Designing for Concurrency

ObjecTime®
www.objectime.com

Designing real-time systems is one of the most challenging areas of software development. One of the things which makes it so challenging is dealing with a world in which many things are going on at once. Concurrent behaviour is the norm in the real world, but the software community at large pays remarkably little attention to it. This paper is intended as a primer on concurrency.

Our approach emphasizes the need to separate fundamental design issues from implementation mechanisms. We focus on the coordination issues which must be addressed when concurrent activities interact with each other, and give some heuristics for choosing the best mechanisms for various design situations encountered in real-world systems.

We also discuss the relationship between object-oriented programming and concurrency. In particular, we show that, while objects with a lightweight message-passing mechanism represent the best underpinnings for real-time concurrent systems, conventional object-oriented languages must be extended to provide an adequate solution base.

What is Concurrency and Why Does it Require Special Design Consideration?

The dictionary can leave us a little confused on the meaning of concurrency. “Concurrent” is defined as “operating at the same time,” or “running parallel.” It also, however, is defined as “meeting or tending to meet at the same point,” or “convergent,” which is certainly not consistent with parallelism, at least in the strict geometric sense. Of course, the word “concur,” meaning “to agree” comes from the same root. Despite the apparent confusion, each of these definitions is relevant to our task in developing real-time software.

Concurrency is a natural phenomenon, of course. In the real world, at any given time many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency. If each concurrent activity evolved independently, in a truly parallel fashion, this would be relatively simple: we could simply create separate programs to deal with each activity. The challenges of designing concurrent systems arise mostly because of the interactions which happen between concurrent activities. When concurrent activities interact, some sort of coordination is required. Vehicular traffic provides a

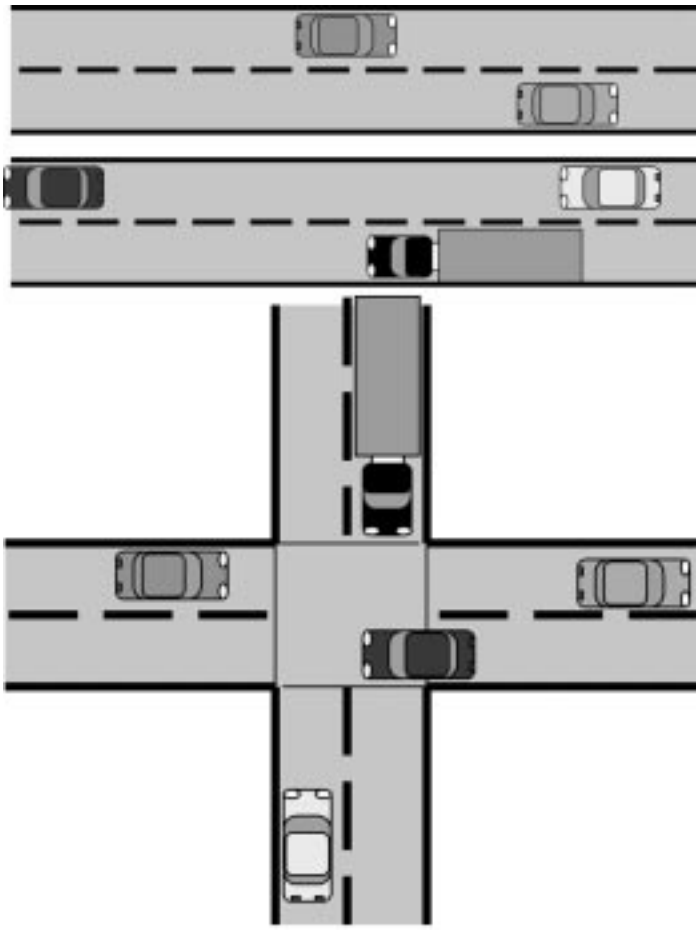

Fig. 1 – Concurrency in vehicular traffic

useful analogy, as in Fig. 1. Parallel traffic streams on different roadways having little interaction cause few problems. Parallel streams in adjacent lanes require some coordination for safe interaction, but a much more severe type of interaction occurs at an intersection, where careful coordination is required.

Why are we interested in concurrency?

Some of the driving forces for concurrency are external. That is, they are imposed by the demands of the environment. In real-world systems many things are happening simultaneously and must be addressed "in real-time" by software. To do so, many real-time software systems must be "reactive." They must respond to externally generated events which may occur at somewhat random times, in somewhat random order, or both. Designing a conventional procedural program to deal

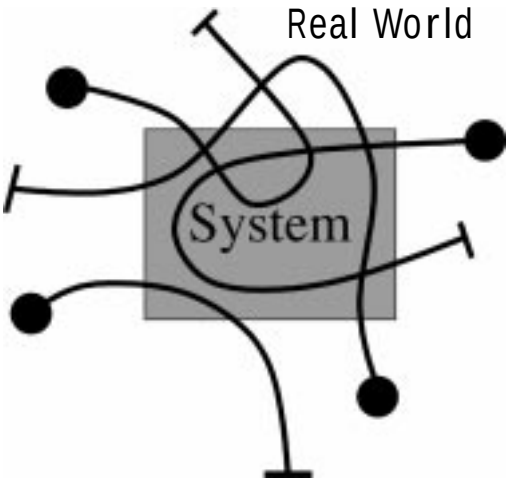

Fig. 2 – External forces for concurrency

with these situations is extremely complex. It can be much simpler to partition the system into concurrent software elements to deal with each of these events. The key phrase here is "can be", since complexity is also affected by the degree of interaction between the events.

There can also be internally inspired reasons for concurrency [Lea]. Performing tasks in parallel can substantially speed up the computational work of a system if multiple CPUs are available. Even within a single processor, multitasking can dramatically speed things up by preventing one activity from blocking another while waiting for I/O, for example. A common situation where this occurs is during the startup of a system. There are often many components, each of which requires time to be made ready for operation. Performing these operations sequentially can be painfully slow.

Controllability of the system can also be enhanced by concurrency. For example, one function can be started, stopped, or otherwise influenced in mid-stream by other concurrent functions—something extremely difficult to accomplish without concurrent components.

With all these benefits, why don't we use concurrent programming everywhere?

What makes concurrent software difficult?

Most computers and programming languages are inherently sequential. A procedure or processor executes one instruction at a time. Within a single sequential processor, the illusion of concurrency must be created by interleaving the execution of different tasks. The difficulties lie not so much in the mechanics of doing so, but

in the determination of just when and how to interleave program segments which may interact with each other.

Although achieving concurrency is easy with multiple processors, the interactions become more complex. First there is the question of communication between tasks running on different processors. Usually there are several layers of software involved, which increase complexity and add timing overhead. Determinism is reduced in multi-CPU systems, since clocks and timing may differ, and components may fail independently.

Finally, concurrent systems can be more difficult to understand because they lack an explicit global system state. The state of a concurrent system is the aggregate of the states of its components.

