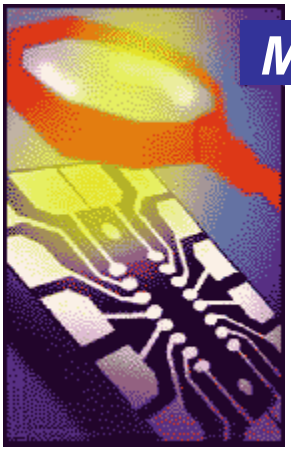

Miro Samek

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

Who Moved My State?

A better approach to event-driven programming using flexible state machines

If you looked closely at just about any computer system around you, you'd probably find out that at any given time it is doing... nothing useful. That is, most of the time the CPU is either hibernating in one of its power-saving modes or busily "twitching its thumbs" by executing an idle loop. (Modern PCs are particularly good at the latter form of doing nothing — they can "twitch their thumbs" a billion times per second all while driving a screen saver.) This wasn't necessarily so throughout most of the history of computing. For decades, CPU time was too expensive to be unused. Consequently, the operating systems and applications were carefully designed to keep the CPU constantly busy with data-processing tasks often organized into batch jobs. Only relatively recently, the nearly zero cost of processor time has radically changed the nature of computing. These days, millions of PCs and billions of embedded devices aren't processing batch jobs. Rather, almost all computers today are event-driven systems, which means that they continuously wait for the occurrence of some external or internal event, such as a mouse click, a button press, a time tick, or an arrival of a data packet. After recognizing the event, they react by performing the appropriate computation that may include manipulating the hardware or generating "soft" events that trigger other internal software components. (That's why event-driven systems are alternatively called reactive systems.) Once the event handling is complete, the software goes back to a dormant state (an idle loop or a power-saving mode) in anticipation of the next event.

Programmatically, this scheme implies that upon arrival of an event the CPU executes only a tiny fraction of the overall code. The main challenge is to quickly pick and execute the right code fragment. Indeed, you can trace the whole progress in understanding reactive systems just by studying the methods used to pick the correct code to execute. The problem is not at all trivial and leads to control inversion compared to the traditional data-processing systems. A conventional program (such as a compiler) starts off with some initial data (e.g., a source file), which it transforms by a series of computations into the desired output data (an object file), maintaining full control of the processing sequence from the beginning to the end. In contrast, an event-driven program gains control only sporadically when handling events. The actions actually executed are determined by events, which arrive in largely unpredictable order and timing, so the path through the code is likely to differ every time such software is run. Moreover, a reactive system doesn't typically have a notion of progression from beginning to end — the job of such a system is never done (unless explicitly terminated by an event). This paradigm still baffles many programmers, who often find it strange, backwards, mind-boggling, or weird.

Obviously, mastering the event-driven computing model is particularly important for embedded developers, given that embedded systems are predominantly reactive by nature. However, reactive systems are of fundamental importance across the whole software industry, because reacting to events is what most computers do most of the time. For example, consider the ubiquitous GUI built into most PCs, Macs, and Unix workstations. For the past two decades, the GUI has been the focus of intense study both in the industry and in academia with thousands of books and man-years devoted to the problem. You might think, therefore, that by now the event-driven paradigm underlying all GUI architectures is well understood and entirely under control. Admittedly, GUI development made tremendous progress from the arcane, black magic approach (recall early-days Windows programming), through application frameworks, such as Microsoft's MFC, Borland's OWL, or Java AWT, to RAD (rapid application development) tools, such as Microsoft Visual Basic or

Boland Delphi — especially the component-based approach originated by Visual Basic enabled GUI programming "for the masses." However, in spite of all these powerful frameworks, tools, and components, astonishingly many GUIs are still lousy. I don't mean here just poorly designed human-machine interactions — I mean applications that crash, lock up, or mislead the user and are a nightmare for the maintainer programmer. So, after all, perhaps understanding of the event-driven paradigm isn't that widespread among programmers.

Oversimplification of the Event-Action Paradigm

All modern GUI development tools and architectures seem to coalesce around the event-action paradigm. The example published in *Constructing the User Interface with Statecharts* by Ian Horrocks (Addison Wesley, 1999) illustrates how this paradigm works in practice. The author discusses a simple desktop calculator distributed as a sample application with Microsoft Visual Basic (versions 3.0 through 5.0, I guess), in which he found a number of serious problems. As Horrocks notes, the point of this analysis is not to criticize this particular application, but to point out the shortcomings of the general principle used in its construction.

