Chapter 2

Real-Time Specification and Design Techniques

Requirements Engineering Process

The processes used for RE (Requirement Engineering) vary widely depending on the application domain, the people involved and the organisation developing the requirements. The process of establishing what services are required and the constraints on the system's operation and development. Requirements engineering help software engineers to better understand the problem they will work to solve. It encompasses the set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants and how end-users will interact with the software.

- Feasibility Study
- Requirements elicitation and analysis
- Requirements Specification
- Requirements Validation

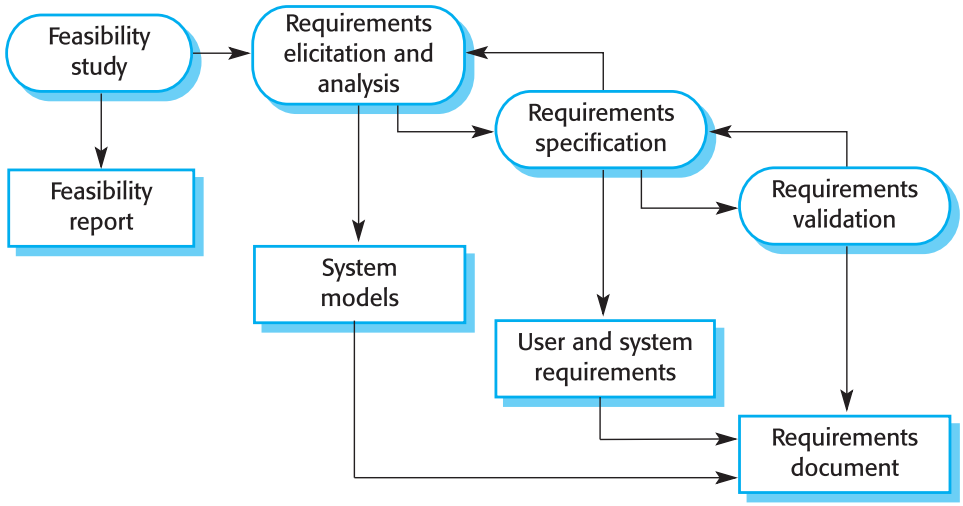

Figure 2.1: The requirements-engineering process depicting documentation in rectangles and activities in smoothed rectangles

A Software Requirements Specification (SRS) – a requirements specification for a software system – is a complete description of the behavior of a system to be developed. It includes a set of use cases that describe all the interactions the users will have with the

software. In addition to use cases, the SRS also contains non-functional requirements (such as performance requirements, quality standards, or design constraints). The software requirement specification document enlists all necessary requirements for project development. To derive the requirements we need to have clear and thorough understanding of the products to be developed.

Types of requirements:

It defines the following kind of requirements:

- Functional
- Non-functional
    - External interfaces
    - Performance
    - Logical database
    - Design constraints (ex: standards compliance)
    - Software system attributes

      Reliability, availability, security, maintainability, portability

2.1 Requirements Specification for Real-Time Systems

There appears to be no particularly dominant approach for specification of real time applications. In general, it seems that real-time systems engineers tend to use one or a combination of the following approaches:

- Top-down process decomposition or structured analysis.
- Object-oriented approaches.
- Program description languages (PDL) or pseudo code.
- High-level functional specifications that are not further decomposed.
- Ad hoc techniques, including simple natural language and mathematical description, and are always included in virtually every system specification.

There are three general classifications of specification techniques:

Formal: Formal methods have a rigorous, mathematical basis.

Informal: If it cannot be completely transliterated into a rigorous mathematical notation and associated rules. For example: flowchart.

Semiformal: Approaches to requirements specification that defy classification as either formal or informal are sometimes called semiformal. For example: UML (Unified Modeling Language) Statechart is formal and other metamodeling techniques it employs have a pseudomathematical basis.

Formal Methods in Software Specification

Formal method attempt to improve requirements formulation and expression by the use and extension of existing mathematical approaches such as propositional logic, predicate calculus and set theory.

Writing formal requirements can often lead to error discovery in the earliest phases of the software life cycle, where they can be corrected quickly and at a low cost.

There are three general uses for formal methods:

Consistency Checking:

This is where system behavioral requirements are described using a mathematically based notation.

Model Checking:

State machines are used to verify whether a given property is satisfied under all conditions.

Theorem Proving:

Here, axioms of system behavior are used to derive a proof that a system will behave in a given way.

Formal methods also offer opportunities for reusing requirements.

The system design document is typically a collection of descriptive, mathematical, procedural, structural or state-based model of the required system. In the following sections we will outline several techniques used to create system specifications and design real-time software.

