Science of Computer Programming 8 (1987) 231-274
North-Holland

STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

David HAREL
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel

Communicated by A. Pnueli
Received December 1984
Revised July 1986

Abstract. We present a broad extension of the conventional formalism of state machines and state diagrams, that is relevant to the specification and design of complex discrete-event systems, such as multi-computer real-time systems, communication protocols and digital control units. Our diagrams, which we call *statecharts*, extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication. These transform the language of state diagrams into a highly structured and economical description language. Statecharts are thus compact and expressive—small diagrams can express complex behavior—as well as compositional and modular. When coupled with the capabilities of computerized graphics, statecharts enable viewing the description at different levels of detail, and make even very large specifications manageable and comprehensible. In fact, we intend to demonstrate here that statecharts counter many of the objections raised against conventional state diagrams, and thus appear to render specification by diagrams an attractive and plausible approach. Statecharts can be used either as a stand-alone behavioral description or as part of a more general design methodology that deals also with the system's other aspects, such as functional decomposition and data-flow specification. We also discuss some practical experience that was gained over the last three years in applying the statechart formalism to the specification of a particularly complex system.

1. Introduction

The literature on software and systems engineering is almost unanimous in recognizing the existence of a major problem in the specification and design of large and complex *reactive systems.* A reactive system (see [14]), in contrast with a *transformational system*, is characterized by being, to a large extent, event-driven, continuously having to react to external and internal stimuli. Examples include telephones, automobiles, communication networks, computer operating systems, missile and avionics systems, and the man-machine interface of many kinds of ordinary software. The problem is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and

* The initial part of this research was carried out while the author was consulting for the Research and Development Division of the Israel Aircraft Industries (IAI), Lod, Israel. Later stages were supported in part by grants from IAI and AD CAD, Ltd.

rigorous, sufficiently so to be amenable to detailed computerized simulation. The behavior of a reactive system is really the set of allowed sequences of input and output events, conditions, and actions, perhaps with some additional information such as timing constraints. What makes the problem especially acute is the fact that a set of sequences (usually a very large and complex one) does not seem to lend itself naturally to 'friendly' gradual, level-by-level descriptions, that would fit nicely into a human being's frame of mind.

For transformational systems (e.g., many kinds of data-processing systems) one really has to specify a transformation, or function, so that an input/output relation is usually sufficient. While transformational systems can also be highly complex, there are several excellent methods that allow one to decompose the system's transformational behavior into ever-smaller parts in ways that are both coherent and rigorous. Many of these approaches are supported by languages and implemented tools that perform very well in practice. We are of the opinion that for reactive systems, which present the more difficult cases, this problem has not yet been satisfactorily solved. Several important and promising approaches have been proposed, and Section 8 of this paper discusses a number of them. However, the general feeling is that many more improvements and developments are necessary. This paper represents a certain attempt to progress along these lines.

Much of the literature also seems to be in agreement that states and events are *a priori* a rather natural medium for describing the dynamic behavior of a complex system. See, for example, [7-9, 19, 23]. A basic fragment of such a description is a *state transition*, which takes the general form "when event α occurs in state A, if condition C is true at the time, the system transfers to state B". Indeed, many of the informal exchanges concerning the dynamics of systems are of this nature; e.g., "when the plane is in cruise mode and switch x is thrown it enters navigate mode", or "when displaying the time, if button y is pressed the watch starts displaying the date". Finite state machines and their corresponding state-transition diagrams (or *state diagrams* for short) are the formal mechanism for collecting such fragments into a whole. State diagrams are simply directed graphs, with nodes denoting states, and arrows (labelled with the triggering events and guarding conditions) denoting transitions. Figure 1 shows a simple self-explanatory state diagram.

However, it is also generally agreed that a complex system cannot be beneficially described in this naive fashion, because of the unmanageable, exponentially growing multitude of states, all of which have to be arranged in a 'flat' unstratified fashion,

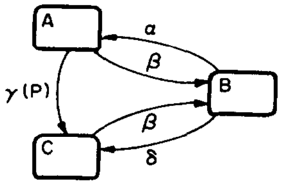


Fig. 1.

resulting in an unstructured, unrealistic, and chaotic state diagram. See, for example, [7, p. 57; 23, p. 380]. To be useful, a state/event approach must be modular, hierarchical and well-structured. It must also solve the exponential blow-up problem by somehow relaxing the requirement that all combinations of states have to be represented explicitly. A good state/event approach should also cater naturally for more general and flexible statements, such as

(1) "in all airborne states, when yellow handle is pulled seat will be ejected",
(2) "gearbox change of state is independent of braking system",
(3) "when selection button is pressed enter selected mode",
(4) "display-mode consists of time-display, date-display and stopwatch-display".

Clause (1) calls for the ability to *cluster* states into a *superstate*, (2) introduces *independence*, or *orthogonality*, (3) hints at the need for more *general transitions* than the single event-labelled arrow, and (4) captures the *refinement* of states.

Conforming to the "one picture is worth a thousand words" aphorism, one would like to find a way to change or extend the good old state/event formalism in ways that will satisfy these needs, while retaining, or even enhancing, the visual appeal of state diagrams. In fact, [10], when surveying flowchart techniques, all but challenges the computer science research community to come up with a good visual medium for describing concurrency; since orthogonality, as described later, represents concurrency, this paper can be viewed as a response to that challenge.

Some of the previous attempts at extending flat state diagrams, such as communicating state-machines and augmented transition-diagrams will be discussed in Section 8.

In Sections 2-5, the main sections of the paper, we introduce *statecharts*[1] as a possible attempt at confronting these problems. Statecharts constitute a *visual formalism* for describing states and transitions in a modular fashion, enabling clustering, orthogonality (i.e., concurrency) and refinement, and encouraging 'zoom' capabilities for moving easily back and forth between levels of abstraction.

Technically speaking, the kernel of the approach is the extension of conventional state diagrams by AND/OR decomposition of states together with inter-level transitions, and a broadcast mechanism for communication between concurrent components. The two essential ideas enabling this extension are the provision for 'deep' descriptions and the notion of orthogonality. Since we strongly believe in the virtues of visual descriptions, the approach is described solely in its diagrammatic terms, although the reader should be able to provide a textual, language-theoretic or algebraic equivalent if so desired. In a nutshell, one can say:

$$\begin{aligned}\text{statecharts} = \text{state-diagrams} &+ \text{depth}\\ &+ \text{orthogonality} + \text{broadcast-communication}.\end{aligned}$$

[1] This rather mundane name was chosen, for lack of a better one, simply as the one unused combination of 'flow' or 'state' with 'diagram' or 'chart'. In the earlier versions of this paper [12], we used the word *statification*, meant as a combination of specification and stratification using states, for the act of preparing statecharts.

The running example used throughout these sections concerns the author's Citizen Quartz Multi-Alarm III wristwatch[2], the example being sufficiently simple to be contained here (almost) in its entirety, but sufficiently complex to serve as an illustration of the method.

Section 6 discusses a number of advanced features that are under investigation as possible additions to the basic formalism, including the integration of statecharts with temporal logic and the introduction of probabilism. Section 7 contains a brief account of the formal semantics of statecharts; a more complete treatment, however, is deferred to a separate paper. Section 8, as mentioned, discusses related work and compares the statechart formalism with some alternative notations suggested for the specification of reactive systems. Section 9 reports briefly on the experience accumulated with the language and on an implementation that is in the workings.

2. State-levels: Clustering and refinement

In deciding upon a graphical representation for capturing depth and hierarchy, there is a real disadvantage in drawing trees or other line-graphs. These media make no use whatsoever of the *area* of the diagram: lines and points are of no width, and no advantage is taken of location. We shall use rounded rectangles (*boxes* in the sequel) to denote states at any level, using encapsulation to express the hierarchy relation. Arrows will be allowed to originate and terminate at any level. The graphics is actually based on a more general concept, the *higraph*, which combines notions from Euler circles, Venn diagrams and hypergraphs, and which seems to have a wide variety of applications. See [13].

An arrow will be labelled with an *event* (or an abbreviation of one) and optionally also with a parenthesized *condition*. (In Section 5 it will be allowed to be labelled also with Mealy-like outputs, or *actions*.) Thus, in Fig. 1 there are three states *A*, *B*, and *C* and, for example, event γ occurring in state *A* transfers the system to state *C*, but only if condition *P* holds at the instant of occurrence.

Now, since event β takes the system to *B* from either *A* or *C* we can *cluster* the latter into a new super-state *D* and replace the two β arrows by one, as in Fig. 2.

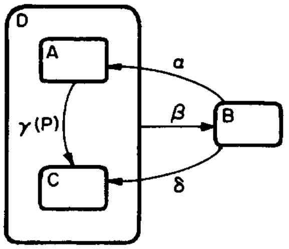


Fig. 2.

[2] This is a product of the Citizen Watch Company of America, Inc.

The semantics of D is then the *exclusive-or* (XOR) of A and C; i.e., to be in state D one must be either in A or in C, and not in both. Thus, D is really an *abstraction* of A and C. The state D and its outgoing β arrows thus capture a *common property* of A and C, namely, that β leads from them to B. The decision to let transitions that leave a super-state, such as the β in Fig. 2, stand for transitions leaving *all* substates turns out to be highly important, and is the main way statecharts economize in the number of arrows. Figure 2 might also be approached from a different angle: first we might have decided upon the simple situation of Fig. 3, and then state D could have been *refined* to consist of A and C, yielding Fig. 4. Having made this refinement, however, the incoming α and β arrows become underspecified, as they do not say which of A or C is to be entered. Extending them to point directly to A and C, respectively, does the job, and if the γ transition within D is added, one indeed obtains Fig. 2. Thus, clustering, or abstraction, is a bottom-up concept and refinement is a top-down one; both give rise to the *or*-relationship between a state's substates.

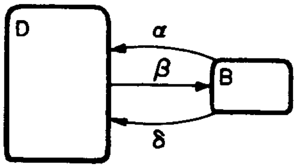


Fig. 3.

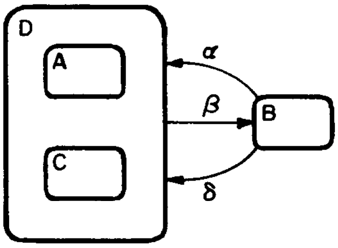


Fig. 4.

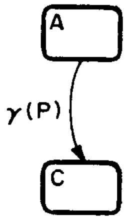


Fig. 5.

Both *zooming-in* and *zooming-out* can be illustrated using this simple example. The first is achieved by looking 'inside' D (disregarding external interface for the time being) and finding simply Fig. 5, and the latter is done by eliminating the inside of D and abstracting Fig. 2 to Fig. 3. These notions will acquire more significance later on.

Suppose now that as far as the 'outside' world is concerned A is the *default state* among A, B and C, in the sense that if asked to enter the A, B, C group of states the system is to enter A unless otherwise specified. In the description given by Fig. 1 this can be captured by a small arrow as in Fig. 6(i). For Fig. 2 one can use the direct notation of Fig. 6(ii), or alternatively, the two-step one of Fig. 6(iii), which

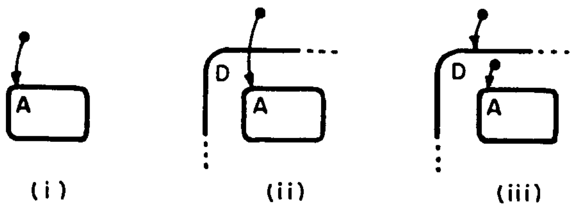


Fig. 6.

says that D is default among D and B, and A is the default among A and C. Of course, for zooming in and out the latter has obvious advantages. Default arrows are thus analogous to the start states of finite-state automata.

Let us now introduce our running example. The Citizen Quartz Multi-Alarm III watch has a main display area and four smaller ones, a two-tone beeper, and four control buttons denoted here a, b, c and d. See Fig. 7. It can display the time (with

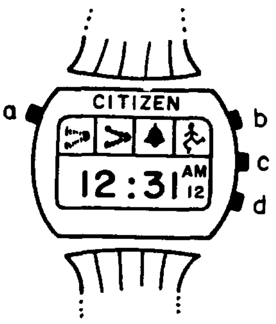


Fig. 7.

am/pm or 24 hour time modes) or the date (day of month, month, day of week), it has a chime (beeps on the hour if enabled), two independent alarms, a stopwatch (with lap and regular display modes, and a 1/100 s display), a light for illumination, a weak battery blinking indication, and a beeper test. We shall assume throughout that the main functions of these are known, and will use liberal terminology, such as 'power weakens' to denote certain events of obvious meaning, though, of course, to make things complete one would have to tie these events up with actual happenings in the physical parts of the system, or to specify them as output events produced in other, separately specified, components.

The main external events will be the depressing and releasing of buttons (e.g., event "a" denotes button a being depressed, and "$\hat{a}$" denotes it being released), and there will be certain internal ones too. The distinction is sharpened in Section 5. We remark here that while the description of the watch presented herein is intended to be as faithful to its actual workings as possible, there are some very minor differences that are not worth dwelling upon here, and that most likely will not be detected in normal use of the watch. The point is, however, that the statechart of the watch (cf. Fig. 31) was obtained by the author using the obviously inappropriate method of observation from the final product; had it been the basis for the initial specification and design of that final product, in the spirit of the gradual development presented below, the undescribed anomalies might have been avoided.