Example of a concurrent, real-time system: an elevator system

As an example to illustrate the concepts to be discussed, we will use an elevator system. More precisely, we mean a computer system designed to control a group of elevators at one location in a building. Obviously there may be many things going on concurrently within a group of elevators—or nothing at all! At any point in time someone on any floor may request an elevator, and other requests may be pending. Some of the elevators may be idle, while others are either carrying passengers, or going to answer a call, or both. Doors must open and close at appropriate times. Passengers may be obstructing the doors, or pressing door open or close buttons, or selecting floors—then changing their minds. Displays need to be updated, motors need to be controlled, and so on, all under the supervision of

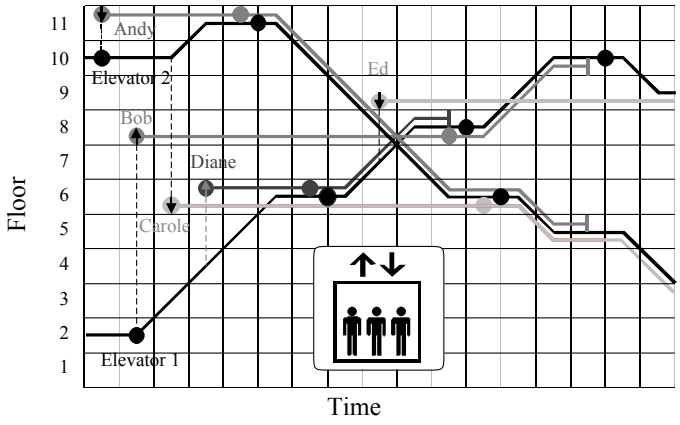

Fig. 3 – Concurrent demands on an elevator system

the elevator control system. Overall, it's a good model for exploring concurrency concepts, and one for which we share a reasonably common degree of understanding and a working vocabulary.

Figure 3 is a schematic diagram of a scenario involving two elevators and five potential passengers distributed over 11 floors. As potential passengers place demands upon the system at different times, the system attempts to provide the best overall service by selecting elevators to answer calls based upon their current

states and projected response times. For example, when the first potential passenger, Andy, calls for an elevator to go down, both are idle, so the closest one, Elevator 2, responds, although it must first travel upward to get to Andy. On the other hand, a few moments later when the second potential passenger, Bob, requests an elevator to go up, the more distant Elevator 1 responds, since it is known that Elevator 2 must travel downward to an as-yet-unknown destination before it can answer an up call from below.

Concurrency as a simplifying strategy

If the elevator system only had one elevator and that elevator had only to serve one passenger at a time, we might be tempted to think we could handle it with an ordinary sequential program. Even for this "simple" case, the program would require many branches to accommodate different conditions. For example, if the passenger never boarded and selected a floor, we would want to reset the elevator to allow it to respond to another call.

The normal requirement to handle calls from multiple potential passengers and requests from multiple passengers exemplifies the external driving forces for concurrency we discussed earlier. Because the potential passengers lead their own concurrent lives, they place demands on the elevator at seemingly random times, no matter what the state of the elevator. It is extremely difficult to design a sequential program that can respond to any of these external events at any time while continuing to guide the elevator according to past decisions.

Abstracting Concurrency

In order to design concurrent systems effectively, we must be able to reason about concurrency's role in the system, and in order to do this we need abstractions of concurrency itself.

The fundamental building blocks of concurrent systems are "activities" which proceed more or less independently of each other. A useful graphical abstraction for thinking about such activities is Buhr's "timethreads." [Buhr] Our elevator scenario in Fig. 3 actually used a form of them. Each activity is represented as a line along which the activity travels. The large dots represent places where an activity starts or waits for an event to occur before proceeding. One activity can trigger

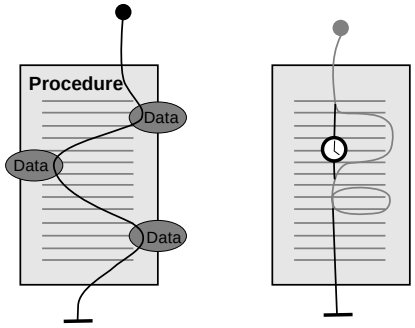

Fig. 4 – Threads of execution

another to continue, which is represented in the timethread notation by touching the waiting place on the other timethread.

The basic building blocks of software are procedures and data structures, but these alone are inadequate for reasoning about concurrency. As processor executes a procedure it follows a particular path depending upon current conditions. This path may be called the “thread of execution” or “thread of control.”[1] We can picture such a thread flowing through a single program using timethreads as in Fig. 4. Of course, as indicated by Fig. 4, the thread of control may take different branches or loops depending upon the conditions which exist at the time, and in real-time systems may pause for a specified period or wait for a scheduled time to resume.

From the point of view of the program designer, the thread of execution is controlled by the logic in the program and scheduled by the operating system. When the software designer chooses to have one procedure invoke others, the thread of execution jumps from one procedure to another, then jumping back to continue where it left off when a return statement is encountered. This is illustrated in Fig. 5.

From the point of view of the CPU, there is only one main thread of execution that weaves through the software, supplemented by short separate threads which are

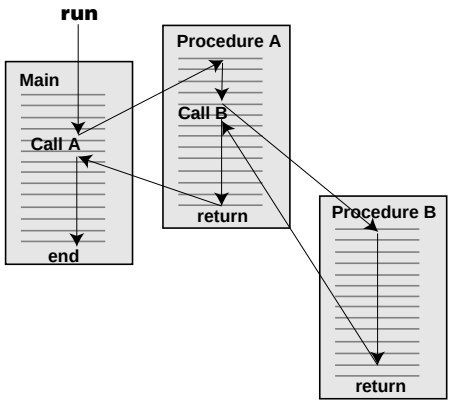

Fig. 5 – Thread of control through procedure calls

executed in response to hardware interrupts (see Fig. 6). Since everything else builds on this model, it is important for designers to know about it. Designers of real-time systems, to a greater degree than designers of other types of software, must understand how a system works at a very detailed level. This model, however,

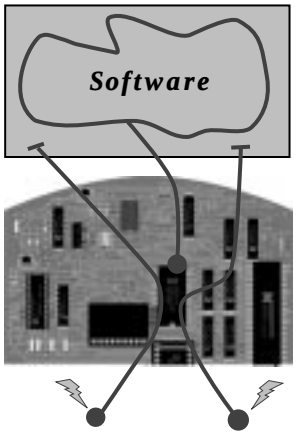

Fig. 6 – Threads of control from the CPU’s viewpoint

[1] For those familiar with operating system threads, we want to emphasize that the “thread of control” we are discussing here is an *abstraction* which is not necessarily associated with any particular artifact in either the OS or the hardware.

is at such a low level of abstraction that it can only represent concurrency very coarse granularity—that of the CPU. To design complex systems, it is useful to be able to work at various levels of abstraction. Abstraction, of course, is the creation of a view or model that suppresses unnecessary details so that we may focus on what is important to the problem at hand.