1. Descriptive Techniques
    a. Natural Language

   b. Pseudocode
   c. Programming Design Lanugages
2. Mathematical Techniques
3. Procedural Techniques
   a. Flowcharts
   b. Data Flow Diagram (DFD)
4. Structural Techniques
   a. Structure Charts
   b. Warnier-Orr Notation
5. State-based Techniques
   a. Finite State Machines
   b. Statecharts
   c. Petri Net

Descriptive Techniques

Natural Language:

The first step in developing a system design to write down a description of what is desired. Either way the form is a descriptive using phrase or sentences outlining the desired result.

Natural languages are the simplest forms of Real time system design. It involves writing the system description in plain English text. Such data are redundant and understood by everyone.

Pseudocode

A more precise descriptive technique is available which greatly reduces the potential for ambiguity. By converting a description into a series of statements and writing them down in a structured way it is possible to create a descriptive form called pseudocode or structured english. This technique adopts a procedural form.

Example of pseudocode for a vending machine:

```
do
     display "Add Coins"
```

```
      wait for coin to be inserted
      identify coin
      count coin value
      if coin value exceeds product cost
      begin
           display "Select Product"
           wait for product selection
           eject product
           calculate change
           eject change
      end
until product dispenser empty
display "No stock"
```

Programming Design Languages (PDLs)

PDL differ slightly from pseudocode, but we will treat them in similar manner. PDL is a type of high-order language that is very abstract; that is, it is detached from the underlying architecture of the machine on which the system will run. PDLs typically can be input into a compiler and translated into high-order languages.

Mathematical Techniques

Most real-time applications are based on the monitoring and/or control of some physical object or process. This presents the design engineer with an ideal opportunity to apply some of that mathematics learnt at university in an attempt to define the physics of the problem as a series of mathematical equations. We use the term attempt because practical systems can be difficult to accurately define as a set of equations, as they are often too complex or exhibit non-linearities.

The concept of using mathematics to define the behavior of a physical system is one of engineering's fundamentals. It provides a concise, commonly understood technique for defining a set of static and dynamic conditions; offers little chance for ambiguity and is in a form which translates easily into software. In addition to these features mathematical

equations can be manipulated to achieve simplifications and optimizations which can greatly improve the performance of real-time software. In many circumstances formal proving of the stability of a system is possible through mathematical analysis.

Mathematical specification is a widely accepted and well understood technique and is recommended for use with real-time systems involved with monitoring and/or control of a physical object or process.

Procedural Techniques

In many cases the design engineer is aware of the processes and procedures that are required to be executed in a real-time system, these include: start-up and shutdown sequences, specific algorithm execution, user data entry, a series of functional decisions, sequences of events and others.

Several design tools are available that help the software engineer to detail procedures which will form an integral part of the real-time system. Some of these techniques have the distinction of being capable of defining increasingly finer levels of detail as the design develops. The following sections outline the most common of these tools.

Flowcharts

Perhaps one of the earliest developed and most widely recognized graphical techniques is the flowchart. Figure 2.2 shows an example of the most commonly used subset of flowcharting symbols. There are actually many other symbols available for flowcharting, but most of them relate to file and record handling for administrative program design.

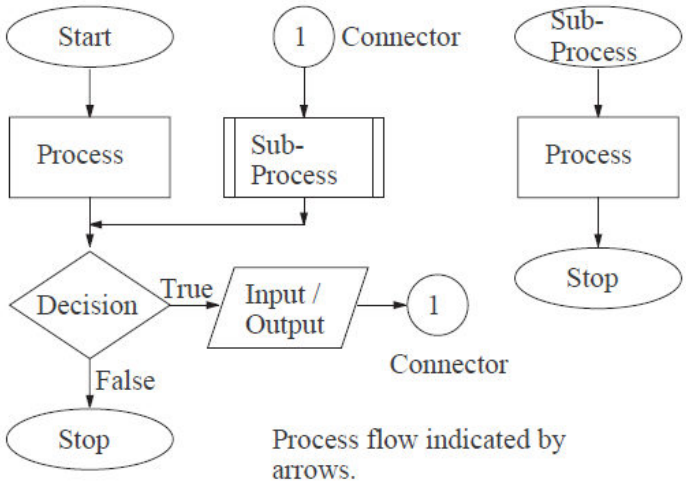

Figure 2.2: Example of the basic set of flowcharting symbols.

The flowchart is designed to show unidirectional program flow, decision points, input/ output, processes and sub-processes. The few basic rules to drawing flowcharts include:

- Each symbol has a maximum of one entry and exit point, except the decision diamond which has two exit points.
- Program flow can only be joined arrow to arrow
- Arrows cannot divide program flow
- Flow should generally be top to bottom

The application of this type of procedural representation is not restricted to software development and is often used to describe all sorts of procedures from changing a tyre to operating a photocopier.

Flow charts are not recommended for use in the specification phase of real-time system design, but can be useful in defining/documenting specific procedures of small parts of a larger system. In multi-tasking systems flowcharts do not easily represent the interaction between the tasks, or between tasks and the operating system. There is also no way of indicating temporal relationships in flowcharting.

