Miro Samek

C/C++ Users Journal™
Advanced Solutions for Professional Developers

Quantum Programming for Embedded Systems: Toward a Hassle-Free Multithreading

It is well known that multithreading changes everything. Even more so with embedded code.

Even the smallest embedded processor must typically handle a variety of activities and manage multiple devices at the same time. Dealing with the concurrency arising naturally in embedded systems has always been one of the most challenging aspects of embedded systems programming. The time-honored approaches range from simple foreground/background loops (`main()`+ISR) to RTOS-based preemptive multithreading with sophisticated mechanisms of inter-task synchronization and communication. Unfortunately, all these methods have serious drawbacks. The simple `main()`+ISR technique suffers typically from poor background loop response, since in the worst case an event signaled by an ISR must wait for processing for the full pass through the background loop. Consequently, the technique is difficult to scale up, because any new functionality extends the loop or adds to the ISRs, which tend to become longer than they should. The RTOS-based approach better addresses the scalability problem, but comes with its own set of pitfalls such as race conditions, deadlock, starvation, priority inversion, and many more. All these hazards make programming with a preemptive RTOS incomparably more difficult, error-prone, and risky than the sequential programming most of us are familiar with. All too many programmers vastly underestimate the true costs and skills needed to program directly with such RTOS mechanisms as semaphores, mutexes, monitors, message queues, mailboxes, critical sections, condition variables, signals, and many others. The truth is all that these mechanisms are tricky to use and often lead to subtle bugs that are notoriously hard to reproduce, isolate, and fix.

In this article, I describe a fundamentally different approach to building concurrent embedded software based on application frameworks, state machines, and active objects. I call this approach QP (Quantum Programming) [1], because it closely resembles the "architecture" of quantum systems composed of microscopic particles governed by laws of quantum mechanics. As it turns out, all microscopic objects (such as atoms or molecules) spend their lives in discrete states (quantum states), can change state only through uninterruptible transitions (quantum leaps), and interact only by exchanging events (intermediate virtual bosons). This article shows you how this model can help you reap the benefits of multithreading (such as low latencies and good CPU utilization) while avoiding most of the risks and pitfalls associated with the traditional approaches.

In the QP model, the main units of software decomposition are active objects that collaborate to collectively deliver the intended functionality. Active objects are concurrent software objects (each embedding a state machine) that interact with one another solely by sending and receiving events. Within an active object, events are processed sequentially in an RTC (run-to-completion) fashion, while the event-passing framework encapsulates all the details of thread-safe event exchange and queuing. These characteristics enable active objects to be internally programmed with purely sequential techniques, thus making the software construction incomparably easier and safer by eliminating most of the traditional concurrency hazards.

Obviously, the active object-based computing model is not new (see the sidebar "From Actors to Active Objects"). Perhaps the most convincing evidence for the practicality and generality of the active objects-based approach provides various design automation tools for embedded real-time systems, which successfully ad-

dress an amazingly wide range of applications (see sidebar "Design Automation Tools for Embedded Real-Time Systems"). Virtually every such tool incorporates a variant of an active object-based framework. For instance, ROOM (Real-Time Object-Oriented Modeling) calls such a framework the "ROOM virtual machine" [3]. A visualSTATE tool from IAR Systems calls it a "visualSTATE engine" [5]. A UML-compliant design automation tool from I-Logix, Rhapsody, calls it an OXF (Object Execution Framework) [6].

From Actors to Active Objects

The concept of autonomous software objects communicating by message passing dates back to the late 1970s, when Carl Hewitt and colleagues developed the notion of an actor [2]. In the 1980s, actors were all the rage within the distributed artificial intelligence community, much as agents are today. In the 1990s, methodologies like ROOM [3], adapted actors for real-time computing. More recently, the UML specification has introduced the concept of an active object that is essentially synonymous with the notion of an actor [4]. Active objects in the UML specification are the roots of threads of control in multitasking systems and engage one another asynchronously via events. The UML specification further proposes the UML variant of statecharts, with which to model the behavior of event-driven active objects.