Figure 8 shows the transitions between the normal *displays* mode and the various beeping states. Here $T1$ and $T2$ stand for the respective internal time settings of the alarms, and T for the current time. Also, $P1$ abbreviates "*alarm*1 enabled $\wedge$ (*alarm*2 disabled $\vee$ $T1 \neq T2$)", and similarly for $P2$, while P abbreviates "*alarm*1

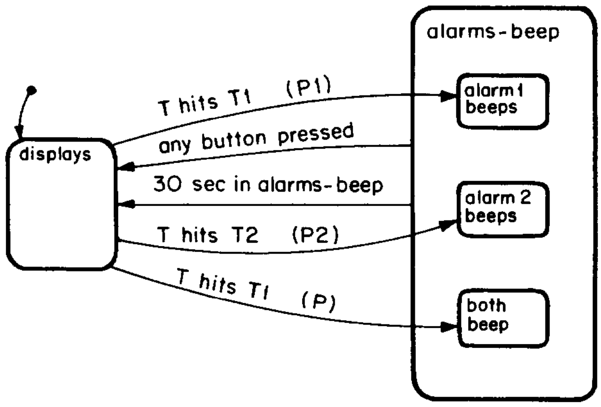


Fig. 8.

enabled ∧ *alarm*2 enabled ∧ $T1 = T2$". These conditions will take on a more precise form when more of the description of the watch is available. Notice the clustering, which replaces six arrows by two. Actually, the displays do not change while the alarm is beeping, but this information should be specified under the topic of activities discussed in Section 5.

A refinement of the *displays* state yields Fig. 9 in which it has been decided that there will be a cycle of displays linked by repeated depressings of *a*; that the time and date displays are linked by *d*'s but that the time display will resume after 2 minutes in *date*. Also, the time display is the default, meaning, among other things, that the entrances from *alarms-beep* in Fig. 8 will actually be entrances to *time*. This will later be changed, but for the time being it is good enough.

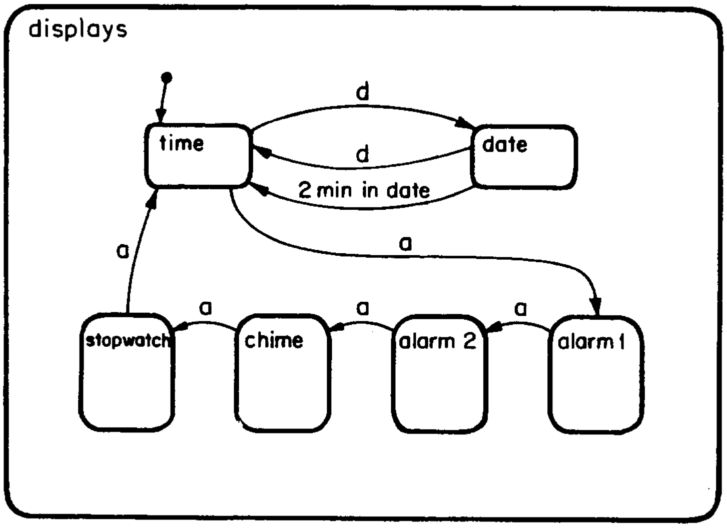


Fig. 9.

One of the most interesting and frequent ways of entering a group of states is by the system's *history* in that group. The simplest kind of this 'enter-by-history' is entering the state most recently visited. In our watch example, we can refine each of the three states *alarm*1, *alarm*2 and *chime* of Fig. 9 with *off* and *on* substates, intended to model the display of the disabled or enabled status of the corresponding feature. The new *alarm*1 state with its *a*-entrance can be described as in Fig. 10(a), meaning "enter the most recently visited of the two, and enter *off* if you are here for the first time". Equivalently, one can use the continuation-arrow notation of Fig. 10(b). In either case, *d* is used to switch between the two substates.

It should be noted that an H generally means that history is applied only on the level in which it appears, so that in Fig. 11(a) history chooses only between *G* and *F*; that is, the system enters *B* if it was in *A* or *B* when it most recently left *K*, and *C* if it was in *C*, *D* or *E* on that last visit. The choice of *B* and *C* here is by the default arrows. History can be made to apply all the way down to the lowest level of states by attaching an asterisk to the H-entry. Thus, in Fig. 11(b) the system will enter the most recently visited state from among *A*–*E*, overriding both defaults.

One can achieve effects in between these one-level/all-levels extremes by additional H-entrances. In Fig. 12 the system will enter *B* if its last visit to *K* was actually to *G*, or that one of *C*, *D* or *E* last visited, otherwise. In this way only the default arrow to *C* is overridden by the history. For specifying more complex notions of history one could use temporal logic, as discussed in Section 6.3.

The *displays* state is now enriched to include updating capabilities for both alarms as well as for the general time/date combination. Entering update modes is achieved via *c*, with a 2-second continuous depressing required in the time/date update mode. In all cases, depressing *b* brings control back to the previous display. The situation of *displays* so far (including the addition of an H*-entrance, modifying the earlier decision to return to *time* after an alarm beep) is given in Fig. 13.

The next step is the refinement of these updating modes. The statecharts for these are given in Figs. 14 and 15. In both cases, the additional exit via *c* could not have been given on the previous level in the naive way of Fig. 16(a), since *c* does not apply to the whole of *update*. However, in such a case one uses the notation of Fig. 16(b) to illustrate the fact that *c* applies to certain parts of *update*, to be specified later. Indeed, zooming out of, say, Fig. 14, would result in the likes of Fig. 16(b), not 16(a).

We should say something about the H-arrows in Figs. 14 and 15. Depressing *d* in any of the updating substates causes exit (but not exit from the encapsulating superstate) and immediae entrance to the most recently visited substate, i.e., that just left! In the case of Fig. 14 this *d*-arrow replaces nine *d*-arrows, one for each substate. For the time being, these *d*-arrows are merely no-op's, but, as discussed later, they actually are responsible for the updating itself, and would require one additional level of states to specify how each pressing of *d* boosts the appropriate internal setting by one unit.

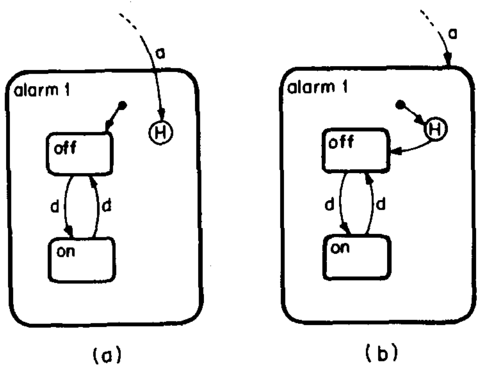


Fig. 10.

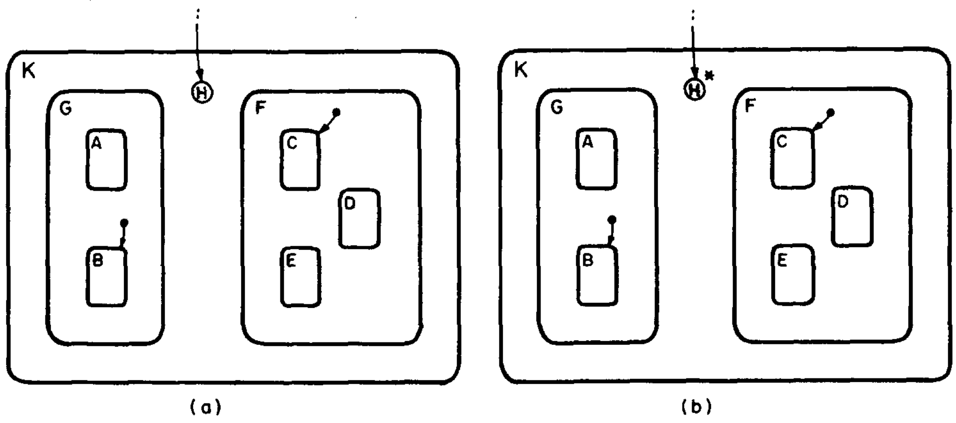


Fig. 11.

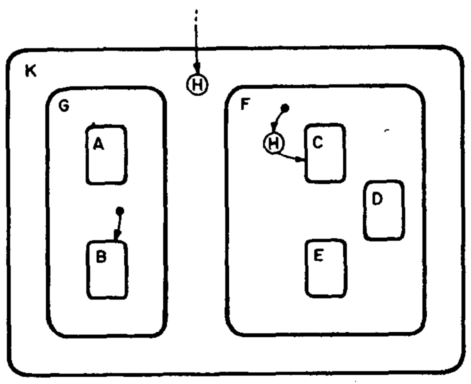


Fig. 12.

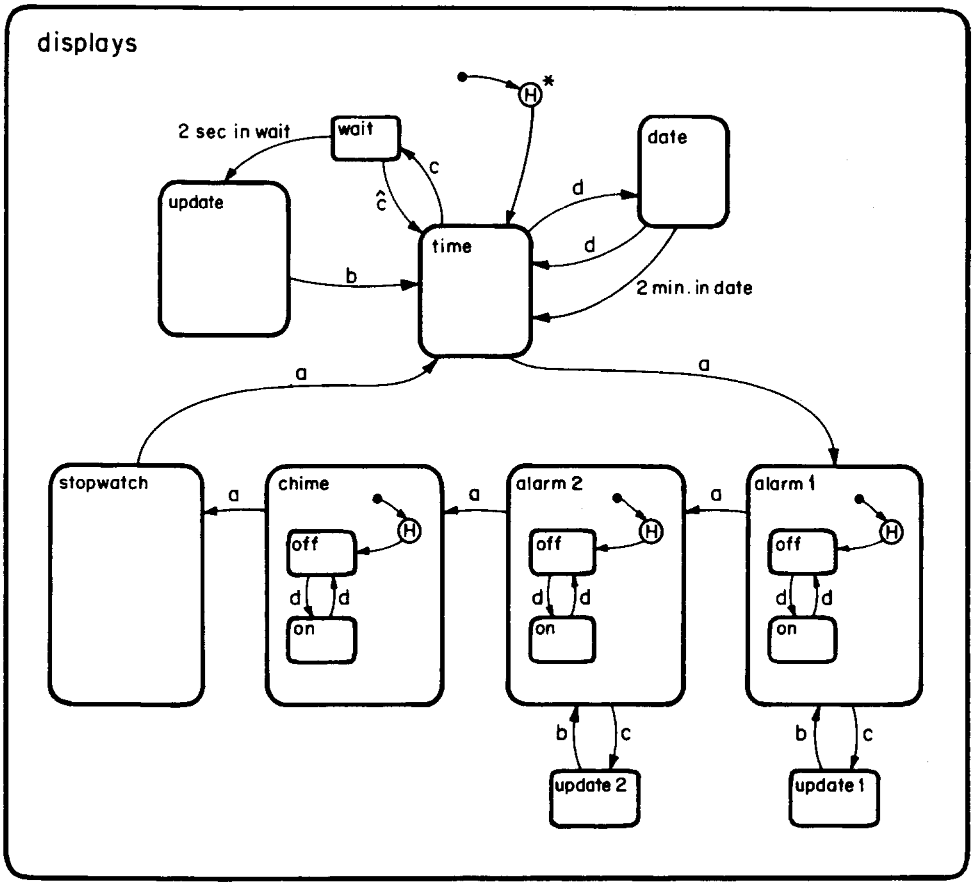

displays
2 sec in wait
wait
c
ĉ
update
b
time
d
d
date
2 min. in date
a
a
stopwatch
chime
alarm 2
alarm 1
a
a
a
off
on
d
d
H
H*
b
c
update 2
update 1

Fig. 13.

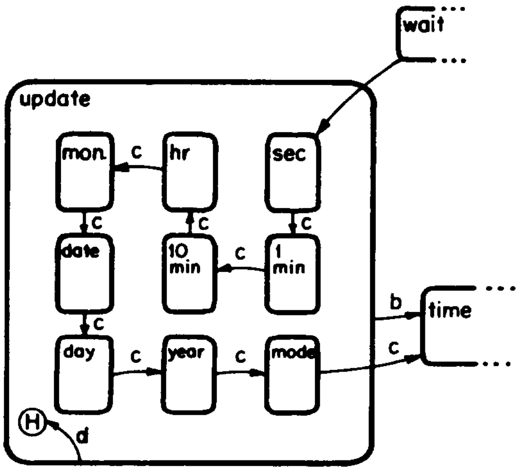

wait
update
mon.
hr
sec
date
10 min
1 min
day
year
mode
c
b
time
d
H

Fig. 14.

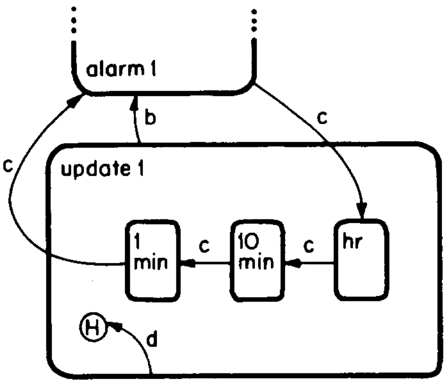


Fig. 15.

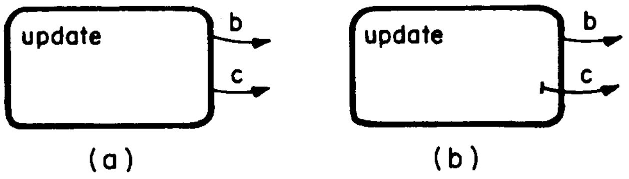


Fig. 16.

We allow for an economical representation of arrows with common sources, targets or events, as in Fig. 17. Note that the variant of Fig. 17(c), in which the arrows are reversed, is a contradiction to the desired determinism of the system. Clearly, more subtle contradictions can occur as a result of the 'deep' character of statecharts, and should be carefully avoided. For example, Fig. 18 shows an α contradiction from A, resulting from the fact that the possible transitions leading out of a state are those emanating from its periphery, as well as those emanating from any of its ancestors' peripheries. Figure 18 also contains a default contradiction upon entering B via β. Had the default arrow entering D been entirely contained within the area of C, there would have been no problem; it would only have influenced the entrance to C via γ. This fact is a consequence of our area-dominated graphical representation. (As it is, γ is underspecified, since C contains no default.)

