. Two steps are taken to extend Chetto's theorem. First is to define the length of schedule time sufficient to limit the schedule constructing process. Second is to actually construct teh schedule. Before proceeding it is necessary to define some terms.

- S_i: Denotes the ith pending sporadic task. These tasks are sorted by their deadlines.
- ad_i: Denotes the absolute deadline for S_i.
- J: Denotes the complete set of periodic tasks in the node.
- J_i: Denotes the active periodic task set after the execution of current pending sporadic task S_i.
- P_{Ji}: Denotes the least common multiple of periods for tasks in J_i.
- S: Denotes the current pending sporadic task set.

Lemma 2. Assume that a periodic task set, J, is feasible and let SC be the schedule produced by ED. A sporadic task S is inserted into the schedule with deadline at time d; let SC' denote the schedule after insertion and let t_1 be the actual completion time (t_1 *leq* d). Then for any $t_2 > t_1$, if $e_i(t_2) > e_i'(t_2)$ then there is no idle time between $[t_1, t_2]$ in SC'. (e_i is the remaining time for T_i in SC and $e_i'$is the remaining time for T_i in SC')

Proof

Because the ED strategy is used, the scheduling sequence of existing tasks will not be affected by the insertion of s.
Suppose there is idle time at t ($t_1 < t < t_2$) for SC'.
Based on ED all the tasks scheduled after t will not be affected by the insertion, so that we must have $e_i(t_2) = e_i'(t_2)$. This is contrary to the assumption; hence there must have been no idle time between $[t_1, t_2]$. *QED.*

Suppose a newly occurred sporadic task, S_i, with absolute deadline ad_i has been inserted into the set of pending sporadic tasks. Let the last of this set be denoted S_m with absolute deadline ad_m. After revising all J_i to J_m, the following lemma can be stated.

Lemma 3. Task S_i is guaranteed if and only if there exists a valid schedule within $[ad_i, ad_m + P_{Jm}]$.

Proof

(*Only if part*)
If s_i is guaranteed, according to the guarantee definition, there exists an infinite valid

schedule which implies a valid schedule within $[ad_i, ad_m+P_{Jm}]$.

(If part)
The schedule that would be produced by ED without S_i is assumed to be valid because all sporadic tasks that occurred previously
were guaranteed. This follows from the fact that S_i is scheduled after all sporadic and periodic task requests with deadline smaller than ad_i. The schedule previously produced within $[t, ad_i]$ (t denotes current time) will not be affected by the insertion of S_i and so remains valid. Therefore, a feasibility
test is only required after ad_i. From ad_m on the active periodic task set will be J_m and the least common multiple periods for J_m is P_{Jm}.

Because all pure periodic task sets are assumed to be feasible, let SC denote the schedule that would be produced for J_m by ED and let SC' denote the schedule after any delay caused by all of the sporadic tasks before ad_m. This can be made equivalent to the insertion of one large sporadic task at ad_m. Given that there is a valid schedule between $[ad_i, ad_m+P_{Jm}]$, we will show that no task requests after ad_m+P_{Jm} will miss their deadlines.

Suppose there is at least one request occurring after ad_m+P_{Jm} that misses its deadline. Denote the index of the first of these as i, occurring at time t' (note: $t' > ad_m+P_{Jm}$. Then we have $e_i(t') < e_i(t'-P_{Jm})$. But according to **Lemma 2** there must be no idle time between $[t'-P_{Jm}, t']$. The total computation time demanded between $[t'-P_{Jm}, t']$ is strictly larger than P_{Jm}, which implies that $\Sigma\{i \textit{ in } J_m\} c_i/p_i > 1$. This is contrary to the assumption that J_m is feasible. From above we know that no task will miss its deadline. Thus, there exists an infinite valid schedule, which implies S_i is guaranteed. *QED.*

If a time length δ can be reserved before time t without disturbing a valid schedule infinitely then we say available time δ is guaranteed before time t.
Corollary 1. Available time δ is guaranteed before time t if
and only if there exist a valid schedule within x

$$x = \textit{cases}\ [t, ad_m+P_{Jm}],\ \text{If } (t \textit{ leq } ad_m).\ [t, t+P_{Jm}],\ \text{If } (t > ad_m).$$

Proof

Formulate a dummy sporadic task S_i with $c_i = \delta$, $ad_i = t$. Assume that this is the newly occurred sporadic task and then follow the same proof as **Lemma 3.** *QED*

Now let us formulate a global task set for all requests made between time period [t, ad_m+P_{Jm}].

$$G = \{G_j(as_j, ac_j, D_j), j=1...,N\}.$$

where

(1) All requests made by sporadic and periodic tasks during the time window are sorted by their deadlines.
(2) For a request G_j, as_j is the absolute start time, ac_j is the absolute computation time required, D_j is the absolute response deadline for the request.

(3) N is the total number of requests remaining in the time window which includes those requests made before t but have not completed execution and those requests made after t with absolute deadline smaller than $ad_m + P_{J_m}$.

(4) For a periodic task $T_i(s_i, c_i, r_i, p_i)$

as_j = *cases*

t, if request is made before t.

$s_i + kp_i$, for some k,

if request is made between $[t, ad_m + P_{J_m}]$.

$ac_j = c_i - e_i(t)$, $D_j = s_i + kp_i + r_i$, for some i, j, k.

(5) For a sporadic task $S_i(c_i,d_i)$, $as_j = t$, $ac_j = c_i - e_i(t)$, $D_j = ad_i$.

Theorem 3. Let q be the index of request in G made by task S_i, then task S_i is guaranteed if and only if *forall* i *in* (q,...,N)

$$\Sigma\{j=1,i\}ac_j \; leq \; D_i - t. \qquad (1)$$

Proof

Only if part
Suppose there is an i *in* (q,...,N) such that (1) does not hold. This would mean that the time demanded by requests (1,...,q) is strictly larger than the available time $D_i - t$. The request i can not meet its deadline. So S_i is not guaranteed. *Contradiction.*

If part
Let i denote the request index in G which is made by S_m.
Suppose there is a request with index j that missed its deadline.
For the case j *leq* i, based on the ED strategy there must be no idle time between (t,D_j). Therefore, (1) cannot hold. *Contradiction.*

For the case j > i, all requests after i are made by pure periodic tasks in J_m because request j missed the deadline. According to **Lemma 2** there must be no idle time between (t, D_j0. Therefore (1) can not hold. *Contradiction. QED.*

7. Design for Factors 3 and 5: Multistate Task Activation and Migration (Dynamic Re-allocation)

7.1 Task Migration (Dynamic Re-allocation)

Once the guarantee test theorem has been applied to a newly presented sporadic task, there are two possible results. One is that the sporadic task can be guaranteed, in which case nothing more need be done. The other is that the sporadic task cannot be guaranteed. Then migration to another processing node must be considered. This section will investigate the issue of migration, in two parts. One is to select the task at the local node for migration; the other is acceptance of the task migration request from a remote node. The first part will be the main issue and focus in this section. The second part can be solved simply by applying the guarantee test theorem at the selected remote node.

7.2 Task Selection

If the system has no State Transition Diagram (STD) and no periodic tasks can be activated or deactivated by the execution of sporadic tasks, the selection of sporadic task for migration presents no problem. The currently occurring S will be the candidate for migration, because before S occured, all tasks supported by the node were guaranteed. If the system has an STD and the currently active periodic task can vary with change of state, the issue of sporadic task selection for migration becomes nontrivial. Execution of S may activate currently non active periodic tasks within the original node of S. Inasmuch as it is assumed that *all* periodic tasks originally assigned to a node are feasible, there at least exists a feasible solution, namely, migrate all sporadic tasks. The problem then becomes that of selecting those sporadic tasks that will in some sense maintain optimal system performance.

7.3 Criteria of Sporadic Task Selection for Migration

Four selection criteria are proposed:

(1) The least number of sporadic tasks.
(2) The least total computation time.
(3) The least time to select either criterion (1) or (2).
(4) The sporadic tasks with greatest slack time (ie, $ad_i-(r_i-e_it))-t$).

The first criterion is based on the assumption that equal effort will be spent to migrate each task regardless of its size. The second is based on the observation that minimum load should be migrated in order to maintain over-all stability (ie, minimize the probability of further migration). The third criterion is to minimize the effort of task selection and therefore reduces the overhead. The fourth criterion identifies tasks that would be more tolerant to migration delay, but it is somewhat more complex and will not be further investigated in this paper; it could be a candidate for future work.