In QP, I use the UML term "active object," rather than the more compact "actor," to avoid confusion with the other meaning of the term "actor" that the UML specification uses in the context of use cases.❑

QP incorporates a minimal implementation of an active object-based framework, which I call QF (Quantum Framework), with goals similar to the ROOM virtual machine or Rhapsody's OXF. However, unlike the frameworks buried inside design automation tools, the QF is intended for direct (manual) coding and is currently available either in C or in C++. The framework has a small memory footprint (around 5 KB in most cases) and executes the applications deterministically. It can be freely embedded in commercial products.

Design Automation Tools for Embedded Real Time Systems

Some of the design automation tools for embedded real-time systems capable of automatic code synthesis from visual models include:

* Statemate and Rhapsody (I-Logix, <www.ilogix.com>),
* Rational Suite Development Studio Real-Time (Rational Software Corp., <www.rational.com>),
* BetterState (WindRiver Systems, <www.wrs.com>),
* Stateflow (MathWorks, <www.mathworks.com>),
* VisualState (IAR, <www.iar.com>),
* ObjectGeode (Telelogic, <www.telelogic.com>).❑

Pitfalls of Conventional Multithreading

The classic DPP ("dining philosophers" problem) posed and solved by Edsger Dijkstra [7] illustrates well the basic challenges of multithreading. As shown in Figure 1, five philosophers are gathered around a table with a big plate of spaghetti in the middle. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each philosopher is a fork. The life of a philosopher consists of alternate periods of thinking and eating. When a philosopher wants to eat, he tries to acquire forks. If successful in acquiring two forks, he eats for a while, then puts down the forks and continues to think. (An alternative oriental version replaces spaghetti with rice and forks with chopsticks.) The key question is: Can you write a program for each philosopher that never gets stuck?

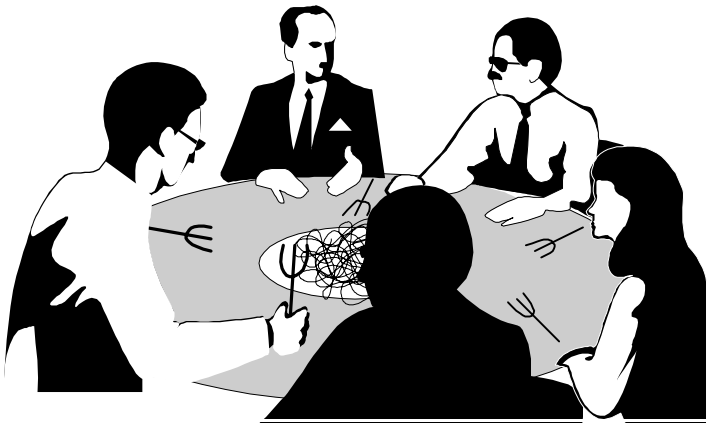

Figure 1 The Dining Philosophers

Although mostly academic, the problem is motivated by the practical issue of how to assign resources to concurrent processes that need the resources to do their jobs; in other words, how do you manage resource allocation. The idea is that a finite set of threads is sharing a finite set of resources, and each resource can be used by only one thread at a time.