To move up one level, we commonly think of software in terms of layers. At the most fundamental level, the Operating System (OS) is layered between the hardware and the application software. It provides the application with hardware-based services, such as memory, timing, and I/O, but it *abstracts* the CPU to create a virtual machine that is independent of the actual hardware configuration.

Fig. 7 – Fundamental layers

Realizing Concurrency: Mechanisms

To support concurrency, a system must provide for multiple threads of control. The abstraction of a thread of control can be implemented in a number of ways by hardware and software. The most common mechanisms are variations of one of the following. [Deitel], [Tanenbaum]

- Multiprocessing—multiple CPUs executing concurrently
- Multitasking—the operating systems simulates concurrency on a single CPU by interleaving the execution of different tasks
- Application-based solutions—the application software takes responsibility for switching between different branches of code at appropriate times

Multitasking

When the operating system provides multitasking, a common unit of concurrency is the *process*. A process is an entity provided, supported and managed by the operating system whose sole purpose is to provide an environment in which to execute a program. The process provides a memory space for the exclusive use of its application program, a thread of execution for executing it, and perhaps some means for sending messages to and receiving them from other processes. In effect, the process is a virtual CPU for executing a concurrent piece of an application.

Fig. 8 – Process

Each process has three possible states:

- blocked—waiting to receive some input or gain control of some resource;
- ready—waiting for the operating system to give it a turn to execute;
- running—actually using the CPU.

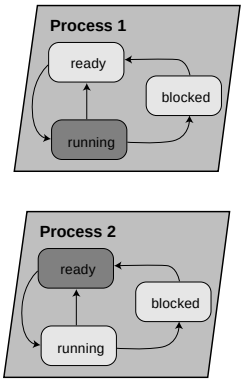

Fig. 9 – Process states

Processes are also often assigned relative priorities. The operating system kernel determines which process to run at any given time based upon their states, their priorities, and some scheduling policy.

Multitasking operating systems actually share a single thread of control among all of their processes. As we said before, from the point of view of the CPU, there is only one thread of execution. Just as an application program can jump from one procedure to another by invoking subroutines, the operating system can transfer control from one process to another on the occurrence of an interrupt, the completion of a procedure, or some other event. Because of the memory protection afforded by a process, this "task switching" can carry with it considerable overhead. Furthermore, because the scheduling policy and process states have

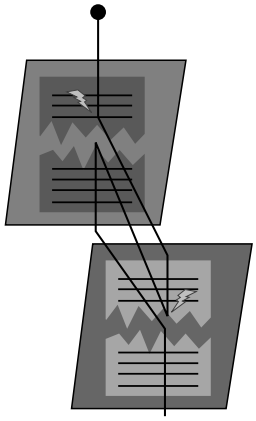

Fig. 10 – Multitasking

[2] The terms "task" and "process" are often used interchangeably. Unfortunately, the term "multitasking" is generally used to mean the ability to manage multiple processes at once, while "multiprocessing" refers to a system with multiple processors (CPUs). We adhere to this convention because it is the most commonly accepted. However, we use the term "task" sparingly, and when we do, it is to make a fine distinction between the unit of *work* being done (the task) and the entity which provides the resources and environment for it (the process).

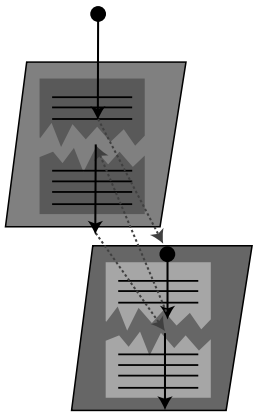

Fig. 11 – A separate thread of control for each process

little to do with the application viewpoint, the interleaving of processes is usually too low a level of abstraction for thinking about the kind of concurrency which is important to the application. [2]

In order to reason clearly about concurrency, it is important to maintain a clear separation between the concept of a thread of execution and that of task switching. Each process can be thought of as maintaining its own thread of execution. When the operating system switches between processes, one thread of execution is temporarily interrupted and another starts or resumes where it previously left off (Fig. 11).

Multithreading

Many operating systems, particularly those used for real-time applications, offer a "lighter weight" alternative to processes, called "threads" or "lightweight threads."[3] Threads are a way of achieving a slightly finer granularity of concurrency within a process. Each thread belongs to a single process, and all the threads in a process

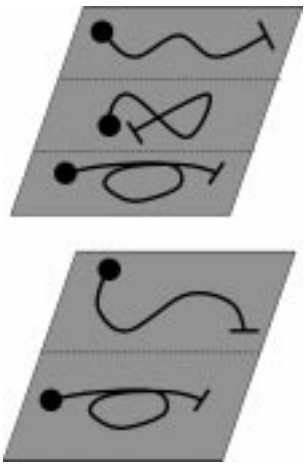

Fig. 12 – Multithreading

share the single memory space and other resources controlled by that process. Usually each thread is assigned a procedure to execute.

[3] It is unfortunate that the term "threads" is overloaded. When we use the word "thread" by itself, as we do here, we are referring to a "physical thread" provided and managed by the operating system. When we refer to a "thread of execution," or "thread of control" or "timethread" as in the foregoing discussion, we mean an *abstraction* which is not necessarily associated with a physical thread.

Multiprocessing

Of course, multiple processors offer the opportunity for truly concurrent execution. Most commonly, each task is permanently assigned to a process in a particular processor, but under some circumstances tasks can be dynamically assigned to the next available processor. Perhaps the most accessible way of doing this is by using a "symmetric multiprocessor." In such a hardware configuration, multiple CPUs can access memory through a common bus. Operating systems which support symmetric multiprocessors can dynamically assign threads to any available CPU. Examples of operating systems which support symmetric multiprocessors are SUN's Solaris and Microsoft's Windows NT.

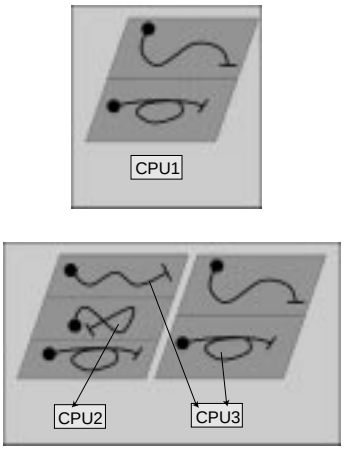

Fig. 13 – Multiprocessing and symmetric multiprocessing

Fundamental Issues of Concurrent Software

Earlier we made the seemingly paradoxical assertions that concurrency both increases and decreases the complexity of software. Concurrent software provides simpler solutions to complex problems primarily because it permits a "separation of concerns" among concurrent activities. In this respect, concurrency is just one more tool with which to increase the modularity of software. When a system must perform predominantly independent activities or respond to predominantly independent events, assigning them to individual concurrent components naturally simplifies design.

The additional complexities associated with concurrent software arise almost entirely from the situations where these concurrent activities are *almost* but *not quite* independent. In other words, the complexities arise from their interactions. From a practical standpoint, interactions between asynchronous activities invariably involve the exchange of some form of signals or information.