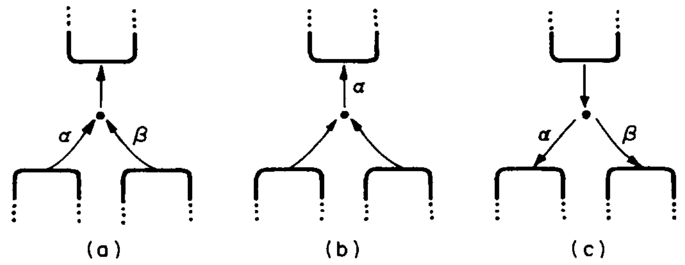


Fig. 17.

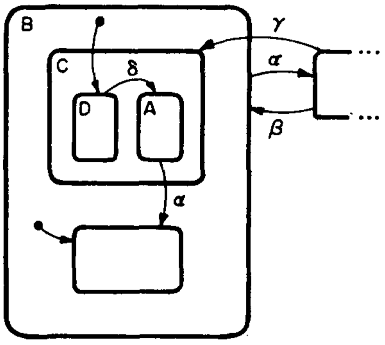


Fig. 18

3. Orthogonality: Independence and concurrency

The capabilities described in the previous section represent only one part of the story, namely, the XOR (exclusive or) decomposition of states, and some related concepts and notations. In this section we introduce AND decomposition, capturing the property that, being in a state, the system must be in *all* of its AND components. The notation used in statecharts is the physical splitting of a box into components using dashed lines.

Figure 19 shows a state Y consisting of AND components A and D, with the property that being in Y entails being in some combination of B or C with E, F or G. We say that Y is the *orthogonal product* of A and D. The components A and D are no different conceptually from any other superstates; they have defaults, internal transitions, etc. Entering Y from the outside, in the absence of any additional information, is actually entering the combination (B, F) by the default arrows. If

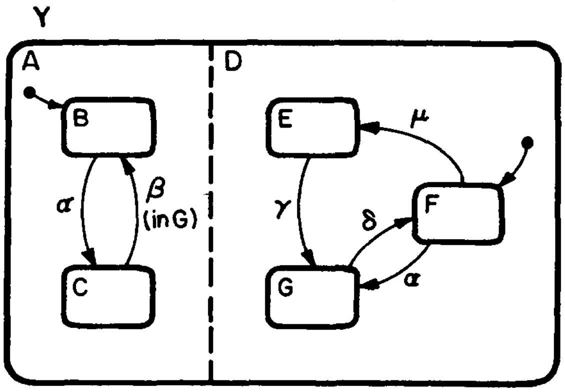


Fig. 19.

event α then occurs, it transfers B to C and F to G *simultaneously*, resulting in the new combined state (C, G). This illustrates a certain kind of *synchronization*: a single event causing two simultaneous happenings. If, on the other hand, μ occurs at (B, F) it affects only the D component, resulting in (B, E). This, in turn, illustrates a certain kind of *independence*, since the transition is the same whether the system is in B or in C in its A component. Both behaviors are part of the *orthogonality* of A and D, which is the term we use to describe the AND decomposition.

Figure 20 is the conventional AND-free equivalent of Fig. 19. The reader will no doubt realize that Fig. 20 contains six states because the components of Fig. 19 contained two and three. Clearly, two components with one thousand states each would result in one *million* states in the product. This, of course, is root of the exponential blow-up in the number of states, which occurs when classical finite-state automata or state diagrams are used, and orthogonality is our way of avoiding it.

Note that the β-transition from C to B has the condition "in G" attached to it, with the obvious consequences, shown explicitly in Fig. 20. Thus, while Y has indeed been split into two orthogonal components, there will in general be some dependence. The "in G" condition causes A to depend somewhat on D, and indeed to 'know' something about the inner states of D. Formally, orthogonal product is a generalization of the usual product of automata, the difference being that the latter is usually required to be a *disjoint* product, whereas here some dependence between components can be introduced, by common events or "in G"-like conditions.

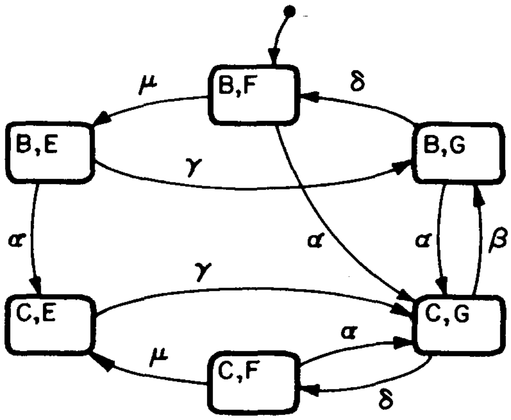


Fig. 20.

One slightly bothersome notational problem is the lack of an appropriate location for the name "Y". The product state Y will, in general, lie within some superstate Z, to which the area outside the borderline of the (A, D) box 'belongs'. Of course, it is possible to use an additional box as in Fig. 21(a). We prefer to try managing without the name Y or simply to attach it to the outside as in Fig. 21(b).

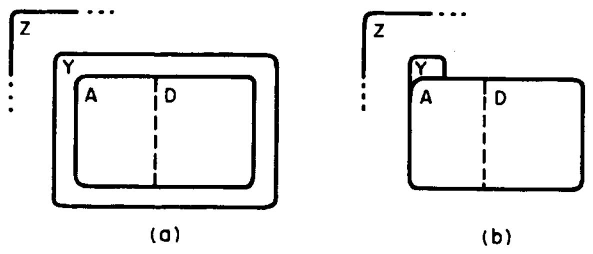


Fig. 21.

An obvious application of orthogonality is in splitting a state in accordance with its physical subsystems. This typically occurs on a very high level of the specification. In an avionics system, for example, one might have a *general-mode* component, and orthogonal components for subsystems, such as the radar. An overly simplified first attempt might look like Fig. 22.

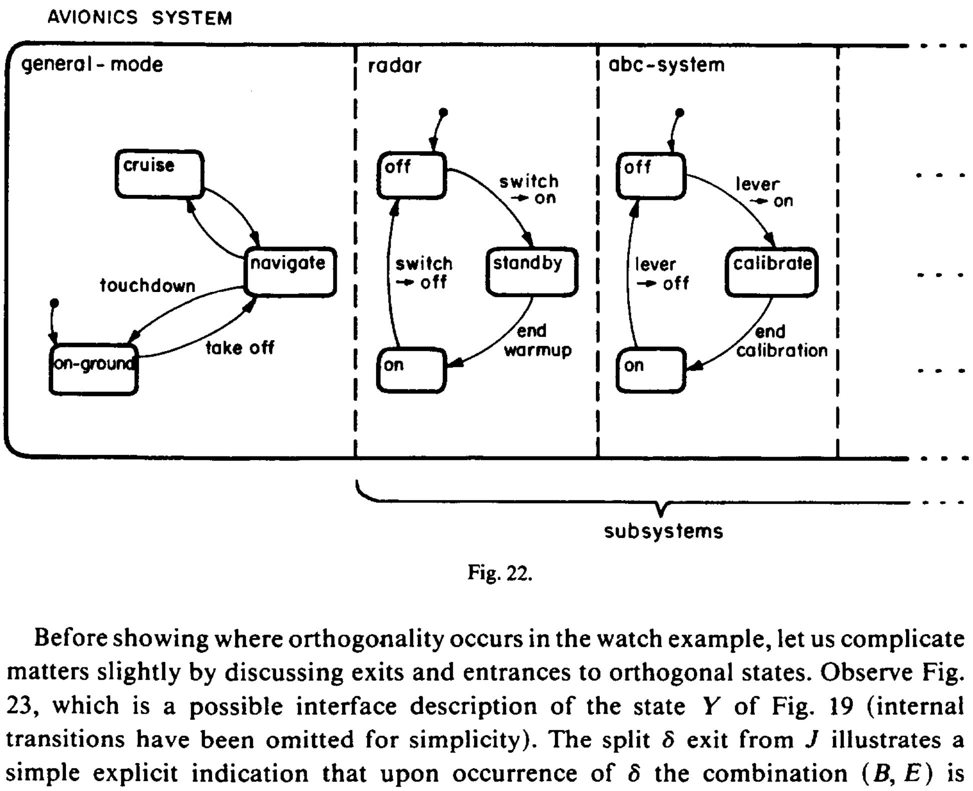


Fig. 22.

Before showing where orthogonality occurs in the watch example, let us complicate matters slightly by discussing exits and entrances to orthogonal states. Observe Fig. 23, which is a possible interface description of the state Y of Fig. 19 (internal transitions have been omitted for simplicity). The split δ exit from J illustrates a simple explicit indication that upon occurrence of δ the combination (B, E) is entered. An α-event in K causes the system to enter (C, F); C by the arrow and F by default, and a ν-event from J causes entrance to the default (B, F). A β-event at L causes entrance to the combination of C with the most recently visited state in D, and an ω-event in combination (B, G) causes transfer to K. An alternative to

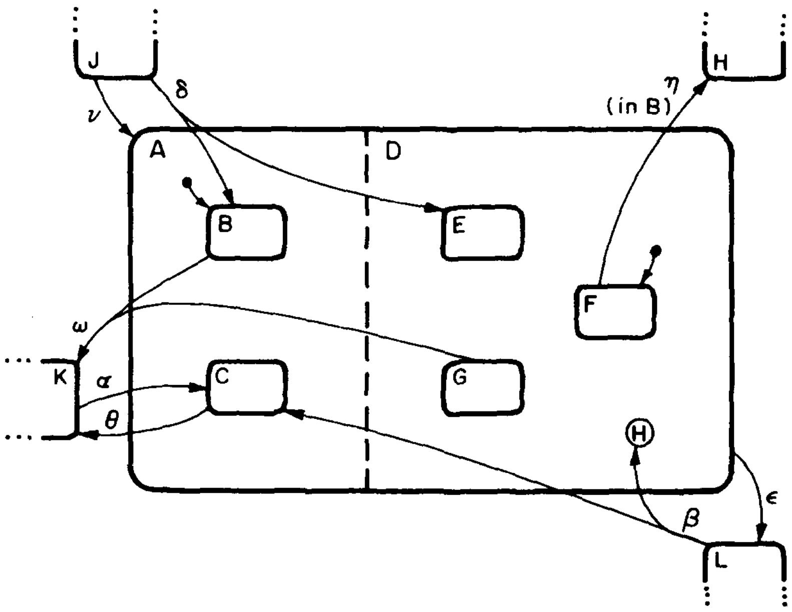


Fig. 23.

this last possibility is to replace one of the outgoing branches of the merging arrows by a condition, as in the η-arrow from F, applicable actually only in (B, F). The θ-arrow from C, on the other hand, is a typical 'exit independently' transition: it states that the product state $A \times D$ is left and K entered, depending only on the fact that the A component is actually C. The most general kind of exit is the ε-arrow causing control to leave $A \times D$ unconditionally. It is perhaps worth following up Fig. 23 with its zoom-out, in which stubbed entrance arrows are used when the entrance crosses the state boundary (that is, when it does not rely on the default). See Fig. 24.

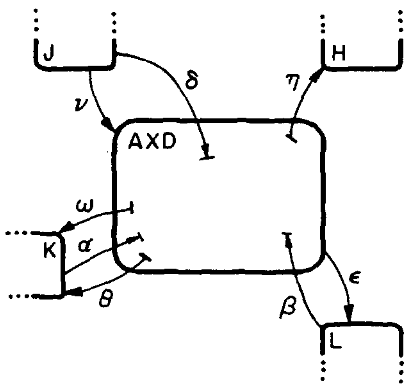


Fig. 24.

Figure 25 contains a refinement of the *stopwatch* display state of Figs. 9 and 13 using orthogonality, and should be self-explanatory. In it, *regular* and *lap* are two kinds of displays and *zero* is the special state in which the stopwatch is off but in its initial position. This description could have been the outcome of a separate person or group dedicated to specifying the behavior of the stopwatch.

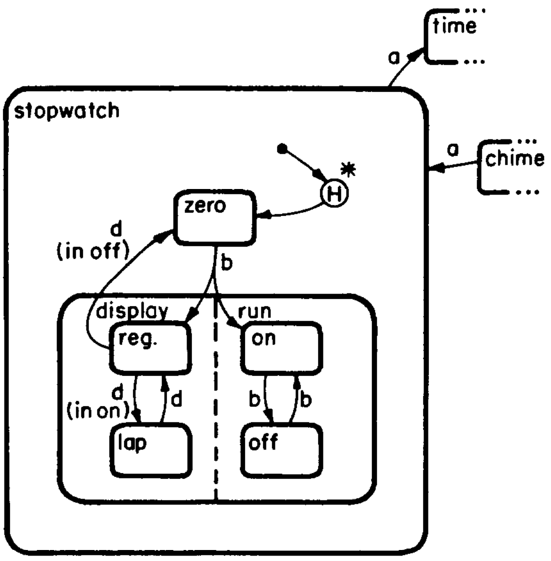


Fig. 25.

Orthogonality appears in the Citizen watch on the high levels too. One might start a top-down behavioral specification of the watch, accounting for battery insertion and removal, as in Fig. 26, and then decide (see Fig. 27) that the *alive* state is to consist of six orthogonal components: a main component containing displays and alarm-beep modes, one component for the enabled/disabled status of each of the alarms and the chime (the latter containing the chime-beeping state too), one for the power status, and one for the light. The resulting levels of the statechart are given in Fig. 28, where the *main* component of the *alive* state has been described in detail earlier.

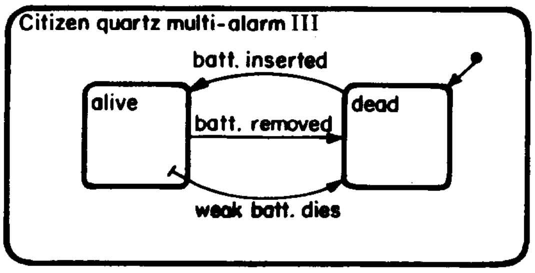


Fig. 26.