The fundamental problem is synchronizing access to the forks. In the simplest (and most naïve) solution, the philosopher threads might synchronize access to the forks using shared memory. To acquire a fork, a philosopher would need to test the corresponding shared flag and proceed only if the flag is cleared. After acquiring the fork, the philosopher would immediately set the corresponding flag to indicate that the fork is in use. However, this solution has a fundamental flaw. If philosopher A preempts philosopher B just after philosopher B acquires a fork but before the flag has been set, then philosopher A could incorrectly acquire the fork that philosopher B already has (the corresponding flag is still cleared). This situation is called a *race condition*. It occurs whenever one thread gets ahead of another in accessing shared data that is changing.

```
 1 enum { N = 5};                        // number of dining philosophers
 2 static HANDLE fork[N];               // model forks as mutex semaphores
 3
 4 void think(long n) {. . . }       // called when philosopher n thinks
 5 void eat(long n) { . . . }          // called when philosopher n eats
 6
 7 long WINAPI philosopher(long n) {           // task for philosopher n
 8    for (;;) {                          // philosopher task runs forever
 9       think(n);           // first the philosopher thinks for a while
10                   // after thinking the philosopher becomes hungry...
11       WaitForSingleObject(fork[(n+1)%N], INFINITE); // get left fork
12       WaitForSingleObject(n], INFINITE);            // get right fork
13       eat(n);                  // got both forks, can eat for a while
14       ReleaseMutex(fork[(n+1)%N]);               // release left fork
15       ReleaseMutex(fork[n]);                    // release right fork
16    }
17    return 0;
18 }
```

Listing 1 Simple (and incorrect) solution to the dining philosophers problem implemented with Win32 API. The explicit fork flags are superfluous in this solution because they are replaced by the internal counters of the mutexes.

Clearly, the philosopher threads need some method to protect the shared flags, such that access to the flags is mutually exclusive, meaning only one philosopher thread at a time can test and potentially set a shared flag.

There are many methods of obtaining exclusive access to shared resources, such as performing indivisible test-and-set operations, disabling interrupts, disabling task switching, and locking resources with semaphores. The solution based on semaphores is, in this case, the most appealing because it simultaneously addresses the problems of mutual exclusion and *blocking* the philosopher threads if forks are unavailable. Listing 1 shows a simple semaphore-based implementation.

The solution from Listing 1 still has a major flaw. Your program might run for a few milliseconds or for a year (just as the first naïve solution did), but at any moment, it can freeze with all philosophers holding their left fork (Listing 1, line 12). If this happens, nobody gets to eat — ever. This condition of indefinite circular blocking on resources is called *deadlock*.

EXERCISE: Execute the dining philosophers example from Listing 1 (the code is available from <www.cuj.com/code>). Name at least three factors that affect the probability of a deadlock. Modify the code to increase this probability. After the system (dead)locks, use the debugger to inspect the state of the forks and the philosopher threads.

Once you realize the possibility of a deadlock (which generally is not trivial), be careful how you attempt to prevent it. For example, a philosopher can pick up the left fork; then if the right fork isn't available for a given time, put the left fork down, wait, and try again. (This is a big problem if all philosophers wait the same amount of time — you get the same failure mode as before, but repeated.) Even if each philosopher waits a random time, an unlucky philosopher could starve (never get to eat). Starvation is only one extreme example of the more general problem of non-determinism because it is virtually impossible to know in advance the maximum time a philosopher might spend waiting for forks (or how long a philosopher thread is preempted in a preemptive multi-tasking system).

Any attempt to prevent race conditions, deadlock, and starvation can cause other, more-subtle, problems associated with fairness and sub-optimal system utilization. For example, to avoid starvation, you might require that all philosophers acquire a semaphore before picking up a fork. This requirement guarantees that no philosopher starves, but limits parallelism dramatically (poor system utilization). It is also difficult to prove that any given solution is fair and does not favor some philosophers at the expense of others. The main lesson of dining philosophers is that multithreaded programming is incomparably harder than sequential programming, especially if you use a traditional approach to multithreading. The conventional design requires a deep understanding of the time domain and operating system mechanisms for inter-thread synchronization and communication, such as various kinds of semaphores, monitors, critical sections, condition variables, signals, message queues, mailboxes, and so on.

Therac-25 Story

The problems associated with multithreading are not just academic. Perhaps the most publicized real-life example of the "free threading" approach to concurrent event-driven software is the Therac-25 story. Between June 1985 and January 1987, a computer-controlled radiation therapy machine called the Therac-25 massively overdosed six people. These accidents have been described as the worst in the 35-year history of medical accelerators. To attribute software failure as the single cause of the accidents is a serious mistake and an oversimplification. However, the Therac-25 story provides an example of an inherently unsafe and practically unfixable software design that resulted mostly from the (still widely applied) "free threading" approach to concurrency combined with the bottom-up (ad hoc) approach to building event-driven software.