For readers unfamiliar with Visual Basic, I quickly summarize how the calculator application (see Figure 1a) is built. The basic steps in creating a GUI application with any such tool (not just with Visual Basic) are pretty much the same. First, you drag and drop various components from the tool palette to the screen designer window. In the case of the calculator, you drag several buttons and a label for the display. After rearranging the components with the mouse, you set their properties, such as the button captions, or the label color and font. At this point — just after a few minutes — you have a complete visual part of the interface. This is fun! You can even launch the application and play with it by pressing the buttons, moving it around the screen, or minimizing the calculator window. Of course, at this stage, the application won't calculate anything. What's missing is the calculator-specific behavior.

Adding the behavior requires some coding, but also seems straightforward (initially, that is). Alongside properties, every component provides events, which are hooks for attaching user code to be executed upon occurrence of the event. For example, the button component provides the Click event, to which you can attach a snippet of code that will be invoked every time the user clicks the particular button. In Visual Basic, you use the IDE to map the event to a Visual Basic procedure. In MFC, you use a message map macro to map an event to a C++ class method. In Java 1.1, you register an event listener with the compatible event source. Whichever way the mapping actually occurs, all modern GUIs impose partitioning of the code by events to which the code responds.

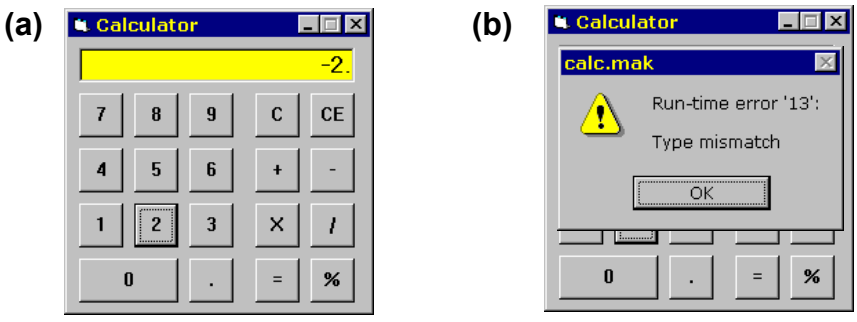

Figure 1 Visual Basic calculator GUI (a), and the GUI after a crash with a run-time error (b)

At first glance, this approach seems to work just fine. Indeed, when you launch the calculator (available for download at <www.cuj.com/code>), you will certainly find out that most of the time it correctly adds, subtracts, multiplies, and divides. What's there not to like? However, play with the application for a while longer, and you'll discover many corner cases in which the calculator provides misleading results, freezes, or crashes altogether. For example, the calculator often has problems with the "-" event — just try the following se-

quence of operations: 2, -, -, -, 2, =. The application crashes with a run-time error (see Figure 1b). This is because the same button "-" is used to negate a number and to enter the subtraction operator. The correct interpretation of the "-" button-click event, therefore, depends on the context, or mode, in which it occurs. Likewise, the CE (Cancel Entry) button occasionally works erroneously — try 2, x, CE, 2, =, and observe that CE had no effect, even though it appears to cancel the "2" entry from the display. Again, CE should behave differently when canceling an operand than canceling an operator. As it turns out, the application handles it the same way, regardless of the context. At this point, you probably have noticed an emerging pattern. The application is especially vulnerable to events that require different handling depending on the context.

EXERCISE: Ian Horrocks found 10 serious errors in this application just after an hour of testing. Try to find at least half of them.

This is not to say that the calculator does not attempt to handle the context. To the contrary, if you look at the code (available at <www.cuj.com/code>), you'll notice that managing the context is in fact the main concern of this application. The code is littered with a multitude of global variables and flags that serve only one purpose — handling the context. For example, `DecimalFlag` indicates that a decimal point has been entered, `OpFlag` represents a pending operation, `LastInput` indicates the type of the last button press event, `NumOps` denotes the number of operands, and so on. With this representation, the context of the computation is represented ambiguously, so it is difficult to tell precisely in which mode the application is at any given time. Actually, the application has no notion of any single mode of operation, but rather tightly coupled and overlapping conditions of operation determined by values of the global variables and flags. For example, the following conditional logic attempts to determine whether the "-" button-click event should be treated as negation or subtraction:

```
. . .
Select Case NumOps
    Case 0
    If Operator(Index).Caption = "-" And LastInput <> "NEG" Then
        ReadOut = "-" & ReadOut
        LastInput = "NEG"
    End If
    Case 1
    Op1 = ReadOut
    If Operator(Index).Caption = "-" And LastInput <> "NUMS" And
                                         OpFlag <> "=" Then
        ReadOut = "-"
        LastInput = "NEG"
    End If
. . .
```

EXERCISE: using the variables listed above write down the logical expression, which would determine whether the CE event should erase the operand from the display.

