Extending the Statechart Formalism: Event Scheduling & Disposition

Arthur Allen, MetaSphere

Dennis de Champeaux, OntoOO

Arthur Allen
MetaSphere Inc.
199 First Street, Suite 340
Los Altos CA 94022
tel: 415-948-8755
FAX: 415-948-7632
email: ada@metasphere.com

Dennis de Champeaux
OntoOO
14519 Bercaw Lane
San Jose CA 95124
tel: 408-559-7264
FAX: 408-371-2713
email: ddc@netcom.com

Abstract

Statecharts are extended to deal with events when no applicable transition is available, and to resolve conflicts relative to event scheduling and response that can arise whenever multiple states can be active simultaneously. "Event closure" and event scheduling are achieved without having to clutter up a basic statechart. The extensions are effected by means of declarative event disposition rules. These rules, together with the statechart topology, determine the contents of one or more disposition matrices. These matrices are combined with the statechart state to determine the response of the event dispatcher to incoming events. The operation of the event dispatcher is also described. A detailed example illustrates these concepts, which are further characterized, for the benefit of working programmers, in the form of a behavioral design pattern. A tool called StateCraft embodies these notions.

OOPSLA '95 Austin, TX, USA

Keywords

Statechart, event closure, declarative disposition rule, disposition matrix, event scheduling, generic response, behavioral design pattern.

1 Introduction

Having to specify only *what* a software system is supposed to do and avoiding the details of *how* a system works has been for many years, and still is, an elusive goal. OO has made progress towards this goal, which is one of its surprises. These days there are already two commercial products available -- from SES and Object Time -- that can translate high level designs directly into executable programs with respectable performance [Selic94+]. It is true that one still has to provide procedural detail for all state transitions within a state transition diagram, but these tasks are typically small in comparison with having to do a full scale design and a subsequent implementation. In addition, maintenance of software generated by these products is greatly simplified since the execution model is very close to the analysis model. In particular, the notion of a single thread of control is preserved.

This paper underwrites the same development philosophy: pushing declarative behavior specifications as far as possible.

The current trend in object oriented analysis and design is, whenever possible, to model object state and state transitions by means of an embedded Statechart. The powerful Statechart formalism, first described by [Harel87], and adopted with some reservations by [Booch93], [Rumbaugh91+], [Firesmith95], and [Selic92+], is visual in nature, and permits modelling of object state and behavior by way of a collection of parallel and nested automata, encompassed within a single diagrammatic formulation.

Software objects are typically viewed as being driven by external events that can manifest in a variety of ways, such as the invocation of a method, arrival of a message, and so forth. Once recognized, an event is typically forwarded to the statechart for processing. Statecharts are very effective for shaping object behavior in response to an event that is *expected*, that is an event that enables a transition rooted in an active state. Unfortunately, this formalism provides little support in the case when an event is *not expected*, or only *partially expected*[1]. Object behavior may depend as much on the one as the other.

The current accepted solution to this problem is to start with a *base statechart* that responds in the prescribed manner to *expected events* occurring in an idealized sequence; to this chart are added sufficient transitions aimed at managing event stimuli such that every event that can occur is always expected (this can be called *event closure)* even when meaningful action is deferred.

For example, an event that is subject to deferral when first announced might be jammed back into a queue by one such transition until explicitly resubmitted upon the occurrence of yet another event. This technique is often supplemented by direct procedural inhibition and subsequent enabling of event stimuli by transitions within the base or augmented statechart.

We believe that these approaches suffer from a number of significant drawbacks:

ad-hoc — The event scheduling policy is ad-hoc and has no formal basis. As a result it becomes increasingly difficult to insure coverage of all possible unexpected event occurrences as the number of events, states and active state combinations grows large.

wasted expressive power — They often lead to a misuse of the formidable expressive power of a statechart, when all that may be required is one of several *generic responses* that could well be state and event independent.

induced procedural cross couplings — A complex statechart will typically have multiple states active simultaneously, one per active hierarchical or parallel state machine. Conflicts will inevitably arise in such settings when two or more active states wish to respond in different and mutually exclusive fashion to a given event (e.g. defer versus accept, defer versus discard). Such conflicts must be recognized and dealt with on a case by case basis within designated transition action procedures. This practice is difficult to carry out in practice, and unavoidably leads to undesirable procedural cross-couplings[2]. These cross-couplings between the states of a complex statechart compromise modularity, and represents a

[1] An unexpected event is one that does not enable any transition currently eligible to fire. A partially expected event is one that drives two or more state machines, at least one of which expects the event while others do not, and where there is a need for synchrony (see the example in sections 2 and 5).

[2] It is perhaps in order to circumvent this problem that a number of methods (e.g. [Selic92+], [Rumbaugh91+])

serious obstacle to the use of multiple inheritance as a vehicle for constructing composite statecharts.

loss of clarity

The original base statechart becomes beclouded by a profusion of inter-related transitions that induce additional hidden state not apparent from an examination of the statechart diagram, rendering resulting software difficult to understand, maintain, and its behavior difficult to predict even when code generators are employed.

In this paper we describe a method, called the method of dispositions[3], that proposes a clean separation of responsibilities between a base statechart, lying at the heart of an object, and an enclosing event management layer. Under this scheme, the base statechart is responsible for shaping behavior in response to *ideally sequenced or anticipated external stimuli.* Of course, stimuli of this kind will not always occur in practice. Real execution environments are subject to faults (e.g. erroneous input from an operator, message corruption, loss or duplication over a communication network) or to unpredictable, asynchronous interaction sequences involving two or more communicating objects. The burden of compensating for the "naivete" of the statechart devolves upon an active event management layer[4]. This layer assumes responsibility for scheduling and serializing event stimuli in time, dispatching, deferring, discarding, re-submitting, or otherwise disposing of events as needed. As will be shown, while carrying out these tasks this layer is deeply informed about the entire hierarchical state of the statechart, in particular with its receptivity and disposition to all possible events.