Interactions between concurrent threads of control give rise to a set of issues which are unique to concurrent systems, and which must be addressed to guarantee that a system will behave correctly.

Asynchronous vs. synchronous interaction

When interactions are necessary between concurrent activities, the designer must choose between a *synchronous* or *asynchronous style*. By synchronous, we mean that two or more concurrent threads of control must *rendezvous* at a single point in time. This generally means that one thread of control must wait for another to respond to a request. The simplest and most common form of synchronous interaction occurs when concurrent activity A requires information from concurrent activity B in order to proceed with A's own work. Synchronous interactions are, of course, the norm for non-concurrent software components. Ordinary procedure calls are a prime example of a synchronous interaction: when one procedure calls another, the caller instantaneously transfers control to the called procedure and effectively "waits" for control to be transferred back to it. In the concurrent world, however, additional apparatus is needed to synchronize otherwise independent threads of control.

Asynchronous interactions do not require a rendezvous in time, but still require some additional apparatus to support the communication between two threads of control. Often this apparatus takes the form of communication channels with message queues so that messages can be sent and received asynchronously.

Contention for shared resources

Concurrent activities may depend upon scarce resources which must be shared among them. Typical examples are I/O devices. If an activity requires a resource which is being used by another activity, it must wait its turn.

Race conditions: the issue of consistent state

Perhaps the most fundamental issue of concurrent system design is the avoidance of "race conditions." When part of a system must perform state-dependent functions (that is, functions whose results depend upon the present state of the system), it must be assured that that state is stable during the operation. In other words, certain operations must be "atomic." Whenever two or more threads of control have access to the same state information, some form of "concurrency control" is necessary to assure that one thread does not modify the state while the other is performing an atomic state-dependent operation. Simultaneous attempts to access the same state information which could make the state internally inconsistent are called "race conditions."

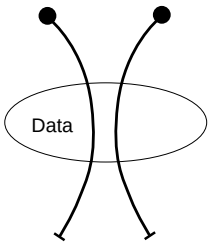

Fig. 14 – Race condition

A typical example of a race condition could easily occur in the elevator system when a floor is selected by a passenger. Our elevator works with lists of floors to be visited when traveling in each direction, up and down. Whenever the elevator arrives at a floor, one thread of control removes that floor from the appropriate list

and gets the next destination from the list. If the list is empty, the elevator either changes direction if the other list contains floors, or goes idle if both lists are empty. Another thread of control is responsible for putting floor requests in the appropriate list when the passengers select their floors. Each thread is performing combinations of operations on the list which are not inherently atomic: for example, checking the next available slot then populating the slot. If the threads happen to interleave their operations, they can easily overwrite the same slot in the list.

Deadlock

Deadlock is a condition in which two threads of control are each blocked, each waiting for the other to take some action. Ironically, deadlock often arises *because* we apply some synchronization mechanism to avoid race conditions.

The elevator example of a race condition could easily cause a relatively benign case of deadlock. The elevator control thread thinks the list is empty and, thus, never visits another floor. The floor request thread thinks the elevator is working on emptying the list and therefore that it need not notify the elevator to leave the idle state.

Other Practical Issues

In addition to the “fundamental” issues, there are some practical issues which must be explicitly addressed in the design of concurrent software.

Performance tradeoffs

Within a single CPU, the mechanisms required to simulate concurrency by switching between tasks use CPU cycles which could otherwise be spent on the application itself. On the other hand, if software must wait for I/O devices, for example, the performance improvements afforded by concurrency may greatly outweigh any added overhead.

Complexity tradeoffs

Concurrent software requires coordination and control mechanisms not needed in sequential programming applications. These make concurrent software more complex and increase the opportunities for errors. Problems in concurrent systems are also inherently more difficult to diagnose because of the multiple threads of control. On the other hand, as we have pointed out before, when the external driving forces are themselves concurrent, concurrent software which handles different events independently can be vastly simpler than a sequential program which must accommodate the events in arbitrary order.

Nondeterminism

Because many factors determine the interleaving of execution of concurrent components, the same software may respond to the same sequence of events in a different order. Depending upon the design, such changes in order may produce different results.

The role of application software in concurrency control

Application software may or may not be involved in the implementation of concurrency control. There is a whole spectrum of possibilities, including, in order of increasing involvement:

1. Application tasks may be interrupted at any time by the operating system (preemptive multitasking).
2. Application tasks may define atomic units of processing (critical sections) which must not be interrupted, and inform the operating system when they are entered and exited.
3. Application tasks may decide when to relinquish control of the CPU to other tasks (cooperative multitasking).
4. Application software may take full responsibility for scheduling and controlling the execution of various tasks.

These possibilities are neither an exhaustive set, nor are they mutually exclusive. In a given system a combination of them may be employed.

Abstracting concurrency

A common mistake in concurrent system design is to select the specific mechanisms to be used for concurrency too early in the design process. Each mechanism brings with it certain advantages and disadvantages, and the selection of the "best" mechanism for a particular situation is often determined by subtle trade-offs and compromises. The earlier a mechanism is chosen, the less information one has upon which to base the selection. Nailing down the mechanism also tends to reduce the flexibility and adaptability of the design to different situations.

As with most complex design tasks, concurrency is best understood by employing multiple levels of abstraction. First, the *functional* requirements of the system must be well understood in terms of its desired behaviour. Next the possible roles for concurrency should be explored. This is best done using the *abstraction* of threads without committing to a particular implementation. To the extent possible, the final selection of mechanisms for realizing the concurrency should remain open to allow fine tuning of performance and the flexibility to distribute components differently for various product configurations.

The "conceptual distance" between the problem domain (e.g., an elevator system) and the solution domain (software constructs) remains one of the biggest difficulties in system design. "Visual formalisms" are extremely helpful for understanding and communicating complex ideas such as concurrent behavior, and, in effect, bridging that conceptual gap. Among the tools which have proven valuable for such problems are:

- module diagrams for envisioning concurrently acting components;
- timethreads for envisioning concurrent and interactive activities (which may be orthogonal to the components);

- interaction diagrams such as “message sequence charts” for visualizing interactions between components;
- Statecharts [Harel] or ROOMcharts [Selic] for defining the states and state-dependent behaviors of components.

Some notations, such as UML or ROOM [Selic], combine some of these elements and define more or less formal ways in which they may be employed together to fully define a system’s structure and behavior.

Objects as Concurrent Components

To design a concurrent software system, we must combine the building blocks of software (procedures and data structures) with the building blocks of concurrency (threads of control). We have discussed the concept of a concurrent activity, but one doesn’t construct systems from activities. One constructs systems from components, and it makes sense to construct concurrent systems from concurrent components. Taken by themselves, neither procedures nor data structures nor threads of control make very natural models for concurrent components, but objects seem like a very natural way to combine all of these necessary elements into one neat package.

An object packages procedures and data structures into a cohesive component with its own state and behavior. It encapsulates the specific implementation of that state and behavior and defines an interface by which other objects or software may interact with it. Objects generally model real world entities or concepts, and interact with other objects by exchanging messages. They are now well accepted by many as the best way to construct complex systems.

