On Synchronized Statecharts

Thesis for the degree of
Doctor of Philosophy

by
Doron Drusinsky

Submitted to the Scientific Council of
The Weizmann Institute of Science
Rehovot, Israel

April 1988

On Synchronized Statecharts

Thesis for the degree of
Doctor of Philosophy

by
Doron Drusinsky

Submitted to the Scientific Council of
The Weizmann Institute of Science
Rehovot, Israel

April 1988

27/3/88 [illegible]

This work was carried out at the

Faculty of Mathematics, Department of Computer Science
The Weizmann Institute of Science
Rehovot, Israel 76100

Under the Supervision of

Professor **David Harel**

ACKNOWLEDGEMENTS

I wish to dedicate this work to my wife Dganit and to my daughter Dana for their endless love, patience and understanding; and to my parents Harry and Luba for many years of love and support which enabled me to reach so far.

I would like to express my sincere appreciation and thanks to my advisor Professor David Harel for introducing statecharts and for many hours that he has spent discussing this work with me. Most of the research reported upon here is a result of collaboration with him, although all errors in this thesis are mine alone. He has been a constant source of inspiration for me, greatly helping me in all aspects of my doctoral research. He also extensively reviewed this thesis, suggesting many improvements, and teaching me the art of best organizing the exposition of technical material.

I also wish to thank the rest of my thesis committee, Professors Amir Pnueli and Smil Ruchman for their constructive suggestions, and for pointing out several issues that needed further investigation. I would like to thank A. Caspi, T. Lamdan, and M. Cohen for helpful discussions over the hardware synthesis methodology, and R. Heiman, R. Rosner, M. Safra and M. Shalev for their constructive feedback over the theoretical issues within Chapter 4. I thank Rafi Heiman for suggesting the clear three-dimensional rendition of Figure 33.

I wish to thank my friend and room-mate, Roni Rosner, for debugging most of the ω-input material (especially Proposition 19) and for many hours of discussion and patience during the last two years.

I thank Yehuda Barbut for doing a superb job with the figures, which constitute most of this thesis.

Table of Contents

ABSTRACT

Finite State Machines (FSM's) are undoubtably one of the most important models in computer science and engineering. Their drawbacks however, are well-known, two of which are their sequentiality and flatness. Harel [24] has suggested statecharts as extention of FSM's which makes an attempt to overcome these limitations. This thesis makes an attempt to show that statecharts do indeed overcome these drawbacks both in applications and in their theoretical power.

In the application parts (Chapters 2 and 6) we investigate two important applications: the design and synthesis of digital control systems and of communication protocols. We illustrate some of the main features of the approach, including hierarchical decomposition, multi-level timing specifications and flexible concurrency and synchronization capabilities. The thesis also presents a VLSI synthesis methodology, through which layout area and delay periods can be reduced relative to the conventional FSM synthesis method, and a polynomial-time sythesis algorithm for statecharts that guarantees communication progress.

In the more theoretical domain (Chapters 4 and 5) the framework of finite-state systems is used to investigate the relative power of three fundamental notions: nondeterminism and $\forall$-parallelism, the two facets of alternation, and statecharts' *cooperative concurrency*, whereby configurations consist of states between which communication can occur. We exhibit an exhaustive set of upper and lower bounds on the ability to inter-simulate these features over Σ^*, and an almost exhaustive set for the Σ^ω case, establishing that (a) each of the three features represents an exponential saving in succinctness of the representation, in a manner that is independent of the other two and additive with respect to them, and (b) of the three, the cooperative concurrency of statecharts is by far the strongest, representing a similar exponential saving when it is substituted for each of the others. In particular, we prove an exponential lower bound on the simulation of deterministic statecharts by AFAs and a triple-exponential lower bound on the simulation of alternating statecharts by DFAs. We also compare statecharts with (propositional) Temporal Logic (TL), examine the power of hierarchy and communication, and establish the complexity of several decision problems.

Introduction

Finite State Machines (FSM's) are probably the most fundamental single computational model within computer science. They have been used in almost every domain of computer science (e.g., specification, verification and synthesis of software, digital hardware, and communication protocols; lexical analyzers; text editors; computer learning; logic; and computer science theory). Two of the major drawbacks of FSM's, however, are their sequentiality and flatness. Without catering naturally for concurrency and multi-level descriptions a state-based approach is bound to be unsuitable for describing the behavior of large and complex systems. These facts seem to be almost universally accepted as indicating the inherent limitations of state-machine descriptions.

Harel's statecharts, introduced in [24], extend the familiar FSM's in several ways, while retaining both their formality and their visual appeal. Statecharts enable modular, hierarchical descriptions of system behavior, catering for multi-level descriptions, concurrency and state-history. Generally speaking, statecharts are state-diagrams[†] augmented with hierarchy (depth), orthogonality (concurrency), broadcast communication, multi-tape (multi-channel) I/O and state-history.

Throughout this thesis, our main interest is in the realm of 'difficult' systems according to the following dichotomy:

* systems that are not fully sequential and not fully concurrent;
* systems that are non-transformational (*reactive*[‡]); and
* systems that have multi-channel I/O.

Intuitively, a fully-concurrent system is a system that consists of k sequential processes that do not communicate. Such systems seem to be only k times more difficult to design than a single process. Non-fully concurrent systems, on the other-hand, must consider the complications of mutual dependencies and relationships between the communicating processes.

As stated by Pnueli [52] "*reactive systems are systems that cannot adequately be described by the* relational *or* functional *view. The relational view regards programs as functions (or as relations, in the nondeterministic case) from an initial state to a terminal state. Typically, the main role of reactive systems is to maintain an interaction with their environment, and therefore must be considered (and specified) in terms of their on-going behavior. Some of the interaction may also*

[†] State-diagrams are the graphical representations of FSM's.