The structure of this paper is as follows. In section 2 we sketch a simple example, which illustrates our contention that "normal" statecharts are subject to the shortcomings discussed above. Sections 3 and 4, the core of the paper, describe our framework and the method of dispositions. Section 5 is devoted to an illustration of the method of dispositions as applied to the example formulated in Section 2. Section 6 describes related efforts, and section 8 has the summary and ties up some loose ends. A behavioral design pattern for the method of dispositions, introduced in section 7, can be found in the Appendix.

2 An Example

2.1 Description

Suppose that we wish to provide a communication protocol object that implements a simple reliable message delivery service using the underlying services of a communication fabric that delivers messages on a best-efforts basis. Given the possibility for message corruption or loss, it is desired that this new service provide an acknowledgement of message receipt, and that re-transmissions be initiated automatically up to a given number of times should an acknowledgement (from the peer communication object at the destination) not be forthcoming within a specified time interval. Conversely, incoming messages must each give rise to the transmission of an acknowledgement message back to the initiating communication object.

An object requiring reliable message delivery begins by instantiating a protocol object. During initialization, the communication object obtains use of a lower level socket object which provides bi-directional access to the communication fabric, as shown in figure 1. Communication cannot begin until the socket is bound to a network address specified to the communication object by the client, and lasts until the socket is unbound.

---

disallow the use of parallel state machines: only one leaf state can be active at a time, and an event can trigger one and only one transition.

[3]A patent application is pending in the US patent office.

[4]The design pattern introduced in a subsequent section maps the functions performed by this layer onto a separate Disposition Object.

2.2 Auxiliary Components

In implementing our example protocol, shown in context in figure 2, use is made of three ancillary constructs[5]:

Banded Event Queue A queues comprises one or more bands. Bands are used to prioritize a queue. Within a band, events are queued in FIFO order. In our example, three separate event queues are used: the Request queue receives events from the client object; the Administrative queue accommodates a variety of events required to implement the protocol, and the Indication queue receives events from the communication fabric or the socket.

Delayed Buffer Allocator: Provides delivery of a message buffer of the required size, which is announced by way of an associated event, which may be delayed relative to the request under conditions of buffer shortage. Three of these appear in the example.

Interval Timer Announces an associated event when a specified time interval has elapsed. Can be started (*start_timer()*) and stopped (*stop_timer()*) as required.

2.3 Protocol Statechart

Figure 3 depicts the statechart containing four state machines that implement our simple protocol. All finite state machines (FSM) states and transitions are numbered for ease of reference.

State machine PEER PROTOCOL FSM(2) addresses life-cycle management. Nested within one of its states, DATAXFER, are two additional state machines: message transmission and retries are managed by TRANSMIT FSM(3), whereas message reception and acknowledgement are implemented within RECEIVE FSM(4).

The I/F FSM(1) relates to the service request interface facing upwards to the client object. Each service request manifests at the REQUEST queue by an associated event (see transition 2), which gives rise to a response or acknowledgement event back to the client after the REQUEST_DONE event is broadcast internally (transition 4). The recognized request events are BIND_REQUEST, UNBIND_REQUEST, XMIT_DATA_REQUEST and NAK_REQUEST. NAK_REQUEST (transition 1) is a catch-all event that designates any unrecognizable or erroneous request event. Event BIND_REQUEST is passed down to the socket object to cause it to be bound with a designated address (transition 14); event UNBIND_REQUEST, which is also passed down (transition 16), causes the address binding to be severed. The socket object responds with a BIND_ACK (transition 12), BIND_NAK (transition 15), UNBIND_ACK (transition 13) as appropriate, delivered to the INDICATION queue. XMIT_DATA_REQUEST requests that a given data message be delivered to a specified destination, with acknowledgement of receipt or failure. Prior to transmission (transition 6), this message must be outfitted with a header (transition 5), within which is placed a unique token value that is returned within the associated acknowledgement message, XMIT_DATA_ACK. Re-transmissions are triggered (in transition 7) should an acknowledgement not be forthcoming within the time-out interval. An outgoing data message should, unless it is lost by the network, manifest to the peer protocol at the destination as a RECV_DATA_IND event (transition 10). Within this transition the message header is stripped and the resulting data message is passed up to the client. An acknowledgement buffer is requested also, for use to stage and to transmit a XMIT_DATA_ACK back to the initiator (transition 11).

[5]These can be viewed as typed instance variables of the protocol object.

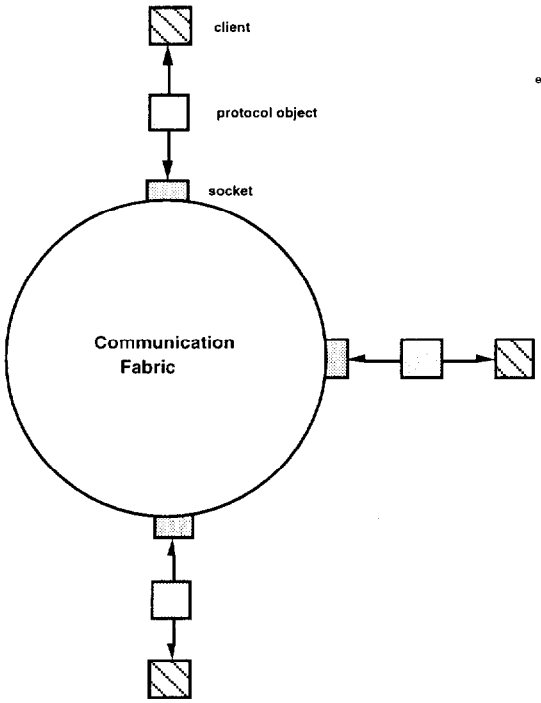


Figure 1: Reliable communication is achieved by interposing a protocol object between the client and the network socket

