18 THE CONCURRENCY VIEWPOINT

Definition	Describes the concurrency structure of the system, mapping functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently, and shows how this is coordinated and controlled
Concerns	Task structure, mapping of functional elements to tasks, interprocess communication, state management, synchronization and integrity, startup and shutdown, task failure, and reentrancy
Models	System-level concurrency models and state models
Problems and Pitfalls	Modeling of the wrong concurrency, excessive complexity, resource contention, deadlock, and race conditions
Stakeholders	Developers, testers, and some administrators
Applicability	All information systems with a number of concurrent threads of execution

Historically, information systems were designed to operate with little or no concurrency, running via batch mode on large central computers. A number of factors (including distributed systems, increasing workloads, and cheap multiprocessor hardware) have combined so that today's information systems often have little or no batch mode operations.

In contrast, control systems have always been inherently concurrent and event-driven, given their need to react to external events in order to perform control operations. It is natural, then, that as information systems become more concurrent and event-driven, they start to take on a number of characteristics traditionally associated with control systems. Similarly, it is unsurprising that, in order to deal with this concurrency, the information systems community has adopted and adapted proven techniques from the control systems community. Many of these techniques form the basis of the Concurrency viewpoint.

The Concurrency view is used to describe the system's concurrency and state-related structure and constraints. This involves defining the parts of the system that can run at the same time and how this is controlled (e.g., defining how the system's functional elements are packaged into operating system processes and how the processes coordinate their execution). To do this, you need to create a process model and a state model: The process model shows the planned process, thread, and interprocess communication structure; the state model describes the set of states that runtime elements can be in and the valid transitions between those states.

Once you have created concurrency and state models, you can use a number of analysis techniques to ensure that the planned concurrency scheme is sound. The use of such techniques is typically part of creating a Concurrency view, too.

It's worth noting that not all information-based systems really benefit from a Concurrency view. Some information systems have little concurrency. Others, while exhibiting concurrent behavior, use the facilities of underlying software packages (e.g., databases) to hide the concurrency model actually in use.

EXAMPLE Data warehouse systems tend to be batch loaded overnight and accessed from a number of desktop machines. These systems do exhibit concurrent behavior—multiple clients can request information from the data warehouse concurrently. However, such a system will typically rely on the underlying database management system to handle all of the concurrency for it (in any way it chooses). Therefore, the process model used is of little architectural significance, and you have little or no control over it. The interesting aspects of the concurrency relate much more to the design of the physical data model and should be handled there.

In contrast, however, many of today's information systems are inherently event-driven, reactive, concurrent systems. This is particularly the case when considering infrastructure such as middleware products. Systems of this type typically sit idle until an external event occurs and then process the event. Given that many external events can occur simultaneously and that the interarrival time of such events may be lower than the time taken to process them, this kind of information-based system is inherently concurrent, with many operations being executed at once.

EXAMPLE Consider an e-commerce system that uses a message-based approach to processing transaction requests. In such a system, when a request arrives, it is translated into a message that is queued for the appropriate functional element that can process it. In order to prevent message queues growing too long and to make efficient use of processing resources, the processing element will need to process a number of messages concurrently. In this case, there may be a large number of concurrent operations within the functional element, each one needing access to shared resources.

The Concurrency view is extremely relevant to systems that exhibit this kind of behavior. Use of the Concurrency view allows the concurrency design of such systems to be made explicit and helps interested stakeholders understand concurrency constraints and requirements. It also allows you to analyze the system to avoid common concurrency problems such as deadlocks or bottlenecks.

CONCERNS

Task Structure

The most important aspect of creating a Concurrency view is establishing the system's process structure, which identifies the overall strategy for using concurrency in the system and the set of processes across which the system's workload is partitioned. The process structure also defines how the functions of the system are distributed across the set of processes identified. It may be necessary to consider the use of operating system threads within processes or to abstract away from individual processes and consider groups of similar processes instead.

Note: Throughout this chapter, the word *task* is used as a generic term to describe a processing thread—whether it is a single operating system process, one thread within a multithreaded process, or some other software execution unit. Where the difference is significant, the terms *process* and *thread* are specifically used.

The aspects of the system's task structure that this view needs to address depend very much on the kind of system you are dealing with.

EXAMPLE A complex, small-footprint system may have only one or two operating system tasks but may need to use a very complex thread model to meet its efficiency and responsiveness goals. In this case, the focus of the task structure activity needs to be at the thread level.

A large enterprise system may be comprised of literally hundreds of concurrent processes, many containing dozens of threads. In this sort of system, the task structure activity needs to be at the level of groups of similar processes in order to focus on the architecturally significant aspects of the concurrency.

Mapping of Functional Elements to Tasks

The mapping of functional elements to tasks can have a significant effect on the performance, efficiency, resilience, reliability, and flexibility of your architecture, so this needs careful consideration. The key question to address is which functional elements need to be isolated from each other (and so placed in separate processes) and which need to cooperate closely (and so need to run within the same process).

Interprocess Communication

When functional elements reside within a single operating system process, communication among them is relatively simple because of their shared address space. While some coordination may be required (see the Synchronization and Integrity subsection), you can use any number of data structures to pass information among them. Similarly, a number of easily used control mechanisms (such as the procedure call and variants of it) can transfer control among elements as needed.

In contrast, when elements reside in different operating system processes, communication among them becomes more complex. This complexity increases if the processes also reside on different physical machines.