Dataflow Diagrams

A dataflow diagram is a simplified system representation showing major flow of data through a series of processes. The four main elements of data flow diagrams are the data source/sink, data storage, processes and data flow arrows. Figure 2.3 shows the symbols used to represent these elements.

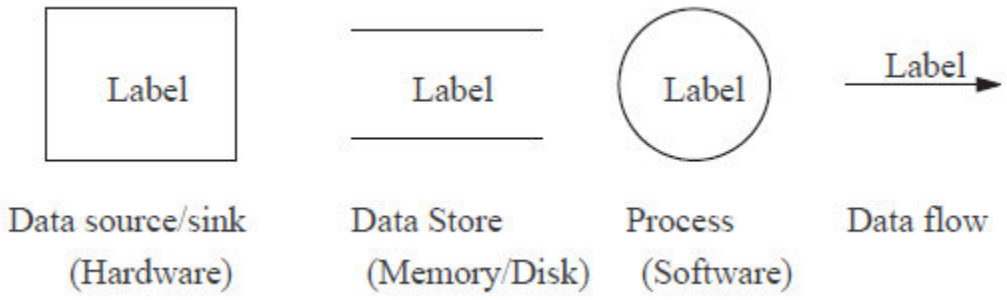

Figure 2.3: Basic symbol for dataflow diagram elements.

Data sources/sinks are typically hardware elements such as peripherals and Input/Output devices. The steps used to create a dataflow diagram include:

- Identify the major data flows based on the system requirements

- Starting on the outer edges draw the sources and sinks for data, usually the system hardware element
- Draw and label arrows indicating data flow between hardware, memory and software modules
- Ensure symbols are clearly labeled according to their function
- Do not show initialization or flow-of-control and keep detail to a minimum

Dataflow diagrams are highly recommended and widely used in the design of real time systems. They offer a structured approach for identifying the main data flows in a system and for partitioning software into modules (processes). Interrupts can be shown as an input from a hardware source that triggers an interrupt service routine. Dataflow diagrams can be used to form a hierarchy of system structure with varying levels of detail. Processes in upper layers can be represented by their own dataflow diagram showing any underlying processes.

Structural Techniques

When adopting an engineering approach to real-time software design it is important to establish a good structural framework around which to build an application. This is why procedures such as top-down design are so successful, because they encourage modularity and hierarchy in a design. Several design tools are available to help the software engineer create well-structured and modular code. The following sections outline a few of these tools.

Structure Charts

Structure Charts are widely used for describing the hierarchical structure of a system.

They can be used to describe, not only software, but any system where a layered structure exists. In software the layers are related to the depth of subroutine and function calls. In other applications the structure chart may depict the hierarchical structure of a chain of command, a written document or the physical components of a device.

The elements which form a structure chart are quite simple: a box represents a process, vertical position indicates hierarchy, lines show links between processes and arrows show major data and control flow. The advantages of structure charts include:

- execution sequence is shown as a left-to-right progression across each layer in the diagram
- they encourage top-down design
- the help identify the modularity of a system

Some variants of the structure chart can be used to illustrate decisions and interrupt processing. Figure 2.4 shows an example of a structure chart including a decision and interrupts.