The detailed analysis performed by Nancy Leveson in *Safeware: System Safety and Computers* (Addison Wesley, 1995) revealed that the ultimate root causes of all the Therac-25 accidents were various *race conditions* within the software. For example, one such race condition occurred between the processes of setting up the bending magnets in preparation for treatment and accepting treatment data from the console. If a skillful operator could enter all the required data within about eight seconds (the time needed to set up the magnets),

then occasionally the machine could end up in an inconsistent configuration (partially set for X-ray treatment and partially set for electron treatment). These exact conditions occurred on April 11, 1986, when the machine killed a patient.

Although the Therac-25 software was developed almost three decades ago and was written in PDP-11 assembly language, it illustrates well the main problem of not knowing exactly which mode of operation (state) the software is in at any given time. Actually, the software has no notion of any single mode of operation, but rather tightly coupled and overlapping conditions of operation defined ambiguously by values of multiple variables and flags, which are set, cleared, and tested in complex expressions scattered throughout the code. This approach can lead to subtle bugs in an application of even a few hundred lines of code. The Therac-25 case, however, shows that, when additionally compounded with concurrency issues, the ad hoc approach leads to disastrous, virtually uncorrectable designs. For example, in an attempt to fix the Therac-25 race condition described earlier, the manufacturer (Atomic Energy of Canada Limited) introduced another shared variable controlled by the keyboard handler task that indicated whether the cursor was positioned on the command line. If this variable was set, then the prescription entry was considered still in progress and the value of the `Tphase` state variable was left unchanged. The following items point out some inherently nasty characteristics of such ad hoc solutions.

- Any individual inconsistency in configuration seems to be fixable by introducing yet another mode-related (extended state) variable.
- Every new extended state variable introduces more opportunities for inconsistencies in the configuration of the system. Additionally, if the variable is shared among different threads (or interrupts), it can introduce new race conditions.
- Every such change perpetuates the bad design further 3 and makes it exponentially more difficult (expensive) to extend and fix, although there is never a clear indication when the code becomes unfixable (prohibitively expensive to fix).
- It is practically impossible to know when all inconsistent configurations and race conditions are removed. In fact, most of the time during computations, the configuration is inconsistent, but you generally won't know whether it happens during the time windows open for race conditions.
- No amount of testing can detect all inconsistencies and timing windows for race conditions.
- Any change in the system can affect the timing and practically invalidate most of the testing.

You might think that the Therac-25 case, albeit interesting, is no longer relevant to contemporary software practices. This has not been my experience. Unfortunately, the architectural decay mechanisms just recounted still apply today, almost exactly as they did three decades ago. The modes of architectural decay haven't changed much because they are characteristics of the still widely practiced bottom-up approach to designing reactive systems mixed with the conventional approach to concurrency.

Computing model of QP

For the reasons just listed and others, experienced embedded programmers have learned to be extremely wary of shared (global) data and various mutual exclusion mechanisms (such as mutexes). Instead, they structure their applications around the event-passing paradigm. For example, Jack Ganssle offers the following advice [9]:

> *Novice users all too often miss the importance of the sophisticated messaging mechanisms that are a standard part of all commercial operating systems. Queues and mailboxes let tasks communicate safely... the operating system's communications resources let you cleanly pass a message without fear of its corruption by other tasks. Properly implemented code lets you generate the real-time analogy of object-oriented programming's (OOP) first tenet: encapsulation. Keep all of the task's data local, bound to the code itself and hidden from the rest of the system.*