Consider an object model for our elevator system (Fig. 15). A call station object at each floor monitors the up and down call buttons at that floor. When a prospective passenger depresses a button, the call station object responds by sending a message to an elevator dispatcher object, which selects the elevator most likely to provide the fastest service, dispatches the elevator and acknowledges the call. Each elevator object concurrently and independently controls its physical elevator counterpart, responding to passenger floor selections and calls from the dispatcher.

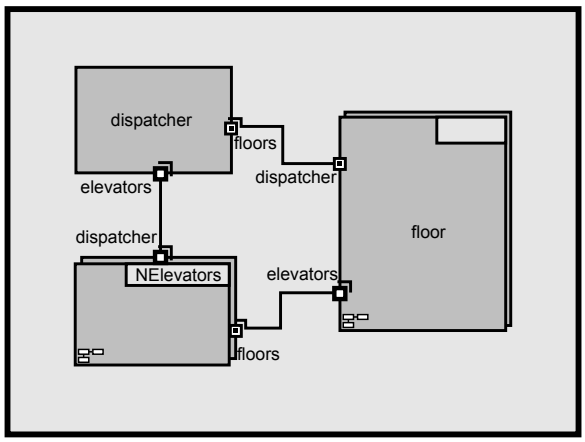

Fig. 15 – Object model of elevator system

Concurrency can take two forms in such an object model. Inter-object concurrency results when two or more objects are performing activities independently via separate threads of control. Intra-object concurrency arises when multiple threads of control are active in a single object. In most object-oriented languages today, objects are "passive," having no thread of control of their own. The thread(s) of control must be provided by an external environment. Most commonly, the environment is a standard OS process created to run an object-oriented "program" written in a language like C++ or Smalltalk. If the OS supports multi-threading, multiple threads can be active in the same or different objects. Figure 16 illustrates this type of environment. In this figure, the passive objects are represented by the circular elements. The shaded interior area of each object is its state information, and the segmented outer ring is the set of procedures (methods) which define the object's behavior.

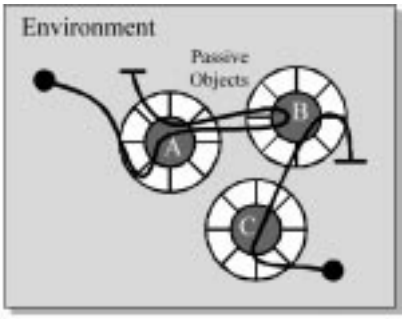

Fig. 16 – Run-time environment for objects

Intra-object concurrency brings with it all of the challenges of concurrent software, such as the potential for race conditions when multiple threads of control have access to the same memory space—in this case, the data encapsulated in the object. One might have thought that data encapsulation would provide a solution to this issue. The problem, of course, is that the object does not encapsulate the thread of control.

Although inter-object concurrency avoids these issues for the most part, there is still one troublesome problem. In order for two concurrent objects to interact by exchanging messages, at least two threads of control must handle the message and access the same memory space in order to hand it off. A related (but still more difficult) problem is that of distribution of objects among different processes or even processors. Messages between objects in different processes requires support for interprocess communication, and generally require the message to be encoded and decoded into data that can be passed across the process boundaries.

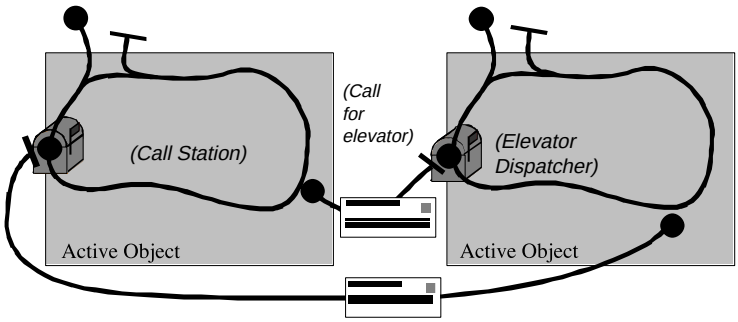

Fig. 17 – Active objects

None of these problems is insurmountable, of course. In fact, as we pointed out in the previous section, every concurrent system must deal with them, so there are proven solutions. It is just that "concurrency control" causes extra work and introduces extra opportunities for error. Furthermore, it obscures the essence of the application problem. For all of these reasons, we want to minimize the need for application programmers to deal with it explicitly. One way to accomplish this is to build an object-oriented environment with support for message passing between concurrent objects (including concurrency control), and minimize or eliminate the use of multiple threads of control within a single object. In effect, this encapsulates the thread of control along with the data.

The active object model

Objects with their own threads of control, such as those illustrated in Fig. 17, are sometimes called "active objects" or "actors." [Agha] In order to support asynchronous communication with other active objects, each active object is provided with a message queue or "mailbox." When an object is created, the environment gives it its own thread of control, which the object encapsulates until it dies. Like a passive object, the active object is idle until the arrival of a message from outside. The object executes whatever code is appropriate to process the message. Any messages which arrive while the object is busy are enqueued in the mailbox. When the object completes the processing of a message, it returns to pick up the next waiting message in the mailbox, or waits for one to arrive. Good candidates for active objects in the elevator system include the elevators themselves, the call stations on each floor, and the dispatcher.

Depending upon their implementation, active objects can be made to be quite efficient. They do carry somewhat more overhead, however, than a passive object. Thus, since not every operation need be concurrent, it is common to mix active and passive objects in the same system. Because of their different styles of communication, it is difficult to make them peers, but an active object makes an ideal environment for passive objects, replacing the OS process we used earlier. In fact, if the active object delegates all of the work to passive objects, it is basically the equivalent of an OS process or thread with interprocess communication facilities. More interesting active objects, however, have behavior of their own to do part of the work, delegating other parts to passive objects (Fig. 18).

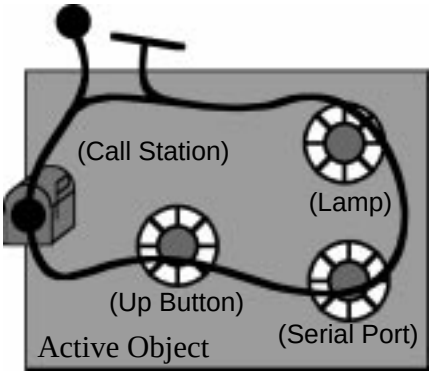

Fig. 18 – Active object as an environment for passive objects

Good candidates for passive objects inside an active elevator object include a list of floors at which the elevator must stop while going up and another list for going down. The elevator should be able to ask the list for the next stop, add new stops to the list, and remove stops which have been satisfied.

Because complex systems are almost always constructed of subsystems several levels deep before getting to leaf-level components, it is a natural extension to the active object model to permit active objects to contain other active objects (Fig. 19). Although a single-threaded active object does not support true intra-object concurrency, delegating work to contained active objects is a reasonable substitute for many applications. It retains the important advantage of complete encapsulation of state, behavior, and thread of control on a per-object basis, which simplifies the concurrency control issues.