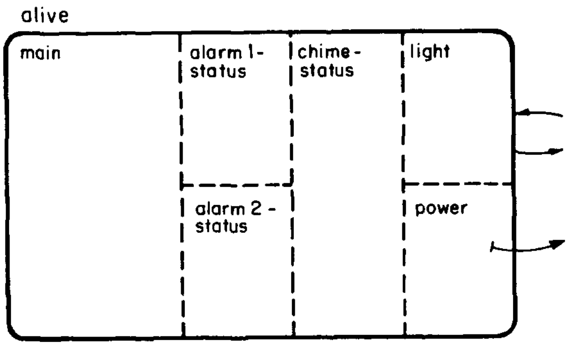


Fig. 27.

Although the two-state *light* component looks rather innocent, it is actually quite subtle because of its scope. Orthogonality of the component on this level prescribes that depressing *b*, say, in the *update* state of Figs. 13 and 14 will simultaneously turn on the light and cause the system to exit the updating state. We shall see even more interesting combinations later.

Note that the *disabled*/*enabled* status of the alarms and the chime are directly linked to the corresponding *off*/*on* substates of *displays*; this is one way of modeling a display change and its 'hidden' consequences. Of course, there are other ways, and constructing statecharts, like writing programs, should encourage many possibilities, depending, among other things, also on personal style.

At this point, the situation permits the conditions *P*1 and *P*2 for the beeping alarms to be made more precise. For example "*alarm*1 enabled" is to be replaced by "in *alarm*1-*status.enabled*".

Note that our previous H-entrances (e.g. in Figs. 13 and 25) can no longer be interpreted without reservation, as "enter most recently visited". Now that we have catered for battery removal and death within the specification, these attain a more sophisticated meaning whereby history is to be 'forgotten' if *dead* has been entered in the meantime. To deal with this more complex historical criterion we use the special actions *clear-history*(*state*) and *clear-history*(*state**), which cause the forgetting of recently visited states on the first level, or all levels, respectively, of *state*. (The combination *clear-history* is abbreviated as *clh*.) Once forgotten, H-entrances do not apply, and defaults are employed. We have chosen to attach the action to the transitions entering *dead*, to the right of a "/", as discussed in Section 5 below.

Here are two final features of the Citizen watch that seem to nicely illustrate the painless way certain kinds of changes can be made to statecharts. We can think of these features as being handed down to the team of specifiers/designers from above at a late stage in the process of specification. The first is a beep-testing feature, and the second involves a 2-minute automatic return to *time* from all displays other than the stopwatch, on condition that no button has been depressed in the interim.

As to the beeper test, depressing both *b* and *d* causes a healthy beeper to beep. Clearly, this test, modeled in detached form in Fig. 29 (assuming, for simplicity,

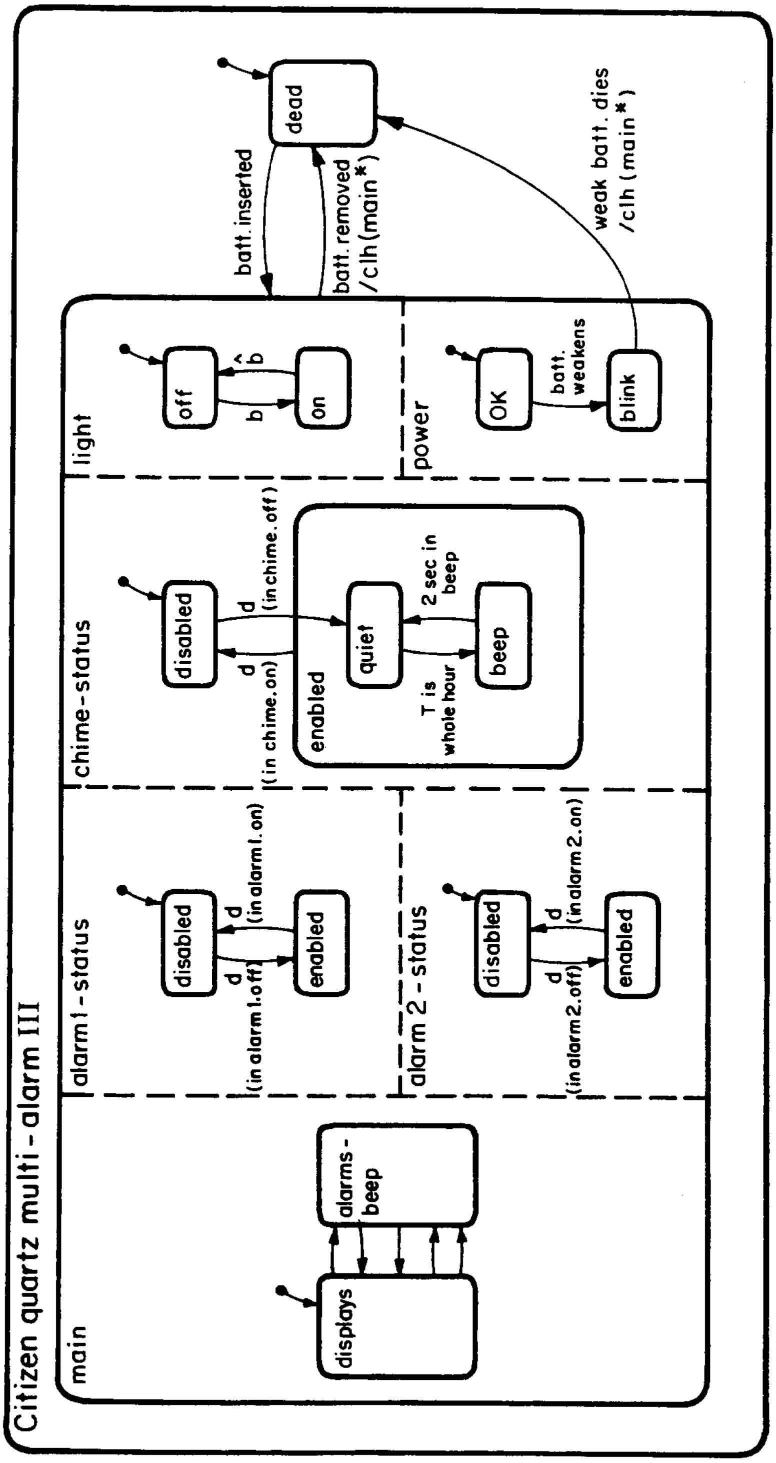

Citizen quartz multi-alarm III
dead
batt. inserted
batt. removed /clh (main*)
weak batt. dies /clh (main*)
light
off
on
b
b
power
OK
blink
batt. weakens
chime-status
disabled
enabled
quiet
beep
d (in chime.off)
d (in chime.on)
2 sec in beep
T is whole hour
alarm1-status
disabled
enabled
d (in alarm1.on)
d (in alarm1.off)
alarm 2-status
disabled
enabled
d (in alarm2.on)
d (in alarm2.off)
main
displays
alarms-beep

Fig. 28.

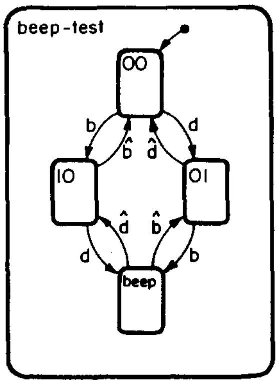


Fig. 29.

that *b* and *d* cannot be depressed simultaneously), is applicable in some states of the watch and not in others. For example, it is obvious that it should not be relevant in the *dead* state and probably not in the other beeping states either. The claim now is that a behavioral designer of the watch can attach this piece of the description as an orthogonal component to whichever portion desired. One reasonable choice might be the *time* display mode, as in Fig. 30. The significance of this kind of

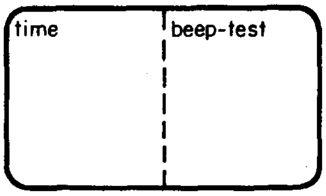


Fig. 30.

decision should be clear: the test will work only when the system is in *time*, in this case. As it happens, the beep-test of the author's actual watch is applicable precisely in both the *time* and *date* states and, curiously enough, also throughout the general *update* sequence. It is not applicable (though it took the author quite a while, and required some strenuous finger-twisting, to discover the fact) in the 2-second *wait* period. A new box was therefore drawn around the relevant portions, their common property being the applicability of the beep-test, and the test itself was attached properly to it. See Fig. 31. We might add here that Citizen's documentation of the watch lists both the *light* feature and the *beep-test* in the same way: if you press so-and-so this-and-that will happen; no indication of the scope is provided, despite the major difference between the two.

As to the second feature, here life is even easier: the addition involves merely drawing another box around the relevant displays, with the appropriate event and condition on the outgoing arrow. Figure 31 (see fold-out) contains the full statechart of the watch so far.

It is instructive to work through certain linear (or branching) sequences of events, called *scenarios*, and observe their effects as prescribed by the statechart. For example, assume the system is in the *mon* updating state (see Fig. 31), and that, for some reason, the user has an urge to try out his beeper. Suppose he depresses *d* and *b* in that order without releasing either of them. The *regular* component says we shall end up in *time* (with the month updated one step, although this fact does not show up in Fig. 31), the *beep-test* component says we shall end up in the *beep* state, and, finally, the *light* component (orthogonal on a higher level to the other two) says we shall end up with the light *on*. This is in fact quite the case; we shall end up in *time*, one month ahead, with the beeper beeping and the light on!

As a small example of a subtle anomaly of the Citizen watch, it so happens that the continuous beeping during a beep test stops upon depressing *a* and resumes upon letting go. Figure 32 shows a refinement of the *beep* state capturing this fact, but it is not included in Fig. 31.

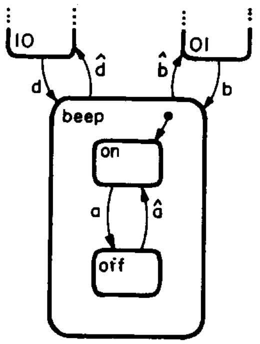
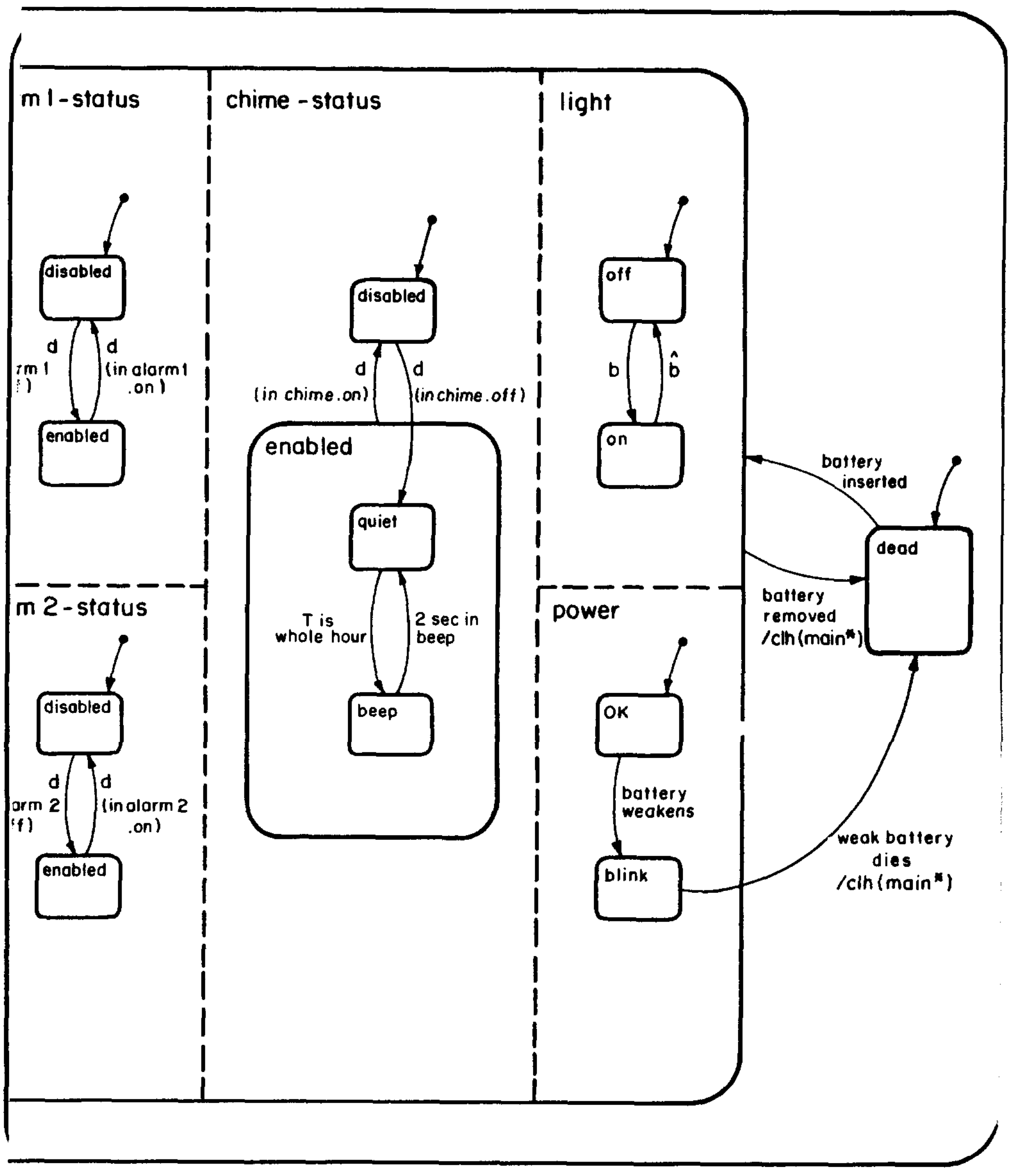


Fig. 32.