A number of interprocess communication mechanisms can be used to link elements in different processes, including remote procedure calls, messaging, shared memory, pipes, queues, and so on. Each has its own strengths, weaknesses, and constraints, and inappropriate use of these mechanisms can cause problems at the system level (e.g., queue latency causing response time problems). In order to deliver a system with an acceptable set of quality properties, the Concurrency view needs to consider and identify the set of interprocess communication mechanisms that will be used to provide the interelement communication required by the system's functional structure.

State Management

In many systems, the runtime state that system elements may be in is important to the correct operation of the system. These systems tend to be event-driven and exhibit a high degree of concurrency.

For these systems, a concern of the Concurrency view is to clearly define the set of states that each functional element of the system can be in at runtime, the set of valid transitions between those states, and the causes and effects of the interstate transitions. Such careful state management is a major factor in ensuring reliability and correct behavior for most concurrent systems. Again, if you are using a formal architectural style, it may define how the system's runtime state should be handled.

Note that this concern refers to the state of the runtime elements of the system. Another type of state management important to many information systems is the set of valid states and transitions for their core persistent information (business objects). However, this is a distinct concept of state, and we refer to persistent object state models as *lifecycles* to avoid any confusion between the two. Object lifecycles are discussed in Chapter 17 about the Information viewpoint.

Having said this, it is quite reasonable to consider state management in the Functional view—after all, the state of functional elements is what we're considering. However, our experience is that the design of the system's state management usually fits better in the Concurrency view. Those systems where state is important are usually those where concurrency is important too, and considering system-level state usually involves the consideration of the concurrency around it as well.

Synchronization and Integrity

As soon as more than one thread of control exists in the system, it is important to ensure that concurrent execution cannot result in corruption of information within the system. This concern applies at a number of levels in the system, from a shared variable within a multithreaded module at one end of the scale to critical corporate transaction data in shared data stores at the other.

An important concern for the Concurrency view to address is how concurrent activity will be coordinated so that the system operates correctly and maintains the integrity of the data within it.

Startup and Shutdown

When you have more than one operating system process in your system, startup and shutdown of the system can become more complicated to manage. Intertask dependencies may mean that tasks need to be started and stopped in very specific orders so that if some tasks fail to start, others will not be started. The system startup and shutdown dependencies are an important part of your concurrency design and need to be clearly understood by developers, testers, and administrators.

Task Failure

When functional elements reside in different processes or run on different threads, dealing with element failure becomes significantly more complex. This is because an element in one task cannot rely on another task being available when it needs to communicate with it, whereas when an element calls another one in the same task, it knows the element will be there. Your concurrency design needs to take into account this added possibility of failure and ensure that the failure of one task doesn't bring the entire system to a halt. In order to address this concern, you need a system-wide strategy for recognizing and recovering from task failure.

Reentrancy

Reentrancy refers to the ability of a software element to operate correctly when used concurrently by more than one processing thread. This is primarily a concern for software developers when designing their software elements. From an architectural perspective, reentrancy is an important constraint for certain elements, so the architecture must clearly define which modules need to be reentrant and which don't.

EXAMPLE If you are developing an e-mail server, the ability to support a great deal of concurrency is likely to be a key concern. Without this, it will be hard to use the e-mail server for large user populations who will want to send and receive e-mail simultaneously. You can take a number of approaches to achieve such concurrency, but for the sake of argument, let's assume that you have decided to implement the server by using a single operating system process and many (perhaps hundreds) of concurrent operating system threads running within it: some sending e-mail, some receiving e-mail, and some managing the server's internal state.

In this sort of environment, it is crucial to decide which of the elements of your system have to be reentrant and which don't. Any element involved in sending and receiving e-mail (e.g., a name resolution library that translates e-mail domains to network addresses) will need to be reentrant to ensure that it can be used simultaneously by many sending and receiving threads. Without such a guarantee, the name resolution library could be the source of many subtle problems later if its internal state could be corrupted by concurrent access.

The reentrancy needs of your architecture can also affect which third-party software elements you can use within the system and where you can use them.

TABLE 18–1 STAKEHOLDER CONCERNS FOR THE CONCURRENCY VIEWPOINT

Stakeholder Class	Concerns
Administrators	Task structure, startup and shutdown, and task failure
Communicators	Task structure, startup and shutdown, and task failure
Developers	All concerns
Testers	Task structure, mapping of functional elements to tasks, startup and shutdown, task failure, and reentrancy

Stakeholder Concerns

Typical stakeholder concerns for the Concurrency viewpoint include those shown in Table 18–1.

MODELS

System-Level Concurrency Models

The Concurrency view maps the functional elements onto runtime execution entities via a concurrency model. The concurrency model typically contains the following items.

- *Processes*: In this context, the term *process* refers to an operating system process, that is, an address space that provides an execution environment for one or more independent threads of execution. The process is the basic unit of concurrency in the design of the system. At the architecture level, the processes are normally assumed to be isolated from each other so that if one process wants to affect the execution of another, it must use an interprocess communication mechanism.
- *Process groups*: At the architecture level, it can often be useful to group individual processes so that a collection of closely related processes can be considered as a single entity at the system level. This can provide a useful abstraction that allows less important concurrency concerns to be deferred until subsystem design. An example is a database management system (DBMS). The important point from the system level is that the DBMS is a functional unit, accessed via well-defined interfaces, which runs in its own process or group of processes. However, the details of the exact number of processes it uses (e.g., how many logging processes run within the DBMS) and the function of each are almost certainly irrelevant to the architecture—indeed, this will probably be decided by a technical

specialist later in the design process. Using a process group in this situation makes it clear that a group of related processes will be used but defers the details of the set until later. The other common use for process groups is simply as a hierarchical structuring technique for large or complex systems that contain many processes. All of the processes may need description, but the use of process groups can make the process model easier to comprehend.

