An Introduction to Software Modeling

Things to Look For…

- Unified Modeling Language overview and diagrams.
- Major static modeling diagrams in UML and the utility of each.
- UML class diagrams and use cases.
- Diagrams for expressing intermodule relationships.
- The need for dynamic modeling.
- Major dynamic modeling diagrams in UML and the utility of each.
- UML state, timing, sequence, and activity diagrams.
- The Philosophy behind the Structured Design methods.
- The purpose and utility of the data and control flow diagram.
5.0 Introduction

As we begin our studies of the design and development of the software side of embedded systems, it’s appropriate that we start to learn about some of the tools that can help us with that portion of the job. On the hardware side, we used the Verilog language to model and to analyze the behavior of the modules prior to physical implementation. In this chapter, we will introduce UML – the Unified Modeling Language and several tools taken from the Structured Design approach to system design for that same purpose. A common theme with both approaches to software modeling is the heavy use of graphics as a first step in dealing with the complexity of many contemporary software systems.

We will begin our studies with a brief history of some of the work that led up to the UML. We will then present and discuss the different diagrams that comprise the UML approach. Our initial goal will be to learn techniques by which we can express and model the static structure of a system. We will then work to capture and model its dynamic behavior.

The static view of a system begins from the outside. Such a view is initially captured by seeking to identify and to express how the user (which may be another system or peripheral device) expects to interact with the system. As the system is analyzed and modeled at increasing levels of detail, the comprising modules, their relationships, and their communication paths are identified, defined and included.

Dynamic models capture the behavior of system while it is performing its intended tasks as well as provide information about interactions amongst tasks. Concurrent task operation and persistence are two of the more important dynamic considerations.

In the design of embedded applications, we work with collections of co-operating objects. These objects may be software entities such as tasks or processes or hardware modules such as processors or various peripheral devices. Some of these objects may be active, that is, centers of independent activity, while others may be inactive. Concurrency expresses the ability of a system to handle many such activities simultaneously. It is the property of objects that models parallel operations through an implementation based upon time sharing a single processor or multiple processors.
The *sequential* execution of a set of instructions in a task or process in an embedded application is called a *thread* or *thread of control*. Systems supporting concurrent operation will have multiple threads of control. Some of the threads may be transitory and others may last the lifetime of the system execution. Concurrency focuses on the notions of abstraction, coordination, and synchronization among those threads. Understanding and modeling this aspect of system behavior is essential in the design of multitasking and multiprocessor embedded systems.

A software object takes up space…it exists for a finite period of time. *Persistence* is the property of an object that describes its existence in space and in time. For instance, a temporary variable may only exist during the evaluation of an expression. Local variables exist only while control flow is within their scope then vanish when the scope is exited. Global variables, for example, may have a lifetime that extends beyond their scope. Other variables persist between executions of a program, between versions of a program, or may outlive the program.

In this chapter, our study of the dynamic aspects of the system will focus on variables and on tasks, whose lifetime falls into the first three categories listed above. Persistence, however, is concerned with more than just data lifetime. The state of the object must also be considered. Values must be consistent, particularly in situations such as physically or temporally distributed systems. The type of an object must be considered. In a distributed application, every element of the system must interpret the data in the same way.

We will conclude our introduction of software modeling with a brief look at some of the structured design methodologies. Structured design, which has been around for over 30 years, provides another rich set of tools for attacking the complexities of contemporary designs. In our studies, we will introduce two of the dynamic modeling tools that are useful for conveniently expressing the flow of data and control within a system. Like the UML, a key aspect of the approach is that it is graphical.

### 5.1 An Introduction to UML

The approach that we’ll use to introduce UML will be to bring in the pieces as we need them. In this section, we’ll provide some initial background, terminology, and vocabulary.
As we’ve mentioned earlier, a wide variety of tools are available to help one design, develop, and test software. Each tool has its strengths as well as weaknesses. There are times and places where they should be used and also times when they should not be used.

Part of being creative, part of design is to be able to see things in ways that perhaps they ‘don’t belong’ or were not originally intended. We introduce UML in a software context, the underlying ideas and approach, however, have much broader application as we’ll see when we study more formal design.

UML evolved from the work of a great many people who were looking for better ways to design and develop object-oriented models and systems. By the mid 1990s, the number of credible approaches was reduced to three. Continuing efforts developed and refined these approaches until by 1997 version 1.1 of UML was submitted and accepted by the Object Management Group – OMG. OMG is an international body that defines standards in many areas of computer science. The current version of UML is 1.3 – which, of course, will be outdated by the time this text gets into print.

As its history suggests, UML evolved with the goal of making object centered design easier; consequently, much of the vocabulary and approach centers around objects. None-the-less, we can extend those ideas to a wide variety of both software and hardware applications. Frequently in the ensuing discussions, we will use the words class and object. The intention here is to describe or refer to an abstract entity (or group of entities) rather than either the intrinsic Java or C++ classes.

Many of the tools that we’ll introduce are graphical in nature. We, as human beings, often find it easier to grasp a new concept when it appears as a picture rather than pages of text. The use of graphics, however, does not eliminate the need for clear, concise, and understandable textual descriptions. A good graphic can quickly capture high level concepts; we still rely on text to express the details.