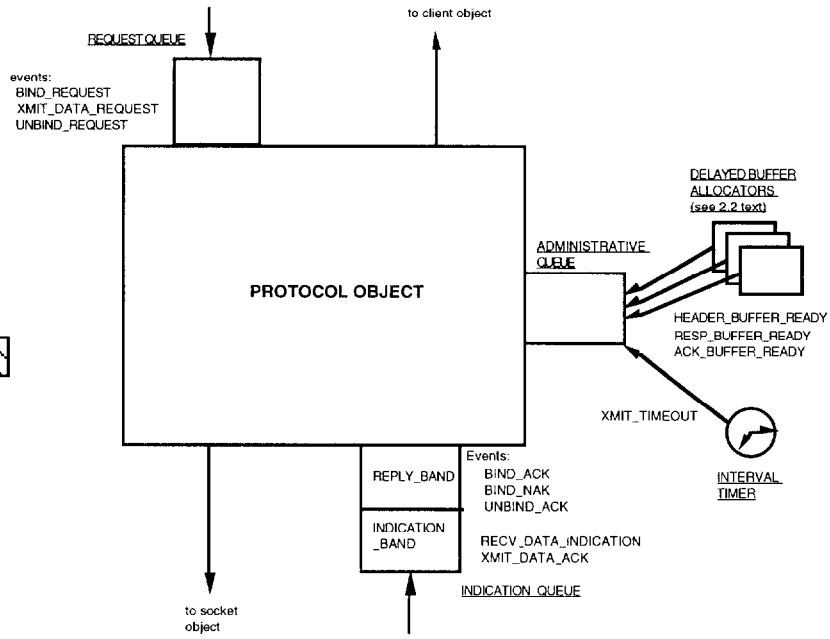


Figure 2: external view of the protocol object showing banded queues, events and ancillary components

The service is asynchronous, i.e., multiple transmission requests may be pending while the peer protocol processes one at a time. Each request is separately acknowledged (transitions 8 or 9 followed by 4).

2.4 Event Closure through Statechart Augmentation

As it stands the example statechart is incomplete: it will function correctly only if events impinge on it in an idealized sequence, which cannot always be expected to occur in practice under protracted use, subject to varying load conditions. Consider the following unresolved issues:

1) Incoming indications and acknowledgements are not discarded as they should when the object is unbound.

2) Incoming acknowledgements are not discarded when there is a token mismatch.

3) Incoming acknowledgements are not discarded when no longer expected.

4) Incoming data messages are not held off as they should while a prior message is still being processed (i.e. when state machine 4 is in state 9 due to a buffer shortage).

5) New interface requests are not deferred as they should until the present one is fully processed. For example, suppose that state machine 3 returns to state 5, while state machine 1 is still in state 3 because of a buffer shortage preventing the allocation of a response message to the client. In that case a pending XMIT_DATA_REQUEST is *partially expected* and should be deferred.

6) Requests issued "out of state" (for example a XMIT_DATA_REQUEST while the object is in state UNBOUND) do not give rise to a NAK response.

The events that could occur are not all expected. Event closure can be achieved by adding transitions in the appropriate places that effect the necessary response. An alternative non-procedural approach to event closure will be demonstrated in section 5.