- *Threads*: In this context, the term *thread* refers to an operating system thread, that is, a thread of execution that can be independently scheduled within an operating system process. Threads are known as *lightweight processes* by some operating systems. At the level of system architecture, threads can often be ignored, with the details of their use being the responsibility of subsystem designers (perhaps with you guiding their use via design patterns in the Development view). For some systems, you do want to model the use of threads in at least some parts of the system. Threads are normally represented in process models via a decomposition of a process.
- *Interprocess communication*: When processes are running, they are assumed to be isolated from each other so that one process cannot change anything in another process. However, in most concurrent systems, processes do need to interact in order to coordinate their execution, request services from each other, and pass information among themselves. They achieve these interactions via a number of interprocess communication mechanisms, which are the connectors in the system's runtime architecture.

  The mechanisms available vary depending on the underlying technology platforms in use. However, interprocess communication mechanisms generally fall into one of these groups.

  - *Procedure call mechanisms* are all some sort of variation on an interprocess function call and are usually based on some form of remote procedure call or some sort of message-passing operation.
  - *Execution coordination mechanisms* allow two or more processes (or threads) to signal to each other when certain events occur. Coordination mechanisms include semaphores and mutexes and are typically limited to coordination between processes or threads running on the same physical machine.
  - *Data-sharing mechanisms* allow a number of processes to share one or more data structures and access them concurrently (possibly coordinating this access via coordination mechanisms). Data-sharing mechanisms include shared memory, distributed tuple spaces (like Linda and JavaSpaces) and simple, traditional mechanisms such as client/server databases and shared file storage.

As the architect, you need to specify the interprocess communication mechanisms used with some care because their selection can significantly affect the quality properties (such as performance and reliability) that the system exhibits. It is also important to note that the interprocess communication mechanisms you choose must be compatible with the element interfaces defined in the Functional view. If one functional element uses an interface from another functional element and the two are mapped to different processes, the processes must be linked by an interprocess communication mechanism that can support the constraints of the interface being used.

NOTATION You can represent the Concurrency view in a number of ways. Some of the more common notational approaches include UML, other formal notations, and informal notations, which we describe briefly here.

- *UML*: UML does not have particularly strong concurrency modeling facilities built into it but does include the notion of an active object that can be stereotyped as a process or as a thread. The problem for the architect is that you normally don't want to consider objects, just coarser-grained entities such as components. The solution to this is to use simple objects (defined only by their names) to represent the processes and threads and to nest threads inside processes and components inside both. One complication if you do use this approach is that many UML tools will not let you nest objects on your diagrams. If this is a concern, consider using a stereotyped package instead of a stereotyped object. You may also need to consider adding a process group stereotype, depending on the level of concurrency model you need to represent.

  Simple examples of interprocess communication, like remote procedure calls, can be represented by using standard UML intercomponent associations, with arrowheads indicating the direction of communication (and possibly using tagged values on the association to make the communication mechanism clear). More complex forms of interprocess communication (shared memory, semaphores, and so on) can be represented quite effectively by introducing further stereotypes and showing associations between the components in the tasks and the interprocess communication mechanisms they use.

  Figure 18–1 shows an example of UML being used for a concurrency model.

  This model shows how the system is implemented by using three processes (a client, a statistics service, and a statistics calculator) along with a process group to implement the Oracle DBMS instance. The concurrent activity between the Statistics Accessor and Statistics Calculator components needs to be coordinated because they are in different processes;