It should be noted that neither the time elapsing activity itself nor the internal values of the time, date, and alarm settings are included in Fig. 31. These parts can be modelled appropriately, the first as an additional component orthogonal to the *alive* state and the latter in the form of one extra level of states within the three *update* states (cf. Figs. 38 and 39 and the accompanying text in Section 6.1). Alternatively, one can regard these as involving variables that change values, and postpone their specification to the *activity* part of the system; see Section 5. Also, no mention is made of the contents of displays, though some of the state names are suggestive; this also is taken up in Section 5.

4. Additional statechart features

Here are a number of features that are part of the basic statechart formalism, but did not show up in the watch example.



m 1-status
disabled
d
rm 1
f)
d
(in alarm1
.on)
enabled
m 2-status
disabled
d
arm 2
f)
d
(in alarm 2
.on)
enabled
chime-status
disabled
d
(in chime.on)
d
(in chime.off)
enabled
quiet
T is
whole hour
2 sec in
beep
beep
light
off
b
b̂
on
power
OK
battery
weakens
blink
battery
inserted
dead
battery
removed
/clh(main*)
weak battery
dies
/clh(main*)

Fig. 31

4.1. Condition and selection entrances

There are two circled connectives, similar to the **H**, for abbreviating more complicated entrances to substates of a superstate than a simple direct arrow. The first is a **C**, for *conditional*, and is illustrated, for a particularly simple case, in Fig. 33. Here 33(b) can replace 33(a). If the actual conditions and/or the topology of the arrows are too complex, one can omit the details from the chart and use the simple incomplete form of Fig. 33(c). The user will have to supply the full details separately, and a computerized support system for statecharts would be able to show 33(c) but would enrich it to 33(b) on request.

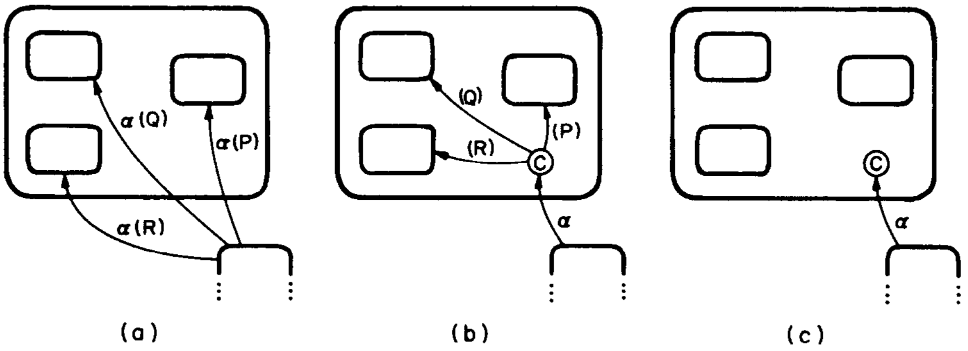


Fig. 33.

The second connective is **S**, for *selection*. Selection occurs when the state to be entered is determined in a simple one-one fashion by the 'value' of a generic event, so that the event is actually the selection of one of a number of clearly defined options and the specifier has chosen to model those options as states. Consider, say, a computerized storage system connected with a display, around which there are four keys marked *type*, *name*, *qty* and *place*, pertaining, respectively, to the type of objects stored, their code name, quantity and physical placement. A typical updating procedure would allow the user to select an option to be updated by pressing the appropriate key and then updating it, possibly repeatedly. Figure 34(a) models the situation, and Fig. 34(b) shows how the selection option (and the unified history entrance) simplifies things. The user will have to specify the event *selection* as being the disjunction of the four lower-level events, as well as the association of each of them with the appropriate state, so that the S-entrance becomes well-defined.

4.2. Delays and timeouts

The present formalism treats time restrictions using implicit timers, as per the exits from *date* and *alarms-beep* in Fig. 31. Formally, this is done using the event

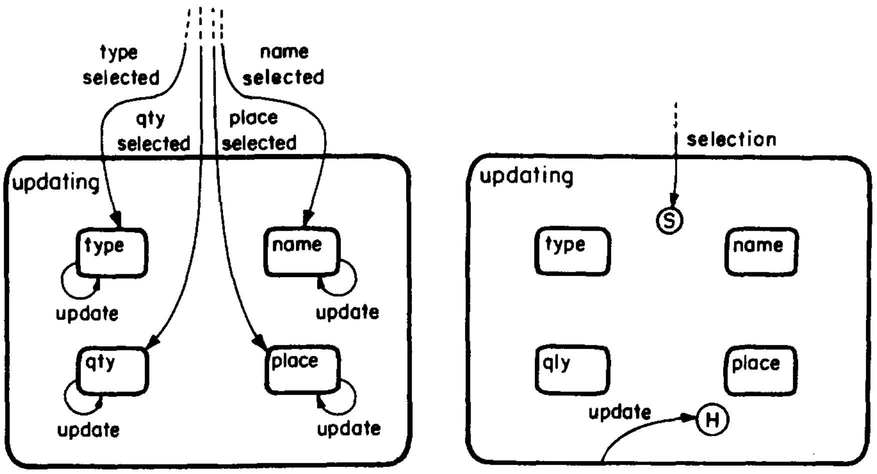


Fig. 34.

expression *timeout*(*event*, *number*), which represents the event that occurs precisely when the specified *number* of time units have elapsed from the occurrence of the specified *event*. The aforementioned exit from *date*, for example, would formally appear as *timeout*(entered *date*, *120*).

However, the need to limit the system's lingering in a state is something that occurs repeatedly in the specification of real systems, especially real-time systems, and it seems worthwhile to supply a special notation for this, a graphical one if possible, and one that makes it obvious that this is a property of the state. Figure 35 shows the notation we use, including a squiggle to indicate that the state comes with a bound, an indication of the bound itself, and a generic event that stands for *timeout*(entered *state*, *bound*), where the state is the source of the transition, and the bound is its specified bound.

Actually, we allow lower bounds too. In general, the syntax of the specification attached to a squiggle is $\Delta t_1 < \Delta t_2$, providing lower and upper bounds on the time in a state. Either one of the Δt_i can be omitted, as is done in the state of Fig. 35.

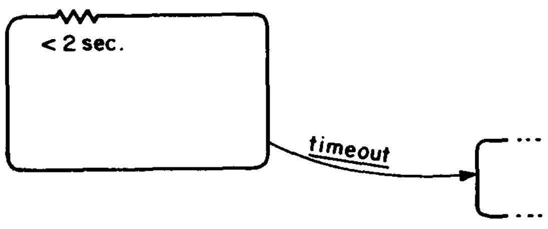


Fig. 35.

The significance of the lower bound is that if they are to cause exits, events do not apply in the state until the lower bound is reached.

4.3. Unclustering

The purpose of this subsection is simply to call the reader's attention to the possibility (useful in both manual and computerized use) of laying out parts of the statechart not within but outside of their natural neighborhood. This conventional notation for hierarchical description has the advantages of keeping the neighborhood small yet the parts of interest large. See Fig. 36. It is a necessary option when the system under description is large.

We are well aware of the fact that taking this to the extreme yields and/or trees, thus undermining our basic area-dominated graphical philosophy. However, adopting the idea sparingly can be desirable.

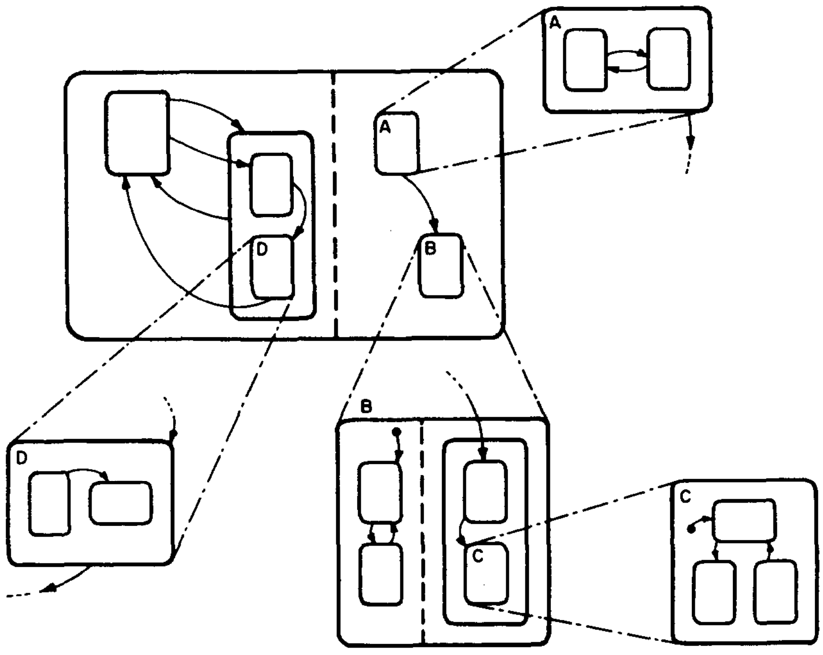


Fig. 36.

5. Actions and activities

There is actually almost nothing in the statecharts presented above, including the extensive Fig. 31, connecting the states, transitions and other objects appearing therein with the 'real' watch! Apart from our convention regarding the use of the letters *a*, *b*, *c* and *d* for the depressing of the 'real' buttons, we only used suggestive words like *displays*, *beeps*, *on* and *off*. Who says that the watch beeps or displays at all? Who says it even keeps the time?

Obviously, what 'pure' statecharts represent is the *control* part of the system, which is responsible for making the time-dependent decisions that influence the system's entire behavior. However, so far, the reactivity part was expressed only by the system changing its internal state-configuration in response to incoming or sensed events and conditions. How else does it respond? How does it influence other components of the system? It seems obvious that what is missing is the ability of statecharts to *generate* events and to change the value of conditions. These can be expressed by the notation "$\ldots/S$" that can be attached to the label of a transition, where S is an *action* carried out by the system. We shall reserve the word *action* for split-second happenings, instantaneous occurrences that take ideally zero time. For us, sending a signal takes zero time, as does a conventional assignment statement. Many actions are output events of the whole system, like "swallow ticket" and "send current balance to display". Not all, however. If we want a transition labelled α in one component of the statechart to trigger another transition in an orthogonal component, without necessarily having any immediate external effects, we can simply label the first α/S and the second S. Upon sensing α the first transition will be taken and the action S will be carried out, generating S as an event that can be sensed elsewhere; and indeed, in the other component the S transition will be taken instantaneously. Thus, events and actions are closely related: the distinction is almost precisely that drawn between input and output events elsewhere.

However, actions are not enough. We need *activities*, which are to actions what conditions are to events. An activity always takes a nonzero amount of time, like beeping, displaying, or executing lengthy computations. Thus, activities are durable—they take some time—whereas actions are instantaneous. In order to enable statecharts to control activities too, we need two special kinds of actions to start and stop activities. Accordingly, with each activity X we associate two special new actions, *start*(X) and *stop*(X), and a new condition *active*(X), all with the obvious meanings. To start the beeping upon entrance to, say, the *alarms-beep* state, one can attach the action *start*(*beeping*) to the entering transitions (or, alternatively, to the entrance to the state), where *beeping* is the required activity.

Obviously, in order to describe such 'real' actions and activities one has to assume some physical and functional description of the system, providing, say, a hierarchical decomposition into subsystems and the functions and activities they support. This description should also identify the external input and output ports and their associated signals. Statecharts can then be used to control these internal activities. Although we are aware of the fact that achieving such a functional decomposition is by no means a trivial matter, we assume that this kind of description is given or can be produced using an existing method. What we want to emphasize here is our conviction that it is most beneficial to directly link these aspects of the system's overall description to the underlying statechart.

To make the idea more precise, a statechart can contain instructions to carry out actions and activities by a simple extension of the ideas in the so-called Mealy and Moore-automata (see e.g. [17]). In the one, actions are allowed along transitions, and in the other they are allowed in states, usually meaning that the action is to be carried out upon entering the state in question. We shall also allow the association of actions with the exit from a state, and will also allow to specify that an activity will be carried out continuously *throughout* the system's being in the state. In other words, saying that the activity X is carried out throughout state A is just like saying that the action $start(X)$ is carried out upon entering A, and $stop(X)$ upon leaving A. We thus enrich the transition labelling to be of the form $\alpha(P)/S$, where α is the event triggering the transition, P the condition that guards the transition from being taken when it is false, and S the *action* (or *output* in automata-theoretic terms) to be carried out upon transition. (We can actually allow Boolean combinations in each component, but we shall not get into a detailed syntax here.)

In Fig. 37, for example, if the system is in (C, D) and event α occurs, the new state-configuration will be (B, D), but the two actions S and V will be carried out, simultaneously, upon entrance. Notice how concurrency of actions is induced by the nested structure of states. Action S will obviously not take place if a transition from state B to F occurs. Now, if event γ occurs, the system exits B and A, causing actions W and T to take place, but also causing the 'internal' action β to be carried out, which in turn, sensed as an event, causes a transition in the orthogonal component. Thus, β is an output event in the left-hand component and an input event in the right-hand one. The new state-configuration will therefore be (C, E) and action U will, therefore, also be taken, and at the very same time. Defining the formal semantics of these action-enriched statecharts is quite a delicate matter. See Section 7.

