Software Design Methods for Real-Time Systems

SEI Curriculum Module SEI-CM-22-1.0

December 1989

Hassan Gomaa
George Mason University

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.
Approved for public release. Distribution unlimited.

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Education Program is developing a wide range of materials to support software engineering education. A **curriculum module** identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in *designing* a course. A **support materials** package includes materials helpful in *teaching* a course. Other materials under development include model curricula, textbooks, educational software, and a variety of reports and proceedings.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

SEI curriculum modules may be copied or incorporated into other materials, but not for profit, provided that appropriate credit is given to the SEI and to the original author of the materials.

Comments on SEI educational publications, reports concerning their use, and requests for additional information should be addressed to the Director of Education, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Comments on this curriculum module may also be directed to the module author.

Hassan Gomaa
Department of Information Systems
and Systems Engineering
George Mason University
4400 University Drive
Fairfax, VA 22030

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position.
It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

221011990

Software Design Methods for Real-Time Systems

Acknowledgements

This module is an outgrowth of my experiences in teaching the graduate course “Software Engineering Methods” at the Wang Institute of Graduate Studies and graduate courses “Software Requirements Analysis, Prototyping, and Design” and “Software Design Methods for Real-Time Systems” at George Mason University. I am indebted to my students for their enthusiasm and feedback, which helped me improve the courses substantially and hence pave the way for this module.

I would also like to gratefully acknowledge the many stimulating discussions I have had with John Brackett, Bo Sanden, and David Weiss that have contributed significantly to my understanding of the design methods described in this module.

I am also indebted to Lionel Deimel for his considerable assistance with all aspects of the production of this curriculum module. Considerable effort was also expended by Jim Rankin in constructing the bibliography.

Thanks are also due to John Brackett and David Budgen, who helped in defining the scope of this module, and to the following reviewers of an earlier draft of this module: Lionel Deimel, Richard D’Ippolito, Gary Ford, Ken Fowler, Frank Friedman, John Goodenough, Roger Van Scoy, and David Wood.

Contents

Software Design Methods for Real-Time Systems

Module Revision History

Version 1.0 (December 1989)	Initial release Approved for publication

Software Design Methods for Real-Time Systems

Capsule Description

This module describes the concepts and methods used in the software design of real-time systems. It outlines the characteristics of real-time systems, describes the role of software design in real-time system development, surveys and compares some software design methods for real-time systems, and outlines techniques for the verification and validation of real-time designs. For each design method treated, its emphasis, concepts on which it is based, steps used in its application, and an assessment of the method are provided.

Philosophy

Real-Time Systems. Real-time systems have widespread use in industrial, commercial, and military applications. These systems are often complex because they have to deal with multiple independent streams of input events. These events have arrival rates that are often unpredictable, although they must be responded to within predefined timing constraints.

Real-time systems are frequently classified as "hard real-time systems" or "soft real-time systems." A hard real-time system has time-critical deadlines that must be met; otherwise a catastrophic system failure can occur. In a soft real-time system, it is considered undesirable, but not catastrophic, if deadlines are occasionally missed.

In spite of the importance of timing constraints in real-time systems, it is a characteristic (and a limitation) of the current state of the art in software design methods for real-time systems that the methods tend to emphasize structural and behavioral aspects of real-time systems and generally pay significantly less attention to timing constraints.

Software Design. A *software design strategy* is an overall plan and direction for performing design. For example, functional decomposition is a software design strategy.

A *software design concept* is a fundamental idea that can be applied to designing a system. Information hiding is a software design concept.

A *software design notation* or representation is a means of describing a software design. It may be diagrammatic, symbolic, or textual. Structure charts and pseudocode are software design notations.

A *software design method* is a systematic approach for carrying out design. It typically describes a sequence of steps for producing a design. A design method is based on a set of design concepts, employs a design strategy or strategies, and documents the resulting design using one or more design notations.

A software design method does not provide a cookbook approach to performing design. A designer must use his skill and judgement in applying the method. It should be pointed out that when a method is deficient in a certain aspect, it is often the case that experienced designers will compensate for this by developing an *ad hoc* solution.

Module Organization. This module builds on the module *Introduction to Software Design* [Budgen89] by focusing on the real-time system domain. It points out the differences between this domain and other application domains. It describes design concepts that are of particular importance to real-time system design, such as concurrent tasks and finite state machines. Life-cycle considerations specific to real-time systems are examined. Design representations for expressing real-time designs are outlined. This module references material in *Introduction to Software Design*, in particular, for those design topics that are of importance to the design of all software systems.

This module surveys several software design methods for real-time systems. The concepts on which each method is based are described first to show what the method attempts to achieve. The steps involved in using the method are then outlined to give an appreciation of the method. This is followed by an assessment of the method. The methods are subsequently compared to one another to point out the similarities and differences among them. Since the methods are best understood by studying an example, the support materials provide an example of applying each design method to solve the same real-time problem.

Criteria for Selecting Software Design Methods. In selecting the design methods to be included in this survey, the following criteria for selection were used:

1. The method must be published in the literature and not be proprietary. This excludes methods such as PAMELA (Process Abstraction for Large Embedded Applications) [Cherry86], whose description is not widely available.
2. The method must actually have been used on a real-world real-time application. This excludes some emerging methods that have recently been published, such as the Box Structured Method [Mills87], ADARTS [Gomaa89b] and Entity Life Modeling [Sanden89].
3. The method must not be oriented toward a specific language. This excludes methods such as that discussed in [Nielsen88], which is oriented toward Ada.
4. The method must be a design *method* and not a design *notation*. A design notation *suggests* a particular approach to performing a design, but does not provide a systematic approach of specific steps for performing design. This excludes design notations such as Statecharts [Harel88b] and MASCOT (Modular Approach to Software Construction and Test) [Simpson86].

Module Interface. Three other SEI curriculum modules provide background for this one related to particular life-cycle phases:

- *Software Specifications: A Framework* [Rombach89] introduces some of the terminology used by this module.
- *Software Requirements* [Brackett89]. The requirements phase precedes the design phase, and its outputs are the inputs to the design phase.
- *Introduction to Software Design* [Budgen89] introduces the principles and concepts involved in the design of large programs and systems. It may thus be viewed as a prerequisite to this module, which focuses specifically on the design of real-time systems.

This module is one of several proposed modules on real-time systems including:

- *Fundamentals of Real-Time Systems*, which should introduce a range of topics relevant to real-time systems, including characteristics of real-time systems, software life cycle overview for real-time systems, and interfacing to hardware—interrupt handling, polling, and sensor/actuator interfaces. In the absence of this module, the reader is referred to introductory books on real-time systems, such as Allworth and Zobel [Allworth87] or Glass [Glass83].

Objectives

A student who has mastered the material presented in this module may be expected to be able to:

- Describe the differences between real-time systems and other kinds of software systems.
- Discuss the design concepts of particular importance to real-time systems.
- Describe design representations for describing real-time designs.
- State the principles behind and steps involved in several design methods for real-time systems. Discuss the similarities and differences between these methods.
- Apply one or more methods to solve small real-time problems.
- Discuss how real-time designs may be verified and validated.

Prerequisite Knowledge

Students should be familiar with the terms and concepts of the software life cycle. They should understand concurrent processing concepts, including process synchronization and mutual exclusion. Students should also have had an introduction to software design.

Module Content

Outline

I. Characteristics of Real-Time Systems
  1. Embedded Systems
  2. Interaction with External Environment
  3. Real-Time Constraints
  4. Real-Time Control
  5. Reactive Systems
  6. Concurrent Processing

II. The Role of Software Design in Real-Time System Development
  1. The Design Process
  2. Real-Time Design as a Software Life-Cycle Phase
    a. Life-Cycle Considerations for Real-Time Systems
    b. Requirements Definition
    c. Architectural Design
    d. Detailed Design
    e. Implementation
  3. Real-Time System Design Concepts
    a. General Design Concepts
    b. Real-Time–Specific Design Concepts
  4. Real-Time Design Representations
    a. Data Flow/Control Flow Diagrams
    b. Task Structure Diagrams
    c. MASCOT Diagrams
    d. Structure Graphs (Buhr Diagrams)
    e. Structure Charts
    f. Entity Structure Diagrams
    g. JSD Network Diagrams
    h. Object Diagrams
    i. Class Structure Diagrams
    j. State Transition Diagrams
    k. Statecharts
    l. Petri Nets
  5. Role of Software Design Methods
  6. Software Design Strategies for Real-Time Systems
    a. Design Methods Based on Functional Decomposition
    b. Design Methods Based on Concurrent Task Structuring
    c. Design Methods Based on Information Hiding Module Structuring
    d. Design Methods Based on Modeling the Problem Domain

III. Survey of Real-Time Software Design Methods
  1. Structured Analysis and Design for Real-Time Systems
    a. Overview
    b. Basic Concepts
    c. Steps in Method
    d. Products of Design Process
    e. Assessment of Method
    f. Extensions and/or Variations
  2. Naval Research Lab Software Cost Reduction Method
    a. Overview
    b. Basic Concepts
    c. Steps in Method
    d. Products of Design Process
    e. Assessment of Method
    f. Extensions and/or Variations
  3. Object-Oriented Design
    a. Overview
    b. Basic Concepts
    c. Steps in Method
    d. Products of Design Process
    e. Assessment of Method
    f. Extensions and/or Variations
  4. Jackson System Development for Real-Time Systems
    a. Overview
    b. Basic Concepts
    c. Steps in Method
    d. Products of Design Process
    e. Assessment of Method
    f. Extensions and/or Variations
  5. DARTS (Design Approach for Real-Time Systems)
    a. Overview

b. Basic Concepts
c. Steps in Method
d. Products of Design Process
e. Assessment of Method
f. Extensions and/or Variations

6. Other Real-Time Software Design Methods

IV. Design Verification and Validation

1. Software Technical Reviews
2. Requirements Tracing
3. Simulation
4. Prototyping
5. Software Testing
   a. Testing Concurrent Software
   b. System Testing

V. Review of Real-Time Software Design Methods

1. Comparison of Real-Time Software Design Methods
   a. Support for Finite State Machines
   b. Support for Concurrent Tasks
   c. Support for Information Hiding
   d. Timing Constraints
2. Trends in Real-Time Software Design Methods
   a. "Eclectic" Design Methods
   b. Domain Specification and Design Methods
   c. Computer Support Tools and Software Development Environments
   d. Executable Specifications and Designs
   e. Performance Analysis of Real-Time Designs
   f. Application of Knowledge-Based Techniques
   g. Application of Formal Methods

Annotated Outline

I. Characteristics of Real-Time Systems

Real-time software systems have several characteristics that distinguish them from other software systems:

1. Embedded Systems

A real-time system is often an embedded system, *i.e.*, the real-time software system is a component of a larger hardware/software system. An example of this is a robot controller that is a component of a robot system consisting of one or more mechanical arms, servo-mechanisms controlling axis motion, and sensors and actuators for interfacing to the external environment. A computerized automobile cruise control system is embedded in the automobile.

2. Interaction with External Environment

A real-time system typically interacts with an external environment, which is, to a large extent, non-human. For example, the real-time system may be controlling machines or a manufacturing processes, or it may be monitoring chemical processes and reporting alarm conditions. This situation often necessitates a sensory interface for receiving data from the external environment and actuators for outputting data to and controlling the external environment.

3. Real-Time Constraints

Real-time systems have timing constraints, *i.e.*, they must process events within a given time frame. These real-time constraints are specified in the software requirements. Whereas, in an interactive system, a human may be inconvenienced if the system response is delayed, in a real-time system, a delay may be catastrophic. For example, inadequate response in an air traffic control system could result in a midair collision of two aircraft. The required response time will vary by application, ranging from milliseconds in some cases, to seconds, or even minutes, in others.

4. Real-Time Control

A real-time system often involves real-time control. *I.e.*, the real-time system makes control decisions based on input data, without any human intervention. An automobile cruise control system, for example, has to adjust the throttle based on measurements of current speed to ensure that the desired speed is maintained.

A real-time software system may also have non–real-time components. For example, real-time data collection necessitates gathering the data under real-time constraints, otherwise the data may be lost. However, once collected, the data can be stored for subsequent non–real-time analysis.

5. Reactive Systems

Many real-time systems are reactive systems [Harel88a]. They are event-driven and must respond to external stimuli. It is usually the case in reactive systems that the response made by the system to an input stimulus is state dependent, *i.e.*, the response depends not only on the stimulus itself, but also on what has previously happened in the system.

6. Concurrent Processing

A feature of most real-time systems is concurrent processing, *i.e.*, there are many events that need to be processed in parallel. Frequently, the order of incoming events is not predictable. Furthermore, the

input load may vary significantly and unpredictably with time.

II. The Role of Software Design in Real-Time System Development

1. The Design Process

Design is a highly creative activity that relies on designer skill, experience, and judgement. Several factors need to be considered in the software design process [Budgen89].

2. Real-Time Design as a Software Life-Cycle Phase

a. Life-Cycle Considerations for Real-Time Systems

Like any software systems, real-time systems should be developed using a life-cycle model. The "waterfall" model [Boehm76, Fairley85] is the most widely used life-cycle model, although, more recently, other models have been used to overcome some of its limitations [Agresti86]. These include the incremental development model (also referred to as evolutionary prototyping) [Basili75, Gomaa86b] and the rapid prototyping model [Agresti86].

b. Requirements Definition

Since a real-time software system is often part of a larger embedded system, it is likely that a system requirements definition phase precedes the software requirements definition. In this case, system functional requirements are allocated to software and hardware before software requirements definition begins [Brackett89]. In this highly constrained environment, the emphasis is usually on producing developer-oriented requirements (D-requirements) [Rombach89].

c. Architectural Design

During this phase, the system is structured into its constituent components. An important factor that frequently differentiates real-time systems from other systems is the need to address the issue of structuring a real-time system into concurrent tasks (processes) [Buhr84]. Depending on the design method used and/or designer decisions, the emphasis at this stage may be on decomposition into tasks, modules, or both. Another important factor is consideration of the behavioral aspects of a real-time system, *i.e.*, the sequences of events and states that the system experiences. This provides considerable insights into understanding the dynamic aspects of the system.

d. Detailed Design

During detailed design, the algorithmic details of each system component are defined. This is often achieved using a program design language (PDL) notation, also referred to as structured English or pseudocode. In real-time systems, particular attention needs to be paid to algorithms for resource sharing and deadlock avoidance, as well as interfacing to hardware I/O devices.

e. Implementation

Since real-time systems are often embedded systems, testing is often more complex than for other systems, possibly requiring the development of environment simulators [Gomaa86a]. Furthermore, performance of the system needs to be tested against the requirements.

3. Real-Time System Design Concepts

a. General Design Concepts

[Budgen89] discusses several important general design concepts.

b. Real-Time–Specific Design Concepts

Design concepts of particular importance to real-time systems are:

(i) Finite State Machines

Finite state machines may be used for modeling the behavioral aspects of a system. Many real-time systems, in particular real-time control systems, are highly state-dependent. A finite state machine consists of a finite number of states and transitions between them. It can be in only one of a given number of states at any given time [Davis88]. In response to an input event, the machine generates an output event and may undergo a transition to a different state. Two notations widely used to define finite state machines are state transition diagrams, a graphical representation, and state transition matrices, a tabular representation. Since large real-time systems typically have large numbers of states, state transition diagrams or matrices can help substantially in providing an understanding of the complexity of these systems.

(ii) Concurrent Tasks (Processes)

A real-time system typically has many activities occurring in parallel. A task represents the execution of a sequential program or a sequential component of a concurrent program. Each task deals with one sequential thread of execution. Overall system concurrency is obtained by having many tasks executing in parallel. A design emphasizing concurrent tasks is often clearer and easier to understand, since it is a more realistic model of the problem domain than a sequential program. Concurrent processes are described in [BrinchHansen73], [Dijkstra68], and [Hoare74].

(iii) Information Hiding

Information hiding is a fundamental software design concept that is relevant to the design of all classes of software systems, not just real-time systems. The information hiding principle was first proposed by Parnas [Parnas72] as a criterion for decomposing a software system into modules. The principle states that each module should hide a design decision that is considered likely to change. Each changeable decision is called the "secret" of the module. The reasons for applying information hiding are to provide modules that are modifiable and understandable, and hence maintainable. Because modules employing information hiding are usually self-contained, they have a much greater potential for reuse than most procedural modules.