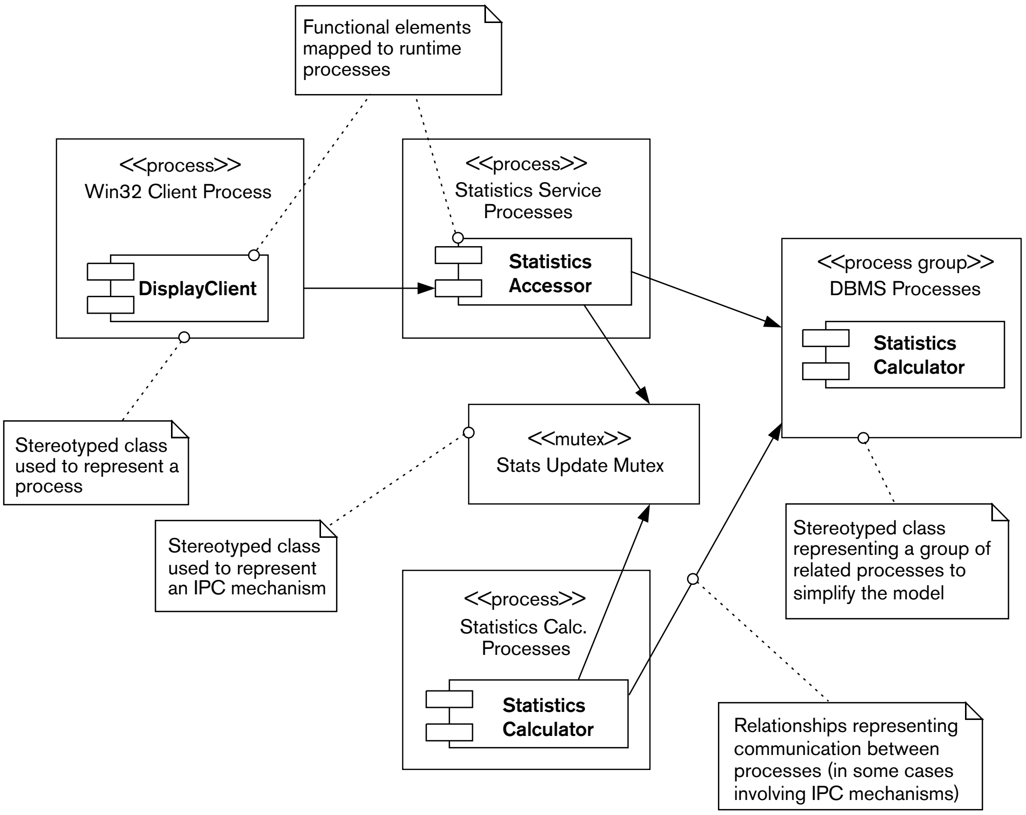
Functional elements mapped to runtime processes
<<process>> Win32 Client Process
DisplayClient
<<process>> Statistics Service Processes
Statistics Accessor
<<process group>> DBMS Processes
Statistics Calculator
Stereotyped class used to represent a process
<<mutex>> Stats Update Mutex
Stereotyped class used to represent an IPC mechanism
<<process>> Statistics Calc. Processes
Statistics Calculator
Stereotyped class representing a group of related processes to simplify the model
Relationships representing communication between processes (in some cases involving IPC mechanisms)

FIGURE 18–1 CONCURRENCY MODEL DOCUMENTED BY USING UML

a mutex is used to achieve this. The illustrated scenario is very simple, and there is little or no architecturally significant thread design in this model. Figure 18–2 shows a more involved model with more architecturally significant threading.

The concurrency model shown in Figure 18–2 illustrates a case where the process structure is very simple, namely, two processes that communicate via a socket stream. However, the thread structure in the DBMS Process instance is architecturally significant, and its structure and interthread coordination strategy needs to be documented and explained. The model shows that there is a single thread containing the Network Listener component, which communicates with between 1 and 40 threads that contain the four main query processing components via an interprocess communication queue. The Disk IO Manager component is hosted on its own thread, and there may be up to 10 instances of this

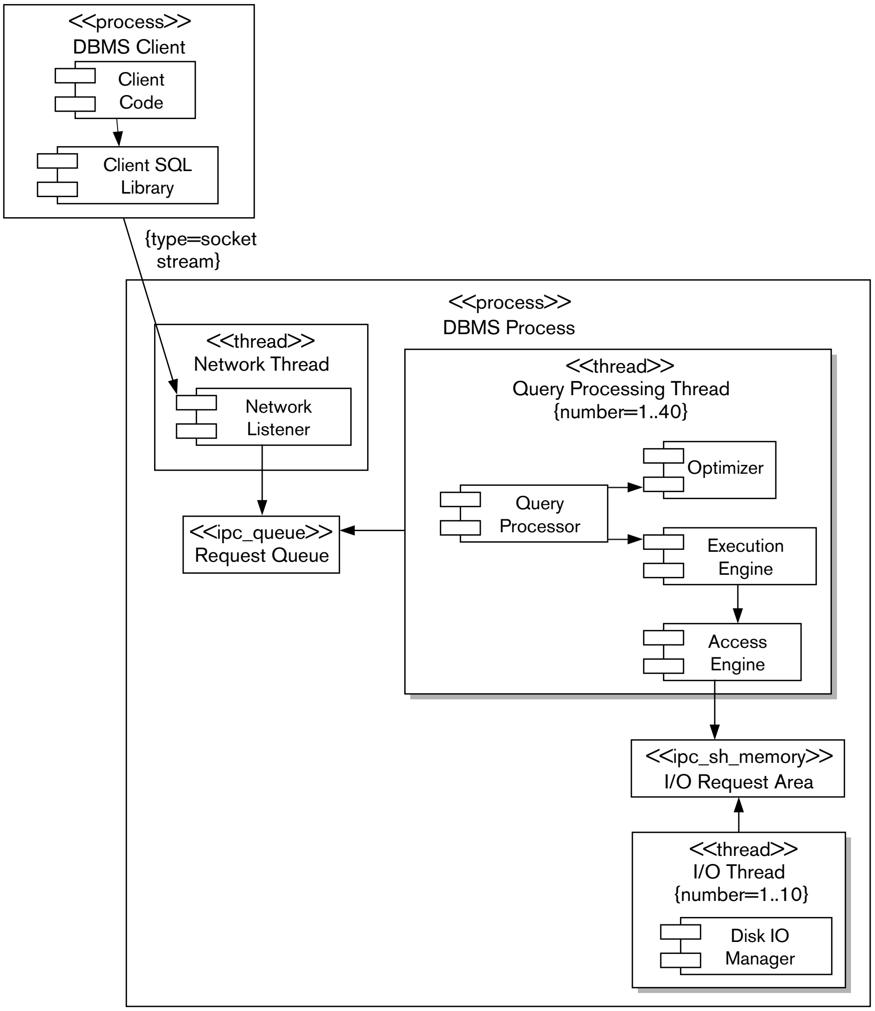

FIGURE 18–2 Thread-Based Concurrency Model

running. The Access Engine component communicates with the Disk I/O Manager instances via the shared memory mechanism.