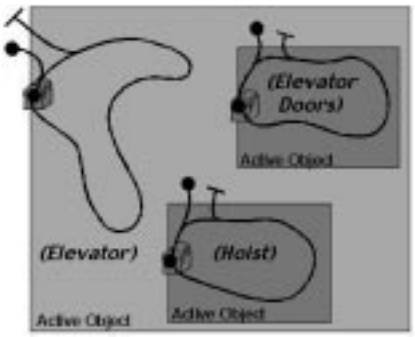

Fig. 19 – Active object containing active objects

In fact, the nature of the interactions between active objects is no different whether they are peers or arranged in a containment hierarchy. The containment or nested object concept is, however, important for other reasons. As a modeling concept, the existence of a system or subsystem implies the existence of its parts. This is more faithfully modeled by nested active objects, where the container is responsible for the instantiation of its components, than by objects in a flattened space.

Consider, for example the partial elevator system described by Figures 20 and 21. Each elevator has doors, a hoist, and a control panel. Each of these components is well-modeled by a concurrent active object, where the door object controls the opening and closing of the elevator doors, the hoist object controls the positioning of the elevator through the mechanical hoist, and the control panel object monitors

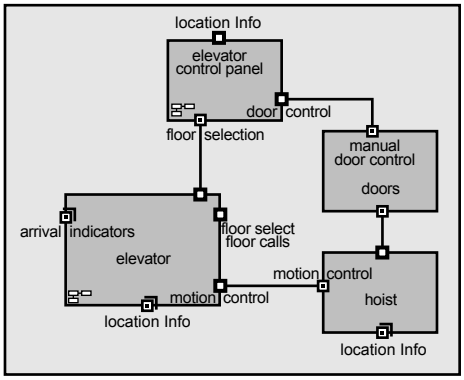

Fig. 20 – Elevator, doors, hoist, control panel as peers

the floor selection buttons and door open/close buttons. Although all of these objects could be modeled as peers in a flat object space (Fig. 20), the nested model (Fig. 21) offers improved modularity and fidelity to the real world. In any event, their concurrency as active objects leads to much simpler software than could be achieved if all this behavior were managed by a single thread of control.

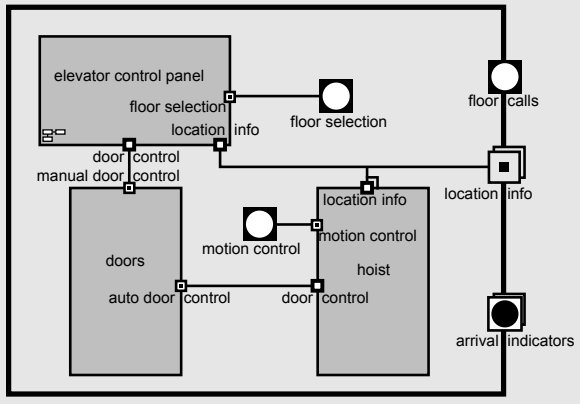

Fig. 21 – Elevator with doors, hoist, and control panel as components

The consistent state issue in objects

As we said when discussing race conditions, in order for a system to behave in a correct and predictable manner, certain state-dependent operations must be atomic. For an object to behave properly, it is certainly necessary for its state to be internally consistent before and after processing any message. During the processing of a message, the object's state may be in a transient condition and may be indeterminate because operations may be only partially complete.

If an object always completes its response to one message before responding to another, the transient condition is not a problem. Interrupting one object to execute another also poses no problem because each object practices strict encapsulation of its state.[4] Any circumstance under which an object interrupts the processing of a message to process another opens the possibility of race conditions and, thus, requires the use of concurrency controls. This, in turn, opens the possibility of deadlock.

Concurrent design is generally simpler, therefore, if objects process each message to completion before accepting another. This behavior is implicit in the particular form of active object model we have presented.

The issue of consistent state can manifest itself in two different forms in concurrent systems, and these are perhaps easier to understand in terms of object-oriented concurrent systems. The first form is that which we have already discussed. If the state of a single object (passive or active) is accessible to more than one thread of control, atomic operations must be protected either by the natural atomicity of elementary CPU operations or by a concurrency control mechanism. This is illustrated in Fig. 22.

[4] Strictly speaking, this is not completely true, as we shall explain shortly.

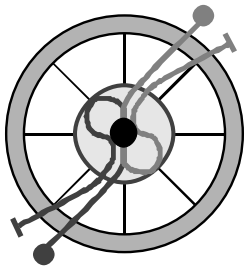

Fig. 22 – Critical section in passive object

The second form of the consistent state issue is perhaps more subtle. If more than one object (active or passive) contains the same state information, the objects will inevitably disagree about the state for at least short intervals of time. (In a poor design they may disagree for longer periods—even forever.) This manifestation of inconsistent state can be considered a mathematical "dual" of the other form.

For example, the elevator motion control system (the hoist) must assure that the doors are closed and cannot open before the elevator can move. A design without proper safeguards could permit the doors to open in response to a passenger hitting the door open button just as the elevator begins to move.[5]

It may seem that an easy solution to this problem is to permit state information to reside in only one object. Although this may help, it can also have a detrimental impact on performance, particularly in a distributed system. Furthermore, it is not a foolproof solution. Even if only one object contains certain state information, as long as other concurrent objects make decisions based upon that state at a certain point in time, state changes can invalidate the decisions of other objects.

There is no magic solution to the problem of consistent state. All practical solutions require us to identify atomic operations and protect them with some sort of synchronization mechanism which blocks concurrent access for tolerably short periods of time. "Tolerably short" is very much context dependent. It may be as long as it takes the CPU to store all the bytes in a floating point number, or it may be as long as it takes the elevator to travel to the next stop.

Design Heuristics for Concurrent Systems

The art of good design is that of choosing the "best" way to meet a set of requirements. The art of good concurrent system design is often that of choosing the simplest way to satisfy the needs for concurrency. One of the first rules for designers should be to avoid reinventing the wheel. Good design patterns and design idioms have been developed to solve most problems. Given the complexity of concurrent systems it only makes sense to use well-proven solutions and to strive for simplicity of design.

The first rule for concurrent system designers is:

Heuristic 1: Focus on interactions between concurrent components.

[5] Admittedly, this would be a *very* poor design, and elevator systems have electrical and mechanical safeguards against such possibilities.

Concurrent components with no interactions are an almost trivial problem. Nearly all of the design challenges have to do with interactions among concurrent activities, so we must first focus our energy on understanding the interactions. Some of the questions to ask are:

- Is the interaction one-directional, bi-directional, or multi-directional?
- Is there a client-server or master slave relationship?
- Is some form of synchronization required?

Once the interaction is understood, we can think about ways to implement it. The implementation should be selected to yield the simplest design consistent with the performance goals of the system. Performance requirements for real-time systems generally include both overall throughput and acceptable latency in the response to externally generated events.