4. Real-Time Design Representations

a. Data Flow/Control Flow Diagrams

Data flow/control flow diagrams are used in Real-Time Structured Analysis [Ward85, Hatley88]. They are an extension of data flow diagrams to include control flows and control transformations. Control flows represent event signals that carry no data. Control transformations control the execution of data transformations and are specified by means of state transition diagrams or decision tables.

b. Task Structure Diagrams

Task structure diagrams are used by the DARTS design method [Gomaa84] to show the decomposition of a system into concurrent tasks and the interfaces between them, in the form of messages, event signals, and information hiding modules.

c. MASCOT Diagrams

MASCOT diagrams [Allworth87, Simpson79, Simpson86] are used to show the decomposition of a system into subsystems consisting of concurrent tasks. The interfaces between tasks are in the form of channels (message queues) and pools (information hiding modules).

d. Structure Graphs (Buhr Diagrams)

Structure graphs are used to describe the structure of a system in terms of concurrent tasks, packages (information hiding modules), and procedures [Buhr84]. These graphs are oriented toward use with the Ada programming language, but they may also be used with languages such as Modula-2.

e. Structure Charts

Structure charts are used in Structured Design [Page-Jones88, Yourdon79] to show how a program is decomposed into modules, where a module is typically a procedure or function.

f. Entity Structure Diagrams

Entity structure diagrams are used in Jackson System Development (JSD) to show the structure of a real-world entity, in the form of the sequence of events experienced by it [Jackson83, Cameron86, Cameron89]. The graphical notation is similar to that used in Jackson Structured Programming (JSP) structure diagrams [Jackson75].

g. JSD Network Diagrams

JSD network diagrams are used to show all the processes in a JSD design and the interfaces between them. Interfaces are represented in the form of data stream (message) communication or state vector inspections [Jackson83, Cameron86, Cameron89].

h. Object Diagrams

Object diagrams are used in object-oriented design (OOD) to show the objects in the system and to identify the visibility of each object in relation to other objects [Booch86].

i. Class Structure Diagrams

Class structure diagrams are used in OOD to show the relationships between classes of objects [Booch86].

j. State Transition Diagrams

State transition diagrams are a graphical representation of finite state machines (FSMs) in which the nodes represent states and the arcs represent state transitions [Allworth87]. They are used by the Real-Time Structured Analysis [Ward85, Hatley88] and DARTS [Gomaa84] methods.

k. Statecharts

Statecharts are an extension of FSMs that provide a notation and approach for hierarchically structuring FSMs and allowing concurrent FSMs that interact with each other [Harel88b]. The objective is to provide a notation that it is clearer and more structured than state transition diagrams.

l. Petri Nets

Petri nets [Peterson81] are a graphical representation for modeling concurrent systems. Two types of nodes are supported: places that are used to represent conditions and transitions that are used to represent events. The execution of a Petri net is controlled by the position and movement of markers called "tokens." Tokens are moved by the firing of the transitions of the net. A transition is enabled to fire when all its input places

have tokens in them. When the transition fires, a token is removed from each input place and a token is placed on each output place. Timed Petri nets are an extension to Petri nets that allow finite times to be associated with the firing of transitions.

5. Role of Software Design Methods

This material is covered in [Budgen89].

6. Software Design Strategies for Real-Time Systems

The various design methods described in this module use different strategies and emphasize different design concepts in decomposing the system into its components. A classification of them, based on the strategy used, is given below.

a. Design Methods Based on Functional Decomposition

This strategy is used by Real-Time Structured Analysis and Design [Ward85, Hatley88]. The system is decomposed into functions (called transformations or processes), and interfaces between them are defined in the form of data flows or control flows. Functions (*i.e.*, data or control transformations) are mapped onto processors, tasks, and modules [Ward85].

b. Design Methods Based on Concurrent Task Structuring

This strategy is emphasized by DARTS [Gomaa84]. Concurrent tasking is considered a key aspect in real-time design. DARTS provides a set of task-structuring criteria to assist the real-time system designer in identifying the concurrent tasks in the system. DARTS also provides guidelines for defining task interfaces.

c. Design Methods Based on Information Hiding Module Structuring

This strategy aims at providing software components that are modifiable and maintainable, as well as being potentially more reusable. This is achieved through the use of information hiding in the design of components. This strategy is used by the Naval Research Lab Software Cost Reduction method [Parnas84] and the object-oriented design method [Booch86].

d. Design Methods Based on Modeling the Problem Domain

This strategy is emphasized by the Jackson System Development method [Jackson83, Cameron86, Cameron89]. With this strategy, the objective is to model entities in the problem domain and then map them onto software processes.

III. Survey of Real-Time Software Design Methods

Below is a survey of real-time structured design methods. Each method treated is described and evaluated in subsections under the following headings:

a. Overview
b. Basic Concepts
c. Steps in Method
d. Products of Design Process
e. Assessment of Method
f. Extensions and/or Variations

1. Structured Analysis and Design for Real-Time Systems

a. Overview

Real-Time Structured Analysis and Design (RTSAD) is an extension of Structured Analysis and Structured Design to address the needs of real-time systems. Real-Time Structured Analysis (RTSA) is viewed by many of its users as primarily a specification method addressing the software requirements of the system being developed. Two variations of RTSA have been developed—the Ward/Mellor [Ward85, Ward86] and Boeing/Hatley [Hatley88] approaches. A third variation, ESML, the Extended System Modeling Language [Bruyn88], is a recent attempt to merge the Ward/Mellor and Boeing/Hatley methods for Real-Time Structured Analysis.

The extensions to Structured Analysis are driven by the desire to represent more precisely the behavioral characteristics of the system being developed. This is achieved primarily through the use of state transition diagrams, control flows, and integrating state transition diagrams with data flow diagrams through the use of control transformations (specifications).

Structured Design [Myers78, Page-Jones88, Yourdon79] is a program design method that uses the criteria of module coupling and cohesion in conjunction with the transform-centered and transaction-centered design strategies to develop a design, starting from a Structured Analysis specification.

b. Basic Concepts

(i) Data and Control Flow Analysis

In RTSAD, the system is structured into functions (called transformations or processes), and the interfaces between them are defined in the form of data flows or control flows. Transformations may be data or control transformations. The system is structured as a hierarchical set of data flow/control flow diagrams that may be checked for completeness and consistency.

(ii) Finite State Machines

Finite state machines, in the form of state transition diagrams, are used to define the behavioral characteristics of the system. The major extension to Structured Analysis is the introduction of control considerations, through the use of state transition diagrams. A control transformation represents the execution of a state transition diagram. Input event flows trigger state transitions, and output event flows control the execution of data transformations [Ward85].

In the Boeing/Hatley and ESML methods, it is also possible for a control transformation to be described by means of a decision table. Process activation tables are also used in Boeing/Hatley to show when processes (transformations) are activated.

(iii) Entity-Relationship Modeling

Entity-relationship diagrams are used to show the relationships between the data stores of the system [Yourdon89]. They are used for identifying the data stores (either internal data structures or files) and for defining the contents (attributes) of the stores. These are particularly useful in data-intensive systems.

(iv) Module Cohesion

Module cohesion is used in module decomposition as a criterion for identifying the strength or unity within a module [Myers78, Page-Jones88, Yourdon79]. Functional and informational cohesion are considered the strongest (and best) form of cohesion. In the early practice of Structured Design [Yourdon79], functionally cohesive modules in the form of procedures were emphasized. The informational cohesion criterion was added later by Myers [Myers78] to identify information hiding modules.

(v) Module Coupling

Module coupling is used in module decomposition as a criterion for determining the connectivity between modules [Myers78, Page-Jones88, Yourdon79]. Data coupling is considered the lowest (and best) form of coupling, in which parameters are passed between modules. Undesirable forms of coupling include common coupling, where global data are used.

c. Steps in Method

During the Real-Time Structured Analysis stage, the following activities take place. (It should be noted that steps (ii) - (v) are not necessarily sequential and that the steps are usually applied iteratively):

(i) Develop the System Context Diagram

The system context diagram defines the boundary between the system to be developed and the external environment. The data flow and control flow interfaces between the system and the external entities that the system has to interface to are defined.

(ii) Perform Data Flow/Control Flow Decomposition

A hierarchical data flow/control flow decomposition is performed, starting from the system context diagram. The Boeing/Hatley approach emphasizes hierarchical decomposition of both function and data. The Ward/Mellor approach starts with an event list, which is a list of input events, and then identifies the functions that operate on each input event. These functions are then aggregated to achieve a top-level data flow diagram and decomposed to determine lower-level functions.

(iii) Develop Control Transformations (Ward/Mellor) or Control Specifications (Boeing/Hatley)

A control transformation is defined by means of a state transition diagram. A control specification may be defined by one or more of state transition diagrams (tables), decision tables, and process activation tables. It is associated with a data flow diagram at any level of the hierarchy.

Each state transition diagram shows the different states of the system or subsystem. It also shows the input events (or conditions) that cause state transitions and actions resulting from state transitions. In the Boeing/Hatley method, the process activation table shows the activation of processes (data transformations) resulting from the actions of the state transition diagram.

(iv) Define Mini-Specification (Process Specification)

Each leaf node data transformation on a data flow diagram is defined by writing a mini-specification, usually in structured English, although other approaches are considered acceptable as long as the specification is a precise and understandable statement of requirements [Yourdon89].

(v) Develop Data Dictionary

A data dictionary is developed that defines all data flows, event flows, and data stores.

Following the RTSA phase, the Ward/Mellor

and Boeing/Hatley approaches diverge. The Boeing/Hatley approach uses system architecture diagrams [Hatley88]. The Ward/Mellor approach continues as follows [Ward85]:

(vi) Allocate Transformations to Processors

The RTSA transformations are allocated to the processors of the target system. If necessary, the data flow diagrams are redrawn for each processor.

(vii) Allocate Transformations to Tasks

The transformations for each processor are allocated to concurrent tasks. Each task represents a sequential program.

(viii) Structured Design

Transformations allocated to a given task are then structured into modules using the Structured Design method. Structured Design uses the criteria of module coupling and cohesion in conjunction with two design strategies, transform analysis and transaction analysis, to develop a program design starting from a Structured Analysis specification.

Transform analysis is a strategy used for transforming a data flow diagram into a structure chart whose emphasis is on input-process-output flow [Myers78, Page-Jones88, Yourdon79]. Thus, the structure of the design is derived from the functional structure of the specification. The input branches, central transforms, and output branches are identified on the data flow diagram and are structured as separate branches on the structure chart.

Transaction analysis is a strategy used for transforming a data flow diagram into a structure chart whose structure is based on identifying the different transaction types [Myers78, Page-Jones88, Yourdon79]. The processing required for each transaction type is identified from the data flow diagram, and the system is structured such that there is one branch on the structure chart for each transaction type. There is one controlling "transaction center" module.

d. Products of Design Process

For the RTSA specification, these consist of:

(i) System Context Diagram

(ii) Hierarchical Set of Data Flow/Control Flow Diagrams

(iii) Data Dictionary

(iv) Mini-Specifications

For each primitive transformation, *i.e.*, one that is not decomposed further, a data transformation is described by a structured English mini-specification, while a control transformation (Ward/Mellor) is defined by means of a state transition diagram.

A control specification (Boeing/Hatley) may be defined by one or more of state transition diagrams (tables), decision tables, and process activation tables, and is associated with each data flow diagram.

(v) Program Structure Charts

For each program, there is a structure chart showing how it is decomposed into modules. Each module is defined by its external specification, namely, input parameters, output parameters, and function. The internals of the module are described by means of pseudocode.

e. Assessment of Method

(i) Strengths

- Structured Analysis and the real-time extensions have been used on a wide variety of projects, and there is much experience in applying the method.
- There are a wide variety of CASE tools to support RTSA.
- The use of data flow and control flow diagrams can assist in understanding and reviewing the system. For example, a good overview of the system can be obtained.
- Emphasizes the use of state transition diagrams/matrices, which is particularly important in the design of real-time control systems.
- The Structured Design module decomposition criteria of cohesion and coupling help in assessing the quality of a design.

(ii) Weaknesses

- There is not much guidance as to how to perform a system decomposition. Consequently, different developers could structure the system in substantially different ways.
- RTSA is usually considered a requirements specification method. However, unlike the NRL Requirements Specification method, which treats the system to be developed as a black box, RTSA addresses system decomposition. Hence, there is a tendency in many projects to make design decisions during this phase, particularly if the specification gets de-

tailed. This makes the boundary between requirements and design fuzzy.

- Although Structured Design can be used for designing individual tasks, it is limited for designing concurrent systems, and hence real-time systems, because of its weaknesses in the areas of task structuring. Thus, Structured Design is a program design method leading primarily to functional modules and does not address the issues of structuring a system into concurrent tasks.
- In its application of information hiding, Structured Design lags behind the Naval Research Lab and object-oriented design methods. This is discussed in more detail in Section V.

f. Extensions and/or Variations

ESML, the Extended System Modeling Language [Bruyn88], is a recent attempt to merge the Ward/Mellor and Boeing/Hatley methods for Real-Time Structured Analysis. As an example, consider the ESML approach to developing state transition diagrams. The Ward/Mellor approach supports events, but not conditions, whereas the Boeing/Hatley approach supports conditions, but not events. Each of these restrictions is overcome in ESML, which supports both events and conditions, in common with the NRL method [Parnas86] and Statecharts [Harel88a, Harel88b].

2. Naval Research Lab Software Cost Reduction Method

a. Overview

The Naval Research Laboratory Software Cost Reduction method (NRL) originated to address the perceived growing gap between software engineering principles advocated in academia and the practice of software engineering in industry and government [Parnas84]. These principles formed the basis of a design method that has been applied to the development of a complex real-time system, namely the onboard flight program for the U.S. Navy's A-7 aircraft. Several principles were refined as a result of experience in applying them in this project.

Applications of the design method is preceded by a specification phase in which a black box requirements specification is produced [Heninger80]. During the requirements phase, consideration is given to factors that could have a profound effect on the future evolution of the system, namely the desirable system subsets (this information is used during the design phase in developing the uses hierarchy) and the likely future changes to the system requirements (this information is used during the design phase in developing the module structure).

The software structure of a system is considered as consisting of three orthogonal structures—the module structure, the uses structure, and the process (task) structure [Parnas74, Parnas84]. The module structure is based on information hiding. Each module is a work assignment for a programmer or team of programmers. The uses structure determines the executable subsets of the software. The process (task) structure is the decomposition of run-time activities of the system.

b. Basic Concepts

(i) Information Hiding

The NRL method applies the information hiding concept to the design of large scale systems [Parnas84]. The use of information hiding emphasizes that each aspect of a system that is considered likely to change, such as a system requirement, a hardware interface, or a software design decision, should be hidden in a separate module. The changeable aspect is called the "secret" of the module. Each module has an abstract interface that provides the external view of the module to its users.

(ii) Information Hiding Module Hierarchy

To manage the complexity of handling large numbers of information hiding modules, the NRL method organizes these modules into a tree-structured hierarchy and documents them in a module guide. Criteria are provided for structuring the system into modules.

(iii) Abstract Interface Specifications

An abstract interface specification defines the visible part of an information hiding module, that is all the information required by the user of the module. It is a specification of the operations provided by the module. The abstract interface to a module is intended to remain unchanged, even if the module's secret changes.

(iv) Design for Extension and Contraction

Design for extension and contraction is achieved by means of the uses hierarchy, which is a hierarchy of operations (access procedures or functions) provided by the information hiding modules. An operation *A* uses an operation *B* if and only if *A* cannot meet its specification unless there is a correct version of *B*. By considering subsets and supersets, designing systems is seen as a process of designing program families.

c. Steps in Method

The following steps in the NRL method are based on [Parnas86]. Reviews are considered an integral part of the method and are conducted for each work product [Parnas85].

(i) Establish and Document Requirements

The software requirements specification is a black-box specification of the system. The method emphasizes the outputs of the system over its inputs. The system is viewed as a finite state machine whose outputs define the system outputs as functions of the state of the system's environment.