- *Formal notations*: The real-time and control systems research community has created a number of concurrency modeling languages that allow the creation of process models. A number of these languages, such as LOTOS, Communicating Sequential Processes (CSP), and the Calculus for Communicating Systems (CCS), are formal and represented textually. Most of these languages are mathematical and fairly abstract, and they aren't widely understood or used in information system development. While this doesn't mean they can't be useful, we have yet to come across a large-scale industrial application of them to information systems. The problem with using these languages tends to center around the need to teach them to the interested stakeholders and their focus on abstract concurrency analysis (such as deadlock analysis) rather than concrete mapping of functional elements to tangible executable entities (such as operating system processes).

- *Informal notations*: In our experience, by far the most common notation used to represent process models is an informal one, created by the author of the model. Given the relatively small number of object types in a process model, an informal notation invented for the problem at hand often works very effectively as a communication tool as long as it is explained well enough. The notation needs to capture processes, process groups, threads, and the set of interprocess communication mechanisms in use. As long as the notation to represent each of these components is well defined, an informal notation often has much to recommend it. In particular, the notation can be kept simple and avoids the potentially awkward process of bending a general-purpose notation like UML to represent the model being described. The risk with informal notations is that people can misunderstand the model because they don't know the notation.

ACTIVITIES

Map the Elements to the Tasks. The first and most fundamental task when creating your process model is to work out how many processes you need and to decide which functional elements will run in which processes. In some cases, this is a very straightforward process—each functional element ends up being a process (or perhaps a process group). In other cases, there is a complex N:M mapping between functional elements and processes, with some elements partitioned between processes and other elements running in shared processes. The important point about this mapping is that you should introduce concurrency only where it is actually required. Concurrency adds complexity to the system and adds significant overhead to interelement communication when it must cross process boundaries. Therefore, add more processes to your system only if you need them for distribution, scalability, isolation, or other reasons guided by the requirements for your system.

Determine the Threading Design. The term *threading design* refers to the process of deciding on the number of threads to include in each system process and how those threads are to be allocated and used. In most cases, threading design is not something that the architect needs to get directly involved in—this is usually the job of the subsystem designers. However, you may get involved in designing and specifying general threading approaches or patterns that should be used at various points in the system in order to meet the system's required quality properties or to ensure consistency across the implementation.

Consider the Approach to Shared Resources. As soon as concurrency is introduced into the system, you must carefully consider how to share resources between concurrent threads of execution. Resource sharing is considered in some other parts of the architecture too (notably, the Information view), and the two activities might be best tackled as a single task. This isn't a book on concurrent computing, so we don't have space to discuss all of the options and potential pitfalls to consider when sharing resources. The simplistic advice is simply that whenever a resource (such as a piece of data in memory, a file, a database object, or a piece of shared memory) is shared among two or more concurrent threads of execution, it must be protected from corruption. This is usually achieved with some form of locking protocol. Like threading design, the details of resource sharing are rarely architecturally significant. Your role in relation to this is to ensure that suitable resource-sharing approaches are used where necessary and that the approach used is suitable in the overall context of the system and does not produce unacceptable side effects for the system as a whole.

Assign Priorities to Threads and Processes. Some tasks in your system may be more important than others. If you have tasks of different importance running on one machine, you need to control their execution so that the more important work gets done before the less important work. The normal method for achieving this is to use the facilities of the underlying operating system to assign priority levels to the different threads and processes. All modern operating systems provide this feature in roughly the same way. Tasks are explicitly or implicitly given runtime priorities. When the operating system's thread scheduler is choosing tasks to run, it considers the higher-priority tasks before the lower-priority ones, thus getting the important work done first. If you can avoid assigning explicit priorities to threads, in general do so—processing priorities can add a lot of complexity to your process model and can introduce subtle but serious problems. However, sometimes you can't avoid it. In these cases, keep the assignment of priorities as simple and as regular as possible, and analyze and prototype your approach to make sure that you aren't introducing problems worse than the one you're trying to solve.

Analyze Deadlocks. Having introduced concurrency into the system, you have also introduced the risk of the entire system grinding to a halt in unexpected ways. Wherever you have concurrency in the presence of shared resources, you always have the possibility of deadlock. In order to avoid this problem, You should check for deadlock if your system requires complex concurrency involving shared resources. You can use a number of modeling techniques for deadlock analysis, such as Petri Nets, which allow you to create a model of your processing threads and shared resources and then analyze the model to catch potential deadlock situations. With experience, it is also usually possible to perform effective deadlock analysis through careful, informal consideration of your concurrency model.

Analyze Contention. Wherever you have a number of tasks and shared resources, you almost always find contention. Contention occurs between tasks when more than one task requires a shared resource concurrently. The introduction of coordination mechanisms (like mutexes) inevitably introduces contention when workloads are high. If contention rises beyond a certain point, the system will slow dramatically, and little useful work will get done. In order to avoid this during normal operations, you need to analyze your shared resources from this point of view. The basis of the technique is to identify each of your possible contention points. Then, for each, estimate the likely number of concurrent tasks contending for the resource and how long each will need the resource for. This allows you to establish the likely wait times that each task experiences at each point and then to estimate how such contention will affect your processing times and throughput. Repeating the exercise for different workloads allows you to estimate the maximum theoretical workloads your system can possibly support.