Such an approach is fertile ground for bugs for at least three reasons:

1. It always leads to convoluted conditional logic.
2. Each branching point requires evaluation of a complex expression.
3. Switching between different modes requires modifying many variables, which all can easily lead to inconsistencies.

Expressions like the one above, scattered throughout the code, are unnecessarily complex and expensive to evaluate at run time. They are also notoriously difficult to get right, even by experienced programmers, as the bugs still lurking in the Visual Basic calculator attest. This approach is insidious because it appears to work fine initially, but doesn't scale up as the problem grows in complexity. Apparently, the calculator application (overall only seven event handlers and some 140 lines of Visual Basic code including comments) is just complex enough to be difficult to get right with this approach.

The faults just outlined are rooted in the oversimplification of the event-action paradigm used to construct this application. The calculator example makes it clear, I hope, that an event alone does not determine the actions to be executed in response to that event. The current context is at least equally important. The prevalent event-action paradigm, however, neglects the latter dependency and leaves the handling of the context to largely ad-hoc techniques. Sample applications such as the Visual Basic calculator have taught legions of Visual Basic programmers to "handle" the context with gazillions of variables and flags, with the obvious results. (For that matter, the same can be said about MFC, Delphi, or Java/AWT/Swing programming.)

A Better Approach

For a better approach, let's turn back to the basics. The main challenge in programming reactive systems is to identify the appropriate actions to execute in reaction to a given event. The actions are determined by two factors: by the nature of the event and by the current context (i.e., by the sequence of past events in which the system was involved). The traditional techniques (such as the event-action paradigm) neglect the context and result in code riddled with a disproportionate number of convoluted conditional logic (in C/C++, coded as deeply nested if-else or switch-case statements). If you could eliminate even a fraction of these conditional branches, the code would be much easier to understand, test, and maintain, and the sheer number of convoluted execution paths through the code would drop radically, perhaps by orders of magnitude. Techniques based on state machines are capable of achieving exactly this — a dramatic reduction of the different paths through the code and simplification of the conditions tested at each branching point.

A state machine makes the event handling explicitly dependent on both the nature of the event and on the context (state) of the system. A state captures the relevant aspects of the system's history very efficiently. For example, when you strike a key on a keyboard, the character code generated will be either an uppercase or a lowercase character, depending on whether the Shift is active. Therefore, the keyboard is in the "shifted" state, or the "default" state. The behavior of a keyboard depends only on certain aspects of its history, namely whether Shift has been depressed, but not, for example, on how many and which specific characters have been typed previously. A state can abstract away all possible (but irrelevant) event sequences and capture only the relevant ones.

The Standard FSM Implementations

I suppose that most readers are familiar with the classical FSMs (Finite State Machines) and with their intuitive graphical representation, such as the UML state diagram of the Keyboard FSM shown in Figure 2. However, when it comes to implementing state machines (in C or C++, say), the literature on the subject presents quite a cloudy picture. The main problem is that state machines cannot operate in a vacuum and require, at a minimum, an execution context (thread) and an event-dispatching mechanism. In any nontrivial use, you also need to provide event-queuing and timing services. All these elements strongly depend on the application domain and operating system support (or lack thereof). Consequently, most writings present state machines intimately intertwined with a specific concurrency model (e.g., polling loops or interrupt service routines) and geared toward a particular event dispatching policy (such as extracting events directly from hardware or through global variables). Obviously, this muddies the water and narrows the applicability of the published state machine code. Nevertheless, I believe that the most popular techniques can be distilled into the following three broad categories:

- A nested switch statement with a scalar "state variable" used as the discriminator in one level of the switch and the event-type in the other. The technique is simple, but scales poorly because the entire state machine is implemented as one monolithic switch statement.
- A state table containing an (typically sparse) array of transitions for each state. The table lists event types (triggers) along one dimension and the states along the other. Each cell in the table contains a pointer to function (action to execute for the specific event-state combination), as well as the succeeding state. In ef-

fect, the table approach converts the traditional conditional code into a table lookup and allows changing the transition criteria by modifying data instead of changing code.