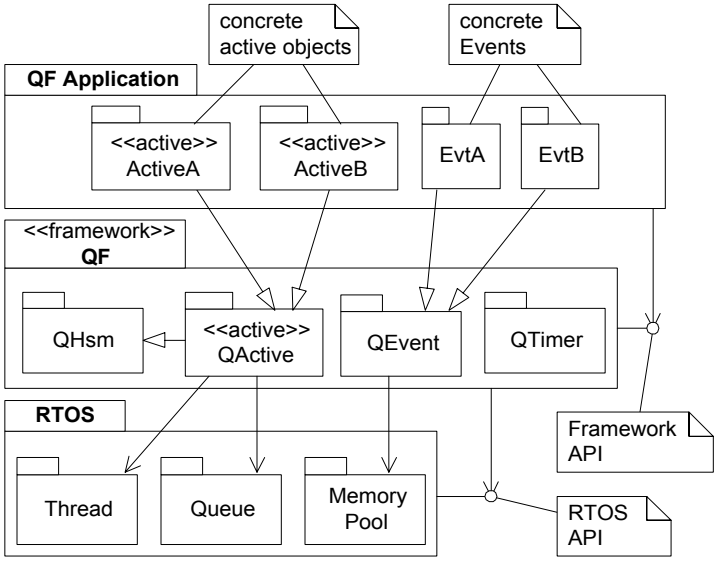

Figure 2 UML package diagram illustrating relationships among RTOS, QF, and QF-Application

In fact, this piece of advice provides a good, albeit incomplete, description of the active object computing model. It captures very well the most important characteristics of active objects—their opaque *encapsulation shell*, which strictly separates the internal structure of an active object from the external environment. The only objects capable of penetrating this shell, both from the outside and from the inside, are event instances.

Another crucial aspect of QP is the universal use of hierarchical state machines (HSMs) for modeling the internal structure of active objects. State machines are ideal for coding event-driven systems, because they make the event handling explicitly dependent on both the nature of the event and on the context (state) of the system. Programmatically, state machines enable the crisp definition of the state at any time, which leads to dramatic reduction of convoluted paths through the code and simplification of the conditions tested at each branching point (recall the Therac-25 story).

The high-level structure of the QF (shown in Figure 2) is typical for any active object–based framework. The design is layered with a real-time operating system (RTOS) that provides a foundation for multithreading and basic services like event queues and memory pools. (For simpler designs, the QF can operate without an underlying RTOS, effectively replacing it.) Based on these services, the QF supplies the `QActive` base class to derive concrete active objects. `QActive` inherits from `QHsm`, which means that active objects inherit are hierarchical state machines. Additionally, `QActive` gives active objects a thread of execution and an event queue. An application built from the QF extends the framework by subclassing `QActive` and `QEvent`. The application uses QF communication and timing services through the QF API; however, the application typically should not need to access the RTOS API directly.

Dining Philosophers Revisited

Earlier in Listing 1, you saw one of the conventional approaches to the dining philosophers problem. Here, I'd like to show you a design based on active objects and the Quantum Framework (QF). The purpose of this discussion is to walk you quickly through the main points without slowing you down with the full-blown details.

Active object–based programming requires a paradigm shift from the conventional approach to multithreading. Whereas in the conventional approach you mostly think about shared resources and various synchronization mechanisms, in the active object–based approach, you think about partitioning the problem into active objects and about exchanging events among these objects. Your goal is to break up the problem in a way that

requires minimal communication. The generic design strategy for handling shared resources is to encapsulate them inside a dedicated active object and to let that object manage the shared resources for the rest of the system (i.e., instead of sharing the resources directly, the rest of the application shares the dedicated active object). When you apply this strategy to the DPP (Figure 1), you will naturally arrive at a dedicated active object to manage the forks (call it `Table` for this example). The `Table` active object is responsible for coordinating `Philosopher` active objects to resolve contentions over the forks. It's also up to the `Table` active object to implement it fairly (or unfairly if you choose). A `Philosopher` active object needs to communicate two things to `Table`: (1) when it is hungry and (2) when it finishes eating. `Table` needs to communicate only one thing to a hungry `Philosopher`: permission to eat. The sequence diagram in Figure 3 shows two scenarios of possible event exchange in the DPP. In the case of `Philosopher` *n*, `Table` can grant permission to eat immediately. However, `Philosopher` *m* has to wait in the hungry state until forks become available (e.g., two forks become free when `Philosopher` *n* is done eating).