State Models

A state model is used to describe the set of states that a system's runtime elements can be in and the valid transitions between the states. The set of states and transitions for one runtime element is known as a **state machine**, and the collection of all of the interesting state machines for your system forms the overall state model.

Usually, you will find that each system task identified in the concurrency model will have one or at most a few functional units mapped to it that are effectively in control of the task. These functional units normally have the system's interesting state models associated with them. If you create a state model, be sure to focus on these system elements so that the state model describes only architecturally significant information. You don't need to capture all of the state machines inside all of the system's elements; the AD needs to describe only state that is visible at the system level, not state that is hidden inside the system's elements.

An important decision to make before you start creating the state model is the set of semantics you want to use in your state machines. Modern state modeling notations (in particular, UML's statechart, discussed later) allow you to introduce a mind-boggling degree of complexity. You need to use such notations carefully if you want to produce a comprehensible model.

A basic state machine in the state model would normally contain the following types of entities.

- *State*: A state is an identifiable, named, stable condition during a runtime functional element's lifetime. States are normally associated with waiting for something (an event) to occur or performing some sort of operation.

- *Transition*: A state transition defines an allowable change in state, from one state to another, following the occurrence of an event. From a modeling point of view, transitions are normally considered to occur in zero time and so cannot be interrupted.
- *Event*: An event is an indication that something of interest has happened in the system (and is normally recognized by an operation being invoked on an element or a time period ending). Events are the triggers that cause transitions between states to occur.
- *Actions*: Actions are atomic (noninterruptible) pieces of processing that can be associated with a transition (so an event causes the transition to occur, and then an action is executed as part of the state transition).

More sophisticated state modeling notations allow additional modeling elements such as *guards* (Boolean conditions governing state transitions), *activities* (long, interruptible items of processing that can be associated with states), and *hierarchical states*.

NOTATION State models are typically represented by using a graphical notation derived in some way from the classic state transition diagram. The most popular variant in use today is probably the UML notation for representing state, the *statechart*. At the end of this subsection we briefly discuss other graphical notations as well as some nongraphical ones, but first we focus on UML's statechart.

- *UML*: A statechart is a flexible notation that can be used in a number of different ways at differing levels of sophistication. Deciding which parts of the notation to use is an important step before getting too far into the modeling process. Figure 18–3 shows a simple UML statechart for the Query Processor element we presented earlier in Figure 18–2.

  This statechart shows the basic notation for a UML statechart, with two normal states and the use of start and end pseudostates to indicate how the element's lifecycle begins and ends. The two normal states are linked with a pair of transitions showing how the element can move between these states. Each of the transition arrows is annotated with an event name and an action name to define how the transition is triggered and what happens as a result of the transition occurring. The event and action normally correspond to operations on the element.

  Note that the statechart shown in Figure 18–3 probably doesn't need to be included in the AD because it doesn't appear to be architecturally significant. The details of how the Query Processor element arranges its internal state don't seem to affect the rest of the system, so these details should be eliminated from the AD and captured in the appropriate software design documents. In fact, our experience is that it is often better to

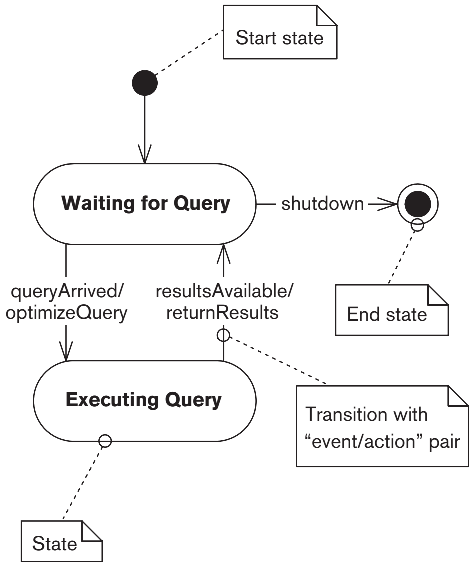

FIGURE 18–3 SIMPLE STATECHART FOR THE QUERY PROCESSOR ELEMENT

leave detailed state modeling to the subsystem design teams rather than to spend much time considering it at an architectural level.

A more sophisticated (and arguably more architecturally significant) example of a statechart is shown in Figure 18–4.

This example illustrates more of the UML statechart notation, in particular, composite and concurrent states. This diagram represents the state machine for a system element that is some sort of calculation engine. It has a single top-level state (Running) that is entered when the element is started and exited when a shutdown event is received (the `reset()` action is performed as part of that transition).

The Running state has been decomposed into four substates that comprise the business of running this element: Waiting for Data, Calibrating Metrics, Calculating, and Distributing Results. The transition arrows indicate the possible transitions between states (along with the events that cause the transitions and the actions that will be executed).

The Calculating state is interesting because it is a concurrent state, as you can see from the dashed line that bisects it. This means that while in the Calculating state, the element is actually in two concurrent substates (Calculating Values and Calculating Risk). When the activity associated with these states completes, the transition from the states is taken, and when both are complete, the element can leave the Calculating state.