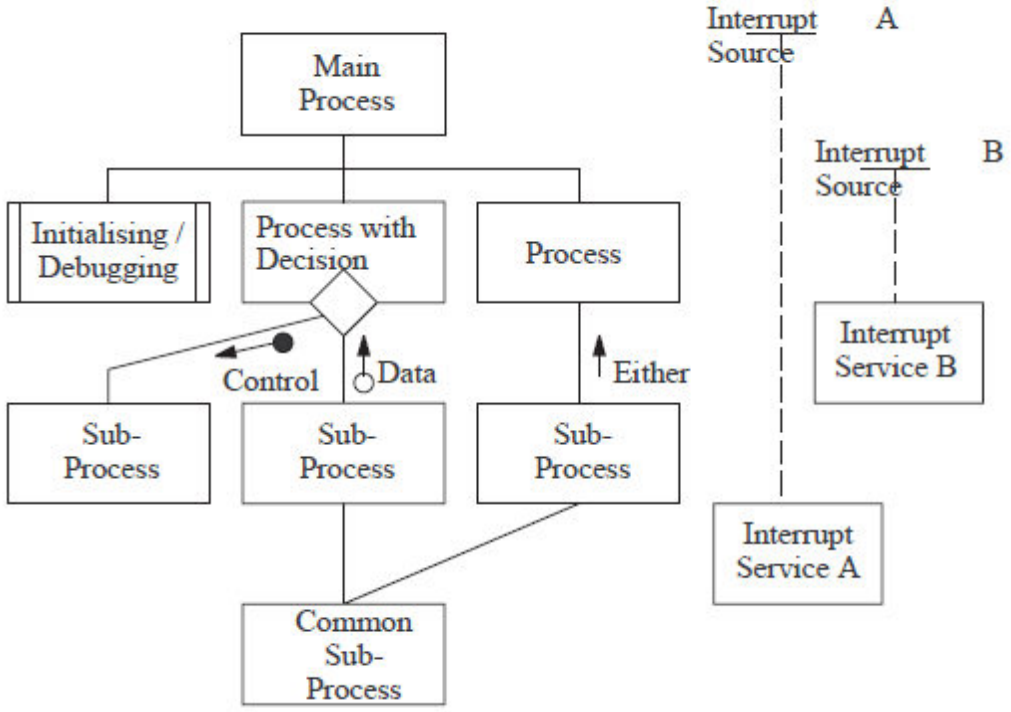

Figure 2.4 Example of the basic set of structure chart symbols.

The position of processes in a structure chart is significant, as it shows the level of depth of subroutine/function call required to reach that module of code, its relationship with processes above and below it and its relative functional level. In the case of interrupt service processes its position and the dashed line above it can be used to indicate the scope of the interrupt. For example, Figure 2.4 shows interrupt service B is only enabled during the execution of the first layer of processes under the main process, while the interrupt service A is capable of interrupting all but the common sub-process.

While structure charts are useful to describe the general structure of a system they lack the ability to depict concurrent processes, significant data storage and temporal

relationships. Thus structure charts are only recommended for use in the initial stages of a design to outline the expected modularity and hierarchy of a system. They may also be used as a good documentation tool to summarize a completed system's structure for later reference.

Warnier-Orr Notation

Warnier-Orr notation is a semi-descriptive, semi-structural system of notation which can represent program structure, data and their conditional relationships. It is somewhat like a structure chart on its side with conditional elements indicating options for execution of sub processes. Warnier-Orr also uses set-theoretic notation to carefully indicate the conditions under which various processes are executed.

Warnier-Orr notation is written top-to-bottom in order of execution and is formed from a combination of sets of steps, a little like psuedo-code. Each set can be made of a combination of other sets and steps. Each step can include a conditional decision which indicates alternate sets for execution. As the notation is written left-to-right increasing levels of detail are introduced. Logical operators of exclusive or (Ex-or) and or (+) are used to indicate mutually exclusive cases and combinational cases respectively. Figure 2.5 illustrates the elements which can be used in Warnier-Orr notation.

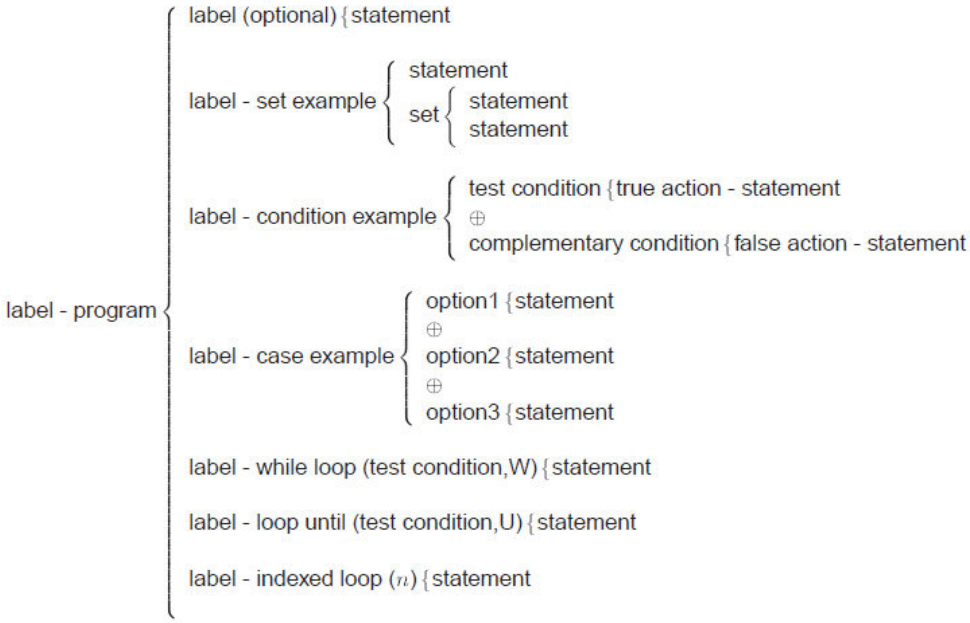

Figure 2.5 Example of Warnier-Orr Notation syntax

In Figure 2.5 each set is identified by a label. In this example they indicate the type of element, but in a real application they should indicate the function or purpose of the element. The elements, in order of appearance include:

- a statement - a statement for execution
- a set - a set of statements or sets
- a condition - requiring a test condition and corresponding true and false actions which are statements or sets
- a case - requiring a series of options and corresponding statements
- a while loop - requiring a test condition, W for a while and statement to be executed while the condition is true
- a loop until - requiring a test condition, U for until and statement to be executed until the condition becomes true
- a indexed loop - requiring a counter n and statement to be executed while *n>0*' which will typically include as statement like n=n-1

Warnier-Orr notation is recommended as an alternative to structure charts as they exhibit the features of modularity, clear sequencing, decision and case capability, looping and counters. Multi-tasking can be incorporated through the inclusion of flags and tests.

