

UML Statecharts

by Bruce Powel Douglass

The term *reactive* is applied to objects that respond dynamically to incoming events of interest and whose behavior is driven by the order of arrival of those events. Such objects are usually modeled and often implemented as finite state machines. A feature of the graphical Unified Modeling Language (UML), called statecharts, makes modeling reactive objects a breeze.

A finite state machine (FSM) specifies the events of interest to a reactive object, the set of states that object may assume, and the actions (and their order of execution) in response to incoming events in any given state. This is crucial in many systems because the allowable sequences of primitive behaviors may be restricted.

State awareness

Consider Figure 1, which shows a statechart of a non-invasive blood pressure (NIBP) monitor. This statechart specifies the various states (shown as rounded rectangles) and the transitions (arrows) that indicate how the system responds to different events. We see that the system responds to a number of events, such as `evMessage`, `evDiastolicFound`, and `evSystolicFound`. We see that to monitor blood pressure, these events must occur in certain predefined sequences. If an event occurs while the system is in a state that doesn't specifically handle that event, it is ignored. Thus, when the system is in the state `waitForSystolic`, any received `evDiastolicFound` events are quietly discarded. The syntax for

transitions is *event-name (parameter-list) [guard-condition] / action-expression*.

The *event-name* is the name of an occurrence of interest that's processed by the object's state machine, such as `evMessage` or `tm` (timeout). If the event-name is left blank, the event fires as soon as the state is reached. When the object is in the predecessor state (from which the transition arises) and the named event is sent to the object, the transition fires and the object moves to the new state executing the specified actions along the way. Parameters may be passed to the object along with the event and manipulated in the actions on the transition.

A Boolean expression inside the square brackets is a *guard*. If the event fires and the guard evaluates to `TRUE`, the transition is taken; if the guard evaluates to `FALSE`, the event is ignored. For example, coming from the `waitForCuffInflation` state, if the `tm` event occurs, the transition is only taken if the guard (`itsSignalProcessor->getValue() < MaxCuffInflationLimit`) is `TRUE`. If not, the timeout event is discarded.

Following the guard, a `/` precedes the *action expression*, which is the set of actions executed when that transition is taken. For example, when the monitor is in the waiting state and a timeout event is received, messages are sent to the associated `Alarm` object (by dereferencing a pointer to that object) and the `Pump` object. Actions may also be executed when a state is entered (called *entry actions*) or exited (*exit actions*).

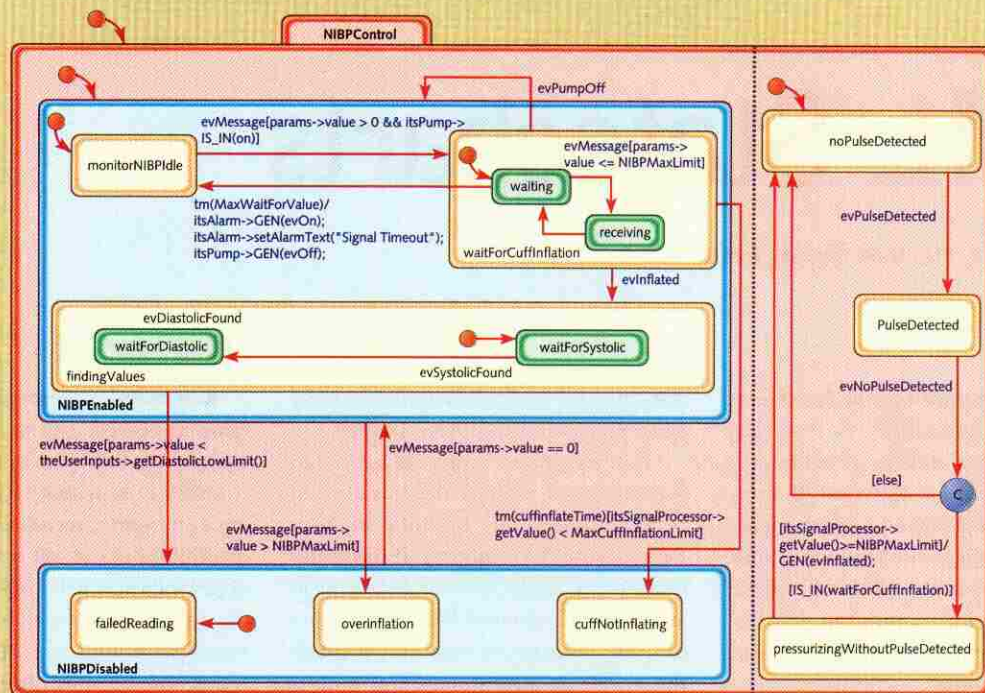
Responses to events may have a time value associated with them (via a UML constraint), but they execute via run-to-completion semantics. This means that once an event is received and the object determines that it will respond to it, the object will execute first the exit actions of the predecessor state, followed by the transition actions, and finally the entry actions of the subsequent state, in that precise order, and not process any new events until that set of actions has completed. Actions are normally specified in the implementation language.