The method uses separation of concerns in organizing the specification document. Sections are provided on the computer (hardware and software) specification, the input/output interfaces, specification of output values, timing constraints, accuracy constraints, likely changes to the system, and undesired event handling. The requirements method is discussed in more detail in [Heninger80]

(ii) Design and Document the Module Structure

To manage the complexity of handling large numbers of modules, the NRL method organizes information hiding modules into a tree-structured hierarchy and documents them in a module guide. The guide defines the responsibilities of each module by describing the design decisions that are to be encapsulated in the module. The module guide helps to provide structure, a check on completeness, and to avoid duplication of function. The guide allows modules to be referenced more easily during the subsequent development and maintenance phases of the project.

The module hierarchy is an "is composed of" relation. Each non-leaf module is composed of lower-level modules. Leaf modules are executable. The main categories of modules, as determined on the A-7 project, are:

- Hardware hiding modules
- Behavior hiding modules
- Software decision modules

Further categorization of modules may be carried out, although this is likely to be application-dependent. Module structuring is described in more detail in [Parnas84].

(1) Hardware Hiding Modules

These are either extended computer modules or device interface modules. The former hide the characteristics of the hardware/software interface that are likely to change, whereas the latter hide the characteristics of I/O devices that are likely to change.

(2) Behavior Hiding Modules

These are modules that hide the behavior of the system as specified by the functions defined in the requirements specification. Thus, if the requirements change, these modules are affected.

(3) Software Decision Modules

These are modules that hide software designer decisions that are likely to change.

(iii) Design and Document Module Abstract Interfaces

The abstract interface specification for each leaf module in the module hierarchy is developed. This specification defines the external view of the information hiding module, *i.e.*, all the information required by the user of the module. It is intended to contain just enough information for the programmer of another module to be able to use it. The interface specification includes the operations provided by the module, the parameters for these operations, the externally visible effects of the module's operations, timing and accuracy constraints, assumptions that users and implementors can make about the module, and definition of undesired events raised. More information on designing abstract interfaces is given in [Britton81]

(iv) Design and Document Uses Hierarchy

The uses hierarchy defines the subsets that can be obtained by deleting operations and without rewriting any operations. This is important for staging system development and for developing families of systems. During this stage, the operations used by each operation (provided by a module) are determined. By this means a hierarchy of operations is developed. The "allowed-to-use structure" defines the possible choices of operations, while the "uses structure" specifies the choice of operations for a particular version (member of the family). More information on the uses hierarchy is given in [Parnas79]

(v) Design and Document Module Internal Structures

After designing the module abstract interface, the internal design of each module is developed. This includes designing the internal data structures and algorithms used by the modules. In some cases, the module may be decomposed further into sub-modules.

During this phase, the process (task) structure of the system is developed [Faulk88]. Separation of concerns is used in designing the task structure. Inter-task synchronization is achieved by means of events. Tasks may be demand or periodic tasks.

d. Products of Design Process

(i) Software Requirements Specification

(ii) Module Guide

(iii) Module Abstract Interface Specifications.

(iv) Uses Hierarchy

(v) Module Internal Structures

(vi) Task Structure

e. Assessment of Method

(i) Strengths

- Emphasis on information hiding leads to modules that are relatively modifiable and maintainable.
- In addition to the emphasis on information hiding, the module hierarchy provides a means of managing large numbers of modules by organizing them into a tree-structured hierarchy.
- Emphasis is placed on designing for change. This starts during the requirements phase when likely changes in requirements are considered. It continues into design with the module structure, where each module hides an independently changeable aspect of the system.
- Emphasis is placed on identifying system subsets. This also starts during the requirements phase when desirable subsets are identified. It continues in the design phase with the uses hierarchy.
- There is a clear separation between requirements and design. The requirements present a black-box view of the system, emphasizing inputs, outputs, externally visible states and their transitions, as well as output-oriented functions.
- Emphasizes the use of finite state machines, which is particularly important in the design of real-time control systems.

(ii) Weaknesses

- It is usually difficult to get an overview of the system. In particular, it is often difficult to see how the major components of the system fit together. This is compounded by the lack of any graphical notation.
- There is less emphasis on task structuring. Although recognized as an important software structure, little guidance is given as to how to identify the tasks in the system.
- Proceeding from the software requirements specification to the module structure is often a big step. It is possible for significant components of the system to be omitted, particularly those not directly visible from the requirements specification, *e.g.*, software decision modules.

f. Extensions and/or Variations

The ADARTS method [Gomaa89b, Gomaa89c] uses a set of module structuring criteria that are based on the NRL module structuring criteria for identifying information hiding modules, in addition to a set of task structuring criteria for identifying concurrent tasks.

3. Object-Oriented Design

a. Overview

Object-oriented design (OOD) is a design method based on the concepts of abstraction and information hiding. There has been much debate on whether inheritance is an essential feature of object-oriented design. Two views of OOD are to be found. The first is in the Ada world, and its most widely known advocate is Booch [Booch86, Booch87a, Booch87b]. It holds that inheritance is a desirable but not essential feature of OOD. The second view originated in the object-oriented programming area, as illustrated by Smalltalk [Goldberg83], C++ [Stroustrup86], and Eiffel [Meyer88]. This view states that inheritance is an essential feature of OOD.

In a recent taxonomy of languages supporting objects, Wegner [Wegner87] has referred to languages that support information hiding modules (objects) but not inheritance, such as Ada and Modula-2, as object-based languages, while languages that support objects, classes, and inheritance are considered object-oriented languages. However, a similar taxonomy for object-oriented design methods has not been constructed.

In this section, the Booch view is used, since it is widely referenced in the Ada-based real-time system domain. Booch starts with an English language or RTSA system specification and then provides object structuring criteria for determining the objects in the system.

b. Basic Concepts

(i) Object Identification

Objects are identified by determining the entities in the problem domain. Each real-world entity is mapped onto a software object.

(ii) Abstraction

Abstraction is used in the separation of an object's specification from its body. The specification is the visible part of the object and defines the operations that may be performed on the object, *i.e.*, how other objects may use it. The body of the object, *i.e.*, its internal part, is hidden from other objects. Abstraction is also used in developing object hierarchies.

(iii) Information Hiding

Information hiding is used in structuring the object, *i.e.*, in deciding what information should be visible and what information should be hidden. Thus, those aspects of a module that need not be visible to other objects are hidden. Hence, if the internals of the object change, only this object is impacted.

c. Steps in Method

(i) Identify the Objects and Their Attributes

In OOD, an object is considered to have state, *i.e.*, persistent data. The state of the object changes as a result of operations on the object. The characteristics of an object are that it:

- has state
- is characterized by the actions it suffers (operations it provides) and requires (uses) of other objects
- is a unique instance of some class
- has restricted visibility of and by other objects
- can be viewed either by its specification or implementation

An informal strategy is used for identifying objects. Initially, Booch [Booch87a] advocated identifying objects by underlining all nouns (which are candidates for objects) and verbs (candidates for operations) in the specification. However, this is not practical for large-scale or even medium-size systems.

Booch later advocated the use of Structured Analysis as a starting point for the design, and then identifying objects from the data flow diagrams by applying a set of object structuring criteria [Booch86, Booch87b]. For each external entity on the system context diagram, there is a corresponding software object. For each data store on the data flow diagrams, there is a corresponding software object.

(ii) Identify the Operations Suffered by and Required of Each Object

In this step, the behavior of each object is characterized by identifying the operations that it provides and that are used by other objects, as well as the operations it uses from other objects. Starting with a Structured Analysis specification, operations are identified from the transformations on the data flow diagrams.

(iii) Establish the Visibility of Each Object in Relation to Other Objects

The static dependencies between objects are identified. Visibility is considered on an object basis (corresponding to the Ada "with" clause). A decision might be made to create a new class that defines the common behavior of a group of similar objects. An object diagram is drawn to show these dependencies.

Three kinds of objects are possible: servers (that provide operations for other objects but do not use operations from other objects), actors (that use operations from other objects but do not provide any), and agents (that provide operations and also use operations from other objects).

(iv) Establish the Interface of Each Object

The outside view of each object is developed. The interface forms the boundary between the object's outside view and inside view. An Ada package specification may now be developed for the object.

(v) Implement Each Object

The internals of each object are developed. This involves designing the data structures and internal logic of each object.

d. Products of Design Process

Booch [Booch86] has described four products of an object-oriented design. In addition, each package is specified by means of an Ada package specification.

(i) A Hardware Diagram

This captures the organization of the underlying target hardware system.

(ii) A Class Structure Diagram

This shows the relationships among classes of objects.

(iii) An Object Diagram

This shows the visibility of each object in relation to other objects.

(iv) The Architecture Diagram

This represents the physical design of the system and shows the system structured into Ada packages.

e. Assessment of Method

This assessment is made in terms of how applicable OOD is to the design of real-time systems.

(i) Strengths

- Is based on the concepts of abstraction and information hiding, two key concepts in software design.
- Structuring the system into objects, which are implemented as packages, should make the system more maintainable and components potentially reusable.
- Maps well to languages that support information hiding modules such as Ada and Modula-2.

(ii) Weaknesses

- Does not adequately address the important issues of task structuring, an important limitation in real-time design.
- The form of the solution depends substantially on the informal strategy used for identifying objects.
- Does not address timing constraints.
- The object structuring criteria are not as comprehensive as the NRL module structuring criteria. This is discussed in more detail in section V.

f. Extensions and/or Variations

More recently, object-oriented analysis [Shlaer88] methods have emerged. These methods use entity-relationship modeling techniques for identifying objects in the problem domain. Seidewitz [Seidewitz86, Seidewitz88] has also developed a method called the General Object-Oriented Design (GOOD) method. With this approach, the specification effort begins by identifying entities in the problem domain. The ADARTS method [Gomaa89b, Gomaa89c] applies task structuring criteria in addition to module structuring criteria that incorporate the OOD object structuring criteria.

4. Jackson System Development for Real-Time Systems

a. Overview

Jackson System Development (JSD) is a modeling approach to software design. A JSD design models the behavior of real-world entities over time. Each entity is mapped onto a software process (task). JSD is an outgrowth of Jackson Structured Programming (JSP), which is a program design method [Jackson75]. As JSD has evolved over several years, this section describes JSD as presented in the latest material available to the author [Cameron89].

Although JSD is, in principle, applicable to real-time systems, the emphasis of earlier work [Jackson83, Cameron86] has been on data processing applications. More recently, however, a number of articles have directly addressed the issue of applying JSD to real-time systems [Renold88, Cameron89, Sanden89]. Renold described mapping JSD designs to concurrent processing implementations. A report in [Cameron89] describes mapping a JSD design to the MASCOT notation [Simpson86], which specifically addresses concurrent processing. Sanden [Sanden89] describes a variation on JSD that addresses the needs of real-time systems and also maps directly to Ada.

b. Basic Concepts

(i) Modeling the Real World

A fundamental concept of JSD is that the design should model reality first [Jackson83], before considering the functions of the system. The system is considered a simulation of the real world. The functions of the system are then added to this simulation.

(ii) Entities and Concurrent Processes

Each real-world entity is modeled by means of a concurrent process called a model process. This process faithfully models the entity in the real world. Since real-world entities usually have long lives, each model process typically also has a long life.

(iii) Transformation to Computer Representation

The model of reality in terms of potentially large numbers of software processes is transformed in a series of steps to an implementation version that consists of one or more concurrent tasks.

c. Steps in Method

(i) Model Phase

During the modeling phase, the real-world entities are identified. Each entity is defined in terms of the actions (events) it experiences. The attributes of each action experienced by the entity are defined. Furthermore, the attri-

butes of the entity itself are also defined. An entity structure diagram is developed, in which the sequence of actions experienced by the entity is explicitly shown. A software model process is created for each entity and has the same basic structure as the entity.

(ii) Network Phase

During this phase, the communication between processes is defined, function is added to model processes, and function processes are added.

Communication between processes is in the form of data streams of messages or by means of state vector inspections. In the first case, a producer process sends a message to a consumer, whereas in the latter case, a process may read data belonging to another process. A network diagram is developed showing the communication between the model processes.

The functions of the system are considered next. Some simple functions are added to the model processes, providing they can be directly associated with an action experienced by the process. Other independent functions are represented by function processes. Typical function processes are input data collection processes, error handling processes, output processes, and interactive processes. The network diagram is updated to show the function processes and their communication with other function or model processes.

After the network diagram has been established, the timing constraints of the system are considered. Thus, it can be specified that certain system outputs must be generated within a specified time from the arrival of certain inputs.

(iii) Implementation Phase

During the implementation phase, the JSD specification, consisting potentially of a very large numbers of processes, is mapped onto an implementation version that is directly executable. Originally, with the emphasis on data processing, the specification was mapped onto one program using the concept of program inversion [Jackson75]. Each process is transformed into a subroutine; a scheduler (supervisory) routine decides when to call the process routines.

During the implementation phase, JSD specifications can be mapped to real-time designs. Mappings have been defined from JSD to MASCOT subsystems, activities, channels, and pools [Cameron89]. With this approach, there is little or no need for program inversion. Mappings to Ada implementations have also been defined [Cameron89].

d. Products of Design Process

(i) Process Definitions

Given as structure diagrams and structure text. In the case of model processes, this also includes the definition of the entity attributes, as well as each input action and its attributes.

(ii) System Network Diagram

Shows the concurrent model and function processes in the system and their data stream and state vector interfaces.

(iii) System Implementation Diagram

Shows the physical implementation of the system, as well as structure text for the system implementation.

e. Assessment of Method

This assessment is made in terms of how applicable JSD is to the design of real-time systems.

(i) Strengths

- The emphasis on modeling real-world entities is a theme that has since been followed by several of the object oriented analysis and design methods.
- Modeling each real-world entity by emphasizing the sequence of events experienced by the entity is especially relevant in real-time system design.
- Concurrent processing is a central theme of the method.
- Clear steps are provided for mapping a JSD design to an implementation.

(ii) Weaknesses

- Since the entity structure—and consequently the process structure—models the sequence of events in the real-world so faithfully, relatively small changes in the real world can impact the software structure. This could make maintainability more difficult and is a potential hindrance to reuse [Cameron89].
- It is often easier to model event sequences in a complex entity using a state transition diagram than an entity structure diagram. This is particularly the case in real-time systems

where complex event sequences are not unusual, a fact recognized by some real-time system advocates of JSD [Renold88, Sanden89].

- The guidelines for the identification of function processes are rather vague. In many JSD examples [Jackson83, Cameron86, Cameron89], there are substantially more function processes than model processes.
- JSD does not emphasize data abstraction and information hiding. This could have a negative impact on maintainability.

f. Extensions and/or Variations

Sanden [Sanden89] describes a variation on JSD that addresses the needs of real-time systems and also maps directly to Ada. The approach, called Entity-Life Modeling, eliminates the distinction between model and function processes, maps processes directly onto Ada tasks, uses state transition diagrams instead of entity structure diagrams when this is considered desirable, and uses information hiding modules to encapsulate data structures and state vectors.

5. DARTS (Design Approach for Real-Time Systems)

a. Overview

The DARTS design method emphasizes the decomposition of a real-time system into concurrent tasks and defining the interfaces between these tasks. The method originated because of a perceived problem with a frequently used approach for real-time system development. This involves using Structured Analysis, and more recently Real-Time Structured Analysis (RTSA), during the analysis phase, followed by Structured Design during the design phase. The problem with this approach is that it does not take into account the characteristics of real-time systems, which typically consist of several concurrent tasks (processes).

The DARTS design method addresses these issues by providing the decomposition principles and steps for allowing the software designer to proceed from a Real-Time Structured Analysis specification to a design consisting of concurrent tasks. DARTS [Gomaa84, Gomaa86a, Gomaa87] provides a set of task structuring criteria for structuring a real-time system into concurrent tasks, as well as a mechanism for defining the interfaces between tasks. These criteria may be applied to a specification developed using RTSA. Each task, which represents a sequential program, may then be designed using Structured Design.

The DARTS method has evolved over time. Initially, it started with a Structured Analysis specification [Gomaa84]. Later, after the introduction of Real-Time Structured Analysis [Ward85], it was extended to start with a Real-Time Structured Analysis specification [Gomaa87]. A more recent development has been an extension to DARTS called ADARTS, Ada-based Design Approach for Real-Time Systems [Gomaa89b, Gomaa89c], to address structuring a real-time system into concurrent tasks and information hiding modules. Another extension, DARTS/DA [Gomaa89a], deals with structuring a real-time application into distributed real-time subsystems. More information on these extensions is given below.

b. Basic Concepts

(i) Task Structuring Criteria

A set of task structuring criteria are provided to assist the designer in structuring a real-time system into concurrent tasks. These criteria are a set of heuristics derived from experience obtained in the design of concurrent systems. The main consideration in identifying the tasks is the concurrent nature of the functions within the system. In DARTS, the task structuring criteria are applied to the transformations (functions) on the data flow/control flow diagrams developed using Real-Time Structured Analysis. Thus, a function is grouped with other functions into a task based on the temporal sequence in which the functions are executed.