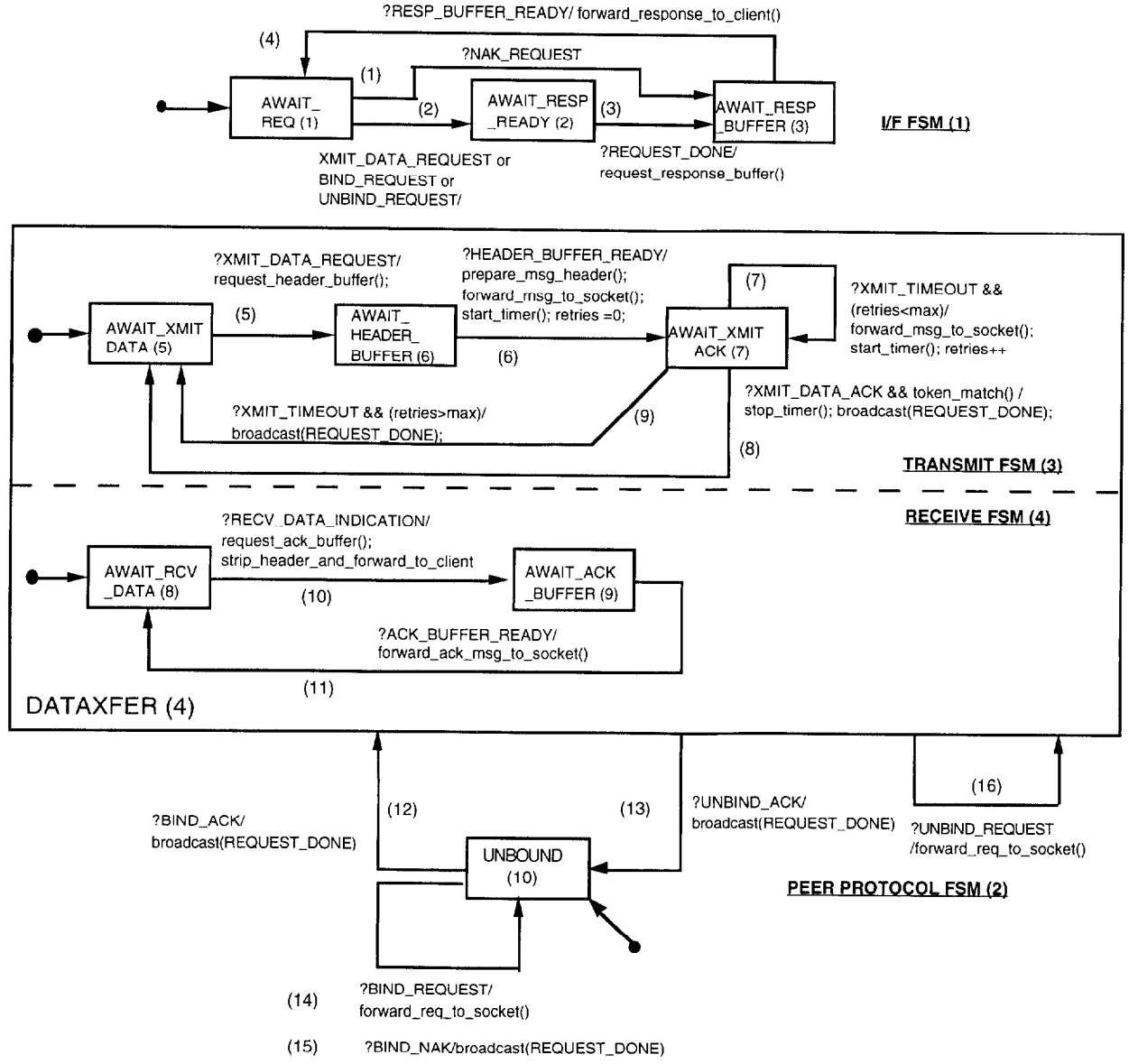


Figure 3: the statechart for the protocol object comprises 4 state machines and 10 states.

3 A Computational Framework

3.1 Overview

The collection of state machines comprising a statechart can be embedded within a larger computational framework, depicted in figure 4, suited to the implementation of event driven objects. The added elements comprising the event management layer are queues, bands and the event dispatcher.

The statechart formalism, as described in [Harel87] and [Harel94+], is accepted in its entirety, including concurrent states and internal event broadcasting[6]. The event management layer concerns itself with

[6]Internal event broadcasting is a powerful mechanism (albeit with complex semantics) that can be used as a source of internal stimuli.

external events, not internal *broadcast*[7] *events* which are assumed to be dispatched by mechanisms internal to the statechart. The event management layer considers events for disposition when the chain reaction of broadcast events resulting from the last external event has subsided, and the statechart has reached a quiescent state.

3.2 Events, Queues and Bands

A *queue* is a channel for *event objects* to the statechart. The set of *events* [8]*classes* associated with a queue, and allowed usage sequences, define a distinct *protocol class*. An event queue typically has a physical basis which is the actual source of event stimuli. For example, a message queue, or an API. Every queue will typically have an associated *event recognizer*, that is, one or more procedures responsible for mapping a physical occurrence into a corresponding integer valued *event identifier* that distinguish it from all other events that can impinge on the object.

All incoming events are placed in an associated *queue* and within each queue, within a particular *event band*. Every band within a *banded queue* has an associated priority. Events within a band are serviced in FIFO sequence. Band priorities establish a priority that spans all bands for all queues attached to an object. Figure 3 illustrates an object endowed with four queues endowed with three separate protocol classes. The queue instance of type B contains three bands, assigned priorities 3, 4 and 7 respectively. The last of these has lower priority than the first two bands of all instances of queue type C[9].

In summary, an event object is queued within an associated band in FIFO order, and subsequently dispatched by the *Event Dispatcher* according to band priority and band state.

4 The Method of Disposition

4.1 Definitions

An event **E** is *acceptable* or *expected* by a Statechart at time T if it enables at least one transition originating from a state that is active at time T.

The *state event guard* for event **E** in state **S** is defined as the disjunction (logical OR) of all event guards associated with transitions enabled by **E** in **S**[10]. If at least one transition enabled by event **E** in a state **S** does not have a boolean guard, the state event guard for **E** in **S** will always be true --it is a tautology. Similarly, if a given state **S** has no transitions enabled by event **E**, the state event guard for **E** in **S** is always false. In practice state event guards can be synthesized by hand or by a code generator from an analysis of the statechart topology. A non-trivial state event guard is one which is neither a tautology nor a falsehood (as knowable from the statechart topology). In the discussion that follows state event guards are always assumed to be non-trivial. As will be shown, every state event guard is employed as a look-ahead probe, to establish the present receptivity of the associated state to the associated event.

4.2 Event Dispositions

A *disposition* specifies a generic event handling policy to the event management layer. Each disposition has a numeric identifier, an assigned priority and an action block. The identifier serves to distinguish one disposition from another. Priority assignments must establish a strict ordering among all dispositions defined for a queue type. Disposition action blocks contain program statements (expressions without side-effects[11] involving object instance variables, message and band manipulations,

---

[7]The same assumption holds true for *spontaneous transitions* [Firesmith95].

[8][Gamma94+] refers to events as we understand them as Commands.

[9]Instead of introducing bands and band priorities, one might assign priorities to individual events. This approach will, however, typically entail greater overhead than grouping events of equal priority into equivalence classes called bands.

[10] Whenever more than one transition is enabled by E from S, they are considered in a priority sequence established by the implementer.

[11]At the very least side-effects must not modify the value of guards within the statechart.

etc.) that define and implement the disposition semantics.