The UML identifies four different kinds of events: `SignalEvents`, `CallEvents`, `ChangeEvents`, and `TimeEvents`. A *SignalEvent* is an asynchronous event sent from a source and queued by the receiver until it's ready to handle it. A *ChangeEvent* is a synchronous event wherein the sender blocks until the receiver has handled the event. *ChangeEvents* occur when a state value is changed for an object. A *TimeEvent* occurs when time passes from entry to the state; if the object leaves the state prior to the timeout, the timer is logically discarded. This is shown in the figure with the `tm(MaxWaitForValue)` transition. The keyword `after` is sometimes used instead of `tm`.

State hierarchy

The states `NIBPEnabled` and `NIBPDisabled` in Figure 1 are called *or-states* because the object is in either one or the other. Also note that `NIBPEnabled` is a *composite state* with *nested states* inside; these nested states are *or-states* with respect to each

FIGURE 1 Statechart example



other. The ability to nest states greatly enhances the applicability of statecharts to more complex problems.

Statecharts may also have *and-states*, which are always nested states of an enclosing composite state. The primary difference between *or-states* and *and-states* is that when the object is in the enclosing composite state, it must be in every active *and-state* simultaneously (in other words, logically concurrent).

The figure only shows a single state at the highest level of abstraction (NIBPControl). This top-level state has two *and-states* (separated by the dotted line). These are independent aspects of the class, the one on the left focusing on the high-level measurement management, and the one on the right detecting blood pressure pulses.

There are various graphical annotations that are neither states nor transi-

tions to be found in the figure. These annotations are called *pseudostates*. These symbols represent special semantics to be applied in the given situation. The transition emanating from a black dot is the initial pseudostate. It indicates the default *or-state* at some specific abstraction level. At the outermost level, it indicates the initial state when the object is created. The circumscribed C symbol (alternative notation: diamond) is a conditional pseudostate and indicates a branch point; each outgoing transition from the conditional pseudostate has a guard; when the event occurs, the various outgoing transitions are examined until a TRUE guard is found. If none of the guards evaluates to TRUE, then the event is discarded, just as with the simple transition; at most a single branch of the conditional pseudostate will be taken.

The UML recognizes about a dozen pseudostates.

Statecharts are a great way to specify behavior as a set of actions to be executed when an object is in some given state. The statechart remembers the condition of existence for the object as the object's state. Through the use of separation of the object's lifecycle into different states, hierarchical nesting of states, and the *or-* and *and-states*, you can model any mundane or highly complex behavior. Continuous algorithms can be modeled by statecharts as well.

esp

Bruce Powel Douglass has over 25 years experience designing safety-critical real-time applications in a variety of hard real-time environments and is the author of several books including Real-Time UML, Doing Hard Time, and Real-Time Design Patterns. He can be reached at bpd@ilogix.com.

EMBEDDED SYSTEMS PROGRAMMING (ISSN 1040-3272) is published monthly by CMP Media LLC., 600 Harrison Street, San Francisco, CA 94107, (415) 905-2200. Please direct advertising and editorial inquiries to this address. SUBSCRIPTION RATE for the United States is \$55 for 12 issues. Canadian/Mexican orders must be accompanied by payment in U.S. funds with additional postage of \$6 per year. All other foreign subscriptions must be prepaid in U.S. funds with additional postage of \$15 per year for surface mail and \$40 per year for airmail. POSTMASTER: All subscription orders, inquiries, and address changes should be sent to EMBEDDED SYSTEMS PROGRAMMING, P.O. Box 3404, Northbrook, IL 60065-9468. For customer service, telephone toll-free (877) 676-9745. Please allow four to six weeks for change of address to take effect. Periodicals postage is paid at San Francisco, CA and additional mailing offices. EMBEDDED SYSTEMS PROGRAMMING is a registered trademark owned by the parent company, CMP Media LLC. All material published in EMBEDDED SYSTEMS PROGRAMMING is copyright © 2003 by CMP Media LLC. All rights reserved. Reproduction of material appearing in EMBEDDED SYSTEMS PROGRAMMING is forbidden without permission. EMBEDDED SYSTEMS PROGRAMMING is available on microfilm/fiche from University Microfilms International, 300 N. Zeeb Rd., Ann Arbor, MI 48106, (313) 761-4700.