[‡] This term is due to Amir Pnueli [52].

include a final result, but other reactive systems such as operating systems, airline reservation systems and process control systems are not supposed to terminate. In such systems the notion of a final state, which is essential to the relational view, is undefined or identified with error. Clearly the relational description is much simpler and neater than the behavioral description, and should therefore be preferred whenever it is possible. On the other hand, it is obvious that for some systems, such as those listed above, the behavioral view is inescapable."

The multi-tape requirement is simply property of practical systems which usually interact with many systems within their environment.

In this context, FSM limitations seem all the more severe. We make an attempt to show, formally and informally, that statecharts are indeed suitable for describing 'difficult' systems. We shall examine some examples and focus on the main appropriate statechart features, namely, flexible concurrency and hierarchy, and visual synchronization capabilities. We investigate this issue formally and show decisively that a purely concurrent model is 'weaker' than a (statechart based) model which is not purely concurrent and permits cooperative concurrency. This weakness is expressed by exponential upper and lower bounds for the simulation of the 'stronger' model on the 'weaker' model, and by polynomial upper and lower bounds for the simulation of the 'weaker' model on the 'stronger' one. We prove similar results for infinite computations (which best describe reactive systems).

On the practical front, this thesis also makes an attempt to show that, at least in the realm of the description and synthesis of digital control units and communication protocols, statecharts are a 'good' formalism. Hence we consider a spectrum of different domains of computer science and engineering:

* computer aided system design;
* computer engineering; and
* computer science theory.

It is important to notice that there is a strong tradeoff between these domains. For example, formal languages that are very expressive are usually economical, but have the property that the appropriate decision problems are very complex and sometimes are even undecidable; also, they are problematic from the standpoint of implementation and synthesis, relatively to less expressive languages. In most applications it usually turns out that the best language is the weakest one that is capable of describing the desired set of problems clearly. For this reason so many specification languages are no more powerful than FSM's.

The thesis covers the spectrum as follows. Chapters 2 and 6 examine two possible applications: (1) the description and synthesis of digital control units; and (2) the description and synthesis of communication protocols. In these chapters statecharts are examined from an informal point of view, and rather efficient synthesis procedures are presented. Throughout Chapter 2 we use the informal definition of statecharts as introduced in [24] and overviewed in Section 1.1. Chapter 6, on the other hand, uses our formal syntax and semantics of synchronized statecharts, and hence it is presented towards the end of the thesis.

Chapter 3 introduces the formal syntax and semantics of statechart transducers and acceptors over finite and infinite inputs. These syntax and semantics use only the most elementry features of the formalism as investigated in Chapters 4 and 5. They are then extended to various other versions of statecharts, including a version that incorporates alternation. Also, the notion of equivalence between multi-tape acceptors is investigated and defined.

Chapters 4 and 5 investigate the theoretical properties of statecharts. Cooperative concurrency, such as that in statecharts, is compared with alternation, for both finite and infinite inputs, and statecharts are compared with temporal logic. Hierarchy and communication are investigated from the formal standpoint, and several decision problems for statecharts are analyzed.

Parts of this thesis are results of joint work with Prof. David Harel. The material of Chapter 2 appears in [12,13,15], that of Chapter 4 appears in [10,14], and parts of Chapter 6 appear in [1].

1.1 An Overview of the Statechart Formalism

Statecharts are based on states, events and conditions, with combinations of the latter two causing transitions between the former. Both states and transitions can be associated in various ways with output events, called *actions*, which can be triggered either by executing a transition or by entering, exiting, or simply being in a state. The system's inputs are thus the (external) events and its outputs are the (external) actions; their union comprises the interface set E of externally observable events, conditions and outputs.

This idea is well-known, and is actually a simple combination of the Moore and Mealy definitions of conventional finite state automata. The allowed sequences over E correspond to the language accepted by the automaton. Moreover, such automata come complete with a standard visual rendition, the state-diagram. In its naive form, this classical state transition method has been all but abandoned as a way of specifying the behavior of complex systems since it provides no modularity or hierarchical structure, and suffers acutely from the exponential blowup in the number of states that need be considered. Indeed, a state/event description seems to have to consider all possible combinations of states in all the components of the system; hence the exponential growth.

The statechart method is rooted in an attempt to revive this old and natural way of thinking about a system's behavior, by extending it to overcome these difficulties. The extensions apply to the underlying nongraphical formalism too, but we prefer to present the ideas in terms of the graphical version. Some of the extensions are now briefly described, but the reader is urged to consult [24] for a fuller treatment.

States in a statechart can be repeatedly combined into higher-level states (or, alternatively, high-level states can be refined into lower-level ones) using AND and OR modes of clustering. Fig. 1 shows a state B whose meaning is "to be in B the system must be in precisely one of D, E or F," and Fig. 2 shows a state A whose meaning is "to be in A the system must be both in B and in C." Notice, however, that in Fig. 2, B and C are themselves OR states, thus the actual possibilities are the state configurations (D,G), (D,H), (E,G), (E,H), (F,G), and (F,H). We say that D, E and F are *exclusive* and B and C are *orthogonal.*