7.4 Task Selection Theorem

As described in Section 2, a global task set G is formed before applying the guarantee test theorem.

$$G = \{G_j(as_j, ac_j, D_j), j = 1,...,N\}$$

The set G consists of all current active periodic and sporadic tasks sorted by their absolute deadlines D_i. The current sporadic task set is $S_1,...,S_m$, sorted by their absolute deadlines. The current sporadic task has an index h, ie, S_h. Each sporadic task corresponds to one task in G, i.e. $S_i = G_j$ for some i and j. The guarantee test is applied to G. In the following, when we say a test failed at k, it means that it failed at task G_k (which may either be a periodic or sporadic task. Suppose k is the smallest index value of G that fails the guarantee test of Theorem 3 (ie, $\Sigma\{j=1,k\}$ ac_j *leq* D_k - t does not hold), then we say that the test is guaranteed at k-1. The failure of the test is caused by the presence of sporadic tasks which compete with periodic tasks for CPU time. According to the ED strategy, a sporadic task with a later absolute response deadline could be scheduled before some periodic task with earlier absolute response deadline if there is any idle time before that periodic task. If a guarantee test fails at k then we say that a sporadic task is *related* to the failure if it consumes any CPU time before G_k. Otherwise it is *not related*. Let $S_1,...,S_m$ be the current pending sporadic tasks which are sorted by their absolute response deadlines (ie, ad_1 *leq* ,..., *leq* ad_m).

The following theorem states that if a guarantee test fails at k then any sporadic tasks with absolute response deadlines later than D_k will not be *related* to the test failure. The significance of this theorem is that it will narrow the range of possible candidates for migration.

Theorem 4. Suppose **Theorem 3** fails at k (ie, $\Sigma\{j=1,k\}$ ac_j *leq* D_k - t does not hold). If ad_i *leq* D_k *leq* ad_{i+1} then $S_{i+1},...,S_m$ are not related to the failure of the test at k.

Proof:
Let af_i be the actual completion time S_i and af_i *leq* ad_i. Because the ED is used and af_i *leq* D_k, there is no idle time between [t, af_i]. According to **Lemma 2** there is also no idle time between [af_i, D_k]. That is, from t to D_k the CPU is completely busy. Because the absolute response deadlines of all the sporadic tasks of $S_{i+1},...,S_m$ are after D_k they will not have a chance to consume any CPU time. So they are not related to the failure of the test at k. *QED.*

The above theorem shows how to narrow the possible *related* sporadic task candidates. For these task candidates, the following theorem will state an adequacy condition for migration to guarantee local tasks. First a lemma will be stated for resolving a single test failure.

Lemma 4. If a test is guaranteed at k-1 and fails at k by δ (i.e. $\Sigma\{j=1,k\}$ ac_j = D_k-t+δ), and if \$$ad_i$ *leq* D_k *leq* ad_{i+1} then migrating any task S_j, where 1 *leq* j *leq* i, such that δ *leq* c_j , will guarantee the test at least at k.
(Without loss of generality, S_j may be considered as several tasks combined such that their total computation time equals c_j.)

Proof:
It is obvious that migrating a task to another node will not cause the test failure before k. Suppose that after migrating some S_j, where 1 *leq* j *leq* i, such that δ *leq* c_j, the test still fails at k. According to the ED strategy and **Lemma 2** there must be no idle time between [t, D_k]. The CPU time previously consumed by S_j must be used by G_k or there must be idle time. Contradiction. *QED.*

Now suppose that a test fails at k by δ. The migration of some S_j (where $c_j = \delta$) does not guarantee that the test will not fail after k. Theorem 5 presents the migration adequacy condition for resolving multiple test failure, where a multiple test failure is defined as follows. Suppose there are p guarantee test failures, at $k_1,...,k_p$, which fail by $\delta_1,...,\delta_p$. Each value of δ_i (where 1 *leq* i *leq* p) is derived by assuming that the previous failures have been recovered by migrating certain sporadic tasks with total computation time equal to $\delta_1 + \delta_2 + ... + \delta_{i-1}$.

Theorem 5. If a guarantee test fails at $k_1,...,k_p$ by $\delta_1,...,\delta_p$ and if a sporadic task set Γ can be chosen that satisfies the following condition: *forall* i (1 *leq* i *leq* p), *therexists* a Γ_i (Γ_i *subset* Γ) with all absolute deadlines earlier than D_{k-i} and the total computation time for the sporadic tasks in Γ_i *geq* $\Sigma\{j=1,i\}$ δ_j, then after the migration of Γ, all tasks can be guaranteed at the local node.

Proof:
Consider first the failure point k_1. According to **Lemma 4**, migration of some sporadic task(s) with computation time δ_1 is sufficient to recover failure point k_1. Now consider failure point k_2. Migration of some sporadic task(s) with computation time $\delta_1 + \delta_2$. From the preceding, δ_1 is used to recover failure point k_1, and according to **Lemma 4**, δ_2 is sufficient to recover failure point k_2. By induction, the rest of the failure points can be recovered in the same way. *QED.*

7.5 Preliminary thoughts on pre-emptive strategies

Sporadic task selection is further complicated if each node is required to execute a sporadic task either completely or not at all. For example, consider three sporadic tasks, S_1, S_2, S_3, in absolute response deadline order, ie, ad_1 *leq* ad_2 *leq* ad_3. Assume that they arrive in reverse order. S_3 arrives first, partially execute, and then S_2 arrives. S_2 partially executes, and S_3 arrives. The guarantee test algorithm detects that the node is overloaded. If a sporadic task is required to be executed by a node either completely or not at all then the only candidate for migration is S_3. If the migration of S_3 cannot solve the problem then the system has to fail. On the other hand, if S_2 were premigrated before S_3 arrived, the system may survive after S_3 arrives. We define this situation where the system has more than one partially executed sporadic task as an *unsafe condition.*

A major significance of the *unsafe condition* is that it leaves little migration choice when the node is overloaded. The system may or may not fail, depending on the values of the deadlines and the δs, but this uncertain condition does limit the adaptive capability of the system. It is therefore desirable to prevent the *unsafe condition.*

7.5.1 Heuristics for Preventing System Failure Caused By *Unsafe Condition*

If there is incomplete knowledge about the occurrence of sporadic tasks then there is no way to completely prevent the *unsafe condition*; however, some observations provide ways to reduce the chances of incurring it and also of preventing the worst case of it.

Let $S_1,...,S_m$ be the current active sporadic task set, sorted by their absolute response deadlines but triggered (ie, arriving) in reverse order. $S_2,...,S_m$ are executed partially. S_1 is the current task and is the only candidate for migration if it can not be guaranteed. We define this condition as the **worst unsafe condition.**

The *worst unsafe condition* gives the least task selection choice for migration when the system is overloaded. In order to prevent potential system failure caused by this condition it should be eliminated completely if possible or prevented when it tends to develop. Several heuristics can be suggested for accomplishing this. objective.

First, we observe the fact that if all sporadic tasks are ordered by their relative response deadlines then we have $S_1, S_2,...,S_m$, which means that d_1 *leq* d_2 *leq* ... *leq* d_m. If $d_1 = d_2 = ... = d_m$ then it is easy to see that no matter in what order they arrive their absolute response deadlines will be in that order. This implies that no *unsafe condition* can develop. In this case the *unsafe condition* is completely eliminated.

This ideal suggests the following heuristics:

(1) At design time, assign those sporadic tasks that have close length of relative response deadlines to the same node.

(2) At run time, migrate some sporadic task when the *worst unsafe condition* tends to develop. How to determine the time when this precautionary migration should be performed is still a subject of our research.

7.5.2 The Request for Migration

After a candidate sporadic task is selected the local node will broadcast a request for migration to all other nodes. The request for migration will consist of the following information.

(1) *Request identifier*: denotes the node where the request originated.

(2) *Task identifier*: denotes the task to be migrated.

(3) *Computation time*: denotes how much CPU time isn needed to execute this task.

(4) *Absolute deadline*: denotes the time when the execution of the task must be complete. When an acceptance from another node is received the task is migrated immediately. If no node can accept this task the local node has to guarantee it if it can be guaranteed or to invoke failure handling if it cannot be guaranteed at the local node.