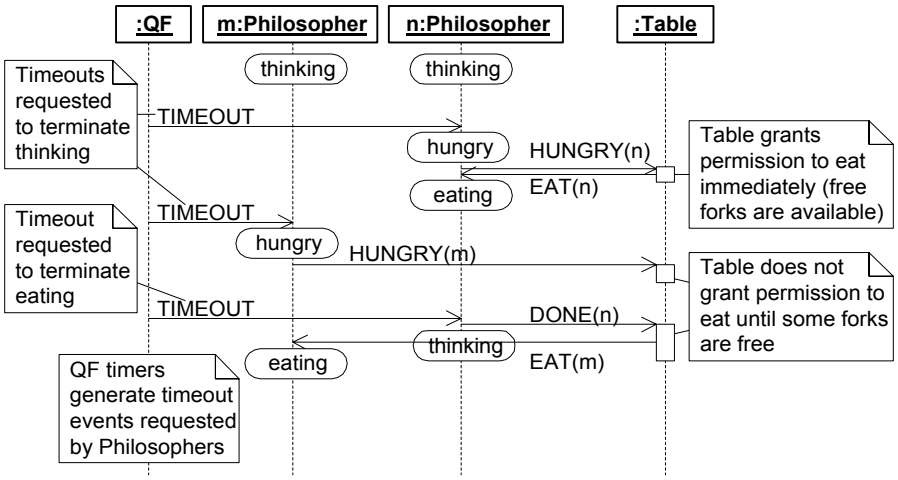

Figure 3 Sequence diagram showing event exchange in the active object–based solution of the DPP

The class diagram in Figure 4 shows that the QF application comprises the `Table` and `Philosopher` active objects and the specialized `TableEvt` class. This diagram has a typical structure for an application derived from a framework. Concrete application components (active objects and events in this case) derive from framework base classes (from `QActive` and `QEvent`) and use other frame-work classes as services. For example, every Philosopher has its own `QTimer` (quantum timer) to keep track of time when it is thinking or eating.

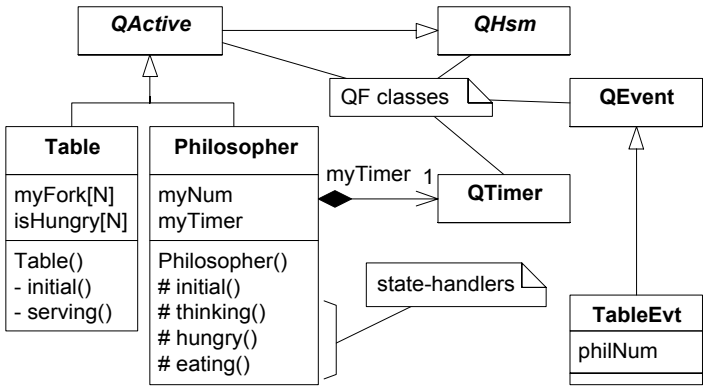

Figure 4 Table and Philosopher active objects and the TableEvt class derived from the QF base classes

The `Table` and `Philosopher` active objects derive indirectly from `QHsm`, so they are state machines. In fact, your main concern in building the application is elaborating their statecharts. Figure 5a shows the statechart

associated with `Table`. It is trivial because `Table` keeps track of the forks and hungry philosophers by means of extended state variables (`myFork[]` and `isHungry[]` arrays, Figure 4), rather than by its state machine.