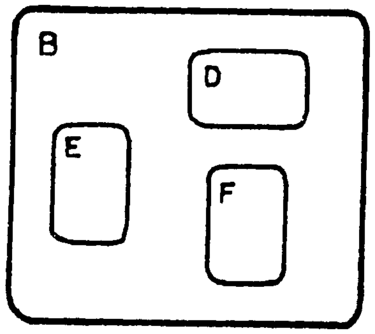


Figure 1. OR-ing states

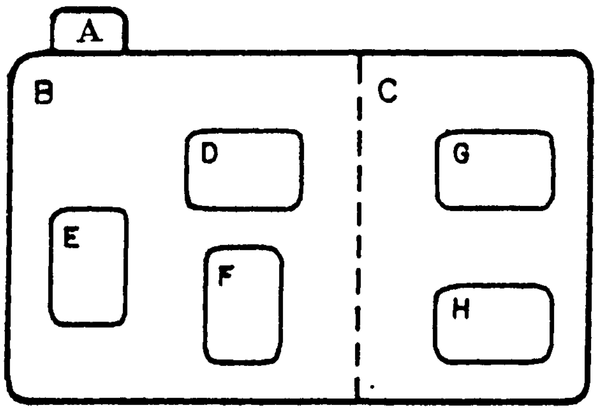


Figure 2. AND-ing states

Transitions in a statechart are not level-restricted and can lead from a state on any level of clustering to any other. A transition whose source state is a superstate means "the system leaves this state no matter which is the present configuration within it." In this way, while event a in Fig. 3 causes a simple transition from state K to L, the event b exemplifies a concise way of causing the system to leave L or M, i.e. any possibility of being in J, and to enter K. Likewise, c causes the system to exit any one of the A-configurations listed above and enter M. If the target of a transition is a superstate, as in the case of events d or e in Fig. 3, a default arrow must be present, indicating which of the lower-level states is actually to be entered (L or the combination (E, G) in this case).

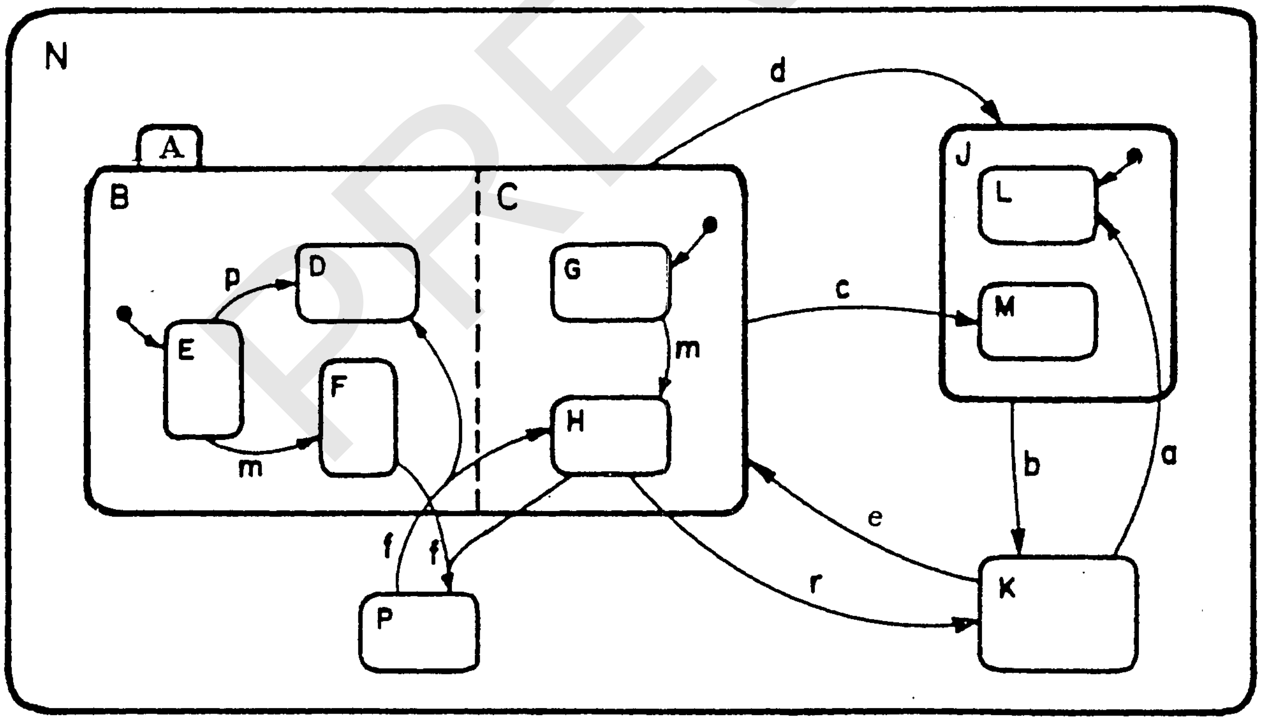


Figure 3. An output-free statechart

Actually, transitions are in general from configurations to configurations, owing to the possibility of orthogonal components in the source and target states. Thus, in Fig. 3 if event f takes place in configuration (F, H) the system enters P and if the same happens in P the system enters (D, H). Concurrency and independence are both made possible by orthogonality: on the one hand event m causes simultaneous transitions in B and C if the configuration is (E, G), and on the other p causes E to be replaced by D regardless of, and with no change to, the present state in C. It is noteworthy that orthogonality (and hence the possibilities it raises) is allowed on any level of detail. Accordingly, a configuration can be layered too, containing orthogonal state components on many levels.

Outputs can now be associated with transitions as in Mealy automata by writing a/b along an arrow; the transition will be triggered by a and will in turn cause b to occur. Similarly, b can be associated with (entering, exiting, or simply being in) a state, in line with Moore automata. In either case b can be an external event or an internal one, in the latter case triggering perhaps other transitions elsewhere in some orthogonal state. Formal syntax and semantics for statecharts appear in Chapter 3.

The formalism presented in [24] offers a number of additional features, among which the following is rather important for hardware applications: one is allowed to specify upper and lower bounds on the time to be in a state, as in the self-explanatory diagram in Fig. 4.

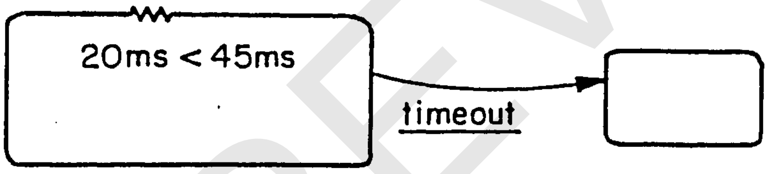


Figure 4. Upper and Lower time bounds

Specifying behavior by statecharts encourages thinking in terms of the system's conceptual states and their interconnections, and caters for modular "chunking" of behavior by using exclusivity and orthogonality of states. Note that the exponential blowup in states is avoided, as the option of using orthogonality on any level eliminates the need for explicit consideration of all state combinations.