Patterns to deal with events

It is not going too far to say that all actions in real-time systems are triggered by the occurrence of externally generated events. One very important externally generated event in real-time systems is simply the passage of time itself, as represented by the tick of a clock. Other external events come from input devices connected to external hardware, including user interface devices, process sensors, and communication links to other systems.

In order for software to detect an event, it must be either blocked waiting for an interrupt, or periodically checking hardware to see if the event has occurred. In the latter case, the periodic cycle may need to be short to avoid missing a short lived event or multiple occurrences, or simply to minimize the latency between the event's occurrence and detection.

The interesting thing about this is that no matter how rare an event is, some software must be blocked waiting for it or frequently checking for it. But many (if not most) of the events a system must handle are rare. Indeed, Weinberg's Law of Twins [Weinberg] tells us that most of the time, in any given system, nothing of any significance is happening.[6]

The elevator system provides many good examples of this. Important events in the life of an elevator include a call for service, passenger floor selection, a passenger's hand blocking the door, and passing from one floor to the next. Some of these events require very time-critical response, but all are extremely rare compared to the time-scale of the desired response time.

A single event may trigger many actions, and the actions may depend upon the states of various objects. Furthermore, different configurations of a system may use the same event differently. For example, when an elevator passes a floor the display in the elevator cab should be updated and the elevator itself must know where it is so that it knows how to respond to new calls and passenger floor selections. There may or may not be elevator location displays at each floor.

[6] Weinberg has a very amusing way of presenting this law and many other important systems engineering principles. The reference is highly recommended.

These observations lead us to several heuristics which are, effectively, corollaries of each other.

Heuristic 2: Isolate and encapsulate external interfaces.

Heuristic 3: Isolate and encapsulate blocking and polling behavior.

Heuristic 4: Prefer reactive behavior to scheduled behavior.

Taken together, these heuristics accomplish several important goals. It is bad practice to embed specific assumptions about external interfaces throughout an application, and it is very inefficient to have several threads of control blocked waiting for an event. Instead, assign a single object the dedicated task of detecting the event. When the event occurs, that object can notify any others who need to know about the event.

This design is based upon a well-known and proven design pattern, the "Observer" pattern [Gamma]. It can easily be extended for even greater flexibility to the "Publisher-Subscriber Pattern," where a publisher object acts as intermediary between the event detectors and the objects interested in the event ("subscribers") [Buschmann].

Simplicity vs. performance

The title above notwithstanding, the goals of simplicity and performance do not necessarily conflict with each other. Given the choices we have for implementing concurrency, we can formulate some guidelines for selecting the best mechanism for each instance.

Heuristic 5: Make heavy use of light-weight mechanisms and light use of heavy-weight mechanisms.

More specifically:

- Use passive objects and synchronous method invocations where concurrency is not an issue but instantaneous response is.
- Use active objects and asynchronous messages for the vast majority of application-level concurrency concepts.
- Use OS threads to isolate blocking elements. An active object can easily encapsulate an OS thread.
- Use OS processes for maximum isolation. Separate processes are needed if programs need to be started up and shut down independently, and for subsystems which may need to be distributed.
- Use separate CPUs for physical distribution or for raw horsepower through real concurrency.

Heuristic 6: Eschew performance bigotry.

In most systems less than 10% of the code uses more than 90% of the CPU cycles. The reactive style of design and isolation of polling and blocking behavior

discussed in Heuristics 2 through 4 go a long way toward isolating and minimizing the CPU-hungry parts of a system. Heuristic 6 says we can go farther, however. Many real-time system designers act as though every line of code must be optimized. Instead, spend your time optimizing the 10% of the code that runs most often or takes a long time. Design the other 90% with an emphasis on understandability, maintainability, modularity, and ease of implementation.

Summary

Real-time systems require concurrent behavior and distributed components. Most programming languages give us very little help with either of these issues. We have seen that we need good abstractions to understand both the need for concurrency in applications, and the options for implementing it in software. We have also seen that, paradoxically, while concurrent software is inherently more complex than non-concurrent software, it is also capable of vastly simplifying the design of systems which must deal with concurrency in the real world.

Perhaps the most important guideline for developing efficient concurrent applications is to maximize the use of the lightest weight concurrency mechanisms. Both hardware and operating system software play a major role in supporting concurrency, but both provide relatively heavy-weight mechanisms, leaving a great deal of work to the application designer. We are left to bridge a big gap between the available tools and the needs of concurrent real-time applications.

Active objects help to bridge this gap by virtue of two key features:

- They unify the design abstractions by encapsulating the basic unit of concurrency (a thread of control) which can be implemented using any of the underlying mechanisms provided by the OS or CPU.
- When active objects share a single OS thread, they become a very efficient, light-weight concurrency mechanism which would otherwise have to be implemented directly in the application.

Active objects also make an ideal environment for the passive objects provided by programming languages. Designing a system entirely from a foundation of concurrent objects without procedural artifacts like programs and processes leads to more modular, cohesive, and understandable designs.

Given the right tools and paradigms, designing complex systems almost begins to look easy.

References

[Agha] Agha, G., *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Buhr] Buhr, R.J.A. and Casselman, R.S., *Use Case Maps for Object-Oriented Systems.* Prentice-Hall, 1996.

[Buschmann] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[Deitel] Deitel, H., *An Introduction to Operating Systems.* Addison-Wesley, 1984.

[Gamma] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Harel] Harel, D., "Statecharts: a visual formalism for complex systems." *Sci. Computer Program.*, vol. 8, 1987.

[Lea] Lea, D., *Concurrent Programming in Java.* Addison-Wesley, 1997.

[Levi] Levi, S-T., and Agrawala, A., *Real-Time System Design*. McGraw-Hill, 1990.

[Selic] Selic, B., Gullekson, G., and Ward, P., *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[Tanenbaum] Tanenbaum, A., *Operating Systems: Design and Implementation*. Prentice-Hall, 1986.

[Weinberg] Weinberg, G. and Weinberg, D., *General Principles of Systems Design*. Dorset House Publishers, 1988 (Formerly published as: *On the Design of Stable Systems.* John Wiley & Sons, 1979).

Real-Time Object-Oriented Modeling (ROOM)

ROOM is a visual modeling language with formal semantics, developed by ObjecTime Limited. It is optimized for specifying, visualizing, documenting, and automating the construction of complex, event-driven, and potentially distributed real-time systems.

ROOM communication model

In ROOM, the occurrence of an event is signalled by the arrival of a message at an object. Message passing is the primary mode of communication between objects. It is convenient both for modeling the asynchronous nature of events as well as for dealing with distributed environments where shared memory may not exist. Note that message-based communication does not necessarily imply that all communication is asynchronous; both synchronous and asynchronous communication are supported in ROOM.

It is common for a collaboration between two objects to take on a specific pattern or *protocol*. A protocol comprises an ordered sequence of invocations and replies between two objects.

Each message that is exchanged in the course of a protocol sequence consists of a named *signal* and an optional passive data object that specifies one or more parameter values.