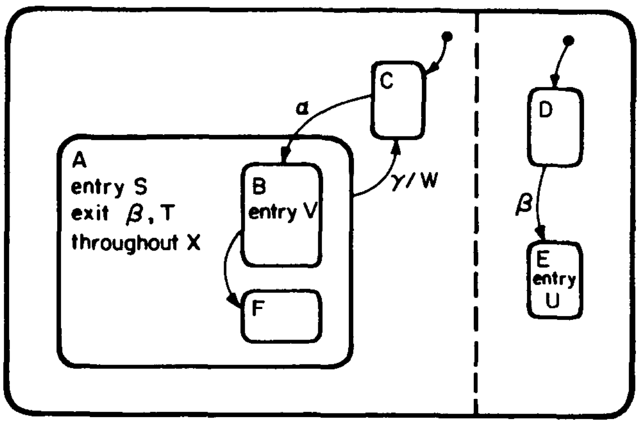


Fig. 37.

The reader should now be able to use our informal descriptions in order to formulate most of the actions and activities that the watch carries out and attach them to their rightful places in the statecharts given above. For example, the appropriately identified action of starting to continuously display the status and internal setting of the first alarm will obviously be associated with the entrance to state *alarm*1, and terminating the display will be associated with several of the exits thereof. (Why not simply associate the durable activity with being in the *alarm*1 state?) Similarly, the activity *start-half-second-beep* will be associated with the two transitions within the *stopwatch.run* state component, since that is precisely what happens when the stopwatch is turned on and off. Clearly, all these actions and activities must also be associated with the various physical components, such as displays, beepers, internal processors and timers.

We should also reemphasize that we do not deal here with the problem of finding the right formalism for specifying the activities themselves. One possibility that comes to mind in case the activities are sequential in nature, is simply to use a conventional programming language. If the activities themselves can involve reactive behavior, we may use separate internal statecharts to control *their* internal behavior. This, in effect, results in a hierarchy of activities, with a statechart controlling the reactive behavior of each, in terms of its own input and output events and its subactivities. The STATEMATE1 system, mentioned in Section 9, contains a graphical formalism for activities, *activity-charts*, and a corresponding one for the physical/structural aspects of the system under description, *module-charts*. Both were designed to fit in nicely with the behavioral aspects, which are described using statecharts, but they are beyond the scope of this paper. See [1].

6. Possible extensions to the formalism

This section contains preliminary ideas on a number of more advanced features that are being currently investigated as possible additions to the basic statechart formalism. The reader should take this into account. For most we have neither a final recommendation for a syntax, nor a satisfactory formal semantics. They are brought here because, in the author's opinion, they represent significant potential strengthenings of the statechart formalism as a tool for specifying real systems. In one way or another, fragments of all of these have been used manually in the experimental projects mentioned later.

6.1. Parameterized states

In many cases (e.g., *alarm*1 and *alarm*2 of Fig. 31) different states have identical internal structure. Some of the most common ones are those situations that are best viewed as a single state with a parameter.

Consider a refinement of state *update.*1*min*, in which the updating by *d* is to be captured. An obvious candidate is that of Fig. 38, in which the condition tests the current time *T*, and the unspecified event *α* denotes *T* crossing a minute borderline. We would like to economize by parameterizing the states, and could choose a notation such as that of Fig. 39. This is actually an example of a 'parameterized-*or*', and one can think of cases in which a 'parameterized-*and*' would be helpful. An example that comes to mind would be the specification of a thousand individual telephones connected through a central network. A portion of the relevant statechart could be given as in Fig. 40. In both cases, however, one should keep in mind that often an underlying programming language with its variables and rich data structures is the best tool for specifying complex kinds of parameterization.

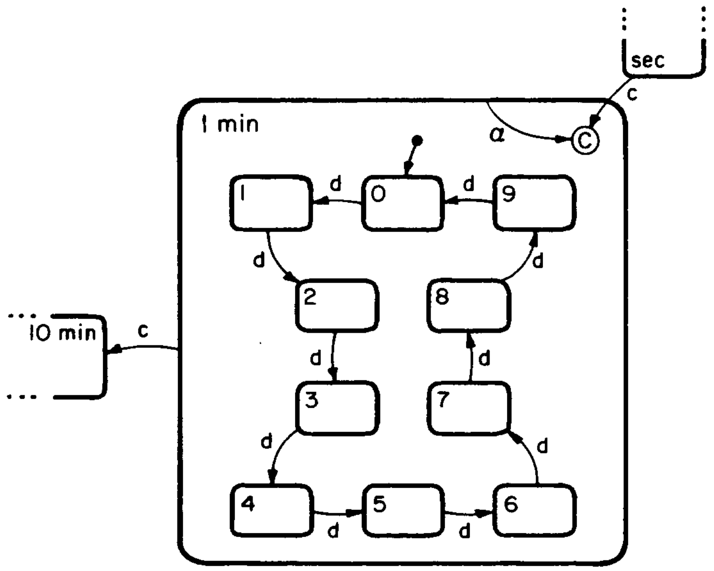


Fig. 38.

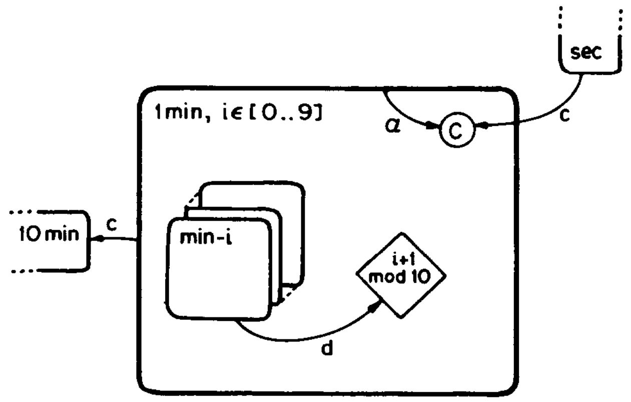


Fig. 39.

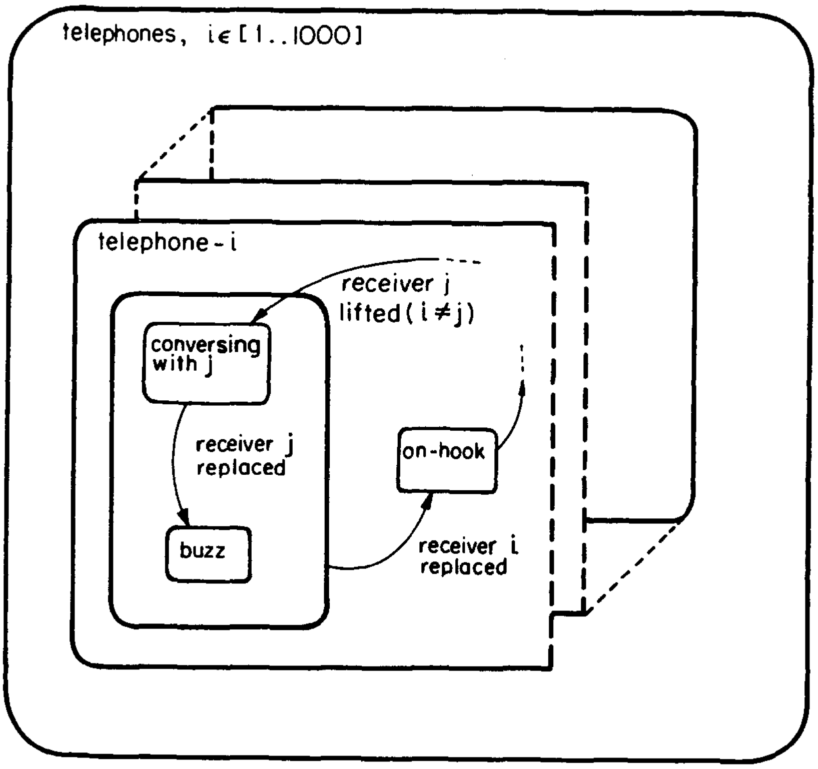


Fig. 40.

6.2. *Overlapping states*

The interrelationship between the states in all the statecharts presented thus far is that of an AND/OR tree; actually an AND/XOR tree. However, there is absolutely no deep reason for this, and statecharts need by no means be entirely tree-like. While the human mind seems to perform well on tree-like hierarchical objects, we definitely do not rule out clustering which is more complex. For example, Fig. 41 shows a situation in which state *C* has two parents. The reason for doing this might

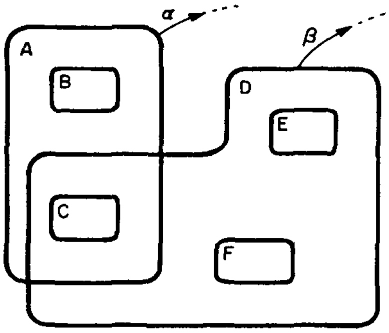


Fig. 41.

be conceptual similarities between the involved states, or merely the pragmatic desire to economize when describing joint exits such as the two transitions appearing in the figure. What is really happening is that states A and D are now related by OR, not XOR. Of course, too much of this kind of overlapping will burden the specification, with incomprehensibility possibly outweighing economy of description. In such a case, one could resort to two copies of C.

Overlapping states, however, are beneficial not only in turning XOR's into OR's. Consider a state A, with substates, internal transitions, etc., that 'lives alone' under some circumstances, but is joined in 'orthogonal marriage' with a state B under others. Figures 42 and 43 show two ways of describing this situation. In the first A

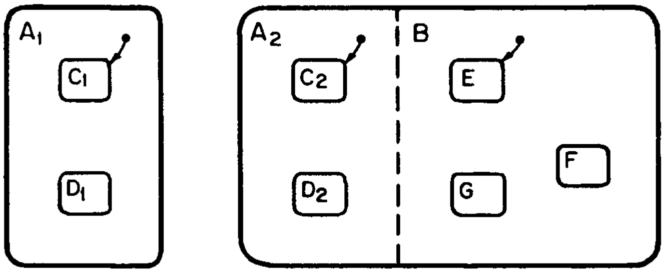


Fig. 42.

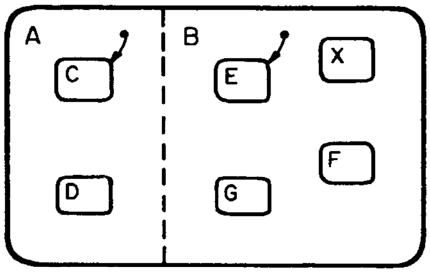


Fig. 43.

appears twice—obviously not an entirely desirable situation, especially if A is large and complex. In the other B contains a special new state that says "this is not really a B state at all"—again, a rather artificial solution. Our recommendation is to use overlapping states, as in Fig. 44. All the transitions appearing there are quite unambiguous, as the reader can verify. In particular, the β-entrance is clearly an entrance to the product of A_1 and B (which, by the defaults is to the pair (C, E)), not to A_2 alone, which is what the α does. The ε-exit leaves D regardless of whether it is matched with a state from B or not, whereas the ω-exit has the condition about being in B attached.

One problem with this use of overlapping is in the ambiguity of entrances to A's substates, like D. It is not at all clear what a transition entering D is supposed to

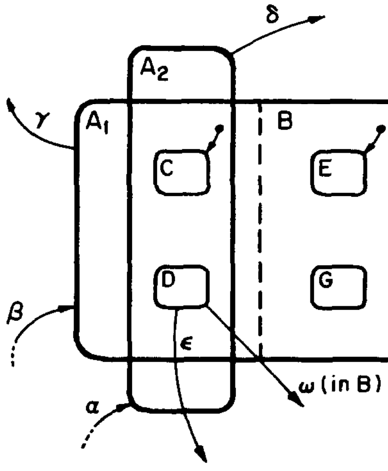


Fig. 44.

mean. We suggest refining the elementary graphical notation of arrows crossing state borderlines, so that one can waive the pleasure of entering a state. Figure 45 shows two entrances to D, the γ causing entrance to (D, E) and the δ causing entrance to D alone.

Overlapping states can be used economically to describe a variety of synchronization primitives, and to reflect many natural situations in complex systems. Several examples appear in [6]. However, as discussed further in Section 7, overlapping states cause semantical problems, especially when the overlapping involves orthogonal components. A first suggestion for a syntax and semantics for overlapping states appears in [20]. We are, however, fully convinced that the addition of an appropriate version of overlapping states to the basic formalism will greatly enhance its potential.

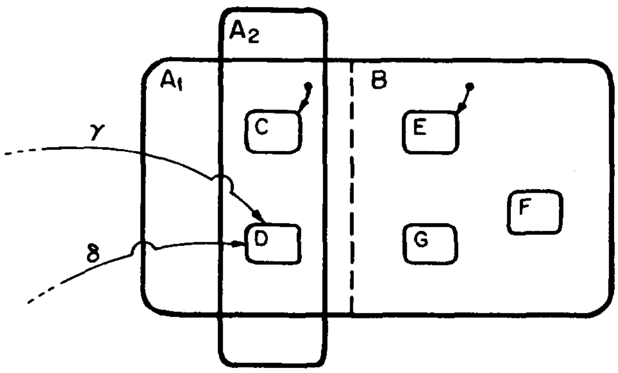


Fig. 45.

6.3. Incorporating temporal logic

Temporal logic (TL) is used quite extensively in the specification of concurrent programs; see [24, 27]. We are of the opinion that TL can be used beneficially together with statecharts in more than one way. It is possible to specify ahead of time many kinds of global constraints in TL, such as eventualities, absence from deadlock, and global timing constraints. Then one would carry out a statechart specification of the system and attempt to *verify* that the statechart-based description satisfies the TL clauses. This approach would seem to require that one develop proof methods using statecharts as the description language and temporal logic as the assertion language. One could attempt to extend methods developed for verifying conventional state-machines against TL formulas, such as those in [5].

Dually, another possibility would be to *synthesize* a 'good' statechart from TL specifications. We feel that this represents an important research topic, since many of the people that were involved in the avionics project mentioned in Section 9 displayed a linear-like scenario-based mode of thinking. They were able to state many desirable scenarios, such as firing a missile or updating the aircraft's location, in precise detail, describing things that they wanted the system to do eventually. Such scenarios are easily described using TL, and it would seem beneficial to be able to derive a reasonable statechart description from a large set of scenarios given in TL. (Zave's sequence diagrams [33] are tailored for scenario-based descriptions too; see Section 8.)