7.5.3 Acceptance of Migration

When a node receives a migration request it will first determine whether the request can be guaranteed. According to **Corollary 1, Theorem 3** can simply be used to test if the remote task can be guaranteed locally or not. If the test is positive it will return a "yes" answer to the requesting node; otherwise it will return a "no" response.

8. Data Structures and Algorithms

This chapter develops the data structures and algorithms that are used in the simulation experiments of Chapter 9. They are based on the theorems developed in Chapters 6 and 7.

In the distributed system environment, all nodes are considered logically identical and will therefore have the same data structures and algorithms.
The data structures and algorithms are presented here as high level abstractions for the purpose of description and preliminary design of the simulatotion. They will be refined and detailed as part of the work of the project.

8.1 Data Structures

(1) A set of all periodic tasks, PTASKS

Each element of PTASKS is denoted as $T_i(s_i,c_i,r_i,p_i)$ which contains initial request time (s_i), computation time (c_i), relative response time (r_i), and period (p_i).

(2) A set of active periodic tasks, A_PTASKS_i

A_PTASKS_i denotes those periodic tasks that will be activated by the current pending sporadic task S_i. It is a subset of PTASKS. A_PTASKS_0 denotes the current activated sporadic tasks.

(3) A set of all sporadic tasks, STASKS

Each element of STASKS is denoted as $S_i(c_i,d_i)$, where c_i is the computation time and d_i is the relative deadline.

(4) An array of active sporadic tasks, A_STASKS

A_STASKS represents all current pending sporadic tasks to be processed. It is a subset of STASKS. An element of A_STASKS is denoted $S_i(c_i, ad_i)$, where ad_i is the absolute deadline. The array is sorted by absolute deadlines.

(5) State Event Matrix, SEM

SEM is a two dimensional array data structure with subscripts *state* and *event*. It represents a finite state machine driven by events. The events are defined by the execution of sporadic tasks. Each entry of the array contains information of next state, and activation and deactivation of periodic tasks.

(6) An array of global tasks, G_TASKS

G_TASKS denotes those active periodic and sporadic tasks which are in the time window [tau, $ad_m + P_{Jm}$] described in Section ???. Each element of G_TASKS is denoted as $G_i(as_i, ac_i, D_i)$, where as_i, is the absolute start time, ac_i is the absolute computation time, and D_i is the absolute deadline. The array is

sorted by absolute deadlines.

(7) An array of failure points, F_POINTS

An element of F_POINTS is denoted F_i(index$_i$, Delta$_i$). It contains the index of task $G_{index\text{-}i}$(index-i), and how much it failed at that point (Delta$_i$). It is sorted by *index-i.*

(8) An array of selected tasks for migration, M_TASKS

M_TASKS represents the sporadic tasks selected for migration. An element of M_TASKS is denoted M_i(index-i), which contains the index of a sporadic task.

Two functions will be used in the algorithms.

(1) ac(G_i)

This function returns the absolute computation time of task G_i.

(2) ad(G_i)

This function returns the absolute deadline of task G_ii.

8.2. Guarantee Test Algorithm

The guarantee test algorithm is based upon Theorem 3 in Section ???. In addtion to performing the test, the algorithm will also return all failure points if the currently occurring sporadic task S cannot be guaranteed. The failure points then will be used by the task selection algorithm for migration.

Algorithm:

guarantee-test

Input:
Current, occurred sporadic task S.

Output:
If S is guaranteed then return "GUARANTEED" else return "NOT GUARANTEED" and F_POINTS.

Body:
Insert S into A_STASKS.
The A_PTASKS$_0$ is known.
Suppose there are m tasks in A_STASKS.
for i := 1 to m do
    use A_PTASKS$_{i-1}$, S$_i$, and SEM to derive A_PTASKS$_i$.
end for.
Build a G_TASKS from A_STASKS and all A_PSTASKS$_i$.

Suppose there are N tasks in G_TASKS and S = G_h.
sum := 0.
for i := 1 to h-1 do
    sum := sum + ac(G_i).
end for.
for i := h to N do
    sum := sum + ac(G_i).
    if sum > ad(G_i) then
      insert a tuple (i, sum - ad(G_i)) into F_POINTS.
      sum := ad(G_i).
    end if.
end for.
if F_POINTS is empty then return "GUARANTEED" else return "NOT GUARANTEED" and F_POINTS.

8.3. Task Selection Algorithm

The task selection algorithm uses a backward search to find those sporadic tasks with deadlines as late as possible in order to guarantee the rest of the tasks. It is a tentative version of the task selection algorithm because it does not necessarily satisfy the criteria. Further analysis is still required.

Algorithm:

task-selection

Input:
F_POINTS.

Output:
If successful then return "SUCCESS" and M_TASKS else return "NOT SUCCESS".

Body:
Because S1,...,Sm is in order, G_{S1},...,G_{Sm} is also in sorted order.
Because F1,...,Fl is in order, G_{F1},...,G_{Fl} is also in sorted order.
Suppose G_{Sk} is the one with the latest absolute deadline, but ad(G_{Sk}) .leq. ad(G_{Fl}).
sum := ac(G_{Sk}).
put k into M_TASKS.
j := k.
i := l.
loop
    if sum > ac(G_{Fi}) then
      if i = 1 then return "SUCCESS" and M_TASKS
      else
          if ad(G_{Sj}) .lt. ad(G_{Fi-1}) then
              sum := sum - ac(G_{Fi}).
              i := i - 1.

```
                else
                        i := i - 1
                        while ad(G_Sj) .gt. ad(G_Fi)
                                j := j - 1.
                                if j = 0 then return "NOT SUCCESS".
                end if.
                        end while.
                        sum := ac(G_Sj).
                        put j into M_TASKS.
                end if.
        end if.
      else
        j := j + 1.
        if j = 0 then return "NOT SUCCESS".
        sum := sum + ac(G_Sj).
        put j into M_TASKS.
      end if.
end loop.
```

References

1. L. MacLaren, "Evolving Toward Ada in Real Time Systems," *ACM Sigplan*, Vol 15, No. 11, Nov 1980.

2. M. Jackson, *Principles of Program Design*, Academic Press, 1975.

3. K. Shin, "Introduction to the Special Issue on Real-Time Systems," *IEEE Transactions on Computers* C-36.8 (Aug. 1987): 901-902.

4. C.D. Locke and D. Vogel, "Problems in Ada Runtime Task Scheduling," *Ada Letters* 7.6 (Fall 1987). International Workshop on Real-Time Ada Issues. Moretonhampstead, Devon, UK. 13-15 May 1987.

5. D. Cornhill, " Four Approaches to Partitioning ADA Programs for Execution on Distributed Targets," *Proc IEEE Computer Society Conf on Ada Applications and Environments*, 1984.

6. I. Lee and V. Gehlot, "Language Constructs for Distributed Real-Time Computing." *Proceedings, Real-Time Systems Symposium.* San Diego, Ca. 3-6 Dec. 1985. 57-66.

7. J. Stankovic J and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Transactions on Computers* C-34.12 (Dec. 1985): 1130-1143.

8. K. Heninger, J. Kallander, J. Shore and D. Parnas, "Software Requirements for the A-7E Aircraft," *NRL Memorandum Report 3876*, November, 1978

9. J. Martin, *Design of Man-Computer Dialogs*, Prentice Hall Inc., Englewood Cliffs, NJ, 1972

10. G. Andrews and F. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys* 15.1, pp 3-43, March 1983.

11. C.A.R. Hoare, "Communicating Sequential Processes," *Communications ACM* 21.8, pp. 666-677, Aug. 1978.

12. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

13. J.D. Ichbiah, *et al*, "Rationale for the Design of the Ada Programming Language," *ACM Sigplan Notices*, Vol 14, No 6, June 1979.

14. H. Gomaa, "Software Development of Real-Time Systems," *Communications ACM* 29.7 pp. 657-668 July 1986.

15. H. Gomaa, "A Software Design Method for Real-Time Systems," *Communications ACM* 27.9, pp 938-949 Sept 1984.

16. H. Simpson and K. Jackson, "MASCOT and Multiprocessor Systems," Systems Designers, Ltd., Surrey, England, Feb 4, 1983.