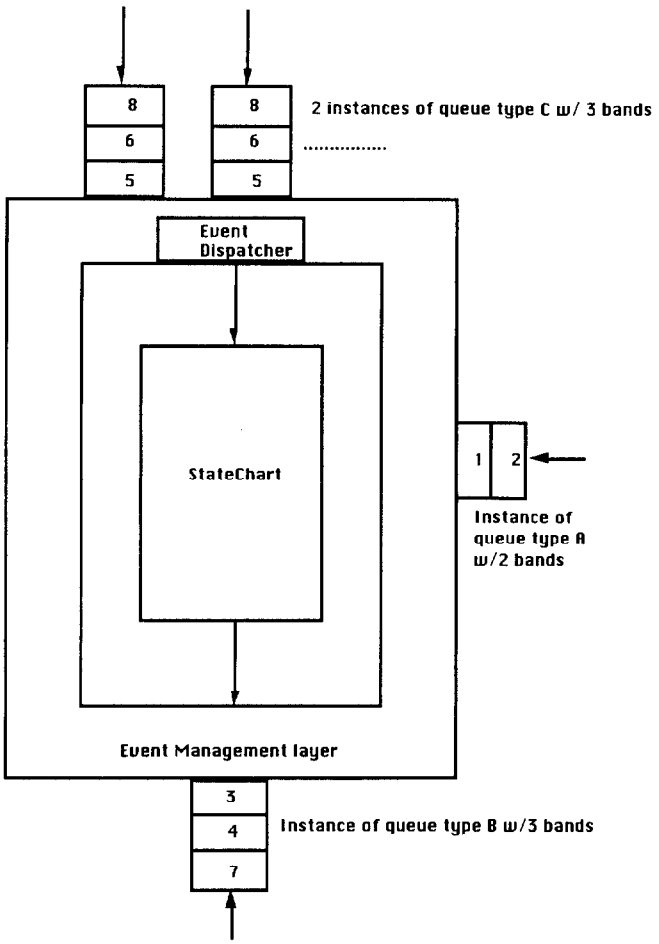


Figure 4: Computational Framework -- the statechart is surrounded by an active event layer that manages the delivery of events arriving at its queues.

According to the method of dispositions, every queue is endowed with a set of dispositions that describes the full range of *generic responses* the statechart may exhibit to any events associated with the queue. A software designer is free to define the set of dispositions and disposition semantics as appropriate for each queue. Nevertheless, each such set must include the reserved *ACCEPT disposition*, the significance of which will become apparent shortly.

The following dispositions, listed by decreasing priority, might be defined for a queue acting as a service interface to a protocol object:

FATAL_PROTOCOL_ERROR
> DEFER
> ACCEPT
> PASS_ON
>WEAK_DEFER
>DISCARD_SILENTLY
>NAK_OUTOFSTATE

The ACCEPT disposition arises for a given event **E** and state **S** whenever **E** enables at least one transition originating in **S**, i.e., the event is expected. Dispositions other than ACCEPT become relevant when event **E** is *not* expected in state **S**, or when there is a state event guard for **E** in **S**.

The reserved *NEUTRAL disposition* (which is discussed further below) serves to desensitize a statechart to an event in a given state. It must always be assigned the lowest priority whenever it is used.

The *default disposition* is the disposition with the lowest assigned priority, with one exception: in those cases where the NEUTRAL disposition is employed, the disposition assigned the next higher priority serves as the default disposition. Whenever ACCEPT is not appropriate (as determined by the statechart topology) the default disposition is used, unless otherwise coerced by a software designer, using *disposition rules* to shape event scheduling in the desired fashion.

Within the set of dispositions listed above, DEFER and WEAK_DEFER share the same action block yet differ by their assigned priorities: the first is stronger than ACCEPT, whereas the second is weaker than ACCEPT but stronger than DISCARD_SILENTLY and the default disposition, NAK_OUTOFSTATE. Had the NEUTRAL disposition be appended to the bottom, NAK_OUTOFSTATE would still serve as the default disposition.

4.3 Guarded Dispositions

As previously discussed, the ACCEPT disposition arises for a given event **E** and state **S** whenever **E** is expected in **S**. By default, the disposition for an event E expected in state **S** is ACCEPT regardless of the value of the state event guard -- the event is announced, and if all guards evaluates to false it has no effect and is forgotten. This is not always the desired behavior. A guarded disposition is one that pertains in lieu of ACCEPT whenever the event state guard is false. If this disposition involves event deferral, the event will be queued and automatically resubmitted when the state event guard later evaluates to true.

4.4 Disposition Rules

A disposition rule is a statement that specifies use of a disposition other than ACCEPT or the default disposition. For instance:

"on all events in bands x, y when in states z, c disposition is DEFER"

"on events z, d for any state disposition is PASS_ON"

"on all events except z, c when in state e disposition is DISCARD_SILENTLY"

Guarded dispositions can be specified in the same fashion. For example the statement:

"on events x, y when in states z, c guarded disposition is DEFER"

4.5 Disposition Priority Matrix

The Disposition Priority Matrix is used to determine the disposition of a protocol object to a proposed event while in a given state. Specifically, the matrix cell value given by disposition_matrix[queue_type, event, state] is the priority of that disposition relative to other dispositions defined for the queue of the event's origin. The actual disposition value can be obtained by function composition[12], as follows:

disposition =

priority_to_disposition(queue_type, disposition_matrix[queue_type,event,state])

As a practical matter, the contents of the disposition matrix, which can grow quite large[13] (50x50 matrices are not uncommon) can be filled in automatically by a code generator based on an analysis of the state machine topology and the disposition rules specified by a programmer. Unless a disposition rule specifies otherwise, all matrix entries for which ACCEPT does not apply receive the default priority value. Guarded dispositions require the evaluation of a *disposition function*[14] that returns a disposition priority value according to the statechart state and the values of any object instance variables used in guard expressions.

4.6 Disposition Map

At any given time a statechart may have a number of active states, each exhibiting its own disposition to a proposed event **E**. The ultimate disposition of the entire statechart must be obtained by a process of resolution.

The method of resolution is very simple and proceeds as follows: for each active state, obtain its disposition priority to event **E**; the disposition with the highest priority prevails[15]. This algorithm is implemented within the *Disposition Map* which, given an object reference, and a proposed event and queue type, returns the event disposition.

---

[12]Each disposition is assigned a unique priority relative to a queue.

[13]Despite their potentially large dimensionality, disposition matrices are quite compact, requiring at most one byte per cell.

[14]A disposition function associated with the disposition matrix can be derived automatically from the corresponding state event guard, which itself can be derived automatically from the statechart topology. As a practical matter, the disposition matrix cells governed by guarded dispositions can hold small indices into a table of disposition functions.

[15] This procedure can be applied to event hierarchies (see [Rumbaugh91]) by considering the disposition of the statechart to **E** as well as the abstract events from which **E** is derived.