5.2 UML Diagrams

UML uses diagrams and models as a first step towards expressing static and dynamic relationships amongst objects. While an important part of the standard, the authors do not see such diagrams as the main thrust of the approach. Rather, a philosophy of a Model Driven Architecture (MDA) in which UML is used as a programming language is more common. The high level goal
is to create an environment in which tool vendors can develop models that can work with a wide variety of other MDA tools. On the user side, designers who work with UML range from those who are putting together a ‘back of the envelop’ sketch to those who utilize it as a formal (high level) design and programming language. UML provides a very good mechanism for quickly exchanging ideas with other designers and for capturing the critical elements of a design.

The above discussion notwithstanding, the current standard recognizes thirteen different classes of drawings. As a design evolves, these different perspectives offer a rich set of tools whereby we can formulate and analyze potential solutions. Such tools enable one to model several different aspects of a design. It’s rare that all of the types are used in a single design.

The different diagram types are presented in Table 5.0.

<table>
<thead>
<tr>
<th>Class</th>
<th>Use Case</th>
<th>Component</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State Chart</td>
<td>Timing</td>
<td>Sequence</td>
<td>Activity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Package</td>
<td>Composite Structure</td>
<td>Interaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Deployment</td>
</tr>
</tbody>
</table>

As is suggested by their names, diagrams in the first four categories provide the means for a developing a static or structural view. The next four add dynamic analysis; the final five bring the pieces together.

We’ll begin with several of the static components and relationships. These will be sufficient to get us started. We will spend only limited time on the diagram types in the last category.
5.3 Use Cases

The first diagram that we’ll look at is the Use Case. The use case gives us an outside view of the system. It describes the public interface for the module or system. It answers the questions, ‘What is the behavior that the user sees?’ What is the behavior the user expects? The use case repeatedly poses the question, ‘What?’ until the external view of the system has been satisfactorily captured.

The use case diagram is intended to present the main components of the system and how the user interacts with those components. Like many of the diagrams we’ll work with, the use case diagram can be hierarchical in nature. From the top level drawing, one can expand each use case into sub use cases as necessary.

The use case diagram comprises three components, the system, the actor(s), and the use case(s). The meaning of system is self evident; after all, that’s what is being designed. It’s expressed in the diagram as a box – we’ll often leave this off the diagram. The actor(s), drawn as simple stick figures, represent any one or any thing that might be using the system. They are viewed as being outside of the system. The use cases, represented as a solid oval, identify the various behaviors of the system or ways that it might be used. They encapsulate the events or actions that must occur to implement the intended behavior of the system and are stated or expressed from the point of view of the user. Accompanying each use case is a textual component fully describing it. Use case diagrams can be a very powerful tool during the early stages of a project when one is trying to identify, define, and capture the requirements for the system.

As we construct the diagram, we place the actor that executes the use case on the left hand side. Supporting actors appear on the right hand side. Supporting actors are not restricted to human users; an actor can be a computer or other system as well. The set of use cases appears in the center of the drawing with arrows indicating the actors involved in the use case.
A generic use case diagram given as seen in Figure 5.0,

![UML Use Case Diagram](image)

We see that the system comprises three use cases. Actor0 is using the system and appears on the left hand side. Actor1 is supporting UseCase2 and is placed on the right hand side.

It’s important to remember to keep things simple when putting the use case diagram together. If a system being designed is showing twenty five to fifty use cases on the top level drawing, then it’s time to rethink the design.

In this next example, we are working on a simple data acquisition system.

**Example 5.0**

A basic data acquisition system that has the ability to measure voltage and temperature is to be designed. The use case diagram for the system begins with the user shown as Actor0. After the data has been collected, it is to be analyzed for trends, alarm conditions, or other specific patterns. In addition, because the temperature sensor is a nonlinear device, a linearization operation must be performed.

The data is to be collected at very high speed from a number of measurement points; as a result, hardware co-processing capability is probably going to be necessary. That entity is included as a second actor and

![Use Case Diagram for a Simple Data Acquisition System](image)
labeled as *Data Processor*. A possible use case diagram for the data acquisition system is given in Figure 5.1.

**Writing a Use Case**

The use case diagram captures a graphical representation of the public interface to the module or system. Associated with each use case is a textual description of what actions the actor is to perform and how the system is expected to respond. Such a description can be decomposed into two pieces: the *normal activity* of the use case and how *exceptional conditions* are to be handled.

Let’s examine the *measure volts* use case for the data acquisition system. We specify how the user is to select the task, any options associated with the task, and how exceptions are handled as is done in Figure 5.2. Do not forget, a use case description is not intended to be *War and Peace*.

![Use Case Diagram](image)

```
User
   Select measure volts mode
   Select measurement range or autorange

System
   If range specified
   Configure to specified gain
   Make measurement
   If in range – display results
   If exceed range – display largest value for range and flash the display
   If auto range
   Configure to midrange gain
   Make measurement
   If in range – display results
   If above or below range adjust gain to next range and repeat the measurement
   If exceed range – display largest value for range and flash the display
```

Figure 5.2  
Writing a Use Case Description

---

**5.4 Class Diagrams**

Once we have identified how the user intends or expects to interact with the system, the next step is to begin to identify and to formulate to modules that give rise to that external behavior. That process begins with the *class diagram*. This diagram gives a description of the objects in a system coupled with the relationships that exist amongst them. Such a description is frequently found
among the foundation elements of most modeling tools. The class diagram enables one to specify
the *public interface* to the object; the interface expressed in the use cases. Such a description
includes the properties and the operations that instances of the object can perform and identifies
any constraints the application imposes on those operations. The public interface should always
be one of the earlier views that one takes of any design. We want to see the design from our user’s
point of view.