- Various object-oriented designs representing a state machine as a dynamic tree-like structure of state and transition objects traversed at run time by a specialized "state machine interpreter." To a large degree, this is a generalized state table approach that attempts to represent a state machine in a structure more memory-efficient than an array (a network of objects in this case).

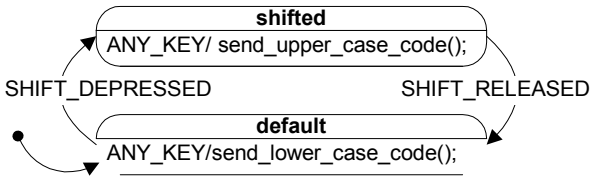

Figure 2 UML state diagram representing the keyboard state machine

All the popular state machine techniques are rather incompatible with the event-action paradigm for at least two reasons. First, the event-action paradigm maps an event to a specific event-handler signature (the action procedure receives the event parameters as specific, strongly typed function arguments), whereas all the state machine techniques require a uniform representation of all events. Secondly, the event-action paradigm already performs one level of event de-multiplexing based on the event type, whereas the state machine techniques perform de-multiplexing based on both the event type and the current state. For example, the event-action paradigm would slice a state table along the state dimension, which would defuse the notion of state.

The Technique of the Pros

Incidentally, the smallest, fastest, and arguably the most natural technique for implementing state machines in C/C++ isn't widely publicized (although I suspect it's in every pro's bag of tricks). The technique hinges on pointers-to-functions in C (and pointers-to-member-functions in C++). Pointers-to-functions are obviously heavily used in other standard techniques (e.g., state tables typically store such pointers). However, the technique I mention here represents the very concept of "state" directly as a pointer-to-function. Please take note, the "state" isn't enumerated; it's not an instance of a `State` class — it is a pointer to a state-handler function, as declared in line 4 of Listing 1.

```
 1: typedef short Signal;
 2: typedef struct Event Event;
 3: typedef struct Fsm Fsm;
 4: typedef void (*State)(Fsm *, Event const *);
 5:
 6: struct Event { /* Event base class */
 7:    Signal sig;
 8: };
 9:
10: struct Fsm { /* Finite State Machine base class */
11:    State state__; /* the current state */
12: };
13:
14: #define FsmCtor_(me_, init_) ((me_)->state__ = (State)(init_))
15: #define FsmInit(me_, e_)     (*(me_)->state__)((me_), (e_))
16: #define FsmDispatch(me_, e_) (*(me_)->state__)((me_), (e_))
17: #define FsmTran_(me_, targ_) ((me_)->state__ = (State)(targ_))
```

Listing 1 The FSM header file

The other FSM elements in this implementation include: the `Event` base class for derivation of events with parameters (Listing 1, lines 6-8) and the `Fsm` base class for derivation of state machine objects (lines 10-17). The `Fsm` class stores the current state in its attribute `state__` and provides four methods ("inlined" by means