The `Philosopher` state machine (Figure 5b) clearly shows the life cycle of this active object consisting of states thinking, hungry, and eating. This statechart publishes the `HUNGRY` event on entry to the hungry state and the `DONE` event on exit from the eating state because this exactly reflects the semantics of these events. An alternative approach — to publish these events from the corresponding `TIMEOUT` transitions — would not guarantee the preservation of the semantics in potential future modifications of the state machine.

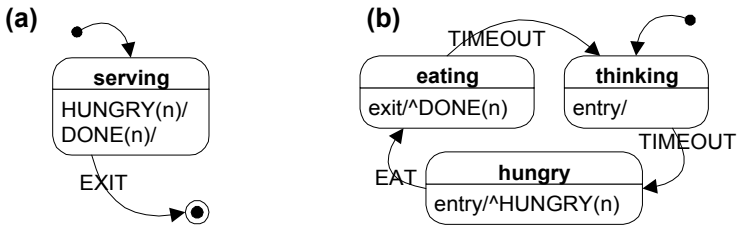

Figure 5 (a) Table statechart and (b) Philosopher statechart; the ^DONE(n) notation indicates propagation of the DONE event with parameter *n* (for the *n* -th Philosopher)

Note that, externally, the active object–based solution makes no reference whatsoever to the forks, only to the philosopher's right to eat (refer to the sequence diagram in Figure 3). Interestingly, Dijkstra [7] proposed a similar solution as the "most natural." His formal analysis of the problem further deduced a need for each philosopher to have an "I am hungry" state, in which the philosopher would wait for permission to eat. Overall, Dijkstra's solution has many remarkable parallels to the active object–based design.

- Each `Philosopher` life cycle goes through thinking, hungry, and eating states (refer to the `Philosopher` statechart in Figure 5b).
- Each `Philosopher` has a private semaphore on which to wait when hungry (waiting on this semaphore corresponds to blocking the `Philosopher` thread on the private event queue inherited from the `QActive` base class).
- A `Philosopher` starts eating when in the hungry state and neither neighbor is eating (in active object–based design, this is handled by `Table`; refer to the handling of the `HUNGRY` event in Figure 3)
- When a `Philosopher` finishes eating, any hungry neighbor starts eating (in active object–based design, this is handled by the `Table` active object; refer to the handling of the `DONE` event in Figure 3)

EXERCISE: Execute the QP version of the dining philosophers (the Windows version is available from <www.cuj.com/code>). Check that the Table hands out the forks correctly to the hungry `Philosophers`.

Unprecedented Flexibility

You might object rightly that the active object-based solution to the DPP is bigger (when measured in lines of code) than a typical traditional solution. However, as I try to demonstrate in the following discussion, none of the traditional approaches to DPP are in the same class as the active object–based solution.

The active object–based solution might be a bit larger than the traditional solution, but the QF-based code is straightforward and free of all concurrency hazards. In contrast, any traditional solution deals directly with low-level mechanisms, such as semaphores or mutexes, and therefore poses a risk of deadlock, starvation, or simply unfairness in allocating CPU cycles to philosopher tasks.

However, what sets the active object–based solution truly apart is its unparalleled flexibility and resilience to change. Consider, for example, how the initial problem could naturally evolve into a more realistic applica-

tion. For instance, starting a conversation seems a natural thing for philosophers to do. To accomplish such a new feature (interaction among the philosophers), a traditional solution based on blocking philosopher threads would need to be redesigned from the ground up because a hungry (blocked) philosopher cannot participate in the conversation. Blocked threads are unresponsive.

A deeper reason for the inflexibility of the traditional solution is using blocking to represent a mode of operation. In the traditional solution, when a philosopher wants to eat and the forks aren't available, the philosopher thread blocks and waits for the forks. That is, blocking is equivalent to a very specific mode of operation (the hungry mode), and unblocking represents a transition out of this mode. Only the fulfillment of a particular condition (the availability of both forks) can unblock a hungry philosopher. The whole structure of the intervening code assumes that unblocking can only happen when the forks are available (after unblocking, the philosopher immediately starts to eat). No other condition can unblock the philosopher thread without causing problems. Blocking is an inflexible way to implement modal behavior.