It is standard for the same protocol specification to be used by many different objects. Rather than recreate the same specification each time, it is more practical to define a single reusable specification and then reference it as necessary. In the object paradigm, this is provided through the *class* concept. A *protocol class* in ROOM defines a reusable message protocol specification.

Active objects in ROOM

The primary structural element of ROOM is called an *actor*. An actor is an *active object* responsible for performing some specific function. It is concurrent in the sense that it can operate in parallel with other actors.

An actor communicates with other objects through one or more interface objects called *ports*. Each port represents an instance of one protocol class. A single actor can simultaneously handle multiple different protocols.

Ports act as intermediaries between an actor's implementation, contained within the encapsulation shell, and its environment. The implementation only interacts directly with ports; it is de-coupled from the environment so that the same actor specification can be used in a variety of contexts.

To make useful systems, it is necessary to combine one or more objects of different kinds into more complex functional aggregates. In ROOM, this is achieved through three primary composition mechanisms: bindings, layer connections, and containment. Bindings and layer connections are used to model communication relationships while containment captures compositional relationships between actors.

A *binding* models a communication channel that connects two ports. Only ports that have compatible protocols can be mutually bound.

In contrast to bindings, *layer connections* are directed relationships and model situations in which there is an asymmetric dependency between two actors. This asymmetry is typically in the form of a client-server relationship: the client cannot function unless a server is present whereas the server can exist and function independently of any particular client.

A layer connection connects one or more *service access points* (SAPs) on the client to a *service provision point* (SPP) on the server. SAPs and SPPs are instances of protocol classes like ports.

The pattern of layer connections and bindings that inter-connect actors defines the complete set of possible communication relationships between actors. This is a crucial aspect of software architectures since it explicitly identifies all the potential causality links between the actors.

An actor class can be subclassed. Subclasses automatically inherit all the attributes of their superclasses (including implementation) and may add new attributes or modify existing ones.

Modeling behavior

The point where structural and behavioral dimensions meet is at the interfaces of an actor. An interface can be directly manipulated by the *behavior component* of an actor. A behavior component can be attached to any actor.

Reactive systems require modeling formalisms that are quite specific. ROOM uses a variant of hierarchical finite state machines called *ROOMcharts*. State machines are relatively well known among practitioners and have been widely used to capture reactive behavior.

In ROOMcharts, an event is represented by the arrival of a message at an actor interface. Depending on the current state of the behavior, the type of signal in the message, and the interface on which it arrives, the event will trigger a transition that leads to a change of state of the behavior.

ROOM uses a *run-to-completion* model of event processing. This simply means that messages are processed one at a time; once message handling is commenced and until it is completed, no other messages are processed by that behavior (this implies queueing of messages at the interfaces). The advantage of this approach is that it provides automatic mutual exclusion of event handlers and, thereby, significantly simplifies the behavior specification. It is justified in situations where the handling of events is relatively short—a property shared by the majority of reactive components.

Hierarchy in ROOMcharts means that a state can contain a state machine. As with structure, hierarchy allows for a complex problem to be addressed gradually, one level of abstraction at a time.

Each transition in a state machine can have an associated Detail Level program sequence that specifies what is to be done as part of the transition. This is where the contents of an incoming message can be processed and messages can be sent to other actors.

Extended state variables are instances of Detail Level passive data objects which are used by the finite-state machine to maintain auxiliary status information. These objects can be accessed by the code inside transitions. Since actors are fully encapsulated, extended state variables are not accessible by other actors.

UML for Real-Time

UML for Real-Time is a complete real-time modeling standard, co-developed by ObjecTime Limited and Rational Software Corporation, that combines UML 1.1 modeling concepts, and special modeling constructs and formalisms originally implemented in *ObjecTime Developer* and defined in the ROOM language.

When the semantics of a UML metaclass must be refined, a new UML stereotype is introduced. UML for Real-Time introduces the important Capsule, Port, and Connector, stereotypes that have been defined to support the modeling of complex real-time systems.

Capsules correspond to the ROOM concept of *actors*. Capsules are complex, potentially concurrent, and possibly distributed active objects. They interact with their surroundings through one or more signal-based boundary objects called ports. Collaboration diagrams are used to describe the structural decomposition of a Capsule class.

A *port* is a physical part of the implementation of a capsule that mediates the interaction of the capsule with the outside world—it is an object that implements a specific interface. Ports realize *protocols*, which define the valid flow of information (signals) between connected ports of capsules. In a sense, a protocol captures the contractual obligations that exist between capsules. Because a protocol defines an abstract interface that is realized by a port, a protocol is highly reusable.

Ports provide a mechanism for a capsule to export multiple different interfaces; each tailored to a specific role. They also provide a mechanism to explicitly *connect* an exported interface of one capsule directly to the interface of another capsule. By forcing capsules to communicate solely through ports, it is possible to fully de-couple their internal implementations from any direct knowledge they have about the environment. This de-coupling makes capsules highly reusable.

Connectors capture the key *communication relationships* between capsules. These relationships have architectural significance since they identify which capsules can affect each other through direct communication.

The functionality of simple capsules is realized directly by the *state machine* associated with the capsule. More complex capsules combine the state machine with an *internal* network of collaborating *sub-capsules* joined by connectors. These sub-capsules are capsules in their own right, and can themselves be decomposed into sub-capsules. This type of decomposition can be carried to whatever depth is necessary, allowing modeling of arbitrarily complex structures with just this basic set

of structural modeling constructs. The state machine (which is optional for composite capsules), the sub-capsules, and their connections network represent parts of the implementation of the capsule, and are hidden from external observers.

About ObjecTime Developer

ObjecTime Limited is the leading vendor of object-oriented tools for automating the development of real-time software systems, and the developer of the ROOM language. *ObjecTime Developer* is a software design automation toolset that implements ROOM constructs. It addresses the entire development lifecycle to deliver complete mission critical real-time applications using its modeling, model execution and *TotalCode*™ application code generation capabilities.

ObjecTime Developer is used for developing a wide variety of complex, real-time, event-driven applications in telecommunications, data communications, defense, aerospace, and other industries. Companies such as Nortel, Lucent Technologies, Motorola and Lockheed Martin use ObjecTime Developer to accelerate real-time software delivery, and to improve the quality and functionality of their real-time products.

ObjecTime Developer automates the ROOM language, and thus implements the UML for Real-Time concepts. All the benefits of UML 1.1 role modeling and the UML for Real-Time extensions, including design time model-visualization; run-time model animation of state machines and message flow (including message sequence charts); and run-time visual model, and source code level, debugging facilities; are available in ObjecTime Developer today. With ObjecTime Developer, all of the code for complete, high performance, mission critical applications is generated. The model is more than a model—the model is the application.

For more information, visit the ObjecTime web site at *http://www.objectime.com*.

To learn more about ObjecTime, visit our website at www.objectime.com.
or contact us at 1-800-567-8463

OBJECTIME®
www.objectime.com

084-0698 Printed in Canada