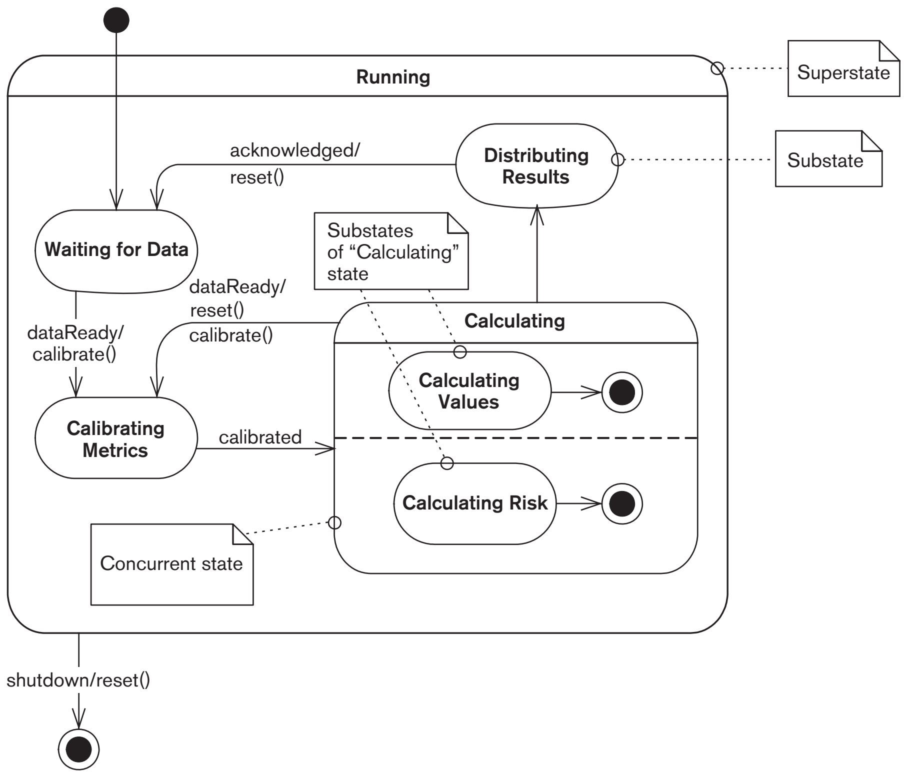

FIGURE 18–4 COMPOSITE AND CONCURRENT STATECHART

There appear to be two architecturally significant aspects to this state machine.

- If new input data becomes ready when the element is in the Calculation state, in-progress results are discarded (by executing the `reset()` action), and calculation starts again. In contrast, if this occurs while results are being distributed, the distribution process is not interrupted.
- No matter the state of the element, if a shutdown event is received, all processing immediately stops, the state is reset, and the element exits.

Of course, whether or not these facts are architecturally significant depends on the situation. However, we can make a reasonable argument that these facts are visible at the system level and thus can affect or be relied on by other system elements, and therefore, these facts need to be captured as part of the architecture.

An interesting point to note about the UML statechart is that its ability to show hierarchical state composition allows you to express architectural constraints on state models without needing to define the entire model. The statechart in Figure 18–5 illustrates this point.

This statechart distills one of the architecturally significant features from the statechart in Figure 18–4, namely, that a shutdown event must be immediately responded to in any running state and a reset of the element performed as part of shutdown. In effect, this documents an architectural constraint that the designer of the corresponding part of the system must respect; but while clearly defining this constraint, the statechart leaves the details of the lower-level states to the subsystem designer.

- *Other graphical notations*: In addition to UML, many other graphical notations exist for modeling state. Some of the better-known ones include simple state transition diagrams, Petri Nets, SDL, and David Harel's original Statecharts. All of these notations have strengths and weaknesses when compared to each other and to UML statecharts, and they are worth considering if UML statecharts cause you problems. However, the standardization of and wide familiarity with UML statecharts means that in general they should probably be your first choice. The Further Reading section at the end of this chapter contains some references to information about these notations.
- *Nongraphical notations*: In principle, the graphical notations can all be represented in a textual form (and indeed many graphical state modeling notations do define an equivalent textual form). Similarly, a number of primarily textual formalisms for modeling and analyzing state can be represented in a graphical form (for an accessible example, look at the Finite State Processes language). A textual state model can be useful when the model needs to be processed in some way by machine, but for human readers it is almost always better to use a graphical notation where possible.

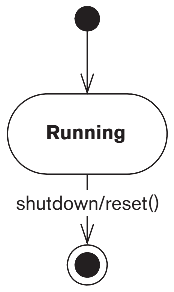

FIGURE 18–5 ARCHITECTURAL CONSTRAINT STATECHART

ACTIVITIES

Define the Notation. Before starting to create your state model, spend some time working out your needs for the modeling notation and defining how you will use your chosen notation.

Identify the States. The core activity when creating a state model is to work out what states your system elements can be in and the processing (if any) associated with each state. Beware of accidentally modeling activities as states; this is a common modeling mistake. If in doubt, try considering your state machine as a UML activity diagram. If you can do this, you have probably modeled activities rather than states. When performing state identification at the architectural level, focus on the states that are visible from outside the element and thus have a system-wide effect.

Design the State Transitions. Once you know what states your elements can be in, design a set of transitions that allows them to move between the states correctly. For each transition, clearly identify how the transition is triggered and any (atomic) actions that must be performed as a side effect of traversing it. Make sure that the events and actions you identify can be supported by the operations and state of the element for which you are designing a state machine.