17. P. Ward and S. Mellor, *Structured Development for Real-Time Systems*, 3 vols, Yourdan Press, New York, NY, 1985-86.

18. D. Hatley and I Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, NY, 1988

19. T. Demarco, *Structured Analysis and System Specification*, Prentice Hall, 1979.

20. D. Wood, "Methods Evaluation Technical Report," Software Engineering Institute, Aug 11, 1989.

21. D. Harel, "Statecharts: A Visual Approach to Complex Systems," *Concurrent Systems*, Feb 1986.

22. J.F. Stay, "HIPO and Integrated Program Design," *IBM System Journal*, No. 2, 1976.

23. E. Yourdan and L. Constantine, *Structured Design*, Prentice Hall, Englewood Cliffs, NJ, 1979.

24. D.L. Parnas, "A Technique for Software Module Specification with Examples," *Comm ACM*, vol. 15, No. 5, May 1972.

25. G. Booch, "Object Oriented Development," *IEEE Trans. on Software Engineering*, vol SE-12, No. 2, Feb. 1986.

26. R. Abbott, "Program Design for Informal English Descriptions," *Comm ACM*, Vol. 26, No.11, Nov 1983.

27. R.J.A. Buhr, *et al*, "Software CAD: A Revolutionary Approach," *IEEE Trans. on Software Engineering*, vol 15, No. 3, Mar 1989.

28. P. Chen, "The Entity-Relationship Model - Toward a Unifying View of Data," *ACM Trans. on Data Base Systems*, vol.1, No. 1, Mar 76, pp 9-36.

29. J. Martin, *Computer Data-Base Organization*, Prentice Hall Inc., Englewood Cliffs, NJ, 1975

30. C.J. Date, *An Introduction to Database Systems*, 3rd Edn, Addison Wesley Pub. Co., Reading, MA 1981.

31. D. Lefkovitz, *File Structures for On-Line Systems*, Sparten Books, 1967.

32. I. Nassi I and B. Schneiderman, "Flowchart Techniques for Structured Programming," *ACM Sigplan Notices*, Aug 1973.

33. R Maaes, "On the Representation of Program Structures by Decision Tables: A Critical Assessment," *Computer Journal*, Vol 21, No. 4, 1978.

34. J.D. Warnier, *Logical Construction of Programs*, Van Nostrand Reinhold, 1974.

35. M. Gordon, "The Byron Program Development Language," *Journal of Pascal and Ada*, p24, May 1983.

36. J.V. Guttag, *et al*, "The Larch Family of Specification Languages," *IEEE Software*, Vol 2, No. 5, 1985.

37. N.S. Prywes, *et al*, "Use of a Non Procedural Specification Language and Associated Program Generator in Software Development," *ACM Trans. on Programming Languages and Systems*, Vol 1, No. 2, 1979.

38. J.R. Abrial, *The specification language Z: basic library*, Oxford Univ. Programming Research Group, 1980.

39. M. Molloy, "Performance Analysis Using stochastic Petri Nets", *IEEE Trans. on Computers*, Vol C-31(9), 913-917, Sept. 1982.

40. A. Albrecht, "Measuring Application Development Productivity," *Proc. IBM Applications Development Symposium*, Monterey, CA, Oct. 14-17.

41. C. R. Symons, "Function Point Analysis: Difficulties and Improvements," *IEEE Trans. on Software Engineering*, Vol. 14, No. 1, Jan 1988, pp 2-11.

42. C. F. Kemerer, "An Empirical Validation of Software Cost Estimation Models," *Comm of the ACM*, Vol. 30 No. 5, May 1987, pp 416-429.

43. J. Stankovic J and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Transactions on Computers* C-34.12 (Dec. 1985): 1130-1143.

44. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol 20 No. 1, pp 46-61, 1973

45. O Serlin, "Scheduling of Time Critical Processes," *Proc. Spring Joint Computer Conf.*, 1972, pp 925-932.

46. H Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Tans. on Software Engineering*, vol 15, No. 10, Oct. 1989

47. H Chetto and M. Chetto, "Scheduling Periodic and Sporadic Tasks in a Real-Time System," *Information Processing Letters*, Feb 27, 1989, pp177-184

48. J. Stankovic and K. Ramamritham, "The Design of the Spring Kernel," *Proceedings, Real-Time Systems Symposium*. San Jose, Ca. 1-3 Dec. 1987. 146-157.

49. E. Jensen, C.D. Locke and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proceedings, Real-Time Systems Symposium*. San Diego, Ca. 3-6 Dec. 1985. 112-122.

50. W. Zhao, K. Ramamritham and J. Stankovic J, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering* SE-13.5 (May 1987): 564-577.

51. C.D. Locke and D. Vogel, "Problems in Ada Runtime Task Scheduling," *Ada Letters* 7.6 (Fall 1987). International Workshop on Real-Time Ada Issues. Moretonhampstead, Devon, UK. 13-15 May 1987.

52. A. Mok, "The Design of Real-Time Programming Systems Based on Process Models," *Proceedings, Real-Time Systems Symposium*, pp 5-17, Dec. 1984.

53. C.D. Locke, "Best Effort Decision Making for Real-Time Scheduling," Dissertation, Carnegie-Mellon University, 1986. CMU-CS-86-134.

54. L. Sha, "On Priority Scheduling and Priority Inversion," *Message to Comp. Lang. Ada* (May 1988).

55. Leung, "A note on Preemptive Scheduling of Periodic Real-Time Tasks", *Information Processing Letters*, Vol. 11, No. 3, November 1980

56. Horn, "Some Simple Scheduling Algorithms", *Naval Res. Logist. Quart*, Vol.21, pp. 177-185, 1974.

57. Ramamritham, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems", *IEEE, Software*, July 1984.

58. Booch G, "Software Engineering with Ada", Benj. Cummings Publishing Co., Menlo Park, CA, 1983.

59. Orr KT, "Introducing Structured Systems Design", Infotech International,Ltd., Maidenhead, England, 1972.

60. Prywes NS, Pnueli A and Shastry S, "Use of a Non Procedural Specification Language and Associated Program Generator in Software Development", ACM Trans. on Programming languages and Systems, Vol. 1, No. 2, pp 196-217, October 1979.

61. D. Lefkovitz, "Event Oriented Design: A Description of the Methodology," Quarterly Report No. 7, InfoSoft Design, Contract No. N00014-85-C-0298, April, 1987.

62. Hamilton M and Zeldin S, "Higher Order Software -- A Methodology for Defining Software", IEEE Trans. on software Engineering, Vol. SE-2, No. 1, p9, March 1976.

63. Buzen JP, et al, "Performance Oriented Design Reference Manual", Contract No. N00039-81-C-0183, BGS Systems, Lincoln, MA, Sept. 1981.

64. Pritsker A, Kiviat P, "Simulation with GASP II", Prentice Hall, (1969).

Appendix A
Program Structure Notation

The control and packaging of modules, characterized by Class **5.2 Control and Packaging,** has become rather complex, because there are five mechanisms operating simultaneously. This means that the design process becomes a synthesis of these five mechanisms, and the implementers, testers and maintainers, who come after, must have a good map with which to follow these complex interactions. The five mechanisms are: (1) A normal subprogram call within a program or task, (2) an intertask call, (3) a task initiation, instantiation or termination, (4) an exception break within a subprogram and (4) module or data structure packaging in the Ada sense. The first four imply a transfer of program control, while the fifth is an aggregation mechanism used for design and developmental reasons. In addition to these four main mechanisms there is a series of sub control mechanisms such as *alternative, sequenced*, and *iterative* subprogram calls, and *selective* and *delayed* task entries as in Ada.

It would be desirable to have a single, convenient but definitive descriptive notation for all of these mechanisms. As discussed in the body of the paper, the closest such notation are a combination of the Buhr diagrams and the Structure Charts of Yourdan, which are variants on those devised by Jackson. They have the advantage of being graphical, which seems to aid both in the conceptualization and synthesis of the design and in its understanding by those who follow; however, they have a few drawbacks. One is that neither alone does the entire job. Second, there are limits to graphics, when the detail involved becomes too great. A textual notation is therefore offered in this Appendix that covers all of the control and packaging mechanisms at once, is expandable to enable any level of detail, can be created and maintained with a word processor, and can be integrated with other textual (or graphical) components of the system documentation, such as requirements, design and the implementation code. It is called the *Program Structure Notation.*

The method uses a canonical notation to describe a tree, the nodes of which are generically called *process elements.* The node notation will define the type of process element and its relationship to other nodes, in the sense of control and/or packaging. The node syntax is:

<prefix> <node number> <<name><[suffix]>>* <<{&}name> {<[suffix]>} > | **

where

< > is the entity symbol
{} is an optional entity
* is an entity repetition of zero or more times
[], & and * are literal symbols

[suffix]

The *suffix* is a mnemonic code that defines the process element type, as follows:

MS	Main subprogram
S	Subprogram (also the default suffix value)

T	Task
P	Package
A	Task entry (Ada *accept*)
C	Call to a task entry
E	Exception
TA	Task Activation

name

This is the name of the process element that is assigned by the designer. There is also a special dummy name designated as *. It represents a node that contains no process element but is a part of the control structure, as specified in the prefix.