The class diagram presents the various kinds of objects in the system
and identifies the relationships called *associations* amongst them.
Objects are expressed as a rectangle, subdivided into three areas as
illustrated in the diagram in Figure 5.3. The top area gives the *name*
of the class or object. The middle section identifies all of the *proper-
ties* of the object. These will generally be declared inside the module
implementation and thereby hidden from the casual user. The third pane identifies the *operations*
that the object is intended to perform. These operations establish the external behavior of the
object; they provide the public interface to the object.

The *properties* of an object provide a mechanism to capture the structural features of that object.
A property may be further elaborated as *attributes* or *associations*. The former describe a particu-
lar characteristic of a property such as the address of an output port. The latter capture how the
object relates to other objects within the system. A property can be quantified by a *multiplicity*
attribute identifying how many objects may fill the property. The wheel on an automobile has a
*multiplicity* of 4.

**Class Relationships**

We can define number of different relationships among classes. These include

- Parent – Child or inheritance or generalization
- Containment or aggregation
5.4.1 Inheritance or Generalization

We express inheritance using a solid line terminating in a hollow arrow. The diagram in Figure 5.4 presents a portion of the design of an external world communications interface in an embedded system. Therein, we represent the relationship between the parent – Driver and two children – Serial and Parallel. We say that Serial or Parallel are a kind of (AKO) Driver.

The diagram captures the requirement (through the parent interface) that each of the different types of interface must support a common subset of capabilities. Specifically, there must be a port number associated with each interface, the driver must provide the address to an I/O buffer, it must manage a status flag, and it must implement the read() and write() functions to execute the transfer. The ‘+’ appearing in the diagram indicates that each of the corresponding elements is publically visible.

While we generally think of inheritance as supported by the Java or C++ languages; the concept naturally applies as we begin the design of an I/O interface and its associated drivers. It seems reasonable that there should be a common way of communicating with each driver. The C language does not formally support inheritance; however, such a limitation should not preclude using the concept as a hardware or software design tool.

5.4.2 Interface

An interface is a wrapper around one piece of functionality that allows us to present a different set of capabilities as a public view. We express an interface in a manner that is similar to that which we use for inheritance. We use a dashed line terminating in hollow arrow.

Here, in Figure 5.5, we illustrate the concept of an interface with a standard laboratory instrument. We’ll apply the same concept shortly when we are working with several different data structures.

In the diagram, the interface measurePressure, gives the underlying voltmeter the public appearance of a pressure meter. The hidden operation, convert(), performs the...
necessary math to transform the raw voltage reading from a transducer into the corresponding and proper pressure reading.

5.4.3 Containment

Containment conveys the idea that one object is made up of several others, that is, a whole – part relationship. Under UML, we can express two different forms of containment, aggregation and composition.

5.4.3.1 Aggregation

Let's look first at aggregation which expresses a whole – part relationship in which one object or module contains another module. A key characteristic of an aggregation is that the owned module may be shared with other modules outside of the aggregation. Under such conditions, rules must be established to ensure proper management of the shared module.

Continuing with the voltmeter system, let's assume one of the design requirements specifies that, in addition to executing pressure measurements, it must also perform several different kinds of analyses on the data that is collects. In partial support of such analysis, we design an algorithm that performs a series of statistical computations such as trend, mean, limits test, or rate of change, on the collected data.

To perform the necessary computations, the algorithm utilizes a number of different library functions. While the individual functions may be collected under the umbrella of the analysis package in the design, they can exist without that module and certainly could be used by other modules within the system as well.

The statistical analysis algorithm is an aggregation of many specific algorithms. The UML diagram for the aggregation relationship, shown in Figure 5.6, presents both the whole and its parts connected via a solid line that originates at an open diamond on the end associated with the whole and terminates on the end associated with the part.
5.4.3.2 Composition

The composition relationship is similar to aggregation. However, the notion of ownership of the parts by the whole is much stronger. The elements of the composition cannot be part of another object and, unlike the aggregation relationship, they cannot exist outside of the whole object. While this may sound a little strange, at the core of the issue is the proper management of memory. The idea is loosely analogous to local variables in a function. Once one leaves the scope of the function, the local variables disappear.

In an embedded system, we often build an application as a collection of tasks. Each of these tasks executes according to a designated schedule. The schedule is made up of a number of intervals. Without the schedule, the intervals have no meaning. We express such a relationship in a composition diagram as shown in Figure 5.7.

The schedule is composed of 1 to n intervals. Observe that the diagram is similar to that for the aggregation. The connecting line now originates in a solid rather than open diamond. We annotate the relationship as a 1 to n composition.