In light of this explanation, we are now in position to show how the NEUTRAL disposition can be used to *desensitize* a statechart to a given event in a given state. This disposition should be employed (as it will be in our example) whenever there is a need to observe or track events submitted to the statechart non-intrusively. One application of this facility is concurrent protocol verification.

First, recall that the NEUTRAL disposition is always assigned the lowest priority. Given an event, let us assume that there are, at all times, one or more one active states that exhibit dispositions other than NEUTRAL to this event. This assumption is easily met in a statechart, such as our example in Figure 3, that has two or more concurrent state machines. Within such a setting, the NEUTRAL disposition will never be decisive, since any other disposition, quite possibly the default disposition, is guaranteed to prevail during resolution.

4.7 Event Dispatching

Event dispatching and disposition behavior is shaped using a list of declarative disposition rules involving a set of prioritized generic responses. In this section we describe how this procedure is carried out at run-time.

A band can be in one four possible states: Enabled, Scheduled, Guarded and Declined. The default state -- Enabled-- is entered when a band is first initialized. This is the state of rest when a band is empty and eligible to submit new events as they are received, at which point the band enters the scheduled state, and is placed in the priority band queue.

The *Event Dispatcher* is entered by a thread of execution bearing a new event object to a queue. It services all scheduled bands in its priority band queue until none remain. Events within a given band are serviced one at a time in FIFO order, until the band is empty (it returns to the Enabled state) or enters one of two possible *states of deferral.*

The event dispatcher *proposes* for disposition the first event in the foremost band in the banded queue. The disposition map yields a disposition value for the event. This value and the queue type are used to identify and invoke an associated generic response embodied within the associated disposition action block. Ultimately, a response procedure must either *accept* or *decline* an event proposal. Acceptance of an event causes the event object to be removed from the band that holds it. If the band is empty it enters the Enabled state; otherwise it remains in the Scheduled state. Whenever an event proposal is declined, the band to which it still belongs enters one of two possible states: the *Guarded state* is entered when the disposition is guarded by a state event guard[16]; otherwise the *Declined state* is entered.

Whenever an event has been accepted, the generic response procedure must, upon completion, indicate to the dispatcher whether to 1) proceed with the event to the statechart[17] (as shown in figure 5), or 2) solicit another event proposal from the foremost scheduled band[18].

Once activated the event dispatcher drives the statechart until no more eligible events can be found. Deferred bands are re-appraised as appropriate during this process: deferred and guarded bands are re-scheduled on every state change. Guarded bands are re-scheduled also if the associated guard evaluates to true.

5.0 Example Revisited

We take up once again the example introduced in section 2, by first defining dispositions[19] and their relative priorities for each queue, followed by a

[16]A state event guard may involve event attribute values.

[17]By construction the ACCEPT generic response must always accept the proposed event and request its submission to the statechart.

[18] An event proposal might be accepted only to be discarded, for example to implement the DISCARD-SILENTLY disposition. In such cases step 2 is appropriate.

[19]We omit a definition of the procedures that implement the disposition semantics: their intended generic purpose should be self-evident, and their implementation within a target run-time environment should be entirely straightforward.

series of disposition rules used to populate the disposition matrices for each queue.

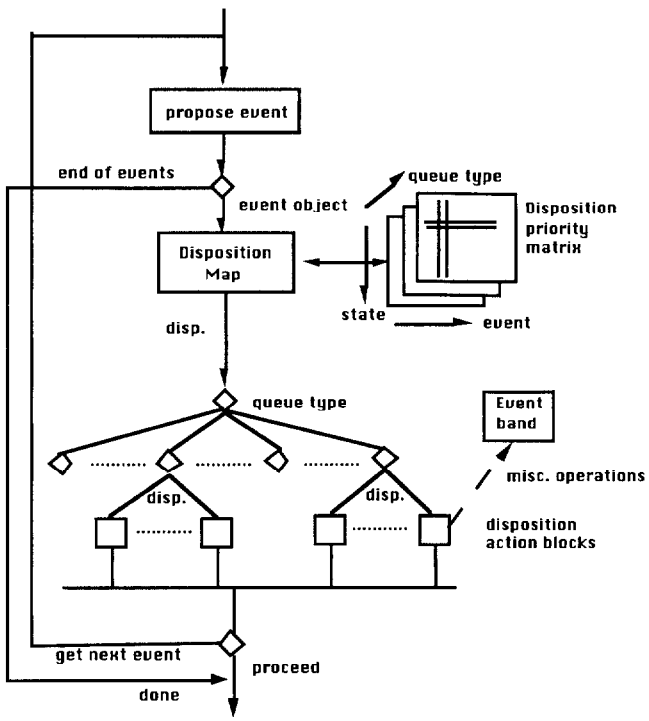


Figure 5: Once entered, the event dispatcher drives the statechart until no more eligible events can be found. Deferred bands are re-appraised as appropriate during the process.

5.1 Queue Descriptions

ADMINISTRATIVE QUEUE:

Bands:

ADMIN_BAND (priority 1) with events:
HEADER_BUFFER_READY,
RESP_BUFFER_READY,
XMIT_TIMEOUT,
ACK_BUFFER_READY

Dispositions: ACCEPT (priority 1)
DISCARD_SILENTLY (priority 2)

INDICATION QUEUE:

Bands:

REPLY_BAND (priority 2) with events:
BIND_ACK,
BIND_NAK,
UNBIND_ACK

INDICATION_BAND (priority 3) with events
RECV_DATA_IND,
XMIT_DATA_ACK

Dispositions: ACCEPT (priority 1)
DISCARD_SILENTLY (priority 2)
DEFER (priority 3)

REQUEST QUEUE:

Bands:

REQUEST_BAND (priority 4) with events:
BIND_REQ,
UNBIND_REQ,
XMIT_DATA_REQ,
NAK_REQ

Dispositions: DEFER (priority 1)
ACCEPT (priority 2)
NAK_OUT_OF_STATE (priority 3)
NEUTRAL (priority 4)

5.2 Disposition Rules

// Incoming indications and acknowledgements are
// discarded when the object is unbound.

on events RECV_DATA_IND and XMIT_DATA_ACK, when in state UNBOUND disposition is DISCARD_SILENTLY;

// Incoming acknowledgements are discarded when
// there is a token mismatch.

on event XMIT_DATA_ACK, when in state AWAIT_XMIT_ACK guarded disposition is DISCARD_SILENTLY;

// Incoming acknowledgements are discarded when
// not expected.

on event XMIT_DATA_ACK when in state AWAIT_XMIT_ACK, AWAIT_HEADER_BUFFER disposition is DISCARD_SILENTLY;

// New interface requests are deferred until the
// present one is fully processed

on any event in band REQ_BAND when in state AWAIT_RESP_READY, AWAIT_RESP_BUFFER disposition is DEFER.