(ii) Task Interfaces

Guidelines are provided for defining the interfaces between concurrent tasks. Task interfaces are in the form of message communication, event synchronization, or information hiding modules (IHMs). Message communication may be either loosely coupled or tightly coupled. Event synchronization is provided in cases where no data are passed between tasks. Access to shared data is provided by means of IHMs.

(iii) Information Hiding

Information hiding is used as a criterion for encapsulating data stores. IHMs are used for hiding the contents and representation of data stores and state transition tables. Where an IHM is accessed by more than one task, the access procedures must synchronize the access to the data.

(iv) Finite State Machines

Finite state machines, in the form of state transition diagrams, are used to define the be-

havioral characteristics of the system. State transition diagrams are an effective tool for showing the different states of the system and the transitions between them.

(v) Evolutionary Prototyping and Incremental Implementation

Evolutionary prototyping and incremental implementation are assisted by the identification of system subsets using event sequence diagrams. These diagrams identify the sequence of execution of tasks and modules that are required to process an external event. System subsets form the basis for incremental development.

c. Steps in Method

(i) Develop Structured System Specification using Real-Time Structured Analysis

The system context diagram and state transition diagrams are developed. The system context diagram is decomposed into hierarchically structured data flow/control flow diagrams. The relationship between the state transition diagrams and the control and data transformations (functions) is established. This step is similar to RTSA steps (i) - (v).

(ii) Structure the System into Concurrent Tasks

The task structuring criteria are applied to the leaf nodes of the hierarchical set of data flow/ control flow diagrams. A preliminary task structure diagram is drawn, showing the tasks identified using the task structuring criteria. I/O transforms that interface to external devices are mapped to asynchronous I/O tasks or periodic I/O tasks. Internal transforms are mapped onto control or periodic tasks and/or may be combined with other transforms according to the sequential, temporal, or functional cohesion criteria.

(iii) Define Task Interfaces

Task interfaces are defined by analyzing the data flow and control flow interfaces between the tasks identified in the previous stage. Data flows between tasks are transformed into either loosely coupled or tightly coupled message interfaces. Control flows are transformed into event signals. Data stores form the basis of information hiding modules.

At this stage, a timing analysis may be performed. Given the required response times to external events, timing budgets are allocated to each task. Event sequence diagrams [Gomaa86a] can help in this analysis by showing the sequence of task execution from external input to system response.

(iv) Design Each Task

Each task represents the execution of a sequential program. Using the Structured Design method, each task is structured into modules. Either transform analysis or transaction analysis is used for this purpose. The function of each module and its interface to other modules are defined. The internals of each module are designed.

d. Products of Design Process

(i) RTSA Specification

See section on RTSA.

(ii) Task Structure Specification

Defines the concurrent tasks in the system. The function of each task and its interface to other tasks are specified.

(iii) Task Decomposition

The decomposition of each task into modules is defined. The function of each module, its interface, and detailed design in PDL, are also defined.

e. Assessment of Method

(i) Strengths

- Emphasizes the decomposition of the system into concurrent tasks and provides criteria for identifying the tasks, an important consideration in real-time system design.
- Provides detailed guidelines for defining the interfaces between tasks.
- Emphasizes the use of state transition diagrams, which is particularly important in the design of real-time control systems.
- Provides a transition from a Real-Time Structured Analysis specification to a real-time design. Real-Time Structured Analysis is probably the most widely used analysis and specification method for real-time systems. Its use is being encouraged by the proliferation of CASE tools supporting the method. However, many designers then find it difficult to proceed to a real-time design. DARTS directly addresses this issue by providing the decomposition principles and steps for allowing the software designer to proceed from a Real-Time Structured Analysis speci-

fication to a design consisting of concurrent tasks.

(ii) Weaknesses

- Although DARTS uses information hiding for encapsulating data stores, it does not use information hiding as extensively as the NRL and OOD methods. Instead, it uses the Structured Design method, not information hiding, for structuring tasks into procedural modules.
- A potential problem in using DARTS is that if the RTSA phase is not done well, this could make task and package structuring more difficult. One of the problems of RTSA is that it does not provide many guidelines as to how to perform a system decomposition. The approach recommended with DARTS is to develop the state transition diagrams before the data flow diagrams, *i.e.*, to pay attention to control considerations before functional considerations.

f. Extensions and/or Variations

(i) DARTS/DA

In large systems, it is usually necessary to structure a system into subsystems before structuring the subsystems into tasks and modules. One approach for structuring a system into subsystems is an extension to DARTS to support distributed real-time applications, called DARTS/DA [Gomaa89a].

(ii) ADARTS

The DARTS weakness in information hiding is addressed by the ADARTS method [Gomaa89b, Gomaa89c]. ADARTS uses the DARTS task structuring criteria for identifying tasks, but it replaces Structured Design with an information hiding module structuring phase in which modules are identified using a set of module structuring criteria. These criteria are based on the Naval Research Laboratory method [Parnas84] module structuring criteria, supported by the object-oriented design [Booch86] object structuring criteria. ADARTS designs may be described using a graphical notation similar to Buhr diagrams [Buhr84].

6. Other Real-Time Software Design Methods

Some other software design methods for real-time systems are briefly reviewed in this section.

The Distributed Computing Design System (DCDS) is an outgrowth of the SREM (Systems Requirements Engineering Methodology) method [Alford85]. DCDS provides a graphical notation for hierarchically decomposing a real-time system design, emphasizing events, as well as both sequential and concurrent functions. With each high-level function, a performance index (*i.e.*, maximum allowed response time) is provided. As the function is hierarchically decomposed, the performance index is divided amongst the lower-level functions. Eventually, at the lowest level of decomposition, sequential and concurrent functions are allocated to the components of the real-time system.

PAMELA [Cherry86] is a software design method that is strongly oriented toward Ada. The method uses a hierarchical decomposition approach, based on data flow diagrams, in which transformations are eventually decomposed into concurrent tasks at the lowest level. The tasks and their interfaces are mapped to Ada.

Some real-time design approaches are actually design notations that suggest a particular approach to performing a decomposition. However, they do not provide the principles and steps for performing a design, and hence are not strictly design methods. Statecharts [Harel88b] are a graphical notation for hierarchically decomposing state transition diagrams. Statemate [Harel88a] is a tool based on statecharts that also includes activity charts and module charts. Statemate can be used to support various specification and design methods. For example, an industrial course is available showing how a Real-Time Structured Analysis [Ward85] specification can be expressed in Statemate.

MASCOT diagrams [Simpson79, Simpson86] are a notation for concurrent systems. This notation has been used in conjunction with JSD, as described in Section III.4 and [Cameron89].

IV. Design Verification and Validation

This section addresses design verification and validation. For more detailed information on software verification and validation, refer to the introduction of [Collofello88b].

1. Software Technical Reviews

The purpose of software technical reviews is to detect errors in software products. Reviews should be carried out throughout the life cycle. Studies have shown that the longer an error goes undetected, the more costly it is to correct [Boehm76]. Technical reviews can be very effective at uncovering design errors. Most design methods do not specifically address reviews. However, software development organizations frequently incorporate design methods and technical review procedures for products of the design process into a software life cycle that is tailored to the organizations' needs.

One design method that also addresses design reviews is the NRL method, which uses a procedure called Active Design Reviews [Parnas85]. With this approach, each reviewer is expected to answer a set of questions about the product under review. Questions are organized by area of expertise.

More information on technical reviews is given in [Collofello88a]. A classic paper on the topic is [Fagan76].

2. Requirements Tracing

Requirements tracing is a means of determining the completeness of a design. This is achieved by checking whether all the software requirements of the system have been incorporated into the design. This is typically carried out using requirements matrices. For checking that the design meets its requirements, the matrix should map each software requirement to one or more design components, such as tasks and/or modules. More information on requirements tracing is given in [Collofello88b].

3. Simulation

Simulation can be an effective way of verifying that the design is sound and that it meets its timing requirements. With this approach, the software system under development, as well as the environment in which it is to operate, are simulated. To be of most value, the simulation should be performed before system development is started. Although much useful information can be obtained from a simulation exercise, simulation models are often very detailed. The time to develop them can therefore be considerable. Care must also be taken to ensure that the assumptions made in the model are realistic.

In many real-time system development projects, environment simulators are used. In this case, the real-time application itself is developed, but the environment in which it is to operate is simulated. This has the advantage of creating a controlled environment that can greatly assist in software regression testing and performance testing [Beizer84, Gomaa86a, Myers79].

4. Prototyping

Agresti [Agresti86] states that "[p]rototyping is the process of building a working model of a system or part of a system. The objective of prototyping is to clarify the characteristics of a product or system by constructing a version that can be exercised." Two main classes of prototypes are throw-away prototypes and evolutionary prototypes [Gomaa86b].

Throw-away prototypes can be used to help clarify user requirements [Agresti86, Gomaa81]. This approach is particularly useful for helping develop the user interface, and it can be used for real-time systems that have a complex user interface. For more detailed information on this topic, refer to [Perlman88]. Throw-away prototypes can also be used for experimental prototyping of the design. They can be used to determine if certain algorithms are logically correct or to determine if they meet their performance goals.

The evolutionary prototyping approach is a form of incremental development, in which the prototype evolves through several intermediate operational systems into the delivered system [McCracken82, Gomaa86b]. This approach can help in determining whether the system meets its performance goals, for testing critical components of the design and for reducing development risk by spreading the implementation over a longer period. Event sequence diagrams may be used to assist in selecting system subsets for each increment [Gomaa86a].

5. Software Testing

Some aspects of software testing of real-time systems are no different than for non–real-time systems. Most differences arise either from the software system's consisting of several concurrent tasks or from its interfacing to external devices.

More information on software testing can be found in [Collofello88b], [Beizer84], and [Myers79].

a. Testing Concurrent Software

A major problem in testing real-time systems—indeed any concurrent system—is that execution of such a system is non-deterministic. An approach for the deterministic testing of concurrent systems is described in [Tai87]. A systematic method for the integration testing of concurrent tasks is described in [Gomaa86a]. A method for analyzing and testing transaction flow through a system is described in [McCabe85].

b. System Testing

System testing is the process of testing an integrated hardware and software system to verify that the system meets its specified requirements [IEEE83]. During system testing, several aspects of a real-time system need to be tested [Beizer84, Myers79]. These include:

- Functional testing to determine that the system performs the functions described in the requirements specification.
- Load (stress) testing to determine whether the system can handle the large and varied workload it is expected to handle when operational.
- Performance testing to test that the system meets its response-time requirements.

System testing of real-time systems can be greatly

assisted by the construction of environment simulators [Gomaa86a, Myers79] that simulate the behavior of the external devices to which the system must interface.

V. Review of Real-Time Software Design Methods

1. Comparison of Real-Time Software Design Methods

In comparing real-time software design methods, the approach taken here is to evaluate how each addresses the three real-time–specific design concepts outlined in section II.3.b, namely, finite state machines for defining the control aspects of a real-time system, concurrent tasks for defining the concurrency in the system, and information hiding for defining modifiable and potentially reusable software components. A fourth criterion is how each handles timing constraints, an important characteristic of real-time systems. A comparison of real-time software design methods is also given in [Kelly87].

a. Support for Finite State Machines

The use of finite state machines is a major consideration in three of the methods, RTSAD, DARTS, and NRL. It is a secondary consideration in OOD. In JSD, a different approach is taken, with event sequences depicted using entity structure diagrams.

The major extension to Structured Analysis for real-time applications is to address the control aspects of a system, primarily through the use of finite state machines. The use of state transition diagrams and tables have been well-integrated into the method through the use of control transformations and specifications.

Finite state machines are also an important feature of the DARTS method, which advocates analyzing the control aspects of the system before the functional aspects. DARTS uses RTSA as a front-end to the design method. Control tasks execute finite state machines, and state transition tables are encapsulated into information hiding modules.

Finite state machines are also an important aspect of the NRL method. A key feature of the specification method is the identification of system modes (super-states) and the transitions between them. In the design phase, each mode transition table is encapsulated in a mode determination module.

In object-oriented design, an object may be defined by means of a finite state machine that is encapsulated within the object. However, OOD does not give as much prominence to finite state machines as the previous three methods.

In JSD, entities in the problem domain are modeled using entity structure diagrams that show the sequence of events experienced by the entity. The regular expression notation used by entity structure diagrams is mathematically equivalent to finite state machine notation. However, for complex entities, where there are comparatively many transitions in relation to the number of states, it is frequently clearer and more concise to use a finite state machine notation, rather than entity structure diagrams.

b. Support for Concurrent Tasks

Although all the methods address concurrent tasks to some extent, there is a wide variation in the emphasis placed on them. Concurrent tasks are fundamental to two of the methods, DARTS and JSD. The NRL and OOD methods place less emphasis on task structuring.

The Ward/Mellor [Ward85] version of RTSAD addresses structuring the system into concurrent tasks, but provides few guidelines for this purpose. Structured Design is a program design method, and hence does not address the issue of task structuring. However, Structured Design can be used for designing individual tasks.

DARTS addresses the weaknesses of RTSAD in the task structuring area by introducing the task structuring criteria for identifying concurrent tasks in the system and by providing guidelines for defining task interfaces.

Concurrent processing plays an important role in JSD, since each external entity is mapped onto a model process. Function processes are then added. Model processes are similar to control tasks in DARTS. In Renold's view, many of the DARTS task structuring criteria are almost equivalent to the criteria for the definition of function processes in JSD [Renold88].

The NRL method views the task (process) structure as an important software structure that is orthogonal to the module and uses structures. However, it provides few guidelines for identifying concurrent tasks.

The OOD method assumes that the same object structuring criteria can be used for identifying tasks (active objects) and information hiding modules (passive objects). This view is contrary to that of the DARTS and NRL methods, which assume that different criteria are required for tasks and modules.

c. Support for Information Hiding

Information hiding is the fundamental underlying principle in two of the methods, NRL and OOD. It is also addressed by the DARTS and RTSAD

methods. Information hiding is not addressed by JSD.

Both the NRL and OOD methods emphasize the structuring of a system into information hiding modules (objects). The NRL module structuring criteria are more comprehensive than those of OOD. In particular, there is a whole category of modules, namely software decision modules, addressed by the NRL method that is not identified in OOD. The NRL method is also more concerned about each module hiding a secret, namely a decision that could change independently. Thus, in the NRL method, a module can hide the details of an algorithm that could potentially change.

Object structuring in OOD does not pay as much attention as the NRL method to each object/module hiding one secret. Thus, an object could hide more than one secret. Consequently, OOD-derived components may not be as modifiable and reusable as NRL-derived components.

RTSAD is weak in the area of information hiding. In its application of information hiding, Structured Design lags behind the NRL method and OOD. Although the concept of informational strength (information hiding) modules was added by Myers [Myers78], the design strategies of transform analysis and transaction analysis do not address information hiding. A designer using this method is liable to arrive at a design that is mainly functional. Because of this, requirements and design changes are likely to have a more severe effect on systems developed using RTSAD.

Although DARTS uses information hiding for encapsulating data stores, it does not use information hiding as extensively as NRL and OOD. It uses the Structured Design method, and not information hiding, for structuring tasks into procedural modules.

d. Timing Constraints

Four of the methods, RTSAD, NRL, DARTS, and JSD, address timing constraints. The required system response times are defined during system specification. During design, the timing requirements for each task are determined. OOD does not specifically address timing constraints.

RTSAD addresses timing constraints during the analysis and design phases. During analysis, the response time specification is developed. This includes response times to external events, sampling times of external inputs, required frequency of periodic output, and response times to user inputs [Hatley88]. During design, the timing requirements of each task are determined. Frequency of task activation and context switching overhead are also considered in arriving at a timing estimate [Ward85].

DARTS uses the RTSA timing specification to allocate time budgets to each task. Event sequence diagrams [Gomaa86a] are used to show the sequence of tasks executed from external input to system response. Percentages of this response time are then allocated to each task in the sequence and to system overhead.