5.5 Dynamic Modeling with UML

Dynamic modeling provides the means to capture, to understand, and to design the intended behavior of a system. The static structure gives an architecture; the dynamic aspects of the design get the real work done. Important elements of a dynamic model include,

- Inter module interaction and communication,
- Ensuring the proper order of task execution,
- Understanding what activities can be done in parallel,
- Selecting alternate paths of execution,
- Identifying which tasks are active and when they are not

In the next several sections, we will study UML diagrams that will enable us to explore, to express, and to make trade-offs on these elements of a design.
5.6 Interaction Diagrams

The first diagram that we will study is the *interaction diagram*. For embedded design, understanding and modeling the dynamic behavior of the system is essential. Dynamic behavior gives information about the lifetime of a task, identifies when that task is active or inactive and models interactions amongst tasks. Such interaction often takes form of messages. A message is a means of communication between two or more tasks. It can take several forms,

- Event
- Rendezvous
- Message

Generally the receipt of a message results in the initiation of one or more actions. Such actions are executable functions within the task and result in a change in the values of one or more attributes associated with the task.

UML explicitly supports five kinds of actions,

- **Call and Return**
  The *call action* invokes a method on an object and the *return action* returns a value in response to call.
- **Create and Destroy**
  The *create action* creates an object; the *destroy action* does opposite
- **Send**
  The *send action* sends a signal to an object.

Each of these actions is directly applicable to later work with tasks. These actions are shown in the following diagrams. The dashed line emanating from each object or class is called a *lifeline*. The lifeline captures the notion of the persistence of the object.
5.6.1 Call and Return

A call action is expressed by a solid arrow from the calling object to the receiving object and the return action by a dashed, open arrow from the receiving object to the calling object. Such an interaction is shown in Figure 5.8.

![Figure 5.8: The Call and Return Interaction Diagram](image)

5.6.2 Create and Destroy

The create action is represented by a solid arrow from the creating object to the created class instance and the destroy action by a solid arrow from the destroying object to the destroyed class instance. This relationship is presented in Figure 5.9.

![Figure 5.9: The Create and Destroy Interaction Diagram](image)
5.6.3  Send

The *send action* is captured by a solid arrow with an open half arrow head from the sending task to the receiving task as seen in Figure 5.10.

The sender does not expect a response.

5.7  Sequence Diagrams

The purpose of a *sequence diagram* is to express the temporal ordering of a series of message exchanges between objects. The diagram comprises the four principal components presented in Figure 5.11.

- **Objects**
  
  *Objects* appear along the top margin of the diagram as they did in the interaction diagrams. In our designs, these will be the tasks.

- **Lifeline**
  
  The *lifeline*, drawn as a dashed line leaving the object, captures the notion of object persistence.

- **Focus of Control**
  
  The *focus of control* reflects the durations in the object's life during which it is considered to be active. It's expressed as a thin rectangular box that straddles the object's lifeline and indicates the time during which object is in control of the flow, that is, executing a method or creating another task. This is the time when a task has the CPU.

- **Messages**
  
  The *messages* show the actions that objects perform either on themselves or on each other.

The drawing in Figure 5.12 gives a *sequence diagram* for making, converting, and displaying a time interval measurement in a simple counter design. The initial selection of the specific function spawns the *measure* task. The measure task retrieves the range and measurement edge information from an internal buffer and sends these to the *execute measurement* task. The execute task
returns the raw reading to the measure task which spawns the convert task to process the raw reading into a format that can be displayed. The convert task will also perform the bounds check on the reading and return the bounds exceeded value if necessary. Finally, the measure task sends the measurement to the display task which presents it to the user via the front panel display.

5.8 Fork and Join

When working with a multitasking embedded system, a common sequence of operations is for a parent process to start then spawn several child tasks to do the real work. The child tasks complete their jobs and terminate thence the parent class follows. The process of splitting the control flow into two or more flows of control or subtasks is called a fork. Each subtask represents a separate, independent thread of control. When the subtasks are brought back together or resynchronized, it is called a join.
Such behavior of tasks and subtasks is modeled using a *fork and join* diagram as reflected in Figure 5.13.

The tasks are represented by a cartouche or rounded rectangle. Sequential flow is given by a solid arrow. Forks and joins are represented by thick bar or rectangle called a *synchronization bar*. The fork occurs after the first parent activity or action completes. Following such an action, we see that the task spawns subtasks then suspends itself until subtasks complete. Once all subtasks have completed, the join occurs, and the parent task resumes its activities.

In the diagram above, the parent spawns two child tasks. One child performs its task and completes; the second similarly finishes its task, then, spawns a second. When all activity completes, the child tasks terminate and the parent continues.

### 5.9 Branch and Merge

Another form of flow of control is the *branch* in which the thread of execution is determined by the value of some control variable. Such a structure permits one to model alternate threads of execution. A *merge* brings the flow back together again. Each is represented by the diamond symbol that is commonly found in the familiar flow chart. Sequential flow is shown by a solid arrow and individual tasks or activities are shown using a rounded rectangle.

A simple diagram with two alternate paths of execution for a portion of the overall task is given in the diagram in Figure 5.14.

Following the completion of the activities in the right hand path, the flow of control merges back to a single path. At each branch point one can associate a
guard condition to stipulate under what conditions the branch is to be taken. The guard condition is shown in square brackets on the transition arrow.