The symbol & prefixed to a name means that the node is further decomposed as a separate tree. The separate tree starts with node number 1, but the actual node numbers of nodes within **$name** is the concatenation of the **&name** number and the absolute number within the **$name** tree.

Each program or task is represented by its own tree. Trees are linked by [C] and [A] type process elements, where the former is the calling task, and the latter is the entry of the called (*accept*) task. The *name* associated with both the [C] and [A] nodes is the entry name. It is analogous to the normal call of a subprogram, where the entire tree substructure from the [A] entry downward subtends the [C] process element. Similarly, the [TA] and [T] process elements specify the activation of a task [T] within a process element [TA] at some point in another tree. (Ada provides two mechanisms for activation: elaboration and declaration or dynamic allocation of a task object.)

A node of the tree may be a composite of process elements; hence, the repetition of names and suffixes. For example, it may be a subprogram [S] within a task [T] within a package [P]. A node inherits all of its immediate ancestor process element types unless they are explicitly changed.

node number

The *node number* is a canonical number, 1, 1.1, 1.1.1,..., 1.2, 1.2.1, ... etc., that represents the position of the node in the structure tree.

prefix

The [C] to [A] and [TA] to [T] connections described above denote internode relations. All of the other internode relations are specified by the *prefix*. Figure A1 presents the prototypical internode transfer of control structure, where A, B and C represent process elements of the types defined by the *suffix*. For example, if A, B and C are subprograms,

then the figure is interpreted as: "subprogram A calls subprograms B and C". If A and B are subprograms and C is an exception, then the figure is interpreted as: "subprogram A calls subprogram B and an exception C is raised and handled in A". The prefixes specify certain relationships among process elements at a given level, such as B and C. The prefixes are defined in Table A1. In the **Level Referenced** column, *immediate* means that the symbol applies to the level of the prefixed node; *next* means that it applies to all nodes at the next nested level, up to a cancel prefix, and *previous* means that it applies to the next higher level.

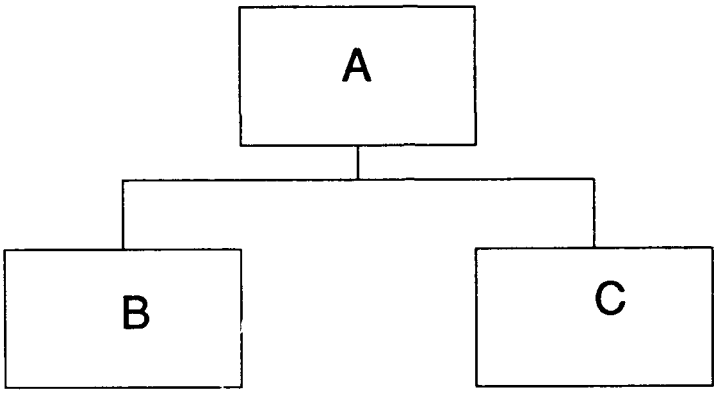


Fig. A1. Prototypical Internode Transfer of Control Structure

TABLE A1
Prefix Definitions

Prefix Symbol	Name	Level Referenced	Meaning
-	sequence	immediate	If B and C (Fig. A1) have a - prefix, they are always called from A and are performed in sequence.
+	alternative (exclusive)	next	If A has a + prefix then either B or C is called, but not both.
@	iteration	next	If A has an @ prefix then level B is performed multiple times. A common configuration would be A with an @ prefix and B and C with a - prefix.
%	alternative (inclusive)	next	If A has a % prefix then B and/or C can be called.

:	(Ada)select	next	If A has a : prefix, where A is a task or a process element within a task, then A is interpreted as a rendezvous, and B and C are alternative entries within the *select*.
/x	cancel "x"	previous	If B or C has a /x prefix, then the prefix x appearing on A is cancelled, where x is assumed to be of type *next*.

Figure A2 presents an example in conventional Structure Chart (Call Tree) format, Figure A3 is the Buhr Diagram, and Figure A4 presents the corresponding Program Structure Notation. From presentation and information content standpoints the graphical Structure Chart is both easy to understand and fairly complete. The Buhr Diagram can become unwieldy as the number of interacting components grows, and the subprogram call structure within the main subprogram or within a package is not shown. What is shown are the packages and their subprogram and task entry interfaces (ie, the visible part or specification). The Program Structure Notation contains the same detail as the Structure Chart but in a purely textual format, so that it amenable to easy update, electronic distribution, and expansion into PDL, further annoation, and ultimately the code itself.

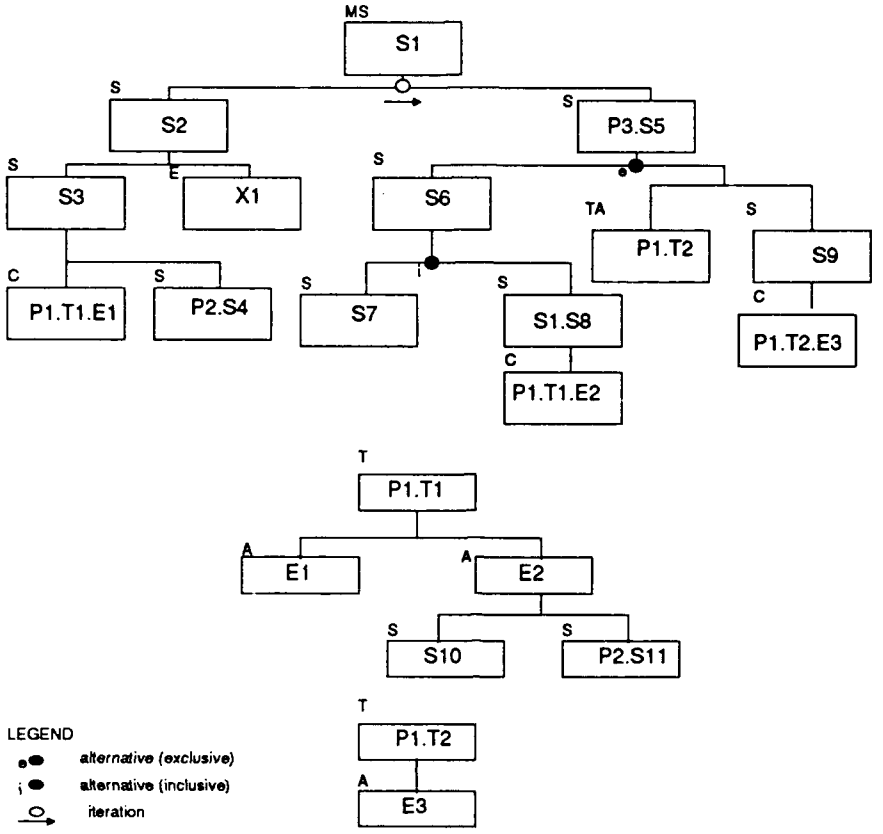


Fig. A2. Example as a Structure Chart (Call Tree)