In the NRL method, timing constraints are specified at the requirements stage for periodic and demand functions that generate system outputs. During design, the timing requirements for each process include its deadline and worst case execution time [Faulk88].

In JSD, timing requirements in the form of system responses to external inputs are analyzed with the assistance of the network diagram to determine timing constraints on individual processes involved in generating the response [Jackson83]. This approach is similar to the use of event sequence diagrams in DARTS.

2. Trends in Real-Time Software Design Methods

Many of the trends in software design methods are not specific to real-time systems. The trend most specific to real-time systems relates to the performance analysis of real-time designs.

a. “Eclectic” Design Methods

Greater efforts are likely to be made to incorporate concepts from different design methods and to integrate them to produce “eclectic” design methods. Efforts in this direction can be seen in ADARTS [Gomaa89b, Gomaa89c] and Entity Life Modeling [Sanden89]. ADARTS attempts to integrate task structuring concepts from DARTS with module and object structuring concepts from the NRL and OOD methods. Entity Life Modeling attempts to integrate JSD concepts with information hiding and Ada tasking.

b. Domain Specification and Design Methods

Existing specification and design methods are for the development of specific systems. In the future, domain methods are likely to be developed for specifying and designing families of systems [Parnas79, Lubars87, Prieto-Diaz87]. Individual target systems are then developed by tailoring the domain specifications and designs to the needs of the target system.

c. Computer Support Tools and Software Development Environments

Many existing computer support tools for soft-

ware specification and design methods are little more than graphical editors with some limited capability for checking for consistency amongst different components of a specification or design. Trends in software development environments [Dart87] are in the direction of making them support the entire software life cycle and orienting them toward supporting specific software design methods by incorporating the rules of the design methods.

d. Executable Specifications and Designs

Computer support tools are being developed to allow specifications and designs to be executed, and hence to allow designers to validate their designs. A good example of these tools is Statemate [Harel88a]. Statemate allows a prototype of the system to be developed that describes the functionality and behavior of the system. The approach of developing executable specifications and designs has been termed the operational approach to software development [Zave84].

e. Performance Analysis of Real-Time Designs

Software design methods for real-time systems need to be integrated with performance analysis techniques to allow real-time designers to analyze their designs from a performance perspective. Alternative designs could then be evaluated, and the designer could select the design that best meets the system objectives. One approach for achieving this is to transform the design into a Petri net model [Peterson81] whose performance can be analyzed using timed Petri net modeling techniques [Coolahan83].

Real-time scheduling is an approach that is particularly appropriate for hard real-time systems that have deadlines that must be met [Goodenough89]. With this approach, the real-time design is analyzed to determine whether it can meet its deadlines.

f. Application of Knowledge-Based Techniques

Many design methods use heuristics, such as the DARTS task structuring criteria and the Structured Design module coupling and cohesion criteria. Heuristics are based on designer experience and are "rules of thumb." Because of this, it is usually not possible to incorporate these heuristics into algorithms. However, knowledge based tools could be developed that incorporate rules embodying these heuristics [Tsai88]. By this means, a designer's assistant [Balzer83] could be provided to help the design team during architectural design.

g. Application of Formal Methods

Another trend in software specification and design methods is in the use of formal methods. A formal method uses a formal specification language, *i.e.*, a language with mathematically defined syntax and semantics. A good example of one of the more mature formal methods is the Vienna Development Method, described in [Pedersen89]. VDM has been successfully applied in the areas of programming language semantics and compiler construction.

Formal methods for real-time systems are currently in the research stage. Methods that show promise include temporal logic and Petri net based methods. A computer support tool with a formal basis is Statemate [Harel88a], which is based on finite state machine theory.

Glossary

Abstract data type

A data type defined by the operations that manipulate it, thus hiding its representation details.

Abstraction

A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information [IEEE83].

Behavior hiding module (NRL)

A module that hides the behavior of the system as specified by a function defined in the requirements specification.

Class

A template for objects.

Cohesion (Structured Design)

The degree to which the functions performed by a module are related (adapted from [IEEE83]).

Context diagram (Structured Analysis)

The highest level data flow diagram in a Structured Analysis specification. It is used to define the boundary between the system to be developed and the external environment.

Control flow (Boeing/Hatley Real-Time Structured Analysis)

A binary signal or multi-valued discrete signal.

Control flow diagram (Boeing/Hatley Real-Time Structured Analysis)

A graphic representation showing the control flows between data and control transformations.

Control specification (Boeing/Hatley Real-Time Structured Analysis)

A specification that describes the behavior of the system in terms of decision tables, state transition tables, state transition diagrams, and/or process activation tables.

Control transformation (Ward/Mellor Real-Time Structured Analysis)

A control function that is defined by means of a state transition diagram.

Coupling (Structured Design)

A measure of the interdependence between modules in a computer program [IEEE83].

Data abstraction

Defining a data structure or data type by the set of operations that manipulate it, thus separating and hiding the representation details.

Data dictionary

A collection of the names of all data items used in a software system, together with relevant properties of those items [IEEE83]. Defines the contents of all data flows, event flows, and data stores in the system (Real-Time Structured Analysis).

Data flow (Structured Analysis)

The data that are passed between a source transformation and a destination transformation or to/from the external environment.

Data flow diagram (Structured Analysis)

A graphic representation showing a network of related functions (transformations) and the data interfaces between those functions.

Data store (Structured Analysis)

A repository of data, usually shown on a data flow diagram.

Design method

A systematic approach to creating a design, consisting of the ordered application of a specific collection of tools, techniques, and guidelines [IEEE83].

Device interface module (NRL)

A module that hides the characteristics of an I/O device. Presents an abstract device interface to its users.

Embedded system

A software system that is a component of a larger hardware/software system.

Event flow (Ward/Mellor Real-Time Structured Analysis)

A signal that indicates an event has taken place.

Information hiding

The technique of encapsulating software design decisions in modules in such a way that the module's interface reveals only what its users need to know; thus each module is a "black box" to the other modules in the system (adapted from [IEEE83]).

Information hiding module

A module that is structured according to the information hiding technique. The module hides some data and is accessed by means of access procedures or functions.

Modularity

The extent to which software is composed of discrete components, such that a change to one component has minimal impact on other components [IEEE83].

Module hierarchy (NRL)

A hierarchical classification of information hiding modules.

Object (OOD)

An instance of a class. An object is an information hiding module that contains both data and operations on that data.

Process (concurrent processing)

Same as task.

Process (Structured Analysis)

A function of the system, also called transformation or bubble.

Real-time

Pertaining to the processing of data by a computer in connection with another process outside the computer, according to time requirements

imposed by the outside process. This term is also used to describe systems operating in conversational mode and processes that can be influenced by human intervention while they are in progress [IEEE83].

Reusability

The extent to which software can be used in multiple applications (adapted from [IEEE83]).

Software decision module (NRL)

A module that hides a design decision that is likely to change.

State transition diagram

A diagram that shows the different states of a system or subsystem and the transitions between them

Task (concurrent processing)

A task represents the execution of a sequential program or a sequential component of a concurrent program. Each task deals with a sequential thread of execution—there is no concurrency within a task.

Task structuring criteria (DARTS)

A set of heuristics for assisting a designer in structuring a system into concurrent tasks.

Transaction analysis (Structured Design)

A design strategy used for transforming a data flow diagram into a structure chart whose structure is based on identifying the different transaction types.

Transform analysis (Structured Design)

A design strategy used for transforming a data flow diagram into a structure chart whose emphasis is on input-process-output flow.

Transformation (Structured Analysis)

A function of the system, also called process or bubble.

Teaching Considerations

Textbooks

There is no one textbook that can be used for teaching the material in this module. Several textbooks address specific topics. An introductory textbook on real-time systems is [Allworth87]. [Pressman87] contains overviews of several design methods, including RTSAD, OOD, JSD, and DARTS. There are several books on Structured Analysis and Design. A comprehensive and up-to-date treatment of Structured Analysis is given in [Yourdon89]. In-depth, though different, treatments of Real-Time Structured Analysis and Design are given in [Hatley88] and [Ward85]. A readable version of Structured Design is given in [Page-Jones88]. OOD is covered briefly in [Booch87b]. A different view of OOD is given in [Meyer88]. Several of Parnas's ideas that form the basis of the NRL method are introduced in [Lamb88]. DARTS is described in [Nielsen88]. Ada-oriented design is described in [Buhr84] and [Nielsen88].

Since it is not practical to expect students to purchase or read all these books, the instructor can assemble a collection of papers covering the material described in this module. A suggested collection consists of those papers classified as "essential" in *Classification of References*, below.

Suggested Course Types

The material in this module may be taught in different ways, depending upon the time available and the knowledge level of the students. Possible treatments include:

1. As part of a graduate-level course on design methods, with special emphasis on the design of real-time systems. In this case, the material can be combined with the material in the curriculum module *Introduction to Software Design* [Budgen89].
2. A variation on the above is to survey several of the design methods but to teach one in more detail, such that students can solve a substantial problem using that method.
3. As part of a graduate-level course on real-time systems. In this case, the material can be preceded by other topics in real-time system development, as described in, for example, [Allworth87] or [Buhr84].
4. As an advanced graduate-level course on software design methods for real-time systems that could follow an earlier course serving as an introduction to software design.

For treatments (1), (2), and (4), students should already be familiar with concurrent processing concepts. In (3), concurrent processing concepts can be taught as part of the real-time systems course.

The material in this module has been used by the author in settings (1), (2), and (4).

In the next section, possible syllabi are outlined.

Suggested Schedules

1. Graduate course on design methods, emphasizing real-time systems:
   - Topics I and II in [Budgen89] (14 hours)
   - Topic III in [Budgen89]: Structured Analysis and Design (2 hours); JSP: (2 hours)
   - Topics I and II in this module (6 hours)
   - Topic III in this module: survey of real-time software design methods (3 hours per method = 15 hours)
   - Review of design methods: based on topic IV in [Budgen89] and topic V in this module (3 hours)

   TOTAL TIME: 42 hours
2. Variation on (1), emphasizing one design method: Expand coverage of selected design method from 3 to 9 hours. This could be done by reducing time allotted to each of the other methods by 1 hour.
3. Graduate course on real-time systems:
   - General material on real-time systems from [Allworth87] or [Buhr84], including topic I in this module (18 hours)
   - Topic II in this module (6 hours)

- Topic III in this module: survey of real-time software design methods (3 hours per method = 15 hours)
- Review of design methods: based on topic V in this module (3 hours)

TOTAL TIME: 42 hours

4. Advanced graduate course on software design methods for real-time systems:
   - Topics I and II in this module (6 hours)
   - Topic III in this module: survey of real-time software design methods (6 hours per method: 3 hours lecture, 3 hours student solution presentations and discussion = 30 hours)
   - Topic IV in this module (3 hours)
   - Review of design methods: based on topic V in this module (3 hours)

   TOTAL TIME: 42 HOURS

Worked Examples

It is difficult, if not impossible, to teach this material without worked examples illustrating the different design methods. It is especially instructive if the same problem can be used to illustrate each of the methods. The author has used the problem of designing an automobile cruise control system to illustrate each of the real-time software design methods. These worked examples are included in the support materials package for this module, which is soon to be released.

The suggested approach for using the worked examples is to first present an overview of a given method and then to follow this by illustrating the method applyed to the cruise control problem.

Exercises

As part of any course treating real-time design methods, students should work on one or more real-time problems, either individually or in groups. Whether one or more problems are tackled depends on the size of the problem(s) and the length of the course. However, sufficient time should be allocated for students to work on problems, since this is the best way for them to really understand the methods.

Real-time problems that may be used are:

- Elevator control system [Jackson83, Sanden89]
- Cruise control system
- Buoy system [Booch86]
- Patient monitoring system
- Automated teller machine system
- Flexible manufacturing system

Problem definitions for the cruise control and flexible manufacturing system problems are given in the support materials package.

Possible teaching approaches to the use of these problems are:

a. Work on one problem throughout the semester using one of the methods. This has the advantage that students get an in-depth appreciation of one of the methods. This approach has been used for a relatively complex flexible manufacturing system.

b. Divide the class up into groups. Each group uses a different method to solve the same problem. Time is allocated at the end of the term for each group to present its solution. A class discussion is held on the strengths and weaknesses of each method, as found through students' application of them to the problem.

c. Work on the same problem using each of the methods. This approach has been used with the elevator problem. Class discussions are held after teaching each method, so that students can compare their solutions.

d. Offer a design lab course in the following term, in which the students work in groups to develop a solution to a substantial real-time problem using one of the methods. In this case, students can also begin implementation.

The author has used approaches (a), (c), and (d). Approach (c) is probably the most demanding and should only be used in conjunction with course type (4). Approaches (a) and (d) can be used in conjunction with course types (1), (2), or (4).

Classification of References

In the lists below, the references in the bibliography are classified by subject matter and by applicability. The categories used in the subject matter classification are:

- **General SE:** General references on software engineering
- **Concurrency:** General references on concurrent processing
- **Real-Time:** General references on real-time systems
- **RTSAD:** Real-Time Structured Analysis and Design
- **NRL:** Naval Research Laboratory Software Cost Reduction method
- **OOD:** Object-oriented design
- **JSD:** Jackson System Development
- **DARTS:** Design Approach for Real-Time Systems
- **Modules:** SEI curriculum modules

Categories in the applicability classification are:

- **Essential:** Instructors and students should read these references, which are directly relevant to the material in this module.
- **Detailed:** More detailed references on the topics covered in the module. These references are likely to be of greater interest to instructors, but may also be relevant to students undertaking more detailed investigations into a particular topics.
- **Background:** Background material that can be covered in courses prior to treating the material in this module. This list includes curriculum modules listed in *Philosophy* under "Module Interface."
- **Additional:** References to additional information on topics related to real-time design. These references are useful for getting a broader view of the area and cover other design methods, testing real-time systems, etc.
- **Advanced:** References for instructors or students wishing to get an in-depth view of current research or advanced development topics of interest in the area of real-time systems.

Classification by Subject Matter

General SE

Agresti86
Balzer83
Basili75
Beizer84
Boehm76
Brooks75
Dart87
Davis88
Fagan76
Fairley85
Freeman83
Gomaa81
Gomaa83
Gomaa86b
Harel88b
IEEE83
Lubars87
Martin85
McCabe85
McCracken82
Mills87
Myers79
Parnas72
Pressman87
Prieto-Diaz87
Tai87
Tsai88
Zave84

Concurrency

Bic88
BrinchHansen73
Buhr84
Dijkstra68
Gehani84
Hoare74
Hoare85
Peterson81
Peterson85

Real-Time

Alford85
Allworth87
Cherry86
Coolahan83
Glass83
Goodenough89
Harel88a
Kelly87
Simpson79
Simpson86
Stankovic88

RTSAD

Bruyn88
DeMarco78
Gane79
Hatley88
Myers78
Page-Jones88
Ward85
Ward86
Yourdon79
Yourdon89

NRL

Britton81
Faulk88
Heninger80
Lamb88
Parnas74
Parnas79
Parnas84
Parnas85
Parnas86

OOD

Booch86
Booch87a
Booch87b
Goldberg83
Meyer87
Meyer88
Seidewitz86
Seidewitz88
Shlaer88
Stroustrup86
Wegner87

JSD

Cameron86
Cameron89
Jackson75
Jackson83
Kato87
Renold88
Sanden89

DARTS

Gomaa84
Gomaa86a
Gomaa87
Gomaa89a
Gomaa89b
Gomaa89c
Nielsen88

Modules

Brackett89
Budgen89
Collofello88a
Collofello88b
Pedersen89
Perlman88
Rombach89

Categorization by Applicability

Essential	Detailed	Background	Additional	Advanced
Booch86	Booch87b	Allworth87	Alford85	Agresti86
Cameron86	Britton81	Basili75	Beizer84	Balzer83
Davis88	Bruyn88	Bic88	Cherry86	Coolahan83
Gomaa84	Buhr84	Boehm76	Collofello88a	Dart87
Gomaa86a	Cameron89	Booch87a	Collofello88b	Goodenough89
Gomaa89a	DeMarco78	Brackett89	Fagan76	Harel88b
Gomaa89b	Faulk88	BrinchHansen73	Freeman83	Hoare85
Kelly87	Gane79	Brooks75	Goldberg83	Kato87
Meyer87	Gomaa87	Budgen89	Gomaa83	Lubars87
Parnas79	Gomaa89c	Dijkstra68	Gomaa86b	Peterson81
Parnas84	Hatley88	Fairley85	Harel88a	Prieto-Diaz87
Parnas85	Heninger80	Gehani84	IEEE83	Tai87
Parnas86	Jackson83	Glass83	Jackson75	Tsai88
Renold88	Lamb88	Gomaa81	Martin85	
Sanden89	Meyer88	Hoare74	McCabe85	
Seidewitz88	Myers78	McCracken82	Mills87	
Ward86	Nielsen88	Parnas72	Myers79	
Wegner87	Page-Jones88	Peterson85	Pedersen89	
	Parnas74	Pressman87	Perlman88	
	Seidewitz86	Rombach89	Shlaer88	
	Ward85		Simpson79	
	Yourdon79		Simpson86	
	Yourdon89		Stroustrup86	
			Zave84	

Bibliography

Agresti86

Agresti, W. W. *New Paradigms for Software Development*. Washington, D. C.: IEEE Computer Society Press, 1986.