5.10 Activity Diagram

An activity diagram permits the capture of all of the procedural actions or flows of control within a task. Such actions may be a branch and merge, a fork and join, or a simple transition from state to state.

The initial node in the diagram is given by a solid black circle; the final node is a solid black circle surrounded by a second circle. The accompanying diagram in Figure 5.15 shows how we might combine our earlier activities into a larger task. Conversely, one can show how a larger task is decomposed into its components.

5.11 State Chart Diagrams

The state chart diagram, like the familiar state diagram, finds its roots in the mathematics of graph theory. Using the diagram, we can begin to capture and to model the state behavior of the (software) system as well as the myriad external and internal events that are affecting that behavior.

5.11.1 Events

Any embedded application must interact with world around it. The system will accept inputs and produce outputs. Inputs generally result in some associated action and the actions may or may not lead to an output. Such inputs, outputs, and actions are referred to by various names. Under the UML umbrella, they are collected under name events. An event is any occurrence of interest to the system; more specifically and typically to one of the tasks in the system.
UML supports the 4 kinds of events given in Figure 5.16,

- **Signal**
  A signal is an asynchronous exchange between tasks.
- **Call Event**
  A call event is a synchronous communication that involves sending message to another task or sending a message to self.
- **Time Event**
  A time event occurs after a specified time duration has elapsed following another event.
- **Change Event**
  A change event occurs after some designated condition has been satisfied.

![Figure 5.16 UML Events](image)

5.11.2 State Machines and State Chart Diagrams

We have studied and used the finite state machines (FSM) to model and to implement a system’s behavior in time. The term state machine is used to describe

- The states that a system can enter during its life time
- Events to which the system can respond
- Possible responses the system can make to an event
- Transitions between possible states

Because of its simplicity, the FSM gives a good first order model of a system’s behavior. UML supports and extends the traditional notion of state machines.

5.11.2.1 UML State Chart Diagrams

A state chart diagram is nothing more than the familiar state diagram with some extensions / modifications under UML. The diagram begins with the notion of a state. A state is written as a cartouche – a rectangle with rounded corners as illustrated in Figure 5.17. Transitions between
states reflect a change in system from one state to another and are expressed as an arrow directed from the source state to the destination state.

Mathematically, the UML state chart is a directed graph. Because cycles are permitted, it’s a cyclic directed graph.

5.11.2.2 Transitions

A transition between states occurs under the following conditions, an event of interest to the system occurs or the system has completed some action and is ready to move to next state. The latter transition is called a triggerless transition. One may associate an action with the transition and a transition to self is permitted.

All four types of transition are illustrated in Figure 5.18.

5.11.2.3 Guard Conditions

A guard condition can be associated with a transition. A guard condition is a Boolean expression that must evaluate to true before the transition can fire. As was done in the branch and merge diagram, a guard condition is shown in square brackets on the transition arrow. UML supports several different kinds of guard.

- An event and a guard condition are written as,

\[
\text{EventName} \ [\text{guardCondition}]
\]

on the state transition edge. If the guardCondition evaluates to false, the transition will not be taken.
• An event, guard condition, and action triple, appear as

\[ \text{EventName} \[\text{guardCondition}] / \text{Action} \]

on the state transition arrow. If the guardCondition evaluates to false, the action is not executed and the transition not taken.

• A guard condition by itself, is described as

\[ \text{guardCondition} \]

Under such a condition, there is a repeated transition to self until the guard condition is met. Through such a mechanism, one can model the polling operation or blocking on an event or variable’s state.

In the following diagram, a solid black circle represents the initial state and a solid circle with a surrounding open circle represents the final state. Illustrated in Figure 5.19 is a transition with an action and a guarded event.

UML also makes the following definitions,

• An entry action is an action that the system always performs immediately upon entering a state. The requirement appears as entry / actionName within the state symbol

• An exit action is an action the system always performs immediately before leaving the state. The constraint appears as exit / actionName within the state symbol

• A deferred event is an event that is of interest to the system. Handling the event is deferred until system reaches another state. The deferred event appears as eventName / defer within the state symbol. Such events are entered into a queue that is checked when the system changes to the new state.

5.11.2.4 Composite States

The states that we've looked at so far are called simple states. UML extends the notion of a simple state to include multiple nested states - called composite states. These come in several different varieties.
Sequential States

If the system exists in a composite state and in only one of the state's substates at a time such substates are called *sequential substates*. Transitions between such substates are permitted as expected. Using sequential substates, the behavior of a state can be decomposed into smaller components as shown in the diagram in Figure 5.20.

History States

When system makes a transition into a composite state, typically the flow of control will start in the initial substate. However, it may be desirable to or necessary to begin in some other state. UML includes the concept of a *history substate* to support such capability. The history substate, shown in the state chart in Figure 5.21 by a small circle enclosing the letter ‘H’, will hold the last state that the system was in before leaving the composite state at an earlier time.

Such a state can be useful when modeling interrupt behavior or if one encounters a situation in which it is necessary to temporarily switch to another context to perform some operation prior to continuing.