Finally, in the statecharts themselves there are conditions that a specifier might want to use that refer to the system's past behavior. We have supplied the simple-minded history entrance for that purpose, but it is obvious that one would like to be able to specify more complex things too. For example, instead of shutting off the history using the *clear-history* action, one could do away with the history entrance in, say, the *chime* state of Fig. 31, and enter *chime* using a conditional entrance with the following condition for the *on* state, and a similar one for the *off* state, and with an 'otherwise' clause leading to *off*, the default. The formula is taken from the past fragment of temporal logic:

$$(\neg(\text{in } \mathit{chime}) \wedge \neg(\text{in } \mathit{dead}))\ \mathbf{since}\ (\text{in } \mathit{chime.on}).$$

It states that the transition is to be taken if sometime in the past we were in *chime.on*, but have not been in *chime* or in *dead* since.

6.4. Recursive and probabilistic statecharts

Although we have seen little need arise in practice, it is worth noting that one can introduce *recursion* (or, in automata-theoretic terms, *context-freeness*) into statecharts, just as augmented-transition-networks [32] (see also [31]) introduce it into conventional state diagrams. This can be done by specifying the name of a separate statechart, the present one being a special case, on a transition arrow. Such an extension would require a notation for terminal states, say a double state-boundary, not only for initial ones (defaults).

One might also want to somehow add *probabilism* to statecharts. One way of doing this would be to allow nondeterminism (say, in the forms outlawed in the discussion around Fig. 18), and then to specify a bias on the coin to be tossed when it arises. This, in fact, would really be tantamount to considering 'Markov-charts', which would be to classical Markov chains what statecharts are to the classical transition diagrams of finite-state automata.

7. Semantics of statecharts

The statechart formalism turns out to be quite a challenge when it comes to providing formal semantics, much more so than simple finite-state automata. The main difficulty is not in the depth of states or the orthogonality constructs themselves; these (as claimed in [12]) can be dealt with by a translation into ordinary automata, although even there delicate problems arise. Consider Fig. 46. The semantics must be able to conclude that when α occurs in (A, B), the system ends up in (C, D), but that one more α leads not to (E, F) but, nondeterministically, either to (E, H) or to (G, F). The more difficult problems arise with the introduction of events and conditions that are generated within the statechart itself, and are sensed in orthogonal components.

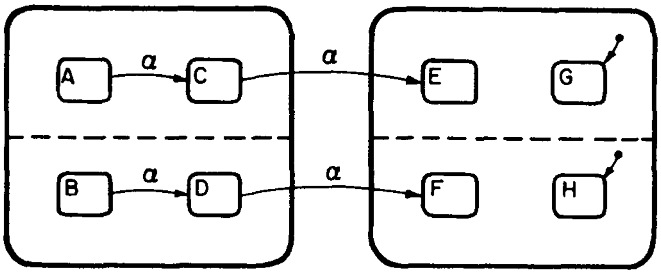


Fig. 46.

To a semanticist this is obvious. The problems with concurrency stem from the cooperation mechanism: shared variables and their access, communication over channels, handshake rendezvous', etc. In this respect, statecharts employ a *broadcast* communication mechanism. One part generates an event (by an action that is placed, say, to the right of a "/" on a transition), and all other parts sense it, acting in response if so specified. Clearly, upon sensing such an event, another component might generate a new event, causing yet others to be generated. Cycles, like that of Fig. 47, have to be dealt with by the semantics, presumably rendering them undefined.

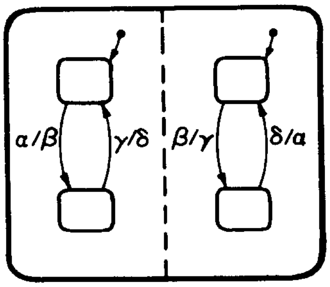


Fig. 47.

Not only the events generated by actions cause problems, but also those generated by the very dynamics of a statechart. We allow statecharts to refer to the condition "in *state*", and to the events that signal changes in that condition, "entered *state*" and "left *state*". At first sight these seem to be very well defined, the first as a time span (durable happening) and the others as instants in time. However, Fig. 48 illustrates a situation in which it is not quite clear what the outcome should be. Starting in (A, C, E), if α occurs, should we end up in (B, C, E), (B, D, E), (B, C, F), or (B, D, F)? Our intuition says, perhaps, (B, C, F), but a formal semantics must supply all the answers.

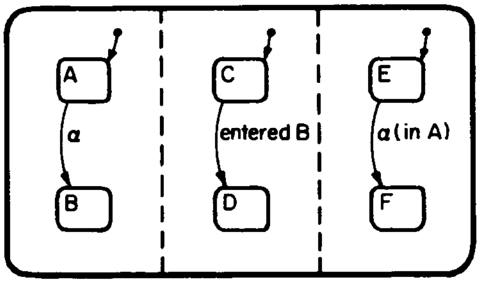


Fig. 48.

Finally, the desire to allow simultaneous events can cause problems, especially if we wish to *negate* events in order to specify that they have not occurred simultaneously. In Fig. 49, for example, the meaning of the two α-exits seems obvious, but do we end up in D if γ occurs in A?

Combining generated events, internal events and conditions, and simultaneous events and their negations causes extremely delicate problems, whose solution is far beyond the scope of this (already quite lengthy) paper. We refer the reader to [15], in which both a formal syntax and a version of formal operational semantics is given for the language, including these features. We shall only provide some hints here as to the basic notions and ideas used therein.

One first notices that any two states in a statechart can be related in one of three ways: exclusive, orthogonal or ancestoral. The best way to see this is to consider

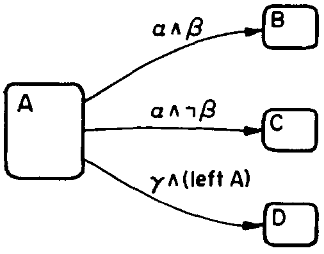


Fig. 49.

the AND/OR state tree. A and B are *exclusive* (respectively *orthogonal*) if they are not on the same path and their least common ancestor is an OR-state (resp., an AND-state), and they are ancestoral if they are on the same path. Using these definitions one defines a legal (partial) *configuration*, to be essentially a set of mutually orthogonal states, and a *full configuration* to be a maximal such set. A *basic* configuration is one all of whose states are atomic (no offspring).

The statechart is given by a state tree together with a set of *transitions*, each of which is an ordered pair of configurations (a *source* and a *target*) labelled with some legal $\alpha(P)/S$ combination. Defaults and history entrances are appropriately included. The heart of the semantics of [15] is the definition of a function *nextstep*(X, C, E), which, for a full basic configuration X, a set of (external) conditions C, and a set of simultaneous (external) events E, provides the set of next possible full basic configurations. (If there are more than one, the system is nondeterministic—an error in the pragmatic sense of Fig. 18 and its discussion, but the standard general case in the framework of the formal semantics.)

The semantics assumes that each step represents some time interval in which new internal events may be generated and responded to. The main steps in defining *nextstep*(X, C, E) are the following. First, all *relevant* transitions are selected. This involves analyzing their *trigger* (the part of their label to the left of the "/") to see whether indeed it is triggered by E, and their source configuration to see whether it is ancestoral to a subconfiguration of X. For each such transition, the semantics takes off separately, analyzing the consequences of applying it (such as the consequences of the action part of its label), and finding new transitions that become relevant as a result. This process is continued until it closes off, reaching a stable and consistent situation. Then the configuration X is replaced by an appropriate X', which is obtained by applying the overall effect just determined to the appropriate ancestoral configuration, and then 'diving' downwards, using default entrances to reach a full basic configuration. The set of all such X' is the final result of *nextstep*. Of course, the process of computing the stable and consistent situation is the tricky part, and its details can be found in [15].

History entrances cause no essential problems, and incorporating them into the semantics entails remembering one state for each state whose area includes an H-connector, and one basic configuration if it includes an **H***-connector. The C's and S's are merely abbreviations of other constructs.

As mentioned earlier, for the advanced features of Section 6 (with the exception of overlapping states; see [20]) we do not even have a satisfactory syntax, let alone a formal semantics.

8. Related work

The state/event approach, in the form of finite-state machines or state transition diagrams, has been suggested numerous times for system specification. A small sample of references includes [19, 23], recommending state machines for the user interface of interactive software, [8], recommending them for the specification of data-processing systems, [7], for hardware system description, [28, 29], for the specification of communication protocols, and [9] for computer-aided instruction.

Many of these papers, and many later ones that have turned to other approaches, have identified the problems associated with a naive use of conventional state-machines or state-diagrams, especially the exponential blow-up in the number of states. A good example of how these problems can cause one to simply give up on the use of state-machines for system description is to be found in [21]. In the table on p. 337 therein the authors claim that state-transition diagrams are hard to read, difficult to draw and change, non-user-friendly, not good for stepwise refinement, cannot be decomposed into executable code, and are no good for large complex specifications.

The need for separate state-components is identified in [21], where the authors recommend using separate diagrams related by 'linkages' (pp. 228–229). However, as in so many of the 'diagramming techniques' of [21], linkages too are not given any semantics, except to point out some subjective correspondence between linked objects.

There have been a number of suggestions for formalisms involving *communicating* finite-state machines. One of the main propositions of this kind is the CCITT language for communication protocols, SDL. See, for example, [4]. The basic SDL formalism enables one-level concurrency, consisting of a number of conventional machines communicating by a broadcast communication mechanism. Input messages are queued, output requests are peeled off the queue, and much of the work in defining the language involves solving problems regarding the implementation of this communication mechanism. No workable way of introducing hierarchy or depth has yet been put forward by the SDL group, and, as a result, even relatively simple protocols have quite complex-looking SDL descriptions.

Augmented transition networks (ATN's, see [31, 32]) provide for hierarchical state/event descriptions by allowing a transition in one machine to be labelled by the name of another machine. When the transition is reached, the 'called' state-machine is started, and the original transition is returned to when this machine reaches a final state. This leads, of course, also to recursive state-machines (see Section 6.4). This approach is definitely one way of introducing hierarchy into state machines, but might not always be the preferable one. In particular, there is no way to introduce interrupt-driven behavior by relating events to high-level collections of states, as does a transition leaving a superstate in a statechart, or to specify event-based entries or exits that cut across levels of detail. The hierarchy of augmented transition networks, therefore, is that of *subroutines*, and is therefore a much more *process*-driven approach than an event-driven one. In addition, ATN's do not address the state blow-up problem at all, so that concurrency and synchronization are not describable within the formalism.

In an interesting paper surveying flowchart techniques from a psychological point of view, Green [10] shows an example of high level states in a state-machine, with transitions allowed to leave the contours of a state at any level. The semantics is precisely that adopted in statecharts—namely, the event is regarded as an interrupt applied to all low-level states within.

Many of the methodologies suggested for the specification of complex systems, such as SADT [26] and HOS [11], concentrate mainly on the structural and functional aspects of these systems, and do not provide any dynamic semantics to cover their precise behavioral aspects. Either explicitly or implicitly, these methods recommend that the control be specified in some suitable programming language or PDL. Usually, they provide only means for functional decomposition with some data- and control-flow information. Consequently, approaches like those of [26, 11], when taken on there own without the addition of control-oriented code, are suitable only for transformational systems. In [30] it is proposed that such methods be augmented with conventional state-machines for behavioral control.

We now discuss a number of approaches that do provide solutions to the problem of behavioral description of complex reactive systems. Since this is not a survey paper, our comments will have to be extremely brief. Also, we have chosen only a small subset of proposed methods, concentrating on those that are, in our opinion,

(i) sufficiently formal to yield direct implementations, full computerized simulation or rapid prototypes, and

(ii) seem to be representative of particularly interesting approaches to the problem.

Petri nets (see, e.g., [25]) might be considered one of the best-known and best-understood solutions. They are graphical and precise, and come complete with an impressive body of research, accumulated over a period of more than 20 years. They are heavily event-driven and, by their very definition, allow for maximum concurrency. One of their drawbacks, however, is the unavailability of a satisfactory hierarchical decomposition. Consequently, there is only one level of concurrency, and the kind of high-level encompassing events that are applicable at once in many lower-level states (or places) are not naturally specifiable. We feel that introducing depth into the definition of a place in a Petri net, with transitions connected to places on all levels, might be one way of overcoming this problem.

Milner's CCS [22] is a formal programming/specification language of algebraic nature, and is also accumulating a large body of research, involving variations, formal semantics and detailed comparisons to other languages. Among the main characteristics of CCS are the adoption of recursion as the main control structure, the availability of nested concurrency, and the naming and aliasing flexibility provided by the use of variables for signals and communication channels. In fact, CCS is emerging as something of a yardstick, against which to measure the features, capabilities and power of new languages and approaches to programming and specifying concurrency. It would seem that further work is needed in order to assess the precise relationship between statecharts and the CCS language.

However, there are some interesting differences between languages like CCS or Hoare's CSP [16] and the statechart formalism. As already stressed above, statecharts are very efficient when it comes to describing 'interrupt-driven' behavior, e.g., by delineating a large set of states and specifying that they all react identically to the occurrence of some event. To describe a similar situation in, say, CCS, we have to explicitly add the branch representing this reaction to each of the relevant processes (corresponding to states). Another main difference is the mechanism of communication between processes. In these languages, the basic communication mechanism is a rendezvous, which enforces synchronization between a single sender and a single receiver. The statechart communication mechanism, on the other hand, is based on broadcast, whereby the sender may proceed even if nobody is listening. It also forces all enabled listeners to accept and respond to the message simultaneously (if a response is indeed specified). These two mechanisms are incompatible, in the sense that it is difficult and cumbersome to implement any one in terms of the other. It would be interesting to find out which of the two leads to more natural and compact specifications of systems. A point worth mentioning, though, is that the broadcast mechanism is not a crucial feature of the statechart formalism. It is conceivable that a satisfactory version of statecharts could be defined with rendezvous replacing broadcasting, while retaining the other benefits afforded by the more central concepts of depth and orthogonality.