State-based Techniques

Many real-time systems have clearly defined conditions or states in which the system operates. The identification of these states allows the design engineer to logically divide a system into distinct operational components. Not only does this method of analysis help partition the software but it also helps identify the system events that trigger changes in system operation. Several design tools are available that use state based techniques to develop well-structured and modular code. The following sections outline a few of these tools.

Finite State Machines

Finite State Machines and Finite State Automata are terms used for the technique of defining a system as a fixed number of unique states between which the system moves in

response to events. A state is identified as a distinct condition a system may occupy, based on system parameters called state variables. Transitions between states are triggered by system inputs (events) or increments of time.

There are actually two types of Finite State Automata, the *Moore* and *Mealy* implementations.

The difference between the two implementations lay in the way that outputs are defined. The Moore machine can only define system outputs in terms of the state variables, where-as the Mealy machine can use input conditions and state variables.

The significance of this distinction will be made clear later when we consider its effect on implementation.

Finite state machines can be represented by mathematical notation, graphically as a State Diagram or State Chart, or in a tabular form. As the graphical and tabular techniques are the most easily applied to software design we shall only consider these as tools for real-time system design.

State Diagrams

The State Diagram is a graphical representation of a finite state machine in which:

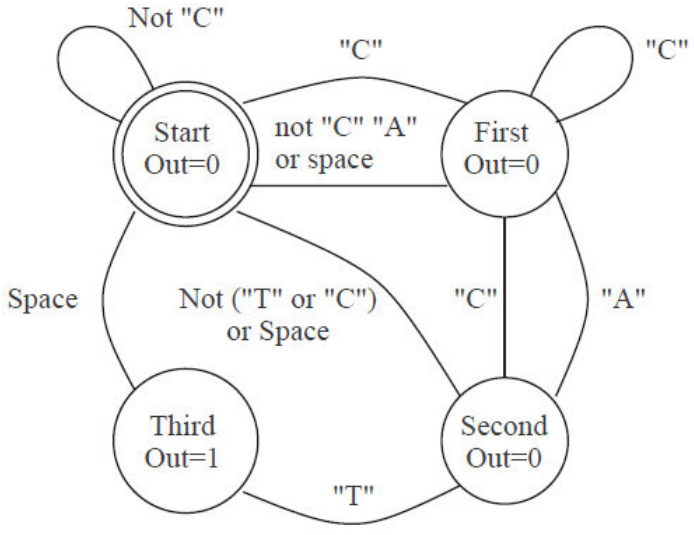

Figure 2.6 Example of a State Diagram (Moore implementation).

- ❖ circles are used to represent system states

- states are typically labeled with capital letters or short descriptions to represent system conditions
- connecting arrows represent transitions between states
- inputs/events are used as labels on transitions indicting trigger conditions
- outputs/actions are either placed inside states or associated with inputs on transitions
- starting or terminating states may be depicted by double circles

Figure 2.6 shows a Moore machine for a simple task to recognize the word CAT from a string of letters. The input is the next letter of the string, the output is a single bit which is1 when CAT is recognized. Note that inputs are shown on the transitions and the outputs are shown inside the states. Recall that a Moore machine's outputs are dependent only on the system state variables and hence are only defined within states. In the Moore machine the outputs are static while the system stays in any individual state.

Figure 2.7 shows the Mealy machine implementation for the same task of Figure 2.6.

Note the outputs are shown on the transitions along with the input causing the transition, separated by a /. In the Mealy machine the outputs are static if based only on state variables or may be transitional if based on input conditions as well.