A very good collection of papers covering critiques of the conventional software life-cycle model, prototyping, operational specification, and transformational implementation.

Good source material for the instructor. Forms an excellent basis for a graduate seminar.

Alford85

Alford, M. "SREM at the Age of Eight: The Distributed Computing Design System." *Computer 18*, 4 (April 1985), 36-46.

Provides a good overview of the DCDS method.

Allworth87

Allworth, S. T., and R. N. Zobel. *Introduction to Real Time Software Design, 2nd Ed*. New York: Springer-Verlag, 1987.

A good introductory book on real-time system design, although much of the discussion is concerned with detailed design issues. Also, good coverage of the MASCOT notation and hardware interfacing issues. Good source material for the instructor and students.

Balzer83

Balzer, R., *et al*. "Software Technology in the 1990's: Using a New Paradigm." *Computer 16*, 11 (Nov. 1983), 30-37.

Advocates a revolutionary paradigm for software development using a transformational approach.

Basili75

Basili, B. R., and A. J. Turner. "Iterative Enhancement: A Practical Technique for Software Development." *IEEE Trans. Software Eng. SE-1*, 4 (Dec. 1975), 390-396.

Abstract: *This paper recommends the "iterative enhancement" technique as a practical means for using a top-down, stepwise refinement approach to software development. This technique begins with a simple initial implementation of a properly chosen (skeletal) subproject which is followed by the gradual enhancement of successive implementations in order to build the full implementation. The development and quantitative analysis of a production compiler for the language SIMPL-T is used to demonstrate that the application of iterative enhancement to software development is practical and efficient, encourages the generation of an easily modifiable product, and facilitates reliability.*

One of the first papers to advocate the incremental development approach to software engineering.

Beizer84

Beizer, B. *Software System Testing and Quality Assurance*. New York: Van Nostrand, 1984.

See comments in [Collofello88b] bibliography.

Bic88

Bic, L., and A. C. Shaw. *The Logical Design of Operating Systems, 2nd Ed*. Englewood Cliffs, N. J.: Prentice-Hall, 1988.

A good reference book on operating systems.

Boehm76

Boehm, B. "Software Engineering." *IEEE Trans. Computers C-25*, 12 (Dec. 1976), 1226-1241.

Abstract: *This paper provides a definition of the term "software engineering" and a survey of the current state of the art and likely future trends in the field. The survey covers the technology available in the various phases of the software life cycle —requirements engineering, design, coding, test, and maintenance—and in the overall area of software management and integrated technology-management approaches. It is oriented primarily toward discussing the domain of applicability of techniques (where and when they work), rather than how they work in detail. To cover the latter, an extensive set of 104 references is provided.*

A classic paper on the waterfall model of the software life cycle.

Booch86

Booch, G. "Object-Oriented Development." *IEEE Trans. Software Eng. SE-12*, 2 (Feb. 1986), 211-221.

Abstract: *Object-oriented development is a partial-lifecycle software development method in which the decomposition of a system is based upon the concept of an object. This method is fundamentally different from traditional functional approaches to design and serves to help manage the complexity of massive software-intensive systems. The paper ex-*

amines the process of object-oriented development as well as the influences upon this approach from advances in abstraction mechanisms, programming languages, and hardware. The concept of an object is central to object-oriented development and so the properties of an object are discussed in detail. The paper concludes with an examination of the mapping of object-oriented techniques to Ada using a design case study.

This paper presents an overview of object-oriented design, as viewed in the Ada world, *i.e.*, with emphasis on information hiding, but not inheritance. The paper outlines how a Structured Analysis specification can be mapped to OOD. The method is illustrated by means of two examples, a cruise control problem and a navigational/weather collection buoy. This paper is also included in [Booch87b].

A good source of material for the instructor and a paper that can reasonably be read by students.

Booch87a

Booch, G. *Software Engineering with Ada, 2nd Ed*. Menlo Park, Calif.: Benjamin/Cummings, 1987.

Describes Ada and its use, with particular emphasis on the features of the language that support large-scale software system development, such as packages, tasks, and generics. It also provides an introduction to a version of object-oriented design that typically can only be readily applied to small programs.

Good source of material for the instructor.

Booch87b

Booch, G. *Software Components with Ada*. Menlo Park, Calif.: Benjamin/Cummings, 1987.

This book presents a large collection of Ada packages that form the basis of a software reuse library. It advocates a "software by composition" approach to software development. Also includes [Booch86].

Brackett89

Brackett, J. W. *Software Requirements.* Curriculum Module SEI-CM-19-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1989.

Capsule Description: *This curriculum module is concerned with the definition of software requirements—the software engineering process of determining what is to be produced—and the products generated in that definition. The process involves all of the following:*

- *requirements identification*
- *requirements analysis*
- *requirements representation*
- *requirements communication*
- *development of acceptance criteria and procedures*

The outcome of requirements definition is a precursor of software design.

BrinchHansen73

Brinch Hansen, P. *Operating System Principles*. Englewood Cliffs, N. J.: Prentice-Hall, 1973.

A classic book on operating systems, although now somewhat dated.

Britton81

Britton, K., R. Parker, and D. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules." *Proc. 5th Intl. Conf. Software Eng.* New York: IEEE, 1981, 195-204.

Abstract: *This paper describes the abstract interface principle and shows how it can be applied in the design of device interface modules. The purpose of this principle is to reduce maintenance costs for embedded real-time software by facilitating the adaptation of the software to altered hardware interfaces. This principle has been applied in the Naval Research Laboratory's redesign of the flight software for the Navy's A-7 aircraft. This paper discusses a design approach based on the abstract interface principle and presents solutions to interesting problems encountered in the A-7 redesign. The specification document for the A-7 device interface modules is available on request; it provides a fully worked out example of the design approach discussed in this paper.*

Describes the application of the information hiding concept to the design of device interface modules.

Good source of material for the instructor.

Brooks75

Brooks, F. *The Mythical Man-Month*. Reading, Mass.: Addison-Wesley, 1975. "Reprinted with corrections" in 1982.

A true classic covering the problems that are frequently encountered in developing and managing large scale software systems, based on the author's experience managing the development of IBM's OS/360 operating system.

This book should be read by all those interested in software engineering.

Bruyn88

Bruyn, W., R. Jensen, D. Keskar, and P. Ward. "ESML: An Extended Systems Modeling

Language." *ACM Software Engineering Notes 13*, 1 (Jan. 1988), 58-67.

> ***Abstract:*** ***ESML (Extended Systems Modeling Language)*** *is a new system modeling language based on the Ward-Mellor and Boeing structured methods techniques, both of which have proposed certain extensions of the DeMarco data flow diagram notation to capture control and timing information. The combined notation has a broad range of mechanisms for describing both combinatorial and sequential control logic.*
>
> This paper presents the basic features of ESML, the recent attempt to merge the Ward/Mellor and Boeing/Hatley approaches to Real-Time Structured Analysis. The ESML method is illustrated by means of a cruise control example.
>
> A good source of material for the instructor.

Budgen89

Budgen, D. *Introduction to Software Design.* Curriculum Module SEI-CM-2-2.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Jan. 1989.

> ***Capsule Description:*** *This curriculum module provides an introduction to the principles and concepts relevant to the design of large programs and systems. It examines the role and context of the design activity as a form of problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations, and main domains of application of these methods.*

Buhr84

Buhr, R. *System Development with Ada*. Englewood Cliffs, N. J.: Prentice-Hall, 1984.

> This book presents a design-oriented introduction to Ada, with special emphasis on concurrent processing. Introduces a graphical design notation—the structure graph—that is gaining widespread acceptance in the Ada community.
>
> A good source of material for the instructor and students, particularly if the orientation of the course is toward Ada.

Cameron86

Cameron, J. "An Overview of JSD." *IEEE Trans. Software Eng. SE-12*, 2 (Feb. 1986), 222-240.

> ***Abstract:*** *The Jackson System Development (JSD) method addresses most of the software lifecycle. JSD specifications consist mainly of a distributed network of processes that communicate by message-passing and by read-only inspection of each other's data. A JSD specification is therefore directly executable, at least in principle. Specifications are developed middle-out from an initial set of "model" processes. The model processes define a set of events, which limit the scope of the system, define its semantics, and form the basis for defining data and outputs. Implementation often involves reconfiguring or transforming the network to run on a smaller number of real or virtual processors. The main phases of JSD are introduced and illustrated by a small example system. The rationale for the approach is also discussed.*
>
> A clear summary of JSD. As the method is still evolving, the steps described are slightly different from [Jackson83]. The method is illustrated by means of a detailed library example. This paper is also included in [Cameron89].
>
> Good source material for the instructor. For a real-time course, a different example would be more appropriate.

Cameron89

Cameron, J., ed. *JSP & JSD: The Jackson Approach to Software Development, 2nd Ed*. Washington, D. C.: IEEE Computer Society Press, 1989.

> A collection of articles and papers describing JSP and JSD and illustrating these methods using a range of examples of reasonable size and complexity. Covers the latest developments in JSD and has some interesting papers on JSD applied to real-time systems, including [Renold88], as well as papers addressing mapping JSD specifications to MASCOT and Ada. Also includes [Cameron86] and a comparison of JSD with OOD.
>
> An excellent source of material for the instructor. Good material for students requiring an in-depth view of JSD.

Cherry86

Cherry G. *The PAMELA Designer's Handbook*. Reston, Va.: Thought Tools, 1986.

> One of the few references on the PAMELA method.

Collofello88a

Collofello, J. *Software Technical Review Process.* Curriculum Module SEI-CM-3-1.5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., June 1988.

> ***Capsule Description:*** *This module consists of a comprehensive examination of the technical review process in the software development and maintenance life cycle. Formal review methodologies are analyzed in detail from the perspective of the review participants, project management and software quality assurance. Sample review agendas are also presented for common types of reviews. The objec-*

tive of the module is to provide the student with the information necessary to plan and execute highly efficient and cost effective technical reviews.

Collofello88b

Collofello, J. *Introduction to Software Verification and Validation.* Curriculum Module SEI-CM-13-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Capsule Description: *Software verification and validation techniques are introduced and their applicability discussed. Approaches to integrating these techniques into comprehensive verification and validation plans are also addressed. This curriculum module provides an overview needed to understand in-depth curriculum modules in the verification and validation area.*

Coolahan83

Coolahan, J., and N. Roussopoulos. "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets." *IEEE Trans. Software Eng. SE-9*, 5 (Sept. 1983), 603-616.

Abstract: *A methodology for the statement of timing requirements is presented for a class of embedded computer systems. The notion of a "time-driven" system is introduced which is formalized using a Petri net model augmented with timing information. Several subclasses of time-driven systems are defined with increasing levels of complexity. By deriving the conditions under which the Petri net model can be proven to be safe in the presence of time, timing requirements for modules in the system can be obtained. Analytical techniques are developed for proving safeness in the presence of time for the net constructions used in the defined subclasses of time-driven systems.*

This paper describes extensions to Petri nets to handle timing requirements for real-time systems.

Dart87

Dart, S., R. Ellison, P. Feiler, and N. Habermann. "Software Development Environments." *Computer 20*, 11 (Nov. 1987), 18-28.

A very good introductory paper on this topic.

Davis88

Davis, A. "A Comparison of Techniques for the Specification of External System Behavior." *Comm. ACM 31*, 9 (Sept. 1988), 1098-1115.

An excellent survey and comparison of different specification techniques. Includes data flow diagrams, finite state machines, Petri nets and statecharts.

An excellent source of material for instructor and students.

DeMarco78

DeMarco, T. *Structured Analysis and System Specification*. Englewood Cliffs, N. J.: Yourdon Press, 1978.

A very popular book on Structured Analysis, although a more up to-date treatment of the subject is given in [Yourdon89].

Dijkstra68

Dijkstra, E. W. "Cooperating Sequential Processes." In *Programming Languages*, F. Genuys, ed. New York: Academic Press, 1968, 43-112.

A classic paper which first introduced the concept of concurrent processes and process synchronization using semaphores. Illustrated by means of several examples.

Good source material for the instructor. However, the concepts have been described in several text books, such as [Bic88] and [Peterson85], which are probably more readable for students.

Fagan76

Fagan, Michael E. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems J. 15*, 3 (1976), 182-211.

See comments in [Collofello88a] bibliography.

Fairley85

Fairley, R. *Software Engineering Concepts*. New York: McGraw-Hill, 1985.

One of the best textbooks on software engineering available. Describes the basic concepts and major issues in the field. Contains a chapter on design that covers fundamental design concepts, design notations, and design methods.

Good source of material for the instructor. Should be read by all software engineering students and is considered a prerequisite to the material in this curriculum module.

Faulk88

Faulk, S. R., and D. L. Parnas. "On Synchronization in Hard Real Time Systems." *Comm. ACM 31*, 3 (March 1988), 274-287.

A detailed description of how concurrent processes are supported in the NRL method.

Excellent source material for the instructor. However, probably rather difficult for students.

Freeman83
Freeman, P., and A. I. Wasserman, eds. *Software Design Techniques, 4th Ed*. Silver Spring, Md.: IEEE Computer Society Press, 1980.

A wide-ranging collection of papers on software design covering basic concepts, analysis and specification, architectural design, detailed design, and management issues.

Very good source material for the instructor and for students who want to get a broad perspective on software design.

Gane79
Gane, C., and T. Sarson. *Structured Systems Analysis: Tools and Techniques*. Englewood Cliffs, N. J.: Prentice-Hall, 1979.

A popular book on Structured Analysis, although a more up-to-date treatment of the subject is given in [Yourdon89].

Gehani84
Gehani, N. *Ada Concurrent Programming*. Englewood Cliffs, N. J.: Prentice-Hall, 1984.

Good book on concurrency in Ada. Several examples are covered, including the multiple readers/ writers problem.

Glass83
Real-Time Software. Glass, R. L., ed. Englewood Cliffs, N. J.: Prentice-Hall, 1983.

An interesting and varied collection of papers and articles on real-time software.

Good source material for the instructor.

Goldberg83
Goldberg, A., and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Reading, Mass.: Addison-Wesley, 1983.

A detailed reference on Smalltalk-80.

Gomaa81
Gomaa, H., and D. B. H. Scott. "Prototyping as a Tool in the Specification of User Requirements." *Proc. 5th Intl. Conf. Software Eng.* New York: IEEE, 1981, 333-339.

Abstract: *One of the major problems in developing new computer applications is specifying the user's requirements such that the requirements specification is correct, complete, and unambiguous. Although prototyping is often considered too expensive, correcting ambiguities and misunderstandings at the specification stage is significantly cheaper than correcting a system after it has gone into production. This paper describes how a prototype was used to help specify the requirements of a computer system to manage and control a semiconductor processing facility. The cost of developing and running the prototype was less than 10% of the total software development cost.*

Describes, with a detailed case study, how prototyping may be used to assist in the requirements specification process.

Gomaa83
Gomaa, H. "The Impact of Rapid Prototyping on Specifying User Requirements." *ACM Software Engineering Notes 8*, 2 (April 1983), 17-28.

Abstract: *Prototyping has been recognized as being a powerful and indeed essential tool in many branches of engineering. Although software prototyping is often considered too expensive, correcting ambiguities and misunderstandings at the requirements specification stage is significantly cheaper than correcting a system after it has gone into production. This paper describes how rapid prototyping impacts the Requirements Analysis and Specification phase of the software life cycle. This is illustrated by describing the experience gained from a prototype used to assist in the requirements specification of a system to manage and control an integrated circuit fabrication facility. The cost of the prototype was less than 10 percent of the total software development cost.*

Describes a prototyping based method for requirements specification and gives an example of its use.