A rather complex example of a statechart for a traffic-light controller appears in Chapter 2.

Application 1: Hardware Description and Synthesis

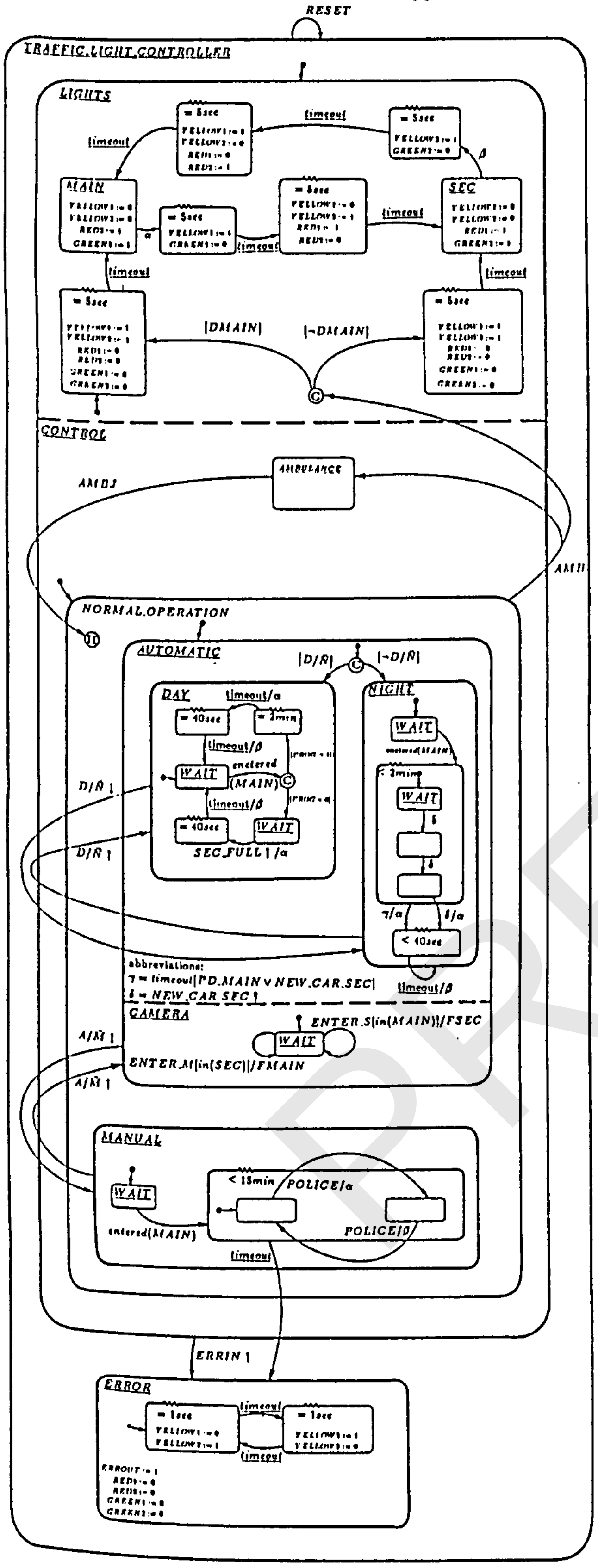
In this Chapter we consider one possible application of statecharts, namely, the description and synthesis of digital control units. Other applications appear after the formal chapters, in Chapter 6. In all of these applications, FSM's have been used extensively in the past, and we make an attempt to apply well-known techniques to the statechart methodology.

Section 2.1 presents an example of the application of statecharts for describing a relatively complex traffic-light controller, and several of the features relevant to hardware description are then discussed. Section 2.2 contains an overview of the conventional synthesis method for FSM's, and presents the principles of a statechart VLSI synthesis methodology. Finally, Section 2.3 presents a programmable approach to statechart synthesis.

Throughout this chapter, we shall use the informal syntax and semantics of statecharts as introduced in [24], and overviewed in Section 1.1.

2.1. Statecharts Via a Traffic-Light Controller.

Fig. 5 describes the behavior of a traffic-light controller whose I/O interface is described in Fig. 6. There are two sets of lights: one is positioned over the main road (MAIN) entering the cross-junction, and the other is over the secondary road (SEC). During day time ($D/\bar{N} = 1$) the controller operates according to one of two possible programs: program A (PROG = 1) gives two minutes to the vehicles in MAIN, and half a minute to the vehicles in SEC, alternatingly, and program B (PROG = 0) gives half a minute to the cars in SEC once the SEC_FULL signal goes high. During the night ($D/\bar{N} = 0$) the controller gives precedence to the cars in MAIN until one of the two possibilities occurs: (1) two minutes have passed since MAIN became green and either a pedestrian wants to cross MAIN (PD_MAIN = 1), or a new car appeared in SEC (NEW_CAR_SEC = 1); or (2) three cars have already appeared in SEC. Once one of these conditions occurs, vehicles in SEC are given half a minute. The controller can be operated manually as well ($A/\bar{M} = 0$). In this mode, whenever POLICE becomes 1 (a policeman pushing a button) a transition is triggered from MAIN to SEC or vice-versa. This manual operation, and any transition from day to night and vice-versa, starts with 5 seconds of flashing yellow lights and then MAIN receiving the green lights. A hidden camera can be operated by the controller when it is in AUTOMATIC mode only.