of macros). The constructor (Listing 1, line 14) initializes the current state, the method `init()` (line 15) triggers the initial transition, the method `dispatch()` (line 16) dispatches events to the state machine, and finally the method `tran()` (line 17) takes a state transition. Listing 2 shows an example of using the technique to implement the `Keyboard` FSM from Figure 2. Lines 3-12 declare the `Keyboard` FSM as a subclass of `Fsm`. (Please note the member `super_` in line 5.) Similarly, the keyboard-specific event `KeyboardEvt` is derived from Event in Listing 2, lines 15-18. The event types must be enumerated (lines 20-24), but the states aren't: they are referenced by the names of state-handler methods. The `Keyboard` FSM has two such state-handlers (`Keyboard_default()` and `Keyboard_shifted()`, declared in lines 11 and 12 of Listing 2, respectively). The `Keyboard_initial()` method (declared in line 10 and defined in lines 30-34) implements the initial transition.

```
 1: #include "fsm.h"
 2:
 3: typedef struct Keyboard Keyboard;
 4: struct Keyboard {
 5:    Fsm super_; /* extend the Fsm class */
 6:    /* ... other attributes of Keyboard */
 7: };
 8:
 9: void KeyboardCtor(Keyboard *me);
10: void Keyboard_initial(Keyboard *me, Event const *e);
11: void Keyboard_default(Keyboard *me, Event const *e);
12: void Keyboard_shifted(Keyboard *me, Event const *e);
13:
14: typedef struct KeyboardEvt KeyboardEvt;
15: struct KeyboardEvt {
16:    Event super_; /* extend the Event class */
17:    char code;
18: };
19:
20: enum { /* signals used by the Keyboard FSM */
21:    SHIFT_DEPRESSED_SIG,
22:    SHIFT_RELEASED_SIG,
23:    ANY_KEY_SIG,
24: };
25:
26: void KeyboardCtor(Keyboard *me) {
27:    FsmCtor_(&me->super_, &Keyboard_initial);
28: }
29:
30: void Keyboard_initial(Keyboard *me, Event const *e) {
31:    /* ... initalization of Keyboard attributes */
32:    printf("Keyboard initialized");
33:    FsmTran_((Fsm *)me, &Keyboard_default);
34: }
35:
36: void Keyboard_default(Keyboard *me, Event const *e) {
37:    switch (e->sig) {
38:    case SHIFT_DEPRESSED_SIG:
39:       printf("default::SHIFT_DEPRESSED");
40:       FsmTran_((Fsm *)me, &Keyboard_shifted);
41:       break;
42:    case ANY_KEY_SIG:
43:       printf("key %c", tolower(((KeyboardEvt *)e)->code));
44:       break;
45:    }
46: }
47:
48: void Keyboard_shifted(Keyboard *me, Event const *e) {
49:    switch (e->sig) {
50:    case SHIFT_RELEASED_SIG:
51:       printf("shifted::SHIFT_RELEASED");
52:       FsmTran_((Fsm *)me, &Keyboard_default);
53:       break;
```

```
54:    case ANY_KEY_SIG:
55:       printf("key %c", toupper(((KeyboardEvt *)e)->code));
56:       break;
57:    }
58: }
```

Listing 2 The `Keyboard` FSM implementation (see state diagram from Figure 2)

Listing 3 shows the test harness for the state machine implemented here as a console application driven from a keyboard with keys "^" and "6" (which emulate pressing and releasing Shift, respectively). Pressing "." terminates the test. The essential elements here are: instantiating the state machine object (line 2), the explicit constructor call (hey, we are coding OOP in C), the state machine initialization in line 4, and dispatching events in line 16. Please note the necessity of explicit casting (upcasting) to the `Fsm` superclass (in lines 4 and 16), because the C compiler doesn't "know" that `Keyboard` is related to (derived from) `Fsm`. Please note how the state machine approach eliminates conditional statements from the code. Actually, in this particular technique even the explicit test of the current state variable (the `state__` member of the `Fsm` class) disappears as a conditional statement and is replaced with more efficient pointer-to-function dereferencing (Listing 3, line 16). This coding aspect is similar to the effect of polymorphism, which eliminates many tests based on the type of the object and replaces them with much more extensible (and efficient!) late binding.

```
 1: int main() {
 2:    Keyboard k;
 3:    KeyboardCtor(&k);
 4:    FsmInit((Fsm *)&k, 0);
 5:    for (;;) {
 6:       KeyboardEvt ke;
 7:       printf("\nSignal<-");
 8:       ke.code = getc(stdin);
 9:       getc(stdin);                        /* discard '\n' */
10:       switch (ke.code) {
11:       case '^': ke.super_.sig = SHIFT_DEPRESSED_SIG; break;
12:       case '6': ke.super_.sig = SHIFT_RELEASED_SIG; break;
13:       case '.': return 0;           /* terminate the test */
14:       default: ke.super_.sig = ANY_KEY_SIG; break;
15:       }
16:       FsmDispatch((Fsm *)&k, (Event *)&ke);  /* dispatch */
17:    }
18:    return 0;
19: }
```

Listing 3 Test harness for the `Keyboard` FSM

EXERCISE: step with a debugger through Listing 3 and look at the machine-language output generated by the `FsmDispatch()` method in line 16. (It's typically only one instruction on a CISC processor!) Compare it to code generated by a virtual method invocation in C++. Aren't they... identical?

OOP in C

Many programmers mistakenly think that Object Oriented Programming (OOP) is possible only with OO languages, such as C++ or Java. However, OOP isn't the use of a particular language or a tool; it is a way of design that can be implemented in almost any language, such as C. Knowing how to code classes, inheritance, and polymorphism in C can be very useful for embedded programmers, who often maintain C code or deal with embedded microprocessors released to the field with nothing but a bare-bones C compiler.