Gomaa84
Gomaa, H. "A Software Design Method for Real Time Systems." *Comm. ACM 27*, 9 (Sept. 1984), 938-949.

This paper describes the DARTS design method and illustrates its use by means of an example of a robot controller system. A later version of the method is given in [Gomaa87]. The task structuring criteria are refined in [Gomaa89b].

Good source of material for instructor and students.

Gomaa86a
Gomaa, H. "Software Development of Real Time Systems." *Comm. ACM 29*, 7 (July 1986), 657-668.

This paper describes how DARTS is used in a software life-cycle context for real-time systems. The paper also describes the use of event sequence diagrams to assist in incremental development.

Good source of material for instructor and students.

Gomaa86b
Gomaa, H. "Prototypes-Keep Them or Throw Them Away?" In *State of the Art Report on Prototyping*, M. E. Lipp, ed. Maidenhead, Berkshire, England: Pergamon Infotech Ltd., 1986, 41-54, 125-126.

Abstract: *This paper describes two different types of software prototype: throw-away prototypes and evolutionary prototypes. The throw-away prototype is a rapid prototype, developed for experimental purposes, and can be used to assist in specifying user requirements—in particular the user interface. The evolutionary prototype is the result of using an incremental development approach. Initially, a subset of the final system is identified and developed, so the prototype is actually an early version of the production system. The paper identifies the main characteristics and benefits of each type of prototype. The impact on the software life-cycle in each case is also described and examples of actual projects which used these approaches, as well as the lessons learned from them, are given.*

This paper points out the differences between throw-away prototyping and evolutionary prototyping and the need for very different approaches when applying these techniques.

Gomaa87
Gomaa, H. "Using the DARTS Software Design Method for Real Time Systems." *Proc. 12th Structured Methods Conf.* Chicago: Structured Techniques Association, Aug. 1987, 76-90.

Abstract: *This paper describes a software design method for real time systems and gives an example of its use. The method is called DARTS, the Design Approach for Real Time Systems. DARTS starts by developing a data flow model of the system using the real time extensions to Structured Analysis. The next stage involves transforming the data flow model into a task structure model defining the concurrent tasks in the system and the interfaces between them. The emphasis of this transformation process is on concurrent processing and data abstraction. Next, each task, which represents a sequential program, is structured into modules using Structured Design.*

This paper describes how DARTS may be used in conjunction with Real-Time Structured Analysis. The robot controller example [Gomaa84] is updated to reflect this.

Gomaa89a
Gomaa, H. "A Software Design Method for Distributed Real-Time Applications." *J. Syst. and Software 9*, 2 (Feb. 1989), 81-94.

Abstract: *This paper describes a software design method for distributed real-time applications that typically consist of several concurrent tasks executing on multiple nodes supported by a local area network. The design method is an extension of DARTS, the design approach for real-time systems, and is called DARTS/DA, DARTS for distributed real-time applications. The method starts by developing a data flow model of the distributed application using structured analysis. The next stage involves decomposing the application into distributed subsystems based on a set of subsystem structuring criteria and defining the interfaces between them. Next, each subsystem is structured into concurrent tasks using the DARTS task structuring criteria and the interfaces between tasks are defined. Finally, each task, which represents a sequential program, is structured into modules using the structured design method. As an example, DARTS/DA is applied to the design of a distributed factory automation system.*

This paper extends DARTS to address the design of distributed real-time applications. The new method, DARTS/DA is illustrated by means of a factory automation example.

Good source material for instructor and students.

Gomaa89b
Gomaa, H. "Structuring Criteria for Real Time System Design." *Proc. 11th Intl. Conf. Software Eng.* Washington, D. C.: IEEE Computer Society Press, May 1989, 290-301.

Abstract: *This paper discusses and compares the criteria used by different design methods for decomposing a real time system into tasks and modules. The criteria considered are coupling, cohesion and information hiding for module structuring and concurrency for tasks. The Structured Design method uses the module coupling and cohesion criteria. The NRL method and Object Oriented Design use information hiding as the primary criterion for identifying modules and objects respectively. The Darts design method uses a set of task structuring criteria for identifying the concurrent tasks in the system. A new design method for real time systems is introduced that uses both task structuring and information hiding module structuring criteria. The method is described and illustrated by mens of an example of an automobile cruise control system.*

Describes the task and module structuring criteria used by different real-time design methods including RTSAD, NRL, OOD, and DARTS. Attempts to blend the task structuring criteria of DARTS with the information module structuring criteria of NRL and OOD into a new method called ADARTS.

Good source material for instructor and students.

Gomaa89c
Gomaa, H. "A Software Design Method for Ada Based Real Time Systems." *Proc. 6th ACM Washington Ada Symposium.* New York: ACM, 1989, 273-284.

Abstract: *This paper describes a software design method for structuring real time systems into concurrent tasks and information hiding packages. The method is called Adarts, an Ada based Design Approach for Real Time Systems. Adarts uses two sets of structuring criteria; task structuring criteria are used to identify the concurrent tasks in the system while package structuring criteria are used to identify the information hiding packages.*

A description of the ADARTS method with particular reference to Ada-based real-time systems.

Of particular interest to Ada-based real-time system design.

Goodenough89
Goodenough, J. B., and C. Sha. *Real-Time Scheduling Theory and Ada.* CMU/SEI-89-TR-14, Software Engineering Institute, Pittsburgh, Pa., 1989.

Abstract: *The Ada tasking model was intended to support the management of concurrency in a priority-driven scheduling environment. In this paper, we review some important results of a priority-based scheduling theory, illustrate its applications with examples, discuss its implications for the Ada tasking model, and suggest workarounds that permit us to implement analytical scheduling algorithms within the existing framework of Ada. This paper is a revision of CMU/SEI-88-TR-33. (The most important revisions affect our discussion of aperiodic tasks and our analysis of how to support the priority ceiling protocol.) A shortened version is also being presented at the 1989 Ada-Europe Conference.*

A readable and informative paper on a complex topic.

Harel88a
Harel, D., *et al.* "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *Proc. 10th Intl. Conf. on Software Eng.* Washington, D. C.: IEEE Computer Society Press, 1988, 396-406.

Abstract: *This paper provides a brief overview of the STATEMATE system, constructed over the past three years by i-Logix Inc., and Ad Cad Ltd. STATEMATE is a graphical working environment, intended for the specification, analysis, design and documentation of large and complex reactive systems, such as real-time embedded systems, control and communication systems, and interactive software. It enables a user to prepare, analyze and debug diagrammatic, yet precise, descriptions of the system under development from three inter-related points of view, capturing* structure, functionality *and* behavior. *These views are represented by three graphical languages, the most intricate of which is the language of* statecharts *used to depict reactive behavior over time. In addition to the use of statecharts, the main novelty of STATEMATE is in the fact that it 'understands' the entire descriptions perfectly, to the point of being able to analyze them for crucial dynamic properties, to carry out rigorous animated executions and simulations of the described system, and to create runing code automatically. These features are invaluable when it comes to the quality and reliability of the final outcome.*

A good overview of the Statemate tool. Also describes how statecharts have been incorporated into Statemate.

Harel88b
Harel, D. "On Visual Formalisms." *Comm. ACM 31*, 5 (May 1988), 514-530.

Abstract: *The higraph, a general kind of diagramming object, forms a visual formalism of topological nature. Higraphs are suited for a wide array of applications to databases, knowledge representation, and, most notably, the behavioural specification of complex concurrent systems using the higraph-based language of statecharts.*

This paper describes a number of important issues concerning design representation. The paper discusses general issues as well as presenting a good introduction to statecharts, illustrated by the digital watch example.

Good source material for instructor and students.

Hatley88
Hatley, Derek J., and I. Pirbhai. *Strategies for Real Time System Specification.* New York: Dorset House, 1988.

A comprehensive description of the Boeing/Hatley approach to Real-Time Structured Analysis. The method is illustrated by means of several examples including the cruise control system and home heating system.

Good source material for the instructor. Probably too detailed for students, unless they are carrying out an in-depth study of the method.

Heninger80
Heninger, K. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications." *IEEE Trans. Software Eng. SE-6,* 1 (Jan. 1980), 2-13.

Abstract: *This paper concerns new techniques for making requirements specifications precise, concise, and easy to check for completeness and consistency. The techniques are well-suited for complex real-time software systems; they were developed to document the requirements of existing flight software for the Navy's A-7 aircraft. The paper outlines the information that belongs in a requirements document and discusses the objectives behind the techniques. Each technique is described and illustrated with examples from the A-7 document. The purpose of the paper is to introduce A-7 document as a model of a disciplined approach to requirements specification; the document is available to anyone who wishes to see a fully worked-out example of the approach.*

An overview of the NRL black-box requirements specification method with examples from the A-7 aircraft project.

Good source material for the instructor. Probably difficult reading for students, however.

Hoare74

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." *Comm. ACM 17*, 10 (Oct. 1974), 549-557.

Abstract: *This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a suitable proof rule. Illustrative examples include a single resource schedule, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.*

A classic paper on operating systems.

Hoare85

Hoare, C. A. R. *Communicating Sequential Processes*. Englewood Cliffs, N. J.: Prentice/Hall International, 1985.

IEEE83

IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. New York: IEEE, 1983. ANSI/IEEE Std 729-1983.

This standard provides definitions for many of the terms used in software engineering.

Jackson75

Jackson, M. A. *Principles of Program Design*. London: Academic Press, 1975.

The original source book on JSP. JSP is also covered in detail in [Cameron89].

Jackson83

Jackson, M. A. *System Development*. Englewood Cliffs, N. J.: Prentice-Hall, 1983.

The original source book on JSD. A more current version of the method is presented in [Cameron86] and [Cameron89]. The book is rather difficult to read, as the description of the method is intertwined with three worked examples. The elevator example has been extracted and included in [Sanden89].

A source of material for the instructor, rather than the student.

Kato87

Kato, J., and Y. Morisawa. "Direct Execution of a JSD Specification." *Proc. COMPSAC 87*. Washington, D. C.: IEEE Computer Society Press, 1987, 30-37.

Abstract: *This paper presents the direct execution of a Jackson System Development (JSD) specification as a part of the Jackson System development Environment (JSE). When we have a tool for executing a JSD specification, we can use it as a rapid prototyping tool of system development. We introduce a language, named the Jackson System development Language (JSL) which is a JSD specification language.*

This paper describes the main part of JSL and explains its interpreter.

Describes a tool to support the execution of JSD specifications.

Kelly87

Kelly, J. "A Comparison of Four Design Methods for Real Time Systems." *Proc. 9th Intl. Conf. Software Eng.* Washington, D. C.: IEEE Computer Society Press, 1987, 238-252.

Abstract: *The purpose of this paper is to compare four design methods which are of current interest in real-time software development. The comparison presents the relative strengths and weakness of each method with additional information on graphic notation and the recommended sequence of steps involved in the use of each method. The methods selected for comparison were:*

- *STRUCTURED DESIGN FOR REAL-TIME SYSTEMS*
- *OBJECT ORIENTED DESIGN*
- *PAMELA (Process Abstraction Method for Embedded Large Applications)*
- *SCR (Software Cost Reduction project - Naval Research Laboratory)*

Readers interested in a framework for comparing methods, an overview of the four selected method-

ologies, and an aid to narrowing candidates for adoption should find this paper helpful

Provides a framework for comparing real-time design methods. Uses this framework to compare RTSAD, PAMELA, OOD, and NRL methods.

Good source material for the instructor and students.

Lamb88

Lamb, David Alex. *Software Engineering: Planning for Change*. Englewood Cliffs, N. J.: Prentice-Hall, 1988.

This book provides a very good overview of many of the ideas of David Parnas that formed the basis of the NRL method.

Good source material for the instructor and students. Chapters 4, 5, and 6 are particularly relevant to this module.

Lubars87

Lubars, M. D., and M. T. Harandi. "Knowledge-Based Software Design Using Design Schemas." *Proc. 9th Intl. Conf. Software Eng.* Washington, D. C.: IEEE Computer Society Press, 1987, 253-262.

Abstract: *Design schemas provide a means for abstracting software designs into broadly reusable components that can be assembled and refined into new software designs. This paper describes a knowledge-based software development paradigm that is based on the design schema representation. It combines design schemas, domain knowledge, and various types of rules to assist in the quick generation of software designs from user specifications. A prototypical environment, IDeA (Intelligent Design Aid), is described that supports the knowledge-based paradigm. The schema-based techniques used in IDeA are presented along with some examples of their use.*

An interesting paper addressing a promising area of research—domain modeling.

Martin85

Martin, J., and C. McClure. *Structured Techniques for Computing*. Englewood Cliffs, N. J.: Prentice-Hall, 1985.

A wide ranging survey of several diagramming techniques and design methods. Compares JSP, Structured Analysis/Design, and the Warnier/Orr method. The book is oriented toward information systems.

McCabe85

McCabe, T., and G. Schulmeyer. "System Testing Aided by Structured Analysis: A Practical Experience." *IEEE Trans. Software Eng. SE-11*, 9 (Sept. 1985), 917-921.

Abstract: *This paper deals with the use of Structured Analysis just prior to system acceptance testing. Specifically, the drawing of data flow diagrams (DFD) was done after integration testing. The DFD's provided a picture of the logical flow through the integrated system for thorough system acceptance testing. System test sets, [*sic*] were derived from the flows in the DFD's. System test repeatability was enhanced by the matrix which flowed from the test sets.*

See comments in [Collofello88b] bibliography.

McCracken82

McCracken, D., and M. Jackson. "Life Cycle Concept Considered Harmful." *ACM Software Engineering Notes 7*, 2 (April 1982), 29-32.

A brief note advocating an evolutionary prototyping approach to software development.

Meyer87

Meyer, B. "Reusability: The Case for Object-Oriented Design." *IEEE Software 4*, 2 (March 1987), 50-64.

An excellent paper describing the benefits of using inheritance in object-oriented design. Illustrated by means of a detailed example of an airline reservation system. The material is covered in more detail in [Meyer88].

Excellent source of material for the instructor. However, students may find the paper difficult and prefer the lengthier treatment given in [Meyer88].

Meyer88

Meyer, B. *Object-Oriented Software Construction*. New York: Prentice-Hall, 1988.

A comprehensive description of designing object-oriented systems using inheritance, in addition to information hiding. Several examples are given using the object-oriented programming language Eiffel.

This book warrants a course of its own on object-oriented software development.

Mills87

Mills, H. D., R. C. Linger, and A. R. Hevner. "Box Structured Information Systems." *IBM Systems J. 26*, 4 (Dec. 1987), 395-413.

A description of the Box-Structured Information System design method.

Myers78

Myers, G. *Composite/Structured Design*. New York: Van Nostrand, 1978.

An early book on the Structured Design method by one of its developers. The book introduces the information hiding concept as a module cohesion criterion, something still not done in later books, *e.g.*, [Page-Jones88].

Myers79

Myers, G. *The Art of Software Testing*. New York: John Wiley, 1979.

See comments in [Collofello88b] bibliography.

Nielsen88

Nielsen, K., and K. Shumate. *Designing Large Real Time Systems with Ada*. New York: McGraw-Hill, 1988.

A detailed book for those interested in developing Ada-based real-time systems. Addresses many Ada-specific issues. The design method is based on DARTS [Gomaa84]. Several detailed case studies are covered, including the robot controller example [Gomaa84] and an air traffic control system.

Good reference material for the instructor. This is a good reference book for a real-time design course oriented toward Ada.

Page-Jones88

Page-Jones, M. *The Practical Guide to Structured Systems Design, 2nd Ed*. Englewood Cliffs, N. J.: Yourdon Press, 1988.

A readable book on the popular Structured Design method. Also has an overview of Structured Analysis. Although recently revived, the book does not cover recent developments in design methods. Unlike [Myers78], it views information hiding as a design heuristic, rather than as a module cohesion criterion, which is probably confusing for both students and practitioners.

A good source of material for the instructor.

Parnas72

Parnas, D. "On the Criteria for Decomposing a System into Modules." *Comm. ACM 15*, 12 (Dec. 1972), 1053-1058.