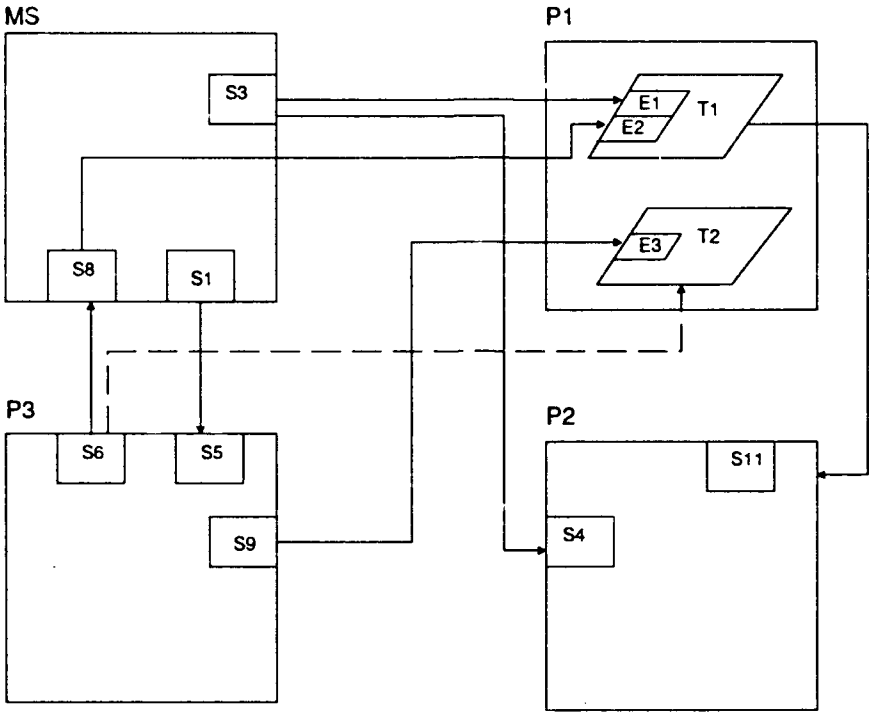


Fig. A3. Example as a Buhr Diagram

```
@1. S1 [MS]
  - 1.1 S2 [S]
        1.1.1 S3
          - 1.1.1.1  P1 [P]
                     T1 [T]
                     E1 [C]
          - 1.1.1.2  P2 [P]
                     S4
        1.1.2 X1 [E]
 - +1.2 P3 [P]
        S5
      % 1.2.1 S6
            1.2.1.1 S7
            1.2.1.2 S1 [MS]
                    S8
                1.2.1.2.1     P1 [P]
                              T1 [T]
                              E2 [C]
        1.2.2 *
          - 1.2.2.1  P1 [P]
                     T2 [TA]
          - 1.2.2.2  S9
                1.2.2.2.1     P1 [P]
                              T2 [T]
                             E3 [C]
: 2. P1 [P]
    T1 [T]
    2.1 E1 [A]
    2.2 E2 [A]
      - 2.2.1    S10 [S]
      - 2.2.2    P2 [P]
                 S11
  3. P1 [P]
    T2 [T]
    3.1 E3 [A]
```

Fig. A4. Example Combined in Program Structure Notation

APPENDIX B

Use of the ESDS in Event Oriented Design

Step 1: Task Decomposition

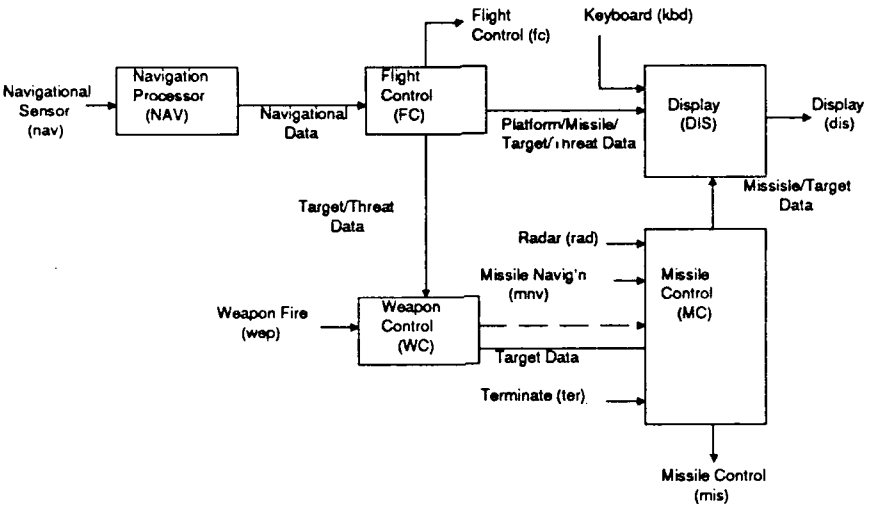


Fig. B1: Data Flow Diagram of the Example

The example has five tasks, abbreviated NAV, FC, DIS, WC and MC. A solid arrow between tasks indicates that the task at the tail calls the task at the head. Data that are consumed by the called task, in response to the call, are shown below the joining arrow. A dashed arrow (as between WC and MC) indicates that the task at the tail dynamically initiates the task at the head. Once initiated a task may also be called as is the case here. The four tasks that are not dynamically initiated are elaborated and automatically initiated at the beginning of the program run. External input signals are designated by lower case mnemonics and appear at the tail of a solid arrow into a task. Ada syntactically handles external interrupts and intertask calls in the same way, ie, as a task entry; therefore, the diagrammatic symbolism is similar, except that the input signals do not appear within a block. Output signals are indicated in the reverse symbolic manner from inputs. They are also designated by lower case mnemonics.

In the example, there is a Navigational Processor (NAV) task that is driven by a Navigational Sensor (nav) signal. The processor calls on the Flight Control (FC) task, which consumes Navigational Data. Task FC has an output, Flight Control (fc). Table

B1 presents the input/output pairs, required response times, and the specified loading in terms of input arrival rates. This information corresponds to that of specification documents (3) and (4).

TABLE B1
Input/Output Response Times and
Input Loading

INPUT NAME	INPUT TYPE	LOADING -- TIME OF OCCURENCE -- MANUAL	LOADING -- TIME OF OCCURENCE -- PERIODIC/RANDOM START	LOADING -- TIME OF OCCURENCE -- PERIODIC/RANDOM PERIOD	OUTPUTS	RESPONSE TIME MIN	RESPONSE TIME REQD
nav	Periodic		25	5	fc	11	13
					dis	10	15
kbd	Random		0	15	dis	1	2
wep	Manual	5			dis	11	15
		20					
		40					
rad	Periodic		10	8	mis	3	5
					dis	6	10
mnv	Periodic		10	7			
ter	Manual	25					
		60					
		80					

The first line of the table indicates that the input *nav* causes the output *fc* to occur and that the required response time is 12 time units. The type of input is specified as *periodic*. The specific loading is given as a periodicity of 5 time units, starting at time 25. The loading information could be entered at any time before the actual run of the simulation and can even be changed from one run to another. It is parametric data and hence not an inherent part of the model specification. Also, under RESPONSE TIME, a minimum (MIN) time is shown, which is the minimum possible time from the indicated input to output. It comes from the Event Diagram.

Task FC calls on the Display (DIS) task, which outputs the signal Display (dis). Table B1 indicates that nav-dis is also an I/O pair with a response time of 15. Task DIS also has an external input, Keyboard (kbd). The table presents it as a random input with a start time of 0 and a period of 15. This means that starting at time 0, a kbd input will appear at a random time (uniform distribution assumed, though other generators could be used) over the 15 unit interval. After the input occurs, another 15 unit interval starts and another kbd input will randomly occur during this next interval. The Periodic and Random input types are both of the automatic type described in Section 3.2. They continue to be regenerated throughout the duration of the run. The kbd input has a dis output with required response time of 2 time units.

The Weapon Control (WC) task calls on the FC task and on the Missile Control (MC) task; however, task MC must first be dynamically initiated by task WC, as indicated by the dashed arrow between them. Task WC has an input, Weapon Fire (wep). Table B1 shows it to be Manual. This means that the designer must provide, at run time, one or more times at which the signal is to occur. Again, these times do not have to be provided at Block Diagram time as part of the model. They are run time parameters. Input wep has a single output, which is dis, to occur within 15 time units.

The Block Diagram is not specific as to what event triggers the dynamic initiation of the MC task. This is left to the Event Diagram. It is assumed that multiple instances of this task can be initiated, so that, at a given time, there may be more than one such task active.

Task MC has three inputs. One is the Radar (rad), which is a periodic input whose outputs are the Missile Control (mis) signal, to occur within 5 time units and the dis signal, to occur within 10 time units. The second input is a Missile Navigation (mnv) signal with no output, because the mnv and rad signals are processed together to produce the mis and dis outputs. It also has a Terminate (ter) signal, which is the means used in this example to terminate the task. In the real world, the task should terminate when the missile no longer requires control, ie, when it hits the target or otherwise terminates its useful flight. For convenience, this input has been made manual and is coordinated with the wep input, which (as will be seen in the Event Diagram) causes task MC to be initiated.

The rad, mnv and ter inputs start when a wep input arrives, while the rad and mnv input generation stops when a ter input arrives. Each wep input inititiates one set of these inputs, and each ter input stops one set of rad and mnv inputs.

The DFD (also referred to as the Block Diagram) and its associated Input/Output table present the basic task decomposition and intertask communication of the system.