In either case, the present state is temporarily exited. Some time in future, flow of control will return to that same state.
Concurrent Substates

A system may be in a composite state and also in more than one of the substates. Such is the situation in which the system may have two or more sets of substates representing parallel flows of control. When system enters a composite state with concurrent substates, it enters into initial state of both flows. Resynchronization is achieved by showing a final state for each flow as in the drawing in Figure 5.22.

We have only touched on some of capabilities of the static and dynamic UML diagrams. This will be sufficient for our work. There is a vast amount of literature available for those who are interested in more detailed study.

5.12 Dynamic Modeling with Structured Design Methods

The next tool that we will study is taken from the Structured Design approach to software modeling. The structured design methodologies, as we noted in our brief study of the UML approach, provide a far richer and more expansive set of tools that we will present here. Our focus will be solely on capturing a high level view of the flow of data and control within a design.

5.12.1 Brief Introduction to the Structured Design Philosophy

Structured design methodologies provide another tool for attacking the complexities of today’s designs. The approach has been around in one form or another for over 30 years. It provides one of the fundamental bases from which many of the modern tools, including UML, grew. A key aspect of the approach is that it is graphical. Its goals are rather simple. The design philosophy presented in this text is an outgrowth of many of its concepts. From top to bottom these goals are,

- To reduce the number of errors made during initial design
- Make it easy to find and fix those that do occur
- Develop robust, reliable, safe software
Its approach, comprised of five fundamental ideas, is equally simple:

1. Use the definition of the problem to guide the definition of solution.
2. Attack problem complexity by partitioning the problem into modules and then organizing the modules into hierarchies.
3. Use tools to help to make complex systems understandable.
4. Develop the solution from well defined statement of the problem.
5. Identify criteria for evaluating the quality of a design.

Many of the static and philosophical approaches to design have already been manifest in earlier discussions and in earlier tools; they are not relevant to the material here. The data and control diagram, however, provides a very simple tool for quickly and easily capturing a high level view of the dynamic structure of a design.

5.12.2 Data and Control Flow Diagrams

The data and control flow (DFD) diagram is used to partition a system into its active components and the data and control interfaces between them. The diagram is also sometimes known as a bubble chart.

5.12.2.1 The Elements

The data flow diagram comprises four graphic elements,

- The data and control flows
- The Processes or tasks and threads
- The data sources and sinks
- Any data stores

Let’s look at each of these.
Data or Control Flow

Data and control flows are expressed using notation that is similar to what we see in many UML diagrams. Data flow is indicated by a closed, solid arrow and control flow by a closed dashed arrow. As the notation in Figure 5.23 indicates, data or control flow in the direction of the arrow.

Processes or Tasks

The processes, modules, functions, or tasks are where the significant work in the application is being accomplished. Using a notation similar to that used in UML for states, these are expressed in a data and control flow diagram by labeled circles. The label identifies the name of the process or task and the level in the hierarchy at which the process resides.

- Level 0 - 1.0, 2.0, 3.0 etc.
- Level 1 - 1.1, 1.2, 1.3; 2.1, 2.2, 2.3; 3.1, 3.2 etc.
- Level 2 - 1.1.1, 1.1.2; 1.2.1, 1.2.2; etc.

The communications portion of an embedded system may contain tasks for managing the send and receive operations in the system. They would be expressed as is drawn in Figure 5.24,
Data Source / Sink

As the name implies, the source identifies where the data originates, for example from an input port and a sink indicates where data goes to, for example to an output port. The source or sink is drawn as a labeled box with an arrow to indicate direction of data flow as seen in Figure 5.25.

![Figure 5.25 Expressing a Data Source or Sink](image)

The source or sink are usually entities that are outside of the system.

Data Store

The final element is the data storage. The data store reflects the temporary storage of data or a time delayed repository of data. The data store is represented by two parallel lines or two parallel lines that are closed on the left hand side. To our electrical engineering students, this should look just like a capacitor – and does much the same job. The graphic is accompanied by a labeled arrow to indicate the direction of the data flow as we see in Figure 5.26.

![Figure 5.26 Expressing a Data Store](image)

Let's look at a simple example.
**Example 5.0**

The diagram in Figure 5.27 below presents a level 0 – top level – data and control flow diagram for a system that accepts commands from a remote source; collects image data at a local site; then sends the information back to the remote site.

![Figure 5.27 Capturing the Data and Control Flow in an Imaging System](image)

Command data comes into the system from the remote site. This input is shown as a data source. The reception is managed by the *Receive* task which brings the information into the system and stores it in the *Receive Buffer*. Once a complete message has been accepted by the *Receive* task, it sends a control message to the *Command Parser* task which parses the data and interprets the command. When it finishes, the *Command Parser* writes the command back into the buffer and sends a message to the *Image Capture* task to execute the capture. The *Image Capture* task collects the data from an external source and stores it into the transmit buffer. When the capture is complete, it signals the *Transmit* task to send the collected data back to the remote site.
The drawing in Figure 5.28 illustrates a hierarchical decomposition of a data flow diagram through three levels. At each level, greater detail is provided.