// We desensitize the interface state machine to
// interface request events (all but NAK_REQ) the
// timeliness and disposition of which is determined
// elsewhere within the statechart. For instance, the
// acceptance of event XMIT_DATA_REQUEST
// should be determined by transition 5 and the
// previous rule, not transition 2 (see figure 3).

on events BIND_REQ and UNBIND_REQ and XMIT_DATA_REQ, when in state AWAIT_REQ disposition is NEUTRAL.

The contents of the disposition matrices and the associated disposition functions and guards implied by these statements are produced automatically by a code generator, and have been omitted.

We note that event closure and scheduling is achieved non-procedurally, without need of any changes to the base statechart, which thus retains its original purposeful simplicity.

6 Related Work

In a recent paper [Harel94+], Harel et al. propose an event management scheme for statecharts that bears some resemblance to the one proposed in this paper. Under their scheme, every object is endowed with one or more event queues. Unexpected events that arrive from a given service direction are automatically subject to deferral within the queue associated with that service. An event thus deferred is automatically resubmitted when it enables at least one transition for which the condition (guard), if any, evaluates to true. Internal broadcast events are handled separately, and are always given precedence over external events, which are considered only when the chain reaction of broadcast events have run their course, and the statechart has reached a quiescent state.

In effect, this scheme can be implemented easily using the method of dispositions with a single disposition (DEFER) defined as the default disposition, and with guarded disposition rules. We submit that the proposed scheme still imposes unnecessary work on the statechart of a generic nature, for instance to discard events that may arrive "too late" rather than early. Moreover it fails to address the potential need for synchrony between multiple state machines relative to a given event or, more generally, to provide a vehicle for conflict resolution between differing dispositions manifested by concurrently active states.

7 A Behavioral Design Pattern

Within the appendix we submit a contribution to the growing catalog of design patterns, the along the lines set forth in [Gamma94+]. The presentation is somewhat compressed owing to limitations on space. Our design pattern applies to objects with state characterized by state complexity, and multi-faceted event-driven interactions with other objects.

8 Final Remarks

Finite state machines are often criticized for their tendency towards combinatorial explosion as the number of states, events and state transitions grows large. Harel's statecharts are the traditional solution to this problem. However, we have observed that even statecharts are insufficient to deal with the inherent complexities of event driven objects subject to "faulty" stimuli or asynchronous, overlapped stimuli from a number of other objects. A major source of trouble is the systematic treatment of unexpected, out-of-sequence or partially expected events -- these are events that cannot be handled immediately (and thus require re-scheduling), or for which an immediate generic response is appropriate.

We have described the method of dispositions that deals with these problems. According to this method, event scheduling, disposition and closure is shaped using a list of declarative disposition rules, involving a set of prioritized generic responses carried out within an active event management layer surrounding a basic statechart. The full expressive power of the statechart can now be focused exclusively on expressing event and state sensitive behavior in response to idealized or anticipated event stimuli, resulting in statecharts of greater clarity and reduced complexity.

This method is quite general, and can be adapted to a broad class of statechart-like methods, of either Moore or Mealy type, that allow transitions to be

enabled by events in an explicit manner. [Firesmith95] surveys several methods that should lend themselves to such an adaptation.

We have constructed a development tool, StateCraft, that comprises a C code generator, document writer, and run-time library. Support for C++ is expected in the near future. The selection of C as our initial target language was necessitated by our application domain, namely object-oriented development of kernel-level protocol software. The StateCraft facility has been used successfully in the implementation, for commercial purposes, of a number of significant communication protocols (e.g. point-to-point protocol, ISO Transport layer class 0) and application programming interfaces (e.g. TLI and sockets). Rough comparisons suggest that the performance and code size of software generated using this tool is on par with implementations done "by hand", but required far less effort to develop, and are arguably easier to understand and maintain.

Continuing research efforts include a study of the implications of this new method relative to single and multiple inheritance of behavior, and development of a drag and drop visual interface for the StateCraft tool.

9 References

[Allen94] Allen, A., Object Oriented Message Handling Sub-system and Method, patent pending with the US patent office, 1994.

[Allen95] A. Allen, "A Method for Object Oriented Protocol Development", submitted to the International Conference on Protocol Specification, Testing & Verification '95, 1995.

[Ansart83+] J.P. Ansart, V. Chari, M. Meyer, O. Rafiq, D. Simon, "Description, Simulation, Implementation of Communication Protocols using PDIL", ACM Sigcomm'83, Symposium on Communication Architectures and Protocols, U. of Texas at Austin, March 1983.

[Bochmann87] G. Bochmann"Usage of Protocol Development Tools: The Results of a Survey", Protocol Specification, Testing and Verification, VII, Elsevier Science Publishers B.V. (North Holland), 1987.

[Bochmann80] G. Bochmann"A general Transition Model for Protocols and Communication Services", IEEE Trans. Communications Vol 28, No 4, April 1980, 643-650.

[Booch93] G. Booch, Object Oriented Analysis and Design, Benjamin/Cummins, 1993.

[deChampeaux93+] D. de Champeaux, D. Lea & P. Faure, Object Oriented System Development, Addison-Wesley, 1993.