Step 2: Task Interfaces

Substeps (1), (3) and (4) are all represented in the Event Diagram of Figure B2. Substep (2), design of the interfacing data structures and decision on method of communication, are omitted for the example, because they do not at present relate to the ESDS model.

Step 3: Representation of Timing Requirements by the Event Diagram

In Step 3 each task of the DFD from Step 1 is decomposed into event sequences of the following nine event types:

1. Task initiation
2. Task entry
3. Sequential processing
4. Call or interrupt to a task entry
5. End of entry
6. End of task

8. Delay (non select)
9. Delay (select)

The Event Diagram is shown in Figure B2.

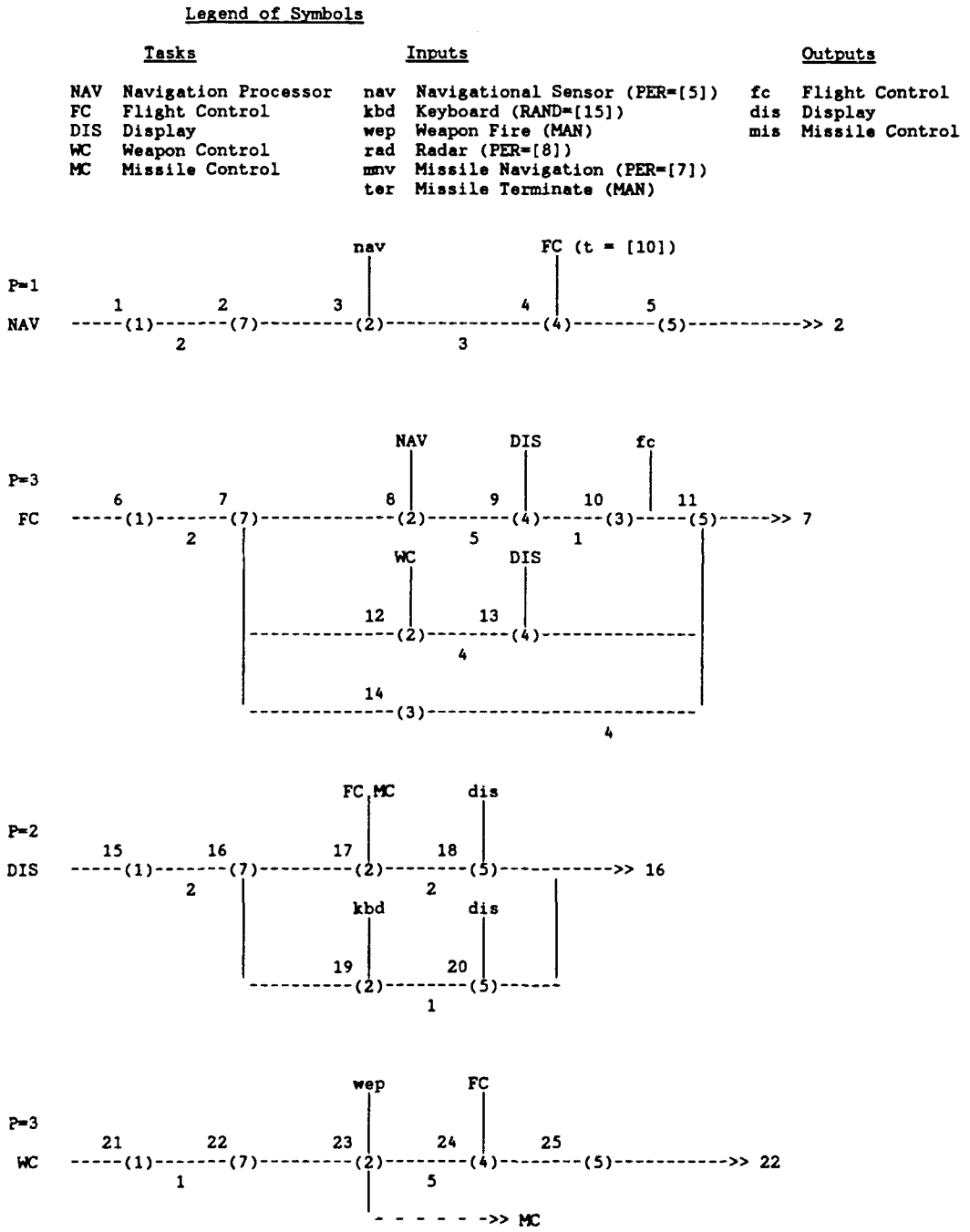


Fig. B2: Event Diagram of the Example

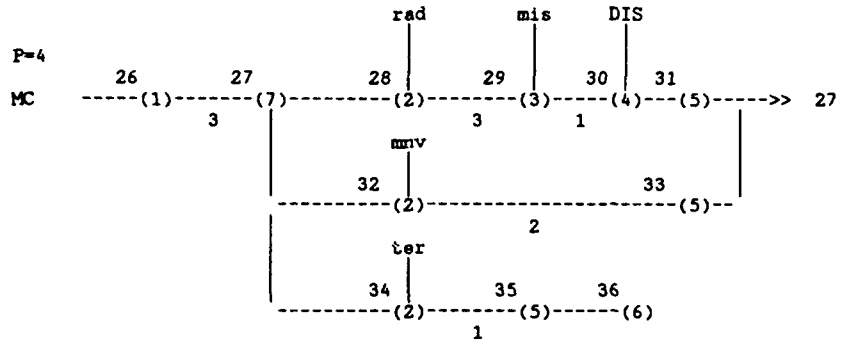


Fig. B2: Event Diagram of the Example (Cont'd)

Each event is identified by two numbers. The number in parentheses is the event type as given in the above list. The number to the left of the parentheses is a unique serial number assigned to each event. Where appropriate the specified execution time of an event appears under the line joining the event to its successor. Of the nine event types only types 1, 2 and 3 have execution times. Thus, in the diagram corresponding to Task NAV, event 1 is a type 1 (task initiation) and takes two time units. Event 2 is a type 7 (rendezvous) and has no execution time specified, because it waits until an *accept* (event 3) arrives. Event 3 is a type 2 (*entry*). The nav input (See Figure B1) enters here. This is followed by event 4, which is a call to task FC. Task NAV is suspended until the call is processed, whereupon task NAV can continue at event 5, which is a type 5, end of entry. The diagram indicates that control then returns to event 2, which is the rendezvous. In this way the task is either waiting at the rendezvous or is processing the nav input within event 3.

Note that the call to FC at event 4 is conditional, as denoted by the notation (t = [10]). This means that if the call is not accepted at FC within 10 time units, it is to be cancelled.

The priority of task NAV is 1, as indicated by the P=1 above the NAV symbol. The Flight Control task (FC) starts at event 6, which is a type 1 (task initiation), runs for two time units and then comes to the rendezvous at event 7. This is an Ada select statement, because there are alternative accepts at events 8 and 11. There is also an else statement at event 14. Note that the entries at events 8 and 11 are from other tasks (NAV, WC and MC), not from inputs. Events 8 and 11 have respective execution times of 5 and 4 and then each calls the Display (DIS) task. Note that event 10 triggers an output (fc), as shown also in the Flight Control block of Figure B2. All three paths return to the rendezvous at event 7. This is a typical control structure for a task.

Table B1 indicates that the nav input has two outputs. One is fc, which must occur within 12 time units; the other is dis, which must occur within 15 units. The minimum execution time for the nav-fc pair is 10 time units. This is determined by tracing the event path: 3-4-8-9-16-17-10, where events 3, 8 and 17 take 3, 5 and 2 units, respectively. A required response time of 12 units will probably turn out to be a fairly tight requirement, because so many calls are made on both the FC and DIS tasks.

The Display task (DIS) has two entries, at events 17 and 19. The former is the entry from task FC. It calculates coordinate information and displays it as the output dis. The latter is the kbd interrupt. A keyboard command is interpreted and appropriate information is displayed. Table B1 indicates that the kbd input has one output, dis, which must occur within 2 time units. The table also shows that kbd is a random input (ie, the pilot does not request data on a periodic basis), over a 15 time unit interval, starting at time 0. Other, more sophisticated distributions could be added to the system, if needed.

The Weapon Control task (WC) will initiate a Missile Control task (MC) whenever there is a weapon fire input (wep) at event 23. These are indicated in the table to be manual inputs occuring at times 5, 20 and 40.