5.13 Summary

In this chapter, we have taken our first steps into basic software design. In doing so, we started to learn about some of the tools that can help us with that job. Specifically, we opened the chapter with a presentation of some of the tools that we take from UML – the Unified Modeling Language. We’ve introduced several different UML diagrams and one Structured Design diagram as tools that we can use to capture and model the static and dynamic relationships in a typical embedded application. We learned that the static models are essential for capturing the structure of the system and the dynamic models are important for expressing the desired behavior of system while it is performing its designated tasks and for providing information about interactions amongst those tasks. Finally, we found that the understanding of concurrent operation of modules within the application and the persistence of software entities comprising the system to be among the more important considerations when designing a system.
5.14 Review Questions

Introduction

5.1. Why are we using the Unified Modeling Language and Structured Design methodologies when developing embedded systems?

5.2. What information does a static view of an embedded system provide? A dynamic view?

5.3. In an embedded application, what does the term *concurrency* mean?

5.4. Please explain the terms *thread* or *thread of control*.

5.5. What does the term *persistence* mean in an embedded software application?

5.6. Are there different forms of *persistence*? If so, briefly describe what these might be.

An Introduction to UML

5.7. Where did UML originate and why was it developed?

5.8. In the context of the work in this text, what is the interpretation of the terms *class* and *object*?

5.9. Many of the tools used in UML are graphical, why?

UML Diagrams

5.10. What is the purpose of UML diagrams?

5.11. What are the major classes of UML diagrams or drawings? Please give a one or two sentence description of the purpose of each type of drawing.

5.12. In the text, the UML diagrams were segregated into three major groupings. What are these?

Use Case Diagrams

5.13. What a *use case* diagram provide for us?

5.14. What are the major components of a *use case* diagram? Briefly describe each.

5.15. Are the *actors* in a *use case* diagram always people?

5.16. A textual description is typically associated with a *use case* diagram. What information should that description contain?

Class Diagrams

5.17. What information should we include in a *class diagram*?
5.18. When we say that the class diagram presents the public interface to an object, what do we mean?

5.19. A class diagram is expressed as a rectangle subdivided into three components. Please identify these components and briefly describe the information contained in each.

5.20. The properties of an object can be decomposed into associations and attributes. Briefly describe what each of these means.

5.21. What are the different kinds of relationships that can be defined among classes or objects?

5.22. What kind of information should be captured in an inheritance diagram?

5.23. For what kind of relationship should an inheritance diagram be used?

5.24. What is the purpose of an interface diagram?

5.25. What kind of relationship do we express with an aggregation diagram?

5.26. What kind of relationship do we express with a composition diagram?

5.27. What is the difference between an aggregation and a composition diagram?

Dynamic Modeling with UML

5.28. What information does a dynamic model give us about a design?

5.29. What are the major elements that should be included in a dynamic model?

5.30. Three different forms of message can be expressed in an interaction diagram. What are these?

5.31. What are the different kinds of actions that are supported in a UML interaction diagram? Briefly describe each action.

5.32. What is the purpose of a UML sequence diagram?

5.33. What are the major elements of a sequence diagram.

5.34. What is a fork and join diagram? When should such a diagram be used?

5.35. What kind of activity does a branch and merge diagram allow us to express?

5.36. For what purpose do we use an activity diagram?

5.37. In the context of the unified modeling language, what is an event?

5.38. What kinds of events are supported by UML? Briefly describe each event.
5.39. A UML state chart diagram is an extension to the familiar state diagram for expressing the behavior of a finite state machine. What is a state diagram intended to describe?

5.40. The UML state chart specifies four kinds of transitions between states. Please identify each of these and briefly describe what each means.

5.41. What is the purpose of a guard condition in a UML state chart?

5.42. What kinds of guard conditions does UML support? Briefly describe what each such condition means.

5.43. What is a composite state in a UML state chart?

5.44. What kinds of composite states does UML support? Briefly describe each one and its intended purpose.

Structured Design Methodologies

5.45. What are the major goals of the Structured Design methodology?

5.46. The Structured Design approach to software modeling consists of five fundamental ideas. What are these?

5.47. What is the purpose of a data and control flow diagram?

5.48. What are the four major elements in a data and control flow diagram? What information does each capture?
5.15 Thought Questions

Introduction

5.1. Why is the modeling of both a static and a dynamic view of an embedded system essential throughout the design process?

5.2. When designing the software for a multitasking embedded system why is the understanding of and the ability to model concurrency important?

5.3. Why is the ability to model persistence in an embedded software application necessary?

Use Case Diagrams

5.4. The use case diagram provides the ability to capture and model the external view of a system. Why is such a view important in the early stages of the design of an embedded system?

5.5. Is the use case diagram limited to a top level / external view of the system?

5.6. Discuss possible benefits of developing a use case analysis for each module comprising a system.

5.7. Is the use case diagram limited to the software components of a system?

5.8. Why is a textual description an important component of a use case diagram?

Class Diagrams

5.9. What is the purpose of developing a class diagram?

5.10. We say that the class diagram presents the public interface to an object, how is this different from a use case diagram?

5.11. The class diagram provides the name of the class. its properties and its operations. Why is it important to capture this information during the early stages of the design of class?

5.12. What role can the class diagram play during the system level definition of an application?

5.13. The properties of an object can be decomposed into associations and attributes. Why is this information important?.

5.14. Why are the different kinds of relationships that can be defined among classes or objects important to understand in the early stages of system definition?