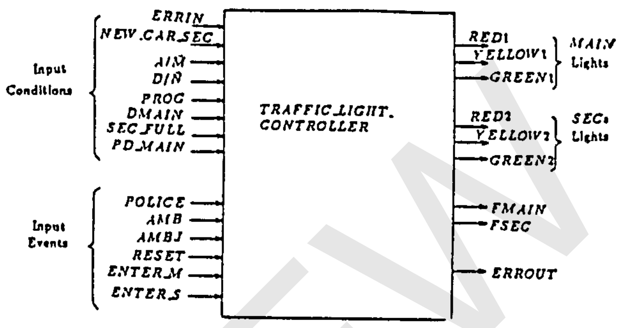


Figure 6. The I/O interface for the traffic-light controller

Figure 5. Statechart for the traffic-light controller

The camera will take a photo of the MAIN entrance to the junction, by producing the FMAIN signal, when MAIN is in the red state and a car enters the junction from MAIN (ENTER_M = 1), and similarly for the SEC entrance (using the ENTER_S signal, and producing the FSEC signal). The controller can receive an ambulance signal (AMB = 1), notifying the controller that an ambulance is approaching the junction from MAIN (DMAIN = 1) or from SEC (DMAIN = 0). It then sets the lights according to the direction of the ambulance, and ignores all other events. Once the ambulance enters the junction the controller is notified (AMBJ = 1), and returns to its previous operation mode; namely DAY or NIGHT. The controller can receive an error message (ERRIN = 1) and then flickers both yellow lights. Another possibility for an error occurs when the controller operates manually for more than fifteen minutes without the policeman pushing the POLICE button, in which case the ERROUT signal is produced. A RESET signal resets the controller to the AUTOMATIC state.

In Fig. 5, we have exclusive states (e.g. DAY and NIGHT), and orthogonal states (e.g. AUTOMATIC and CAMERA). We have default entrances (e.g. the entry to WAIT within MANUAL), and entrances by history (e.g. from AMBULANCE upon the event AMBJ, returning by history only one level backwards, i.e. to AUTOMATIC or MANUAL, and then by default). We have time bounds on the duration of being in a state (e.g. precisely 5 seconds in six of the states in LIGHTS, and at most 15 minutes in two of the states in MANUAL). We use conditional connectors (e.g. the entrance to AUTOMATIC dependent upon $D/\bar{N}$), and uparrows and downarrows to turn condition changes into events (e.g. $D/\bar{N} \uparrow$ is the event occuring when $D/\bar{N}$ changes from 0 to 1).

Actions can appear along transitions as in Mealy automata (e.g. β is generated when making the transition between two states of MANUAL, triggered by the POLICE event). They can also appear in states as in Moore automata, in which case, by convention, they are carried out upon entrance to the state (e.g. the red and green lights are zeroed upon entering ERROR.

Let us try pointing out some of the general capabilities offered by the likes of Fig. 5:

* Hierarchical descriptions.
The ability to provide hierarchical descriptions becomes vital as the complexity of the described system grows. Designing in a hierarchical fashion makes the design process clearer, easier and more manageable. Statecharts enable hierarchical decomposition using a mechanism that condenses information. Thus, in Fig. 5 the event AMB operates on all of NORMAL_OPERATION's substates without the need for explicitly describing that it is "sent" to all the substates. More generally, the event *b* in Fig. 1, for example, operates both on states L and M in J. On the other hand, in a conventional block diagram describing the same configurations, although *b* operates on the whole of J, the description must show how it is sent, explicitly, to both L and M, as in Fig. 7.

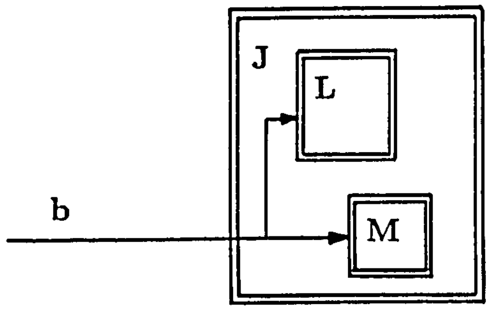


Figure 7. A hierarchical description not condencing information

The capability to condense information does not mearly mean that less lines are to be drawn, but enables *information hiding* where the designer which is responsible for a high-level state may hide the detailes of his module from the designers of the lower-level states.

* Flexible Concurrency Description.
The ability to describe concurrent activities is an important requirement. Statecharts support a natural (and visual) way of describing concurrency at any hierarchical level, without causing sequential descriptions to become an awkward exception. In Fig. 5, CAMERA is orthogonal to AUTOMATIC, but LIGHTS is orthogonal to both of them, on a much higher level.

* Timing specifications.
Statecharts offer local-looking capabilities for timing specification in states. However, since such timing constraints can appear in states on any level, this actually admits global timing constraints too. Figure 8, illustrates multilevel timing constructs, combinations of which can yield quite subtle behavior.

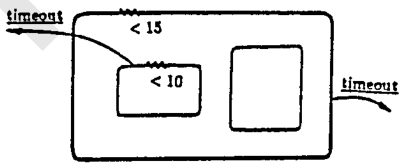


Figure 8. Multi-level time-constraint specification

* Synchronization Methods.
Fig. 5 contains several synchronization tricks, such as the event A/$\bar{\text{M}}$ ↓, which causes a transition from AUTOMATIC to MANUAL. One should notice how the arrow crosses both the NORMAL_OPERATION and CONTROL boundary lines, thus causing the system to exit LIGHTS too and then reenter it (but through the default entrance). The side-effect is to synchronize LIGHTS to its default state upon the A/$\bar{\text{M}}$ ↓ event. Figure 9 illustrates some

similar possibilities: α synchronizes all orthogonal states into their default substates; β synchronizes B into B1 while moving from A1 to A2; C is synchronized into C2 as A enters A2 (i.e A2 is uses as a common state); and D is synchronized into D2 as C enters C2 (i.e. x is used as a common variable).