[Embley92+] D. Embley, B. Kurtz, S. Woodfield, Object-Oriented Systems Analysis , Yourdon Press, Prentice Hall 1992.

[Firesmith95] D. Firesmith, "Object-oriented state modelling using ADM4", Journal of Object-Oriented Programming , 57-65, 1994.

[Gamma94+] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns , Addison-Wesley, 1994.

[Harel87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Sci. Comput. Prog. , 231-274, 1987.

[Harel94+] D. Harel, E. Gery, M. Politi, Object-Oriented Modeling with Statecharts, Technical Review CS94-20, The Weizmann Institute, of Science, Rehovot, Israel, 1994

[Holzmann91] G. Holzmann, Design and Validation of Computer Protocols, Prentice Hall, 1991.

[Liu94] C. Liu, "An Object-Based Approach to Protocol Software Implementation", Proceedings of the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications, 1994.

[Rumbaugh91+] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, Object Oriented Modelling and Design, Prentice Hall, 1991.

[Shankar91] U. Shankar, "Modular Design Principles for Protocols with an Application to the Transport Layer", Proceedings of the IEEE, Vol. 70, No. 12, December 1991.

[Shlaer91+] S. Shlaer & S.J. Mellor, Object Lifecycles: Modelling the World in States, Yourdon Press, 1991.

[Selic92+] B. Selic, G. Gullekson, J. McGee & I.Engelbert, "ROOM: An Object-Oriented Methodology for Developing Real-Time Systems", Proc. 5th International Workshop on CASE, Montreal Canada, 1992.

[Selic94+] Selic, B., R. Tigg, D. Daoust & P. Ward, "Resolved: High Level Efficient Models Can be Formally Transformed into Complete and Efficient Real-Time Implementations", position paper presented at OOPSLA'94.

Appendix: A Behavioral Design Pattern

DISPOSITION Object Behavioral

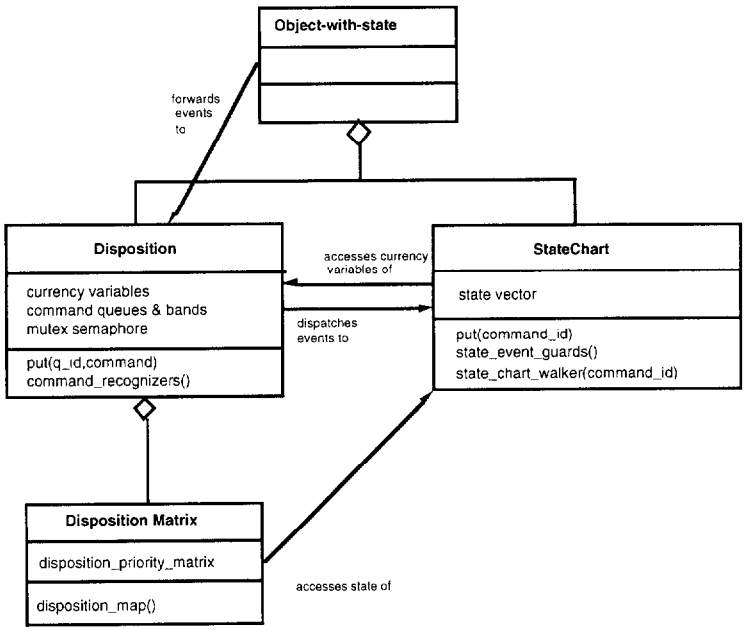
Intent

Given a complex object that incorporates a statechart, provide an event (command[20]) management object that assumes responsibility for command queueing, disposition and dispatching to the statechart, according to the method of dispositions.

Motivation

See section 1: Introduction.

Structure



Applicability

This method applies in connection with objects that are characterized by

- state complexity, modelled by two or more concurrent automata, and nested automata as needed,
- multi-faceted command-driven interactions with other objects, where the interactions may be unpredictably sequenced, prioritized, concurrent, error-prone or asynchronous.

Participants

Object-with-state

- contains instances of the associated Disposition and StateChart classes.
- forwards incoming commands to the Disposition object.

Disposition

- defines the *put()* command invoked by Object-with-state to forward commands.
- performs event recognition, queuing, disposition and dispatching.
- within a concurrent execution environment, serializes access to the statechart object.

StateChart

- implements the required statechart behavior in response to events announced one at a time by the Disposition object.
- implements internal event broadcasting.

Disposition Matrix

- specifies, in matrix form, the disposition priority, and the disposition, for each event in each state.
- implements the disposition map that yields the disposition of the entire statechart (in a given state) to a given event.

Collaborations

- Object-with-state forwards all incoming messages to the Disposition object.

[20] [Gamma94+] refers to events as we understand them as Commands.

- Disposition delegates all event processing to the StateChart object, and invokes the disposition map method of the Disposition Matrix object.
- StateChart maintains a reference to the Disposition object, thereby providing access to Disposition attributes (e.g. misc. currency variables such as current_ event, current_queue, current_ band, and so forth).
- The Disposition Matrix consults the state of the stateChart object while evaluating the disposition map.

Consequences

By entrusting event scheduling and disposition to a specialized event management object, through which all incoming commands must pass, use of this design pattern

- insures event (command) closure.
- simplifies the design of objects that incorporate complex automata.
- can result in improved software reliability.

Implementation

We can expect a code generator to produce an implementation of the Disposition, Disposition Matrix and StateChart object classes, for a specific design captured in a suitable high level description language. Within such a linguistic framework, we would expect low level procedural detail, pertaining to guard expressions, action routines, disposition action blocks, and so forth, to be expressed directly in the targeted language (e.g. C++) within structured procedure blocks. Here again, the code generator can be expected to encapsulate these code blocks within private or public class methods, and associated data structures, so as insure their subsequent activation at run time.

Known Uses

This pattern has been used to implement a variety of command-driven reactive objects, most notably in the domain of communication protocols.

Related Patterns

Command, State, StateChart, Adaptor.