The Missile Control task (MC) accepts radar input (rad) that tracks the target and missile navigational sensor input (mnv). These are both periodic, occuring every 8 and 7 time units, respectively. They are initiated with the wep input and terminate with the ter input, which terminates the missile flight and is another manual input to the MC task.

Step 4: Running of the ESDS and Removal of Timing Bottlenecks

The loading shown in Table B1 turns out to be quite severe for the design architecture represented by the Event Diagram of Figure B4. A series of changes were therefore made, in accordance with the options of Step 4. Table B2 presents the results of each run along with the design change made from the previous run. The results are given simply as the number of I/O failures for each I/O pair. The ESDS provides much more information than simply the I/O failures. QR-7 presents a detailed description of this data, but for the purpose of this discussion, the most basic information is the failure of the system to achieve a required response time, which is referred to here as an I/O failure.

TABLE B2
Results of a Series of ESDS Runs

	DESIGN MODIFICATIONS						I/O FAILURES						
Run	Execution time of event 3	event 8	Task DIS CONFIG'N	Number of PROCESSORS	Req'd Response TIME OF rad-dis	Loading: rad PERIOD	kbd dis	nav dis	rad dis	wep dis	nav fc	rad mis	Total
1	3	5	single	2	10	8	1	3	2	1	2	2	11
2	1	2	single	2	10	8	3		3			3	9
3	1	2	split	2	10	8	2		1	2	4	1	10
4	1	2	split	3	10	8		1	3				4
5	1	2	split	3	14	8		1	2				3
6	1	2	split	3	14	12							0

Run 1 corresponds to the design of Figure B2 and the response and loading specifications of Table B1. The right side of Table B2 records the I/O failures for the indicated I/O pairs. The total number of such failures in run 1 was 11. The objective of the subsequent redesigns was to eliminate these failures. Column 4 of the table is labeled TASK DIS CONFIG'N. "single" refers to the existing, single Display task shown in Figures B1 and B2. "split" refers to a splitting of the entries within this task into two tasks to provide greater opportunity for parallel operation. This was indicated as a queuing bottleneck in Run 2. The split configuration is shown in Figure B3.

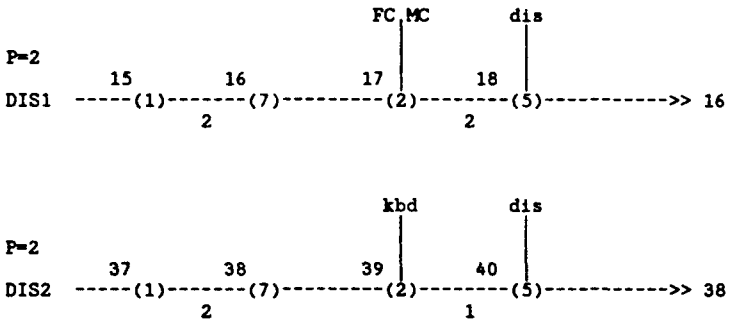


Fig. B3: Split Display Configuration

The result of Run 1 was that large queues developed at event 3, the nav input and at event 8, the entry that NAV calls in FC. This resulted in five failures involving the nav input. (3 for nav-dis and 2 for nav-fc). There were then ripple effects in DIS, where FC and MC contended at the same entry (event 17) as well as with the *kbd* input at event 19. These in turn caused all *dis* outputs to be affected (1 failure for kbd-dis, 2 for rad-dis and 1 for wep-dis). The two rad-mis failures were caused by queuing at event 28 of the *rad* input, resulting from the fact that when MC would call DIS (at event 30) it would be suspended for long periods waiting for the DIS bottleneck to clear. The ESDS analysis shows both the immediate and apparent bottlenecks, as in the case of the *nav* queues, as well as the ripple effects. With this diagnositic information in hand the designer must choose a suitable remedy. In the case of Run 1 it was decided to increase the processing speed of events 3 and 8.

Changing priorities would not have achieved anything, because the NAV task was already getting a high priority as a result of the fact that an input interrupt (*nav*) raises the called task to the highest priority. Another alternative would have been to reduce the loading by increasing the *nav* input interval from 5. This, however, would not be a software design change, but a specification change, which is the last option selected. At the systems level it would be regarded as a design change, that might or might not be acceptable. Thus, priority is given to those changes that the software designer can make without changing either the hardware configuration (ie number of processors) or the timing specifications (ie, response times or loading). Clearly, a bottleneck at entries 3 and 8 can be reduced if the proceessing speed of these events can be increased. The feasibility of this will, in turn, depend upon the algorithms involved, the processor speed, the skill of the programmer, and the quality of the compiler. This will be determined in Step 6. The methodology is an iterative one that proceeds through Step 5 and then can cycle back to any step, as far back as Step 1. Therefore, the designer has the option of reducing the execution time specification to the programmer in order to achieve the required performance before resorting to measures beyond the software boundary. In this case this option was exercized, and the event 3 time was reduced from 3 to 1, while event 8 was reduced from 5 to 2.

The Run 2 row in Table B2 shows that this clearly was effective. In fact, one may ask how the new execution times were determined. With the ESDS it is easy to continuously reduce the time until the desired effect is achieved. Thus, at times 1 and 2 for events 3 and 8, respectively, the five nav-dis and nav-fc failures disappear. However, this reduction of 5 failures did not show up entirely in the total failures in Run 2. The

reason is that more calls were now made from NAV to FC at event 4 and from FC to DIS at event 9, thus exacerbating the Display (DIS) bottleneck. Hence, the kbd-dis, rad-dis and rad-mis failures went up, leaving a net improvement overall of only 2 in the total.

Attention therefore turned to the DIS task. It was decided to change the basic architecture of the design by splitting DIS into two tasks, called DIS1 and DIS2, as shown in Figure B3. The purpose is to allow the kbd to be processed in parallel with the FC and MC calls. As shown in Table B2 for Run 3 it didn't quite work. The failures were redistributed, and our nav-fc problem that was so nicely cleared up in Run 2, reappeared with a vengeance, going from 0 to 4. Analysis of the ESDS results showed that (1) The desired parallelism did not occur, because there were not enough processors to provide real parallel operation, and (2) the two DIS tasks compete for available processors with FC (all are running at the highest priority, which have been inherited from input interrupts), as well as with the other tasks, so that although the nav-dis gets through, the nav-fc does not.

The system is processor bound. No further software solutions can solve the problem; therefore, in Run 4 another processor is added, and the bottleneck is broken. The total number of failures drops from 10 to 4. Runs 3 and 4 illustrate the complexity of these systems inasmuch as a given design change can have unintended or unexpected results, because of the secondary or ripple effect of changes in systems that are so highly interrelated. In Run 3 the splitting of a task just redistributed the bottlenecks, and made some situations worse, that had previously been improved. Overall, the total failures even increased from 9 to 10, though this is not a significant increase. In Run 4, on the other hand, the intended effect was achieved, but the rad-dis failures returned to their former level of 3. Analysis of the run showed the reason for this. The MC call to DIS from event 30 was found to be queued too long at event 17. This also affected nav-dis to some extent.

One solution to this problem would be to create yet another separate DIS task to process the MC and FC calls on DIS1 independently. This would also require another processor. Another option is to go back to the system designer to see if the performance requirements can be relaxed. This is the route that was taken in Run 5. The required response time of rad-dis was increased from 10 to 14. Again, the ESDS could be used to determine the smallest such value that would eliminate all of the failures, but it is assumed in the example that the system designer would only allow an increase to 14 time units. Run 5 shows that this was insufficient. It eliminated only one failure. Next, the load was examined. If the *rad* input rate could be reduced, the DIS1 load would be reduced, and then both the rad-dis and the nav-dis bottlenecks broken. This again requires a system design decision regarding the repetition or sampling rate of the *rad* input signal. Here the system designer was willing to increase the interval from 8 to 12, and in Run 6 all failures were eliminated.

Conclusion

The example has shown that one can design in a systematic way for performance, despite the complexity of multitasking systems, if there exists an analytic tool that is capable of showing the designer where the problems lay and if there is a coordinated methodology that presents options to solve the problems once the tool has identified them. The alternative, which corresponds to current practice, is to design for performance by a trial and error process.

Table B2 is an "end picture" summary of the six runs. In the next report, we will show the detail of intermediate information that pin-points the bottlenecks and enables the designer to see both the direct causes of I/O failure as well as the ripple effects.