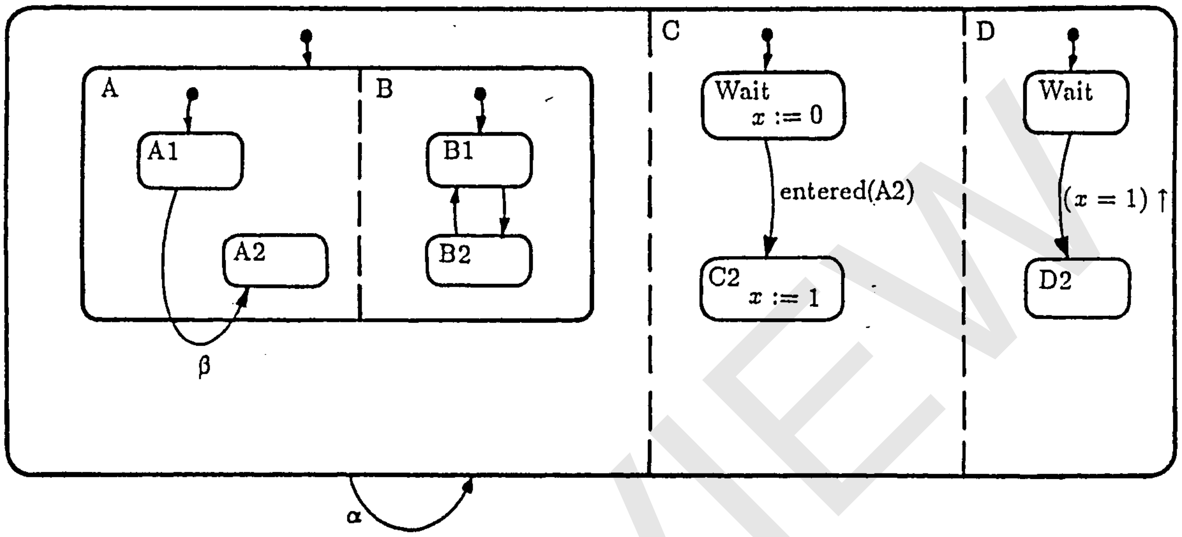


Figure 9. Various synchronization schemes

* Fault Specification.
A discussion of HDL's with respect to fault modelling can be found in [61]. Statecharts supply a hierarchical tool for event-based descriptions, and therefore any event, including those thought of as faults, can be represented at any desired level in the hierarchy. The description takes the form of an event or an impossible event, and alleviates the need for a special effort of 'planting' the test in the basic components of the described system. One should notice, however, that statecharts can only treat high-level behavioral faults and cannot naturally deal with circuit-level faults, for example.

* Visuality.
Statecharts appear to enable visual representations in a somewhat broader spectrum than most common diagramatic systems , as they cover hierarchy, concurrency, timing aspects and synchronization.

The main apparent disadvantage of the statechart approach is that it separates control from data. It is predominantly tailored for control, with the data portions being related to the activities within states or along transitions. A computerized graphical tool, STATEMATE, which is available from i-Logix, Inc. (see [26]), utilizes statecharts for control and, in addition, supports a graphical language of hierarchical data-flow nature, called activity-charts, for the data and functional aspects of the system under description.

2.2. The Statechart Synthesis Methodology

(Note: throughout, n is the number of states in a statechart, i.e., the size of the state tree, and d is the bound on the outdegree of the tree, i.e., the maximal number of immediate substates.)

As discussed above, statecharts are an extension of the Moore and Mealy variants of FSM's. These have typical implementations using PLA's (programable logic arrays) for the specification of the combinatorial logic (Fig. 10).

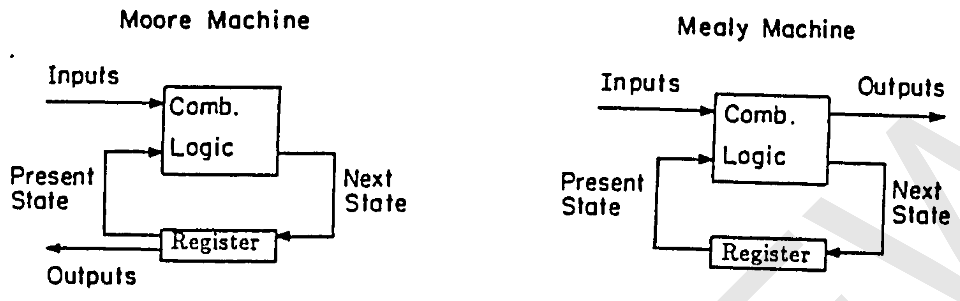


Figure 10. Two implementations for FSM's

Here, an n-state FSM is represented by $O(n^2)$ state transitions,[†] which are represented by a 'next-state' disjunctive normal form formula, implemented on a PLA. The state register contains the 'present-state' between two consecutive state transitions. PLA's enable simple and regular implementations of control units (see [45]) but have the disadvantage of being highly area-consuming as the number of states grows. The area of a PLA for such an implementation is determined mainly by the number of minterm lines, which is on the order of n^2, at least one minterm for each FSM transition. Thus, even without considering I/O wires, FSM area might reach $O(n^2 \cdot \log\ n)$. The clock cycle using this technique is $O(n^2)$, due to the time for the slowest signal to propagate from the state register, through the PLA, back to the state register. Frequently, FSM or PLA folding and partitioning techniques are used to overcome this area blowup. Such techniques however, are usually NP-Complete [23,64]. Our methodology carries out such a 'partition' during the description phase, and uses the designers knowledge of the problem to generate an efficient product.

The basic idea of our synthesis methodology is to trade the concept of a single machine implementing an FSM for a tree of interconnected machines implementing a statechart. Every state at every nonatomic level of the statechart hierarchy is represented by a machine implementing the immediate FSM one level lower.

[†] If the FSM has $k > 1$ input lines, the number of state transitions becomes even larger. However, for simplicity we shall be concerened only with the asymptotic growth of the FSM implementation.