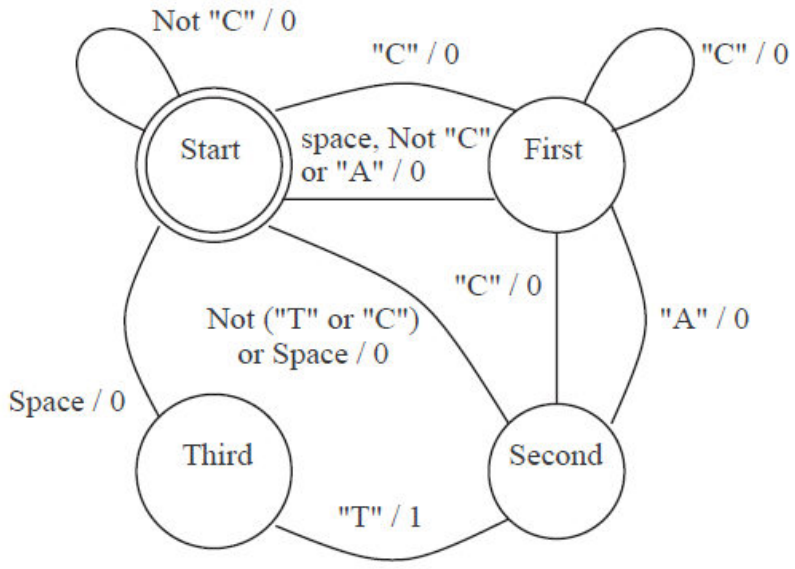

Figure 2.7 Example of a State Diagram (Mealy implementation).

State Diagrams are recommended as a good design technique for real-time systems, particularly for state driven tasks which operate equipment in a range of sequences like traffic light control, medical equipment, aircraft flight control, teller machines etc.

Statecharts

Statecharts are a combination of Finite State Machines and Data Flow Diagrams which feature the ability to depict not only states of operation, but also states within states and orthogonality. The structure of the statechart allows these and other features to be incorporated in the following way.

- States are represented by loops with labels.
- States within states (depth) are represented as loops within loops.
- Orthogonality is represented by a dashed line separating concurrent processes.
- Small letters *a, b…… z* represent events that trigger transitions.
- Small letters in parentheses represent conditions that must be true for the transition to occur.
- Simultaneous transitions in orthogonal states, called *broadcast communications*, are represented by transitions with the same event.
- Outputs are represented as actions associated with states or transitions.
- *Cascade* events can be attached to triggering events.

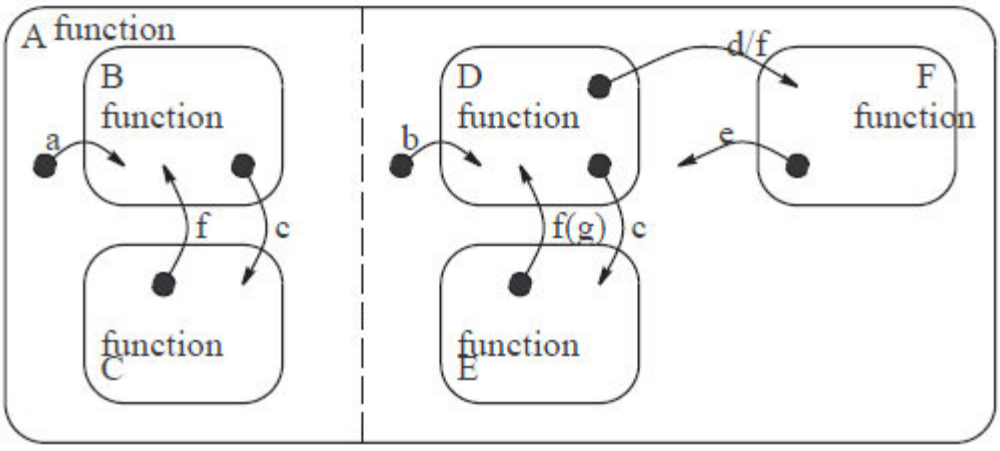

Figure 2.8 Example of a State Chart.

Figure 2.8 shows a sample statechart containing each of the above features. This statechart is comprised of six states A to F, with two orthogonal (concurrent) processes -

one containing states B and C, the other states D, E and F. Each state shows a default label "function" which would be substituted with a descriptive statement of the function of that state. The dynamics of the system are as follows:

- Transition to state B is triggered by event *a*.
- Transition to state D is triggered by event *b*.
- Transition to states C and E are triggered simultaneously (synchronously) by event *c*.
- Control returns to states B and D respectively, triggered by event *f*, with the transition to D conditional on *g*.
- Transition to state F is triggered by event d which causes a cascade trigger of event *f*.
- Control returns to state A from state F on event *e*.

States B to F are said to be nested states of state A, indicating an increasing depth of detail and function. This concept of states-within-states encourages software designers to utilize top-down design principles and create modular code. The concept of depth is similar to sets containing sets in Warnier-Orr notation.

The presence of the dashed line indicates that two processes may be run concurrently, in this case triggered by separate events *a* and *b*. Each process can run independently but some transitions can be synchronized by common events called broadcast communications, such as *c* and *f*.

Statecharts are highly recommended for real-time system design as they offer representations for many of the features required for modern software design, in particular concurrency, modularity and inter task communication. When combined with the state-based decomposition offered by Finite State Machines this design technique is probably the one of the best available.