Abstract: *This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.*

A classic paper that introduces the concept of information hiding as a design criterion.

A good source of material for the instructor.

Parnas74

Parnas, D. "On a 'Buzzword': Hierarchical Structure." *Proc. IFIP Congress 1974.* Amsterdam: North-Holland, 1974, 336-339.

Abstract: *This paper discusses the use of the term "hierarchially structured" to describe the design of operating systems. Although the various uses of this term are often considered to be closely related, close examination of the use of the term shows that it has a number of quite different meanings. For example, one can find two different senses of "hierarchy" in a single operating system [3] and [6]. An understanding of the different meanings of the term is essential, if a designer wishes to apply recent work in Software Engineering and Design Methodology. This paper attempts to provide such an understanding.*

An infrequently referenced paper that describes in detail the interesting view that a software system consists of three orthogonal structures, the information hiding module structure [Parnas84], the uses structure [Parnas79], and the process structure [Faulk88]. A paper that should be read by all system designers, particularly those who believe that the same structuring criteria may be used for tasks and objects.

An essential source of material for the instructor. Students may do better to settle for the instructor's interpretation.

Parnas79

Parnas, D. "Designing Software for Ease of Extension and Contraction." *IEEE Trans. Software Eng. SE-5*, 2 (March 1979), 128-138.

Abstract: *Designing software to be extensible and easily contracted is discussed as a special case of design for change. A number of ways that extension and contraction problems manifest themselves in*

current software are explained. Four steps in the design of software that is more flexible are then discussed. The most critical step is the design of a software structure called the "uses" relation. Some criteria for design decisions are given and illustrated using a small example. It is shown that the identification of minimal subsets and minimal extensions can lead to software that can be tailored to the needs of a broad variety of users.

An important paper that describes the uses structure, a hierarchy of operations provided by modules, and how this structure may be used for determining subsets and extensions of a software system.

A good source of material for the instructor and students, although the example may be difficult to understand.

Parnas84

Parnas, D., P. Clements, and D. Weiss. "The Modular Structure of Complex Systems." *Proc. 7th Intl. Conf. Software Eng.* Long Beach, Calif.: IEEE Computer Society, 1984, 408-416.

Abstract: *This paper discuses the organization of software that is inherently complex because there are very many arbitrary details that must be precisely right for the software to be correct. We show how the software design technique known as information hiding or abstraction can be supplemented by a hierarchically-structured document, which we call a module guide. The guide is intended to allow both designers and maintainers to identify easily the parts of the software that they must understand without reading irrelevant details about other parts of the software. The paper includes an extract from a software module guide to illustrate our proposals.*

A very important paper that describes the application of the information hiding concept to the design of a complex real-time system. Detailed example of the A-7 aircraft.

Essential reading for the instructor and students.

Parnas85

Parnas, D., and D. Weiss. "Active Design Reviews: Principles and Practices." *Proc. 8th Intl. Conf. Software Eng.* Washington, D. C.: IEEE Computer Society Press, 1985, 132-136.

Abstract: *Although many new software design techniques have emerged in the past 15 years, there have been few changes to the procedures for reviewing the designs produced using these techniques. This paper describes an improved technique, based on the following ideas, for reviewing designs.*

1. *The efforts of each reviewer should be focussed on those aspects of the design that suit his experience and expertise.*
2. *The characteristics of the reviewers needed should be explicitly specified before reviewers are selected.*
3. *Reviewers should be asked to make positive assertions about the design rather than simply allowed to point out defects.*
4. *The designers pose questions to the reviewers, rather than vice versa. These questions are posed on a set of questionnaires that requires careful study of some aspect of the design.*
5. *Interaction between designers and reviewers occurs in small meetings involving 2 - 4 people rather than meetings of large groups.*

Illustrations of these ideas drawn from the application of active design reviews to the Naval Research Laboratory's Software Cost Reduction Project are included.

An interesting paper that advocates a highly participatory role by design reviewers.

Parnas86

Parnas, D., and P. Clemens. "A Rational Design Process: How and Why to Fake It." *IEEE Trans. Software Eng. SE-12*, 2 (Feb. 1986), 251-257.

Abstract: *Many have sought a software design process that allows a program to be derived systematically from a precise statement of requirements. This paper proposes that, although we will not succeed in designing a real product in that way, we can produce documentation that makes it appear that the software was designed by such a process. We first describe the ideal process, and the documentation that it requires. We then explain why one should attempt to design according to the ideal process and why one should produce the documentation that would have been produced by that process. We describe the contents of each of the required documents.*

A clear overview of the NRL method that also describes the rationale behind it and stresses the importance of documentation throughout the life cycle. Several aspects of the method are described in more detail in other papers, *e.g.*, [Parnas84].

Essential reading for the instructor and students.

Pedersen89

Pedersen, J. S. *Software Development Using VDM.* Curriculum Module SEI-CM-16-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1989.

Capsule Description: *This module introduces the Vienna Development Method (VDM) approach to software development. The method is oriented toward a formal model view of the software to be developed. The emphasis of the module is on formal specification and systematic development of programs using VDM. A major part of the module deals with the particular specification language (and abstraction mechanisms) used in VDM.*

Perlman88

Perlman, G. *User Interface Development.* Curriculum Module SEI-CM-17-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., April 1988.

Capsule Description: *This module covers the issues, information sources, and methods used in the design, implementation, and evaluation of* user interfaces, *the parts of software systems designed to interact with people. User interface* design *draws on the experiences of designers, current trends in input/output technology, cognitive psychology, human factors (ergonomics) research, guidelines and standards, and on the feedback from evaluating working systems. User interface* implementation *applies modern software development techniques to building user interfaces. User interface* evaluation *can be based on empirical evaluation of working systems or on the predictive evaluation of system design specifications.*

Peterson81

Peterson, J. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, N. J.: Prentice-Hall, 1981.

An excellent reference book on Petri nets, providing a readable treatment of the subject, with many examples.

Peterson85

Peterson, J., and A. Silberschatz. *Operating System Concepts, 2nd Ed*. Reading, Mass.: Addison-Wesley, 1985.

A very good reference book on operating systems.

Pressman87

Pressman, R. *Software Engineering: A Practitioner's Approach, 2nd Ed*. New York: McGraw-Hill, 1987.

A very good introduction to software engineering. Also has chapters on several design methods, including Structured Analysis and Design, DARTS, object-oriented design, and JSD.

A good source of material for the instructor and students.

Prieto-Diaz87

Prieto-Diaz, R. "Domain Analysis for Reusability." *Proc. COMPSAC 87.* Washington, D. C.: IEEE Computer Society Press, 1987, 23-29.

Abstract: *Domain analysis is a knowledge intensive activity for which no methodology or any kind of formalization is yet available. Domain analysis is conducted informally and all reported experiences concentrate on the outcome, not on the process. We propose a model domain analysis process derived from analyzing some domain analysis cases and two existing approaches. After decomposition of the activities analyzed, we were able to capture the domain analysis process in a set of data flow diagrams. The model identifies intermediate activities and workproducts for which support tools can be developed. A project is currently under way to verify our model.*

An interesting research paper that presents an approach to analyzing application domains.

Renold88

Renold, A. "Jackson System Development for Real Time Systems." In *JSP & JSD: The Jackson Approach to Software Development, 2nd Ed.*, J. Cameron, ed. Washington, D. C.: IEEE Computer Society Press, 1989, 235-278.

A good description of how JSD may be used for designing real-time systems. Also includes a comparison of JSD with Structured Analysis/Design and DARTS.

Rombach89

Rombach, D. *Software Specifications: A Framework.* Curriculum Module SEI-CM-11-2.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1989.

Capsule Description: *This curriculum module presents a framework for understanding software product and process specifications. An unusual approach has been chosen in order to be able to address all aspects related to "specification" without confusing the many existing uses of the term. In this module, the term* specification *refers to any plan (or standard) according to which products of some type are constructed or processes of some type are performed, not to the products or processes themselves. In this sense, a specification is itself a product that describes how products of some type should look or how processes of some type should be performed. The framework includes*

- *a reference software life-cycle model and terminology,*
- *a characterizing scheme for software product and process specifications,*

- *guidelines for using the characterization scheme to identify clearly certain life-cycle phases, and*
- *guidelines for using the characterization scheme to select and evaluate specification techniques.*

Sanden89

Sanden, B. "An Entity Life Modeling approach to the Design of Concurrent Software." *Comm. ACM 32*, 3 (March 1989), 330-343.

Describes a variation on JSD that addresses the needs of real-time systems and also maps directly to Ada. Illustrates the method by comparing it to JSD, using Jackson's elevator example [Jackson83].

Seidewitz86

Seidewitz, Ed, and Mike Stark. "Towards a General Object-Oriented Software Development Methodology." *Proc. 1st Intl. Conf. on Ada® Programming Language Applications for the NASA Space Station, vol. II.* Houston: University of Houston-Clear Lake, 1986, D.4.6.1-D.4.6.14.

An early paper on the GOOD method for object-oriented design.

Seidewitz88

Seidewitz, Ed. "General Object-Oriented Software Development: Background and Experience." *Proc. 21st Ann. Hawaii Intl. Conf. System Sciences, vol. II.* Washington, D. C.: IEEE Computer Society Press, 1988, 262-270.

Abstract: *The effective use of Ada™ requires the adoption of modern software-engineering techniques such as object-oriented methodologies. A Goddard Space Flight Center Software Engineering Laboratory Ada pilot project has provided an opportunity for studying object-oriented design in Ada. The project involves the development of a simulation system in Ada in parallel with a similar FORTRAN development. As part of the project, the Ada development team trained and evaluated object-oriented and process-oriented design methodologies for Ada. Finding these methodologies limited in various ways, the team created a general object-oriented development methodology which they applied to the project. This paper discusses some background on the development of the methodology, describes the main principles of the approach and presents some experiences with using the methodology, including a general comparison of the Ada and FORTRAN simulator designs.*

A later paper on the GOOD method. Interesting in its application of entity-relationship modeling to help identify objects in the problem domain.

A good source of material for the instructor and students.

Shlaer88

Shlaer, Sally, and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, N. J.: Yourdon Press, 1988.

A rather narrow view of object-oriented requirements analysis, concentrating on semantic data modeling. However, the treatment given is readable, though somewhat introductory.

Good source of material for the instructor. Probably too narrow for students.

Simpson79

Simpson, H., and K. Jackson. "Process Synchronization in MASCOT." *Computer J. 22*, 4 (Nov. 1979), 332-345.

An early paper on MASCOT, concentrating on the concurrent process synchronization aspects of MASCOT.

Simpson86

Simpson, H. "The MASCOT Method." *Software Eng. J. 1*, 3 (May 1986), 103-120.

A more recent paper on MASCOT that covers the extensions and notation for MASCOT 3.

Stankovic88

Stankovic, J. A., and K. Ramamritham. *Hard Real-Time Systems*. Washington, D. C.: IEEE Computer Society Press, 1988.

A wide-ranging collection of papers covering the specification, design and analysis of real-time systems (with particular emphasis on timing constraints), real-time languages, real time operating systems, architecture and hardware, communication, and fault tolerance.

Good source material for the instructor. Forms an excellent basis for a graduate seminar on this topic.

Stroustrup86

Stroustrup, B. *The C++ Programming Language*. Reading, Mass.: Addison-Wesley, 1986.

A good reference book on this object-oriented language.

Tai87

Tai, Kuo-Chung, and Sanjiv Ahuja. "Reproducible Testing of Communication Software." *Proc. COMPSAC 87.* Washington, D. C.: IEEE Computer Society Press, 1987, 331-337.

Abstract: *Communication software uses timers and constructs such as SEND/RECEIVE and ENQ/DEQ to control synchronization between concurrent processes. As a result, repeated executions of a communication program with the same test sequence may produce different results. This unpredictable program behavior makes the debugging and testing of communication software difficult. The reproducible testing problem is to exercise a given sequence of synchronization events between concurrent processes. In this paper, we present solutions to the reproducible testing problems for SEND/RECEIVE and timers.*

Presents an interesting approach to testing concurrent systems.

Tsai88

Tsai, J. J.-P., and J. C. Ridge. "Intelligent Support for Specifications Transformation." *IEEE Software 5*, 6 (Nov. 1988), 28-36.

Ward85

Ward, P. T., and S. J. Mellor. *Structured Development for Real-Time Systems*. New York: Yourdon Press, 1985-1986. The three volumes in this series are *Introduction and Tools*, *Essential Modeling Techniques*, and *Implementation Modeling Techniques*.

A comprehensive treatment of the Ward/Mellor approach to Real-Time Structured Analysis and Design.

A good source of material for the instructor. Probably too detailed for students, unless they are carrying out an in-depth study of the method.

Ward86

Ward, P. "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing." *IEEE Trans. Software Eng. 12*, 2 (Feb. 1986), 198-210.

Abstract: *The data flow diagram has been extensively used to model the data transformation aspects of proposed systems. However, previous definitions of the data flow diagram have not provided a comprehensive way to represent the interaction between the timing and control aspects of a system and its data transformation behavior. This paper describes an extension of the data flow diagram called the transformation schema. The transformation schema provides a notation and formation rules for building a comprehensive system model, and a set of execution rules to allow prediction of the behavior over time of a system modeled in this way. The notation and formation rules allow depiction of a system as a network of potentially concurrent "centers of activity" (transformations), and of data repositories (stores), linked by communication paths (flows). The execution rules provide a qualitative prediction rather than a quantitative one, describing the acceptance of inputs and the production of outputs by the transformations but not input and output values.*

The transformation schema permits the creation and evaluation of two different types of system models. In the essential (requirements) model, the schema is used to represent a virtual machine with infinite resources. The elements of the schema depict idealized processing and memory components. In the implementation model, the schema is used to represent a real machine with limited resources, and the results of the execution predict the behavior of an implementation of requirements. The transformations of the schema can depict software running on digital processors, hard-wired digital or analog circuits, and so on, and the stores of the schema can depict disk files, tables in memory, and so on.

An overview of RTSA, with some refinement and terminology changes in the notation of [Ward85].

A good source of material for the instructor and students.

Wegner87

Wegner, P. "Dimensions of Object Based Language Design." *Proc. OOPSLA '87*. New York: ACM, 1987, 168-182. Proceedings available as special issue of SIGPLAN *Notices 22*, 12 (Dec. 1987).

Abstract: *The design space of object-based languages is characterized in terms of objects, classes, inheritance, data abstraction, strong typing, concurrency, and persistence. Language classes (paradigms) associated with interesting subsets of these features are identified and language design issues for selected paradigms are examined. Orthogonal dimensions that span the object-oriented design space are related to non-orthogonal features of real languages. The self-referential application of object-oriented methodology to the development of object-based language paradigms is demonstrated.*

Delegation is defined as a generalization of inheritance and design alternatives such as non-strict, multiple, and abstract inheritance are considered. Actors and prototypes are presented as examples of classless (delegation based) languages. Processes are classified by their degree of internal concurrency. The potential inconsistency of object-oriented sharing and distributed autonomy is discussed, suggesting that compromises between sharing and autonomy will be necessary in designing strongly typed object-oriented distributed database languages.

A very interesting paper giving a comprehensive taxonomy of languages supporting objects.

Required reading for the instructor and students who want a clear overview of object-oriented concepts and how they are supported by object-oriented languages.

Yourdon79

Yourdon, E., and L. Constantine. *Structured Design*. Englewood Cliffs, N. J.: Prentice-Hall, 1979.

The classic text on Structured Design, although somewhat dated and not as readable as [Page-Jones88].

Yourdon89

Yourdon, E. *Modern Structured Analysis*. Englewood Cliffs, N. J.: Prentice-Hall, 1989.

Probably the most comprehensive and up-to-date book on the popular Structured Analysis method. Includes material on the real-time extensions to Structured Analysis and Entity-Relationship modeling. There are also two detailed case studies. If you need one book on Structured Analysis, this is probably the one to get.

Very good source material for instructor and students.

Zave84

Zave, P. "The Operational Versus the Conventional Approach to Software Development." *Comm. ACM 27*, 2 (Feb. 1984), 104-118.

This paper advocates an alternative approach to software development in which a problem-oriented executable operational specification is developed, followed by a transformation phase that results in an implementation-oriented specification. A characteristic of the operational specification is that, in order to be executable, it freely interleaves requirements (external behavior) and internal structure.