5.15. Why might an inheritance diagram be useful even if an object oriented language is not being utilized for implementation?

5.16. What information should be expressed in an interface diagram?
5.17. Why do we distinguish an aggregation and a composition diagram?

5.18. What assessments do the aggregation and a composition diagrams enable us to capture about the elements of the corresponding collections?

Dynamic Modeling with UML

5.19. Why is the information about a system that is captured in a dynamic model critical to the design of a modern embedded system?

5.20. What kinds of information should we be including in a dynamic model?

5.21. The event, rendezvous, and message quantify the exchange in an interaction diagram. Characterize the nature of the information in each of these exchanges.

5.22. Is the applicability of an interaction diagram restricted to inside the system?

5.23. Why do we distinguish the three types of information exchanged between entities in a system?

5.24. Please give an example from an embedded application that you are familiar with for each of the types of action modeled in an interaction diagram.

5.25. What information are we trying to understand and model using a UML sequence diagram?

5.26. Should we consider creating a sequence diagram for a complete system.

5.27. For what kinds of systems is a fork and join diagram going to provide useful information?

5.28. When should we be using a branch and merge diagram?

5.29. For what purpose do we use an activity diagram?

5.30. Several kinds of events are supported by UML. Give an example from a commercially available embedded application with which you are familiar where each such type of event might occur.

5.31. A UML state chart diagram is an extension to the familiar state diagram for expressing the behavior of a finite state machine. Give several examples from a commercially available embedded applications with which you are familiar for which the behavior on the software side of the system can be modeled by a state chart.

5.32. The UML state chart specifies four kinds of transitions between states. Give an example from a commercially available embedded applications with which you are familiar in which each such type transition might occur.

5.33. Give several examples of commercially available embedded applications with which you are familiar that might use composite state(s) as an effective aid in
expressing and modeling certain aspects of system behavior? Please explain how the use of composite state(s) has helped.

5.34. Give several examples of commercially available embedded applications with which you are familiar that might use history state(s) as an effective aid in expressing and modeling system behavior? Please explain how the use of history state(s) has helped.

**Structured Design Methodologies**

5.35. Give several examples of commercially available embedded applications with which you are familiar for which a data and control flow diagram might provide a simpler and more appropriate model of behavior that a UML state chart, interaction diagram, or activity diagram. Please explain why the data and control flow diagram is the preferable alternative.
5.16 Problems

UML Modeling - Basic Containers

For each of the data types that one might use in an embedded application please provide the following diagrams.

- Use Case Diagram and textual description of each use case
- Class Diagram for each top level module

5-1. A link in linked list
5-2. A linked list
5-3. A queue
5-4. A stack
5-5. A FIFO Container
5-6. A LIFO Container
5-7. A circular list

UML Modeling - Applications

For the embedded applications listed below, please provide the each of the following diagrams as appropriate.

- Use Case Diagram and textual description of each use case
- A first level decomposition of the application into top level modules
- Class Diagram for each top level module
- An Activity Diagram identifying the major activities in the application
- A State Chart (or State Charts as appropriate) identifying the state behavior of the application
- An Interaction Diagram / Sequence Diagram (or Diagrams as appropriate) identifying the interaction and temporal behavior of the high level modules within the application

5-8. Creating an embedded application.
5-10. A Tic Tac Toe or naughts and crosses game.
5-11. A cell phone.
5-12. A cell phone with 3 way calling.
5-15. An automobile.
5-16. An automobile cruise control.
5-17. A television with VCR / DVD player.
5-19. A washing machine.

   The controls must include the ability to set water temperature, washing start times, modes (presoak, normal, permanent press, delicate), and annunciation of temperature, times, and mode.

5-20. An intrusion detection with 3 doors and timers on each door.

   If a door is left open too long, the intruder alarm is initiated

5-21. An oven control

   The controls must include the ability to set temperatures, cooking start and stop times, modes (bake, broil, clean), and annunciation of temperature, times, and mode.


   The engine cannot start if the seat belt is not fastened. The Doors automatically lock when the engine is started.

5-23. An entertainment system.

   The entertainment system should support the ability to program and control a stereo, television, an in home movie theatre, a gaming console, and route music to any of 6 rooms in the house.


5-25. A television remote control.

5-26. A VCR / DVD record and playback device.

5-27. A stereo.


5-29. A module implementing a four seat passenger entertainment system on a commercial aircraft.

   The entertainment system should support the ability for each of the four passengers to program and control
Movie selection
Audio selection
A gaming console

5-30. An automatic process for filling and capping bottles of juice on an assembly line.

Structured Design Concepts - Data and Control Flow

For the following aspects of an embedded application, please provide a Data and Control Flow diagram.

5-31. Reading / writing from/to a USB port and a general parallel port.
5-32. Accessing and reading a mouse.
5-33. Access and reading keys from keyboard.
5-34. Controlling and accessing a digital to analog converter.
5-35. Controlling and accessing an analog to digital converter.
5-36. Burning a CD.
5-37. Transferring data from an external device to memory then to a display.
5-38. Managing and controlling a video on demand system in a motel or hotel.
5-39. An automatic process for filling and capping bottles of juice on an assembly line.