In contrast, the active object–based computing model clearly separates blocking from the mode (hungry) of operation and unblocking from the signaling of certain conditions (the availability of forks). Blocking in active objects corresponds merely to a pause in processing events and does not represent a particular mode of the active object. Keeping track of the mode is the job of the active object's state machine. A blocked philosopher thread in the traditional solution can handle only one occurrence (the availability of the forks). In contrast, the state machine of a `Philosopher` active object is more flexible because it can handle any occurrences, even in the hungry state. In addition, event passing among active objects is a more powerful communication mechanism than signaling on a semaphore. Apart from conveying some interesting occurrence, an event can provide detailed information about the qualitative aspects of the occurrence (by means of event parameters).

The separation of concerns (blocking, mode of operation, and signaling) in active object–based designs leads to unprecedented flexibility, because now, any one of these three aspects can vary independently of the others. In the dining philosophers example, the active object–based solution easily extends to accommodate new features (e.g., a conversation among philosophers) because a `Philosopher` active object is as responsive in the hungry state as in any other state. The `Philosopher` state machine can easily accommodate additional modes of operation. Event-passing mechanisms can also easily accommodate new events, including those with complex parameters used to convey the rich semantic content of the conversation among philosophers.

Conclusion

QP enables statechart modeling directly in C or C++ through two further fundamental meta-patterns: the HSM and active-object based framework (QF). None of the elements of QP, taken separately, are new. Indeed, most of the fundamental ideas have been around for at least a decade. The contributions of QP are not in inventing new theories of design (although QP propagates a method of design that is not yet mainstream); rather, the most important contributions of QP are stunningly simple implementations of fundamental software concepts.

By providing concrete implementations of such concepts as statecharts and active object-based computing, QP lays the groundwork for a new programming paradigm, no less powerful than OOP. For more on QP, see my book, *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems* [1]. You may also wish to visit the official QP website at <www.quantum-leaps.com>, where you can find code downloads, various QF ports, application notes, design patterns, resources, and more. I welcome contact regarding QP through the email address at the QP website.❑

References

[1] Samek, Miro, *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, CMP Books, 2002, ISBN 1-57820-110-1.

[2] Hewitt, Carl, P. Bishop, and R. Steiger. "A universal, modular actor formalism for artificial intelligence", 3rd International Joint Conference on Artificial Intelligence, pp. 235–245, 1973.

[3] Selic, Bran, Garth Gullekson, and Paul. T. Ward, *Real-Time Object Oriented Modeling*, John Wiley & Sons, 1994, ISBN 0-471-59917-4

[4] Object Management Group, Inc., *OMG Unified Modeling Language Specification v1.4*, http://www.omg.org, September 2001.

[5] IAR Systems visualSTATE® http://www.iar.com/Products/VS/

[6] Douglass, Bruce Powel, *Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999, ISBN 0-201-49837-5

[7] Dijkstra, Edsger W., "Hierarchical Ordering of Sequential Processes", *Acta Informatica 1*, 1971, pp. 115-138.

[8] Leveson, Nancy, *Safeware: System Safety and Computers*, Addison-Wesley, 1995, ISBN: 0-201-11972-2

[9] Jack G. Ganssle. "The Challenges of Real-Time Programming," *Embedded Systems Programming*, July 1998, pp. 20-26.

Miro Samek is the author of *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, CMP Books, 2002. He is the lead software architect at IntegriNautics Corporation (Menlo Park, CA) and a consultant to industry. Miro previously worked at GE Medical Systems, where he has developed safety-critical, real-time software for diagnostic imaging X-ray machines. He earned his Ph. D. in nuclear physics at GSI (Darmstadt, Germany) where he conducted heavy-ion experiments. Miro welcomes feedback and can be reached at `miro@quantum-leaps.com`.