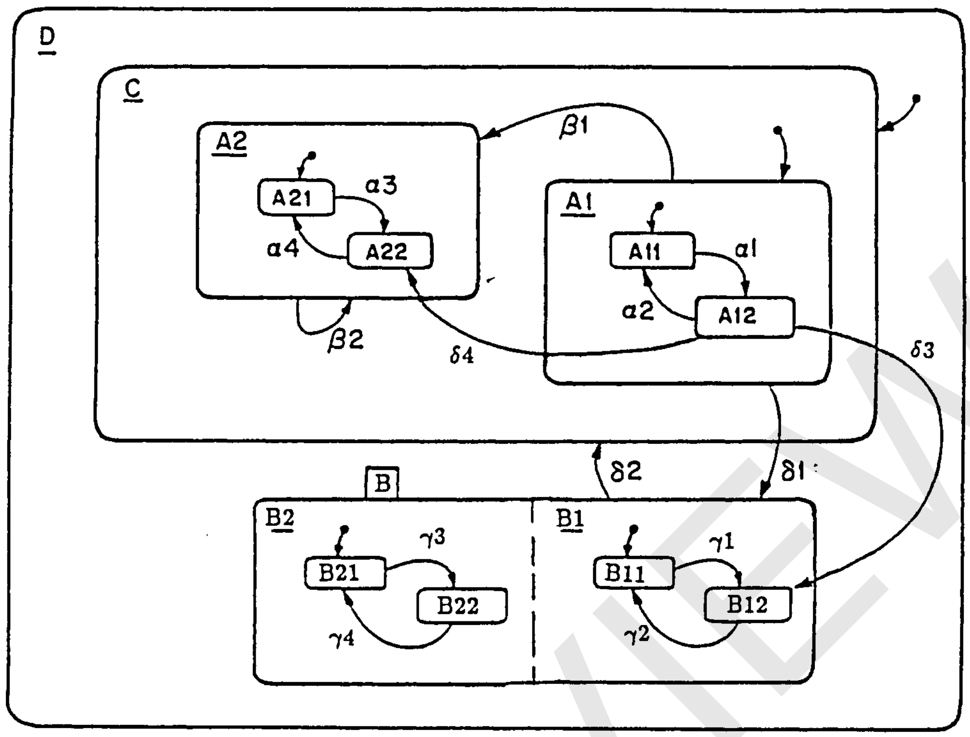


Figure 11. A statechart example

Thus, for the statechart of Fig. 11, fifteen machines are built, implementing the seven FSM's of Fig. 12, and another eight trivial machines for the atomic states; the machine connection scheme is illustrated in Fig. 13.

An event **entered** X is created by the machine one level higher in the hierarchy when it reaches state X. The **left** signals are the duals of the **entered** signals, and notify the lower level machines to move into their *Idle* state. The **leave** signals, running between machines, are created by the lower level states, to notify their predecessors about their termination, such as A1 notifying C in Figures 11, 12. Similarly, the **enter** X signal is created by a high-level state in the case that one of its substates is required to start operating in the X state that is not its default, such as C notifying A2 in Figures 11, 12. Concurrency is implemented in a natural manner as illustrated in Fig. 14 where the **entered** and **left** signals are sent from A to both B and C (i.e. B and C will operate concurrently.)

We use a one-bit code for coding states, with each state having its unique representing bit. This 'horizontal' coding scheme seems exponentially expensive in comparison to the usual 'vertical' coding scheme, but since the coding is per machine (each of limited size), its cost is bounded.