Zave's approach in [33] is to use *sequence diagrams*, which can be viewed as Jackson's tree-like representations of regular expressions [18], with events (inputs) residing at the leaves. A specification of a system is a set of views, each one being, in effect, a possibly large sequence diagram. To enhance flexibility and comprehension events can have aliases, and a special filter is responsible for managing aliases and channeling input events to their relevant views. Sequence diagrams are actually structured *scenarios* (see Section 6.3) and therefore they constitute a natural way of specifying a system's behavior: identify all desired scenarios with respect to some class of events; their union is one view of the system's behavior. Any sequence of events not conforming to all views is illegal and therefore in error.

Concurrency, in Zave's approach, is explicitly present only on the top level: views are concurrent, and within a view there are only sequential, possibly nondeterministic constructs. Any concurrency inherent in the described system is implicitly represented in the sequence diagrams by nondeterministic interleaving of actions of the concurrent elements. We may thus relate Zave's diagrams to statecharts by viewing the latter as folding and conjoining the diagrams together while making the concurrency explicit. This means that sequence diagrams can always be read off (even automatically produced from) the corresponding statecharts, so that they essentially have the same expressive power. However, the statechart formalism encourages the user to explicitly identify the orthogonal and concurrent elements in the behavior he wishes to specify.

It has been our practical experience that when users approach a completely new system, they first specify to themselves a few scenarios, representing central 'threads' in the desired behavior. After experimenting for a while with these, they gradually conceive of the patterns that control them, and at a certain point they are ready to begin constructing statecharts, which compactly represent those patterns. Thus, the two formalisms might be used in a complementary fashion.

Since Zave's descriptions are taken also as 'upper bounds' on the behavior of the environment (non-conforming input sequences are in error), one possible way of merging the two approaches would be to use sequence diagrams as an assertion language, or a *constraint language*, for verifying desired properties of statecharts, as suggested for temporal logic in Section 6.3.

As far as programming languages especially tailored for real-time applications are concerned, the ESTEREL language (see [3]) is one of the most intriguing. It is motivated by concerns very similar to our own, and many of the resulting decisions taken there are strikingly similar to those present in statecharts. Transitions, assignment statements, signal transmitions, etc., are all assumed to be instantaneous, i.e., to take zero time. Durable happenings occur between events and while waiting for them. Events are broadcast, and a process misses an event if it did not happen to be attentive at the instant of occurrence.

One of the most interesting aspects of ESTEREL is the way it is translated, or compiled, into conventional finite-state machines. The compiler analyzes possible interleaving of events and prunes out impossible ones, a practice that often results in automata that are smaller than those obtained by a naive product operation applied bottom-up to the concurrent processes of the original program. This leads one to wonder if the translation techniques of [3] might not be useful in designing an optimizing compiler of sorts for statecharts.

9. Practical experience and implementation

The statechart formalism was conceived and developed while the author was consulting for the Israel Aircraft Industries (IAI) on the development of a complex state-of-the-art avionics system for an advanced airplane. This particular kind of project can demand bewildering levels of competence from many people of diverse backgrounds: pilots, electrical engineers, communication experts, software and hardware professionals, defense and weapons experts, etc. These people have to continuously discuss, specify, design, construct and modify parts of the system, ranging from high-level system structure all the way down to implementation details. The languages and tools they use in doing their work, and for communicating ideas and decisions to others, has an effect on the quality, maintainability, and expedition of the system that cannot be overestimated. Our experience in using the statechart approach in this project has been very encouraging: people were able to 'enter' a behavioral description of any portion of the system in almost no time, and the turnover period for peer review and inter-group feedback has been unusually minute. Statecharts are still used in that project as the main behavioral specification method. In addition to the avionics project, the method is being used experimentally in a number of software, electronics and semiconductor industries.

In all these cases the language is used manually, with all the inevitable consequences. We believe that a serious evaluation of the method, one that will uncover its weaknesses and prompt modifications that will make it more fitting for real use in various application areas, will depend to a large extent on the quality of a computerized graphics-based tool, and especially on its efficiency and user-friendliness. Such an implementation, called STATEMATE1, is in the final stages of construction at a company called AD CAD (Advanced Computer Aided Design), incorporated in Cambridge, Massachusetts. A prototype of STATEMATE1 is in beta-site tests, and a commercially available version is planned for release in June, 1987.

The implemented system uses the statechart language in order to specify the dynamic behavior of the system under development (SUD). In addition, it provides two complementing views of the SUD, the structural view (describing physical modules and channels), and the functional view (describing data-flow and activities). The latter is based on a hierarchy of activities, each controlled by a statechart, as discussed earlier. The three views are closely interrelated, and each is described using a special kind of diagram—*module-charts* for the former and *activity-charts* for the latter. The system enables editing, testing for consistency and completeness, as well as verifying a multitude of more sophisticated properties (such as the absence of nondeterminism or deadlock). It also supports management functions, as well as queries and reports of various kinds. One of the system's main functions is the ability to carry out a wide range of detailed simulations of the SUD, ranging from one-step simulation to random-event interactive simulation, all carried out graphically, and simulating all aspects of the system's description.

The details of this implementation are clearly beyond the scope of this paper. However, the reader should be able to appreciate the multitude of ways in which such a support system may aid the user. Moreover, there seems to be almost no limit to the number and kinds of features that one might want in such an implementation. Desired graphical options range from smooth and useful zoom-capabilities to self-topologizing, whereby the system arranges things in order to minimize, say, arrow length and/or crossover (a feature not yet implemented), and desired query options range from listing states with certain properties to presenting complex views of the system, dominated, say, by events, or by time and actions rather than by states.

10. Conclusion

This plaper is based upon a number of theses:

(1) *Reactive systems* differ from transformational systems, and require different approaches to their specification.

(2) An essential element in the specification of reactive systems is the need for a clear and rigorous *behavioral description*, to serve as the backbone of development from requirements specification all the way to user documentation.

(3) *Statecharts* provide one possible fitting formalism for specifying reactive behavior.

(4) The future lies in *visual languages* and methodologies that, with appropriate structuring elements, can exploit all the obvious advantages of graphical man-machine interaction.

Validating these theses is something a scientific paper cannot really achieve, but an effort has been made here to convince the reader that they are worthy of serious consideration.

Considering thesis (3), we are convinced that people working with complex systems have for a long time appreciated the simplicity and appropriateness of the state/event approach but have lacked a formalism for it that possesses certain elementary properties (such as depth and modularity) that are provided by most programming languages and by many conventional approaches to the physical and functional aspects of system description. The lack of these, as well as the exponential blow-up syndrome and the inherent sequentiality of conventional state machines, seem to have hindered serious use of states and events in the design of really large systems.

As to thesis (4), we believe that before long scientists and engineers will be sitting in front of graphical workstations with large (blackboard size?) displays of fantastic resolution, carrying out their everyday technical and scientific chores. It is quite fair to say that most existing visual description methods in computer science are predominantly intended as aids. The 'real' description of the object is usually given in some textual, algebraic form, and the picture is there only to help see things better, and to assist in comprehending the complexity involved. Here we are suggesting that *visual formalism* should be the name of the game; one uses statecharts as the formal description itself, with each graphical construct given a precise meaning. The language does not consist of linear combinations of icons or of one-level graphs, but of complex multi-level diagrams constructed in nontrivial ways from a few simple constructs. Textual representations of these visual objects can be given, but *they* are the aids (e.g., for users lacking graphical equipment, or for applications requiring textual reports), and not the other way around. See the discussion of *higraphs* in [13].

We are in the midst of several research efforts aimed at evaluating the statechart method in a number of diverse application areas, such as hardware components, communication systems, and interactive software systems. In [6] we propose the adoption of statecharts as a hardware description language, and in [2] for designing communication protocols. Both papers present some rather encouraging results to support these propositions.

Acknowledgments

I would like to thank the many people in the Israel Aircraft Industry who influenced the initial development of the ideas presented herein. In particular I am most grateful to J.Z. Lavi (Loeb) for introducing me to the problems and having the insight to see that solutions were possible, to Y. Shai who was an indispensable part of the development all along, and whose suggestions (such as to use dashed lines, rather than solid ones, for AND, and a quartz watch for illustrating the method) are present throughout, and to A. Kaspi and Y. Livne for presenting, respectively, the avionics engineer's and user's points of view (to one who has neither) in the most competent and patient way possible.

Discussions with and comments from A. Pnueli and R. Sherman have been very useful and thought-provoking and have greatly helped shape the initial ideas into their present form. In particular, Pnueli's ideas concerning reactive systems have served to pinpoint the problems that the paper addresses. Joint work with Pnueli (to whom the term 'reactive' is actually due), R. Sherman and M. Politi on a variety of related issues, including the semantics of statecharts and the functional view of system development have served to clarify many matters. Comments from H. Lachover, A. Pnueli, S. Ruhman and the referees on previous versions of the manuscript are most appreciated. Y. Barbut's superb graphical work and likewise C. Weintraub's TEXing deserve special thanks.

References

[1] The languages of STATEMATE1, Internal Report, Ad Cad, Inc., Cambridge, MA, 1987.

[2] A. Bar-Tur, D. Drusinsky and D. Harel, Using statecharts for describing the communication between complex systems, Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, 1986.

[3] G. Berry and I. Cosserat, The ESTEREL synchronous programming language and its mathematical semantics, in: S. Brookes and G. Winskel, Eds., *Seminar on Concurrency*. Lecture Notes in Computer Science **197** (Springer, Berlin, 1985).

[4] CCITT (International Telecommunication Union), Functional specification and description language (SDL), Recommendations Z.101-Z.104, Vol. VI, Fasc. VI.7, Geneva, 1981.

[5] E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications, *ACM Trans. Prog. Lang. Syst.* **8** (1986) 244-263.

[6] D. Drusinsky and D. Harel, Using statecharts for hardware description, CS85-06, Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, 1985.

[7] M.D. Edwards and D. Aspinall, The synthesis of digital systems using ASM design techniques, in: T. Uehara and M. Barbacci, Eds., *Computer Hardware Description Languages and their Applications* (North-Holland, Amsterdam, 1983) 55-64.

[8] A.B. Ferrentino and H.D. Mills, State machines and their semantics in software engineering, *Proc. IEEE COMPSAC '77 Conference* (1977) 242-251.

[9] S. Feyock, Transition diagram-based CAI/HELP systems, *Internat. J. Man-Machine Studies* **9** (1977) 399-413.

[10] T.R.G. Green, Pictures of programs and other processes, or how to do things with lines, *Behavior Inform. Tech.* 1 (1982) 3-36.

[11] M. Hamilton and S. Zeldin, Higher order software—A methodology for defining software, *IEEE Trans. Software Engrg.* **2** (1976) 9-32.

[12] D. Harel, Statecharts: A visual approach to complex systems, CS84-05, Department of Applied Mathematics, The Weizmann Institute of Science, 1984.

[13] D. Harel, A visual formalism and its applications, in preparation.

[14] D. Harel and A. Pnueli, On the development of reactive systems, in: K.R. Apt, Ed., *Logics and Models of Concurrent Systems* (Springer, New York, 1985) 477-498.

[15] D. Harel, A. Pnueli, J.P. Schmidt and R. Sherman, On the formal semantics of statecharts, *Proc. 2nd IEEE Symposium on Logic in Computer Science* (1987).

[16] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (1978) 666-677.

[17] J. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, MA, 1979).

[18] M.A. Jackson, *System Development* (Prentice-Hall, Englewood Cliffs, NJ, 1983).

[19] R.J.K. Jacob, Using formal specifications in the design of a human-computer interface, *Comm. ACM* **26** (1983) 259-264.

[20] H.-A. Kahana, Master's Thesis, Bar-Ilan University, Israel, 1986 (in Hebrew)).

[21] J. Martin and C. McClure, *Diagramming Techniques for Analysts and Programmers* (Prentice-Hall, Englewood Cliffs, NJ, 1985).

[22] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science (Springer, Berlin, 1980).

[23] D.L. Parnas, On the use of transition diagrams in the design of a user interface for an interactive computer system, *Proc. ACM Conference* (1969) 379-385.

[24] A. Pnueli, The temporal logic of programs, *Proc. 18th IEEE Symposium on Foundations of Computer Science* (1977) 46-57.

[25] W. Reisig, *Petri Nets: An Introduction* (Springer, Berlin, 1985).

[26] D. Ross, Structured analysis (SA): A language for communicating ideas, *IEEE Trans. Software Engrg.* **3** (1977) 16-34.

[27] R.L. Schwartz and P.M. Melliar-Smith, Temporal logic specification of distributed systems, *Proc. 2nd IEEE Interntional Conference on Distributed Computer Systems* (1981) 446-454.

[28] C.A. Sunshine *et al.*, Specification and verification of communication protocols in AFFIRM using state transition models, *IEEE Trans. Software Engrg.* **8** (1982) 460-489.

[29] A.S. Tanenbaum, *Computer Networks* (Prentice-Hall, Englewood Cliffs, NJ, 1981).

[30] P.T. Ward, The transformation schema: An extension of the data flow diagram to represent control and timing, *IEEE Trans. Software Engrg.* **12** (1986) 198-210.

[31] A. Wasserman, Extending state transition diagrams for the specification of human-computer interaction, *IEEE Trans. Software Engrg.* **11** (1985) 699-713.

[32] W.A. Woods, Transition network grammers for natural language analysis, *Comm. ACM* **13** (1970) 591-606.

[33] P. Zave, A distributed alternative to finite-state-machine specifications, *ACM Trans. Prog. Lang. Syst.* 7 (1985) 10-36.