PROBLEMS AND PITFALLS

Modeling of the Wrong Concurrency

When considering the concurrency design of a system, it is easy to get bogged down in the details of the internal concurrency and state design of each element. It's not part of your job as an architect to design detailed thread models that define how individual threads in a server will be allocated, used, and freed, along with all of the coordination between them. Remember that your role is to concentrate on the system rather than all of the details of each element. The concurrency that you should be concerned with is the architecturally significant concurrency, that is, the overall concurrency structure, the mapping of functional elements to that structure, and the system-level state model. You may also be involved in specifying common approaches such as design patterns that need to be used for the concurrency within elements, but in general you should not need to design all of the details—this will only distract you from the system-level problems (which are often quite enough to worry about).

RISK REDUCTION

- Focus on architecturally significant aspects of concurrency.
- Involve the lead software developers as early as possible so they can work on the more detailed aspects of this problem.

Excessive Complexity

Simplicity should always be an aim when designing a system. Simple designs are easier to create, analyze, build, deliver, and support. However, this is particularly important when considering concurrency because it is fundamentally difficult to understand. As we have seen, the price of complex concurrency can be very high at design time, implementation time, and beyond. More software engineering hours have probably been wasted on reworking problematic concurrency than on almost anything else. Simplicity in your concurrency approach will have a major positive impact on the amount of effort required to deliver and support your system.

RISK REDUCTION

- Be sure that all of the concurrency you introduce is justified in terms of stakeholder benefits.
- When designing state models, use the simplest subset of notation possible to capture your state machines in order to encourage a simple state model.

Resource Contention

Resource contention usually manifests itself as excessive activity in small, specific parts of the system (colloquially known as *hot spots*). Careful and early analysis of the concurrency model for potential contention can help you avoid such problems, but in reality, as soon as one resource contention point is eliminated, the next one will emerge. Therefore, tackling resource contention is normally a process of reducing the contention to an acceptable level.

RISK REDUCTION

- Analyze your system as it is being designed to spot resource contention as early as possible, and design around it. Use your usage scenarios to predict which parts of the system are likely to encounter high levels of concurrency, and focus your attention in these regions.
- Reduce contention by decomposing locks on large resources into a number of finer-grained locks, thus reducing the amount of time locks are held.
- Consider alternative locking techniques such as optimistic locking that reduce the time locks are held.
- Eliminate shared resources where you can.
- If possible, reduce the amount of concurrency you need around problematic contention points.

Deadlock

Deadlock occurs when a thread requires access to a resource that has already been locked by another thread. Like resource contention, you can often avoid deadlock through early and thorough analysis of the system. Danger points are those parts of the system where different types of processing threads need access to a number of the same resources. Where you find potential deadlock points, you will probably need to redesign the system to avoid the problem.

RISK REDUCTION

- Where possible, ensure that resources are always allocated to tasks in a fixed order.
- Attempt to isolate parallel threads in such a way that deadlock between them is impossible.
- Certain commercial products that use locks (such as database management systems) provide significant assistance with handling deadlock—in most cases, recognizing it and breaking it by terminating one or more of the problematic transactions. These technologies can be very useful when dealing with deadlock, but their use often needs to be carefully designed into the system so that such deadlock recovery actions are handled correctly.

Race Conditions

A race condition is problematic behavior that results from unexpected dependence on the relative timing of events. It usually occurs when two or more tasks are attempting to perform the same action concurrently. The tasks race for the resource, and the first one to reach the appropriate point in the program code wins and performs the action.

Race conditions are only problematic when they are unplanned because the system has not been designed to cope with more than one task performing the action concurrently. In these cases, information can be corrupted or lost, and the system can behave in unpredictable ways. A classic example is a system-wide data structure in an operating system process that a number of threads can update. If multiple tasks try to update the data structure concurrently (e.g., to increment a counter indicating the number of requests accepted), the resulting value will be undefined and very likely incorrect.

RISK REDUCTION

- Ensure that there are no unprotected, shared system-level resources that can cause race conditions.

- Automatically introduce protection mechanisms for all potentially shared resources.
- Ensure that the definition of each element interface clearly states whether or not the interface is reentrant.

Checklist

- Is there a clear system-level concurrency model?
- Are your models at the right level of abstraction? Have you focused on the architecturally significant aspects?
- Can you simplify your concurrency design?
- Do all interested parties understand the overall concurrency strategy?
- Have you mapped all functional elements to a process (and thread if necessary)?
- Do you have a state model for at least one functional element in each process and thread? If not, are you sure the processes and threads will interact safely?
- Have you defined a suitable set of interprocess communication mechanisms to support the interelement interactions defined in the Functional view?
- Are all shared resources protected from corruption?
- Have you minimized the intertask communication and synchronization required?
- Do you have any resource hot spots in your system? If so, have you estimated the likely throughput, and is it high enough? Do you know how you would reduce contention at these points if forced to later?
- Can the system possibly deadlock? If so, do you have a strategy for recognizing and dealing with this when it occurs?

Further Reading

The area of concurrency has been studied and written about widely, although not many books consider it from an architect's perspective.

A good overview of concurrency in general (albeit with a Java-specific slant) and a good introduction to modeling and analysis appears in Magee and Kramer [MAGE99]; this book also introduces the Finite State Processes language mentioned earlier. Unfortunately, Cook and Daniels [COOK94] is out of print; however, it is well worth tracking down a copy as it contains a