Petri Nets

- First presented by Carl Adam Petri in his dissertation (1962)
- Petri nets are both a graphical and mathematical modeling tool.
- Can model many different kinds of systems.

- Capable of modeling, concurrency, nondeterminism and asynchronous system.

Example of simple Petri net

- Directed graph and a marking.

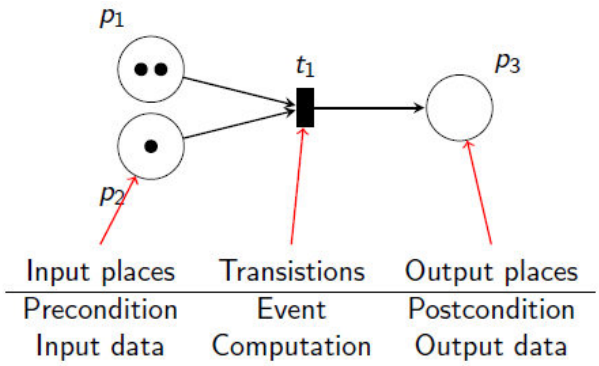

Figure 2.9 Basic symbol of Petri Net

- Petri net consist two types of nodes: *places* and *transitions*. And arc exists only from a place to a transition or from a transition to a place.
- A place may have zero or more *tokens*.
- Graphically, places, transitions, arcs, and tokens are represented respectively by: circles, bars, arrows, and dots.

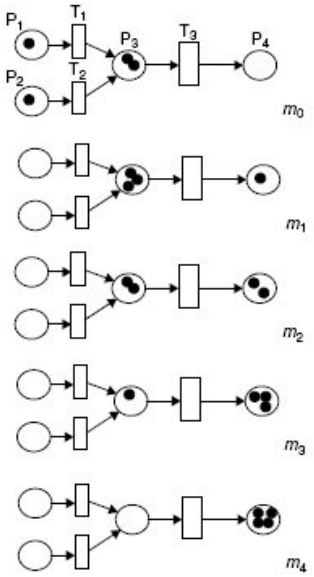

Figure 2.10: Behavior sequence of a slightly more complex Petri net

Firing Table for Petri Net

	P_1	P_2	P_3	P_4
m_0	1	1	2	0
m_1	0	0	3	1
m_2	0	0	2	2
m_3	0	0	1	3
m_4	0	0	0	4

Petri nets can be used to model systems and to analyze timing constraints and race conditions. Certain Petri net sub-networks can model familiar flowchart constructs.

Petri Nets are excellent for representing multiprocessing and multiprogramming systems, especially where the functions are simple. Because they are mathematical in nature, techniques for optimization and formal program proving can be employed. But Petri nets can be overkill if the system is too simple.

Z

Z (pronounced *zed*), introduced in 1982, is a formal specification language that is based on set theory and predicate calculus. As in other algebraic approaches, the final specification in Z is reached by a refinement process starting from the most abstract aspects of the systems. There is a mechanism for system decomposition known as the Schema Calculus. Using this calculus, the system specification is decomposed in smaller pieces called schemas where both static and dynamic aspects of system behavior are described.

Therefore, several extensions for time management have emerged. For example, Z has been integrated with real-time interval logic (RTIL), which provides for an algebraic representation of temporal behavior.

There are other extensions of Z to accommodate the object-oriented approach, which adds formalism for modularity and specification reuse. These extensions, define the system state space as a composition of the state spaces of the individual system objects. Most of these extensions also provide for information hiding, inheritance, polymorphism, and instantiation into the Z Schema Calculus.

2.2 Recommendations on Specification Approach for Real-Time Systems

- Mixing of operational and descriptive specifications.
- Combining low-level hardware functionality and high-level systems and software functionality in the same functional level.
- Omission of timing information.

Real-time system modeling should incorporate the following best practices:

- Use consistent modeling approaches and techniques throughout the specification.
- Separate operational specification from descriptive behavior.
- Use consistent levels of abstraction within models and conformance between levels of refinement across models.
- Model nonfunctional requirements as a part of the specification models, in particular, timing properties.
- Omit hardware and software assignment in the specification.

2.3 Real-Time System Design Activity

The design activity is involved in identifying the components of the software design and their interfaces from the Software Requirements Specification. The principal artifact of this activity is the Software Design Description (SDD).

During the design period, in particular, the real-time systems engineer must design the software architecture, which involves the following tasks:

- Performing hardware/software trade-off analysis.
- Designing interfaces to external components (hardware, software, and user interfaces).
- Designing interfaces between components.
- Making the determination between centralized or distributed processing schemes.
- Determining concurrency of execution.
- Designing control strategies.
- Determining data storage, maintenance, and allocation strategy.
- Designing database structures and handling routines.
- Designing the start-up and shutdown processing.