As a C programmer, you already may have used ADTs (abstract data types). For example, in the Standard C runtime library, the family of functions that includes `fopen()`, `fclose()`, `fread()`, and `fwrite()` operates on objects of type `FILE`. The `FILE` ADT is encapsulated so that the clients

have no need to access the internal attributes of `FILE`. (When was the last time you looked up what's inside the `FILE` structure?) You can think of the `FILE` structure and the associated methods that operate on it as the `FILE` class. The following bullet items summarize how the C runtime library implements the `FILE` "class":

- Attributes of the class are defined with a C `struct` (the `FILE struct`).
- Methods of the class are defined as C functions. Each function takes a pointer to the attribute structure (`FILE *`) as an argument. Class methods typically follow a common naming convention (e.g., all `FILE` class methods start with `f`).
- Special methods initialize and clean up the attribute structure (`fopen()` and `fclose()`, respectively). These methods play the roles of class constructor and destructor.

For example, Listing 2 declares class `Keyboard`. Each method of the class takes the pointer to the attribute structure (`Keyboard *`) as the first argument (the `me` pointer) and starts with the common class prefix (`Keyboard`). The me pointer in C corresponds directly to the `this` pointer in C++. (If you are unfamiliar with the construct `typedef struct Fsm Fsm`, please refer to Dan Saks' article "Tag vs. Type Names," *Embedded Systems Programming*, October 2002.) A simple way of implementing single inheritance in C is by literally embedding the superclass attribute structure as the first member of the subclass structure. (Please convince yourself that the resulting memory alignment lets you always treat any pointer to the subclass as a pointer to the superclass.) In particular, you can always pass this pointer to any C function that expects a pointer to the superclass. (To be strictly correct in C, you should explicitly upcast this pointer.) Therefore, all methods designed for the superclass are automatically available to subclasses; that is, they are inherited.

For example, Listing 2 shows how to derive class `Keyboard` from the base class `Fsm` (lines 3-7). The superclass instance `super_` (Listing 2, line 5) is embedded as the first attribute of the subclass `Keyboard`.

For more information on implementing OOP in C (including polymorphism, which I don't use in the FSM example), see the first chapter of *C+ C++: Programming With Objects in C and C++* by Allen Holub (McGraw-Hill, 1992); *Reusable Software Components, Object-Oriented Embedded Systems Programming in C* by Ted Van Sickle (Prentice Hall, 1997); CUJ articles from July 1990 and July 1993; or my article "Portable Inheritance and Polymorphism in C," *ESP*, December 1997. Needless to say, googling around for "OOP in C" or "Inheritance in C" will reveal countless other resources. ❑

End-around Check

In the context of the ubiquitous event-action paradigm, the State design pattern, as described in Design Patterns by Erich Gamma and colleagues (Addison-Wesley, 1995), takes on special importance. This is the only state machine implementation to my knowledge that is directly compatible with the event-action paradigm. Through heavy use of polymorphism and delegation, the pattern emulates change of an object's class at run time (change of state). The technique, however seems to be hard to extend to hierarchical state machines. I haven't listed this pattern among the most popular techniques, because I don't see it widely used (e.g., out of all ESP state machine articles, not a single one uses the State pattern).

It has been pointed out to me that the technique of changing an object's class at run time without using the heap I described in my last column is not only less elegant, but actually sub-optimal compared to the alternative design, which I included only in the accompanying code. (I referred to this alternative method as Jeff Claar's design.) I agree. After investigating the machine-code output from various C++ compilers, I found out that Jeff's design indeed doesn't incur any overheads either in virtual method invocation (assuming that the `operator->` is inlined), nor in memory usage. In contrast, my technique always leads to generation of an unused virtual table for the `GenericInstrument` class.

Recently, a copy of Spencer Johnson's *Who Moved My Cheese?* (G. P. Putnam's Sons, 1998) fell into my hands. The story takes place in a maze where four amusing characters look for "Cheese" — cheese being a metaphor for what we want in life. We programmers spend our working hours in the most complicated maze of all (a.k.a. source code) striving for our favorite kind of "Cheese" — robust, adaptable, and maintainable software. We use different strategies to find our "Cheese": simple methods like trial-and-error, and sophisticated methods like object-oriented programming, visual tools, and design patterns. Interestingly, techniques based on hierarchical state machines aren't among the popular, mainstream programming strategies. Yet, state machines are perhaps the most effective method for developing robust event-driven code because they give us the map to the software maze. ❑

Miro Samek is the author of *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, CMP Books, 2002. Dr. Samek is the lead software architect at IntegriNautics Corporation (Menlo Park, CA) and a consultant to industry. He 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`.