- Designing algorithms and functional processing.
- Designing error processing and error message handling.
- Conducting performance analyses.
- Specifying physical location of components and data.
- Designing any test software identified in test planning.
- Creating documentation for the system including (if applicable): *Computer System Operator's Manual*, *Software User's Manual*, *Software Programmer's Manual*
- Conducting internal reviews.
- Developing the detailed design for the components identified in the software architecture.
- Developing the test cases and procedures to be used in the formal acceptance testing.
- Documenting the software architecture in the form of the SDD.
- Presenting the design detail information at a formal design review.

Two methodologies, process or procedural-oriented and object-oriented design (OOD), which are related to structured analysis and object-oriented analysis, respectively, can be used to begin to perform the design activities from the Software Requirements Specification produced by either structured analysis or structured design.

Problems with Structured Analysis and Structured Design in Real-Time Applications

There are several apparent problems in using structured analysis and structured design (SASD) to model the real-time systems, including difficulty in modeling time and events. For example, concurrency is not easily depicted in this form of SASD.

Another problem arises in the context diagram. Control flows are not easily translated directly into code because they are hardware dependent. In addition, the control flow does not really make sense since there is no connectivity between portions of it, a condition known as "floating."

Real-Time Extensions of Structured Analysis and Structured Design

It is well known that the standard SASD methodology is not well equipped for dealing with time, as it is a data-oriented and not a control-oriented approach.

Additional tools, such as Mealy finite state machines, are used to represent the encapsulated behavior and process activations. The addition of the new control flows and control stores allow for the creation of a diagram containing only those elements called a control flow diagram (CFD). These CFDs can be decomposed into C-SPECs (control specifications), which can then be described by a finite state machine. The relationship between the control and process models is shown in Figure 2.11.

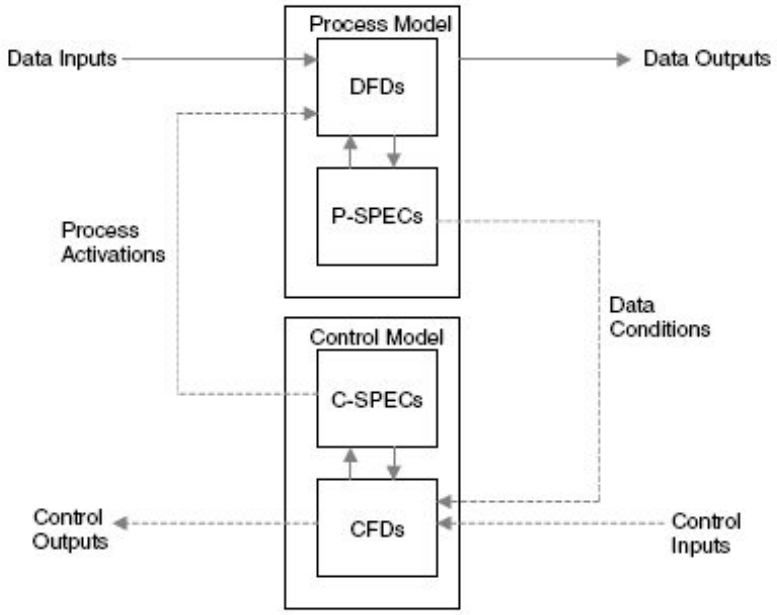

Figure 2.1: The relationship between the control and process model

Real-Time Extensions of Unified Modeling Language

Object-oriented programming languages are those characterized by data abstraction, inheritance, polymorphism and messaging. Data abstraction was defined earlier. Inheritance allows the software engineer to define new objects in terms of previously defined objects so that the new objects "inherit" properties. Function polymorphism allows the programmer to define operations that behave differently, depending on the type of object involved. Messaging allows objects to communicate and invoke the methods that they support.

Object-oriented languages provide a natural environment for information hiding through encapsulation. The state, data, and behavior of objects are encapsulated and accessed only via a published interface or private methods. For example, in the inertial measurement system it would be appropriate to design a class called accelerometer with attributes describing its physical implementation and methods describing its output, compensation, and so forth.

- The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software intensive system.
- The UML is only a language and so is just one part of a software development method.
- Three major elements of model:
  - the UML's basic building blocks,
  - the rules that dictate how those building blocks may be put together,
  - some common mechanisms that apply throughout the UML.

Diagrams in the UML

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram
- Component diagram
- Deployment diagram

Class Diagram

- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.

- These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

Object Diagram

- An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

Use Case Diagram

- A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Sequence and Collaboration Diagrams

- Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. A show an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system.
- A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages; a collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

Statechart Diagram

- A statechart diagram shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system. They are especially important in modeling the behavior of an interface, class, or

collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Activity Diagram

- An activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

Component Diagram

- A component diagram shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

Deployment Diagram

- A deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture. They are related to component diagrams in that a node typically encloses one or more components.