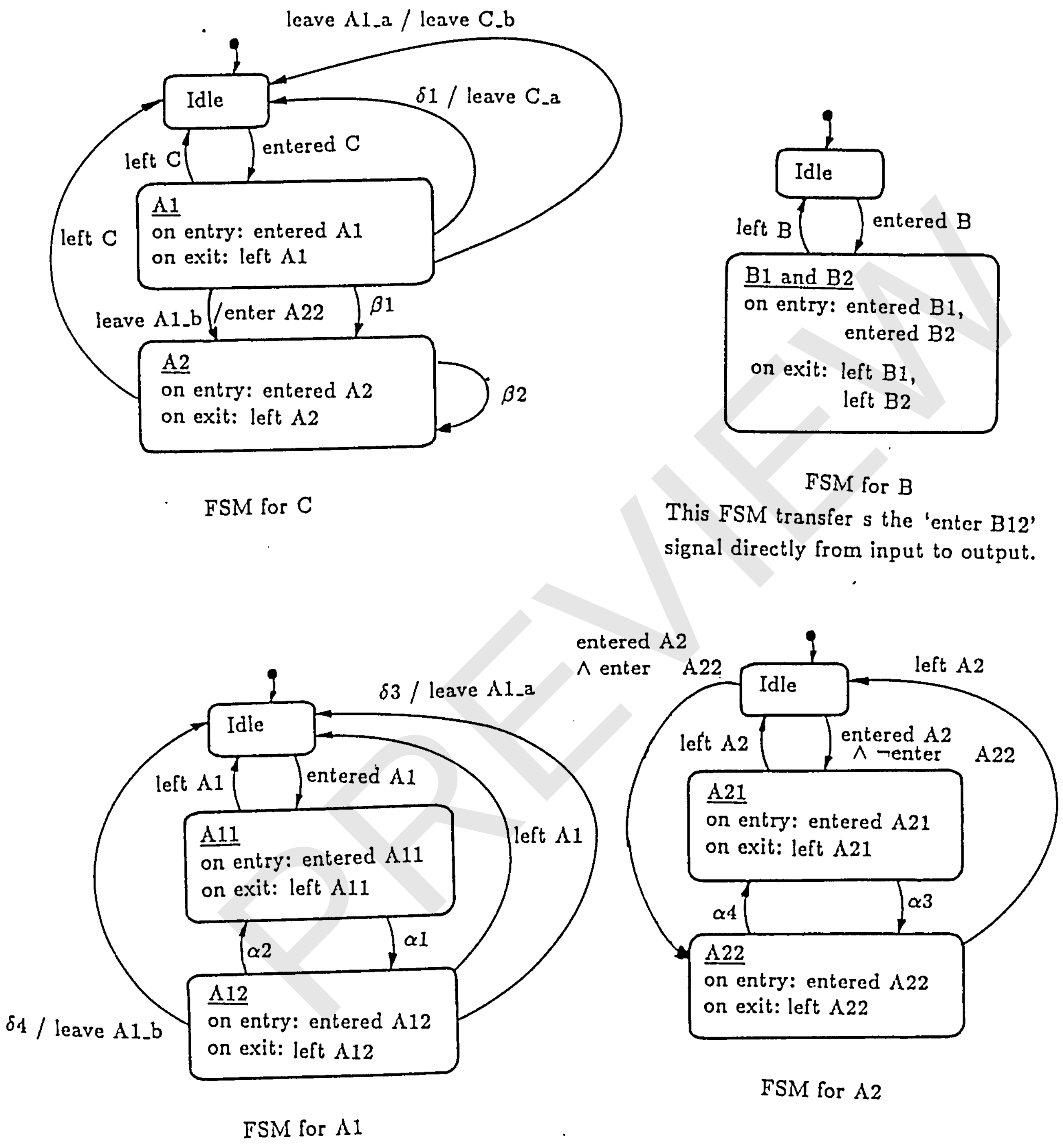
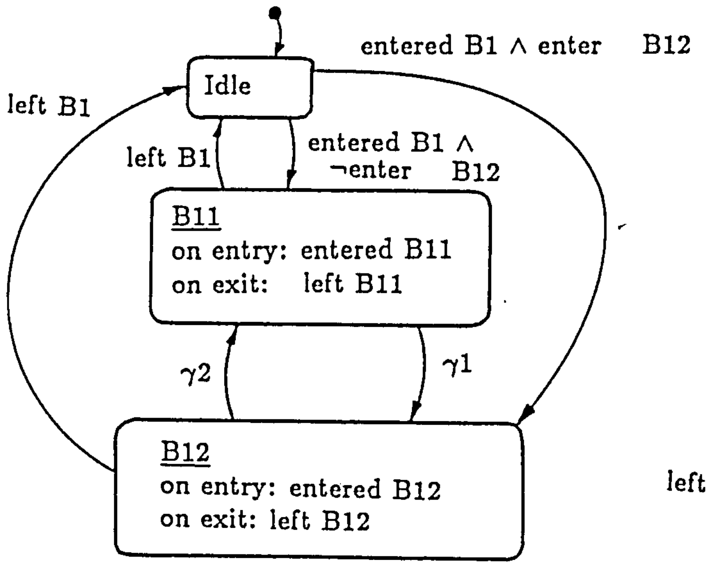
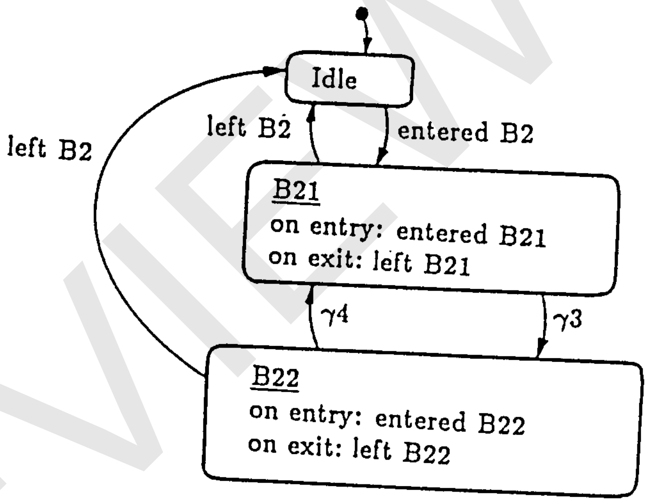


Figure 12. FSM's for Fig. 11.

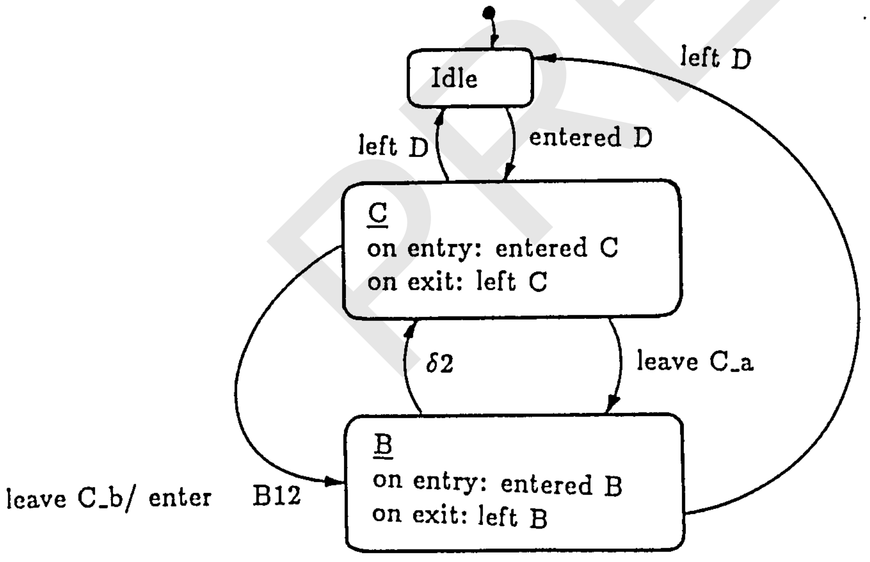
An output along a transition is created by the transition itself.



FSM for B1



FSM for B2



FSM for D

Figure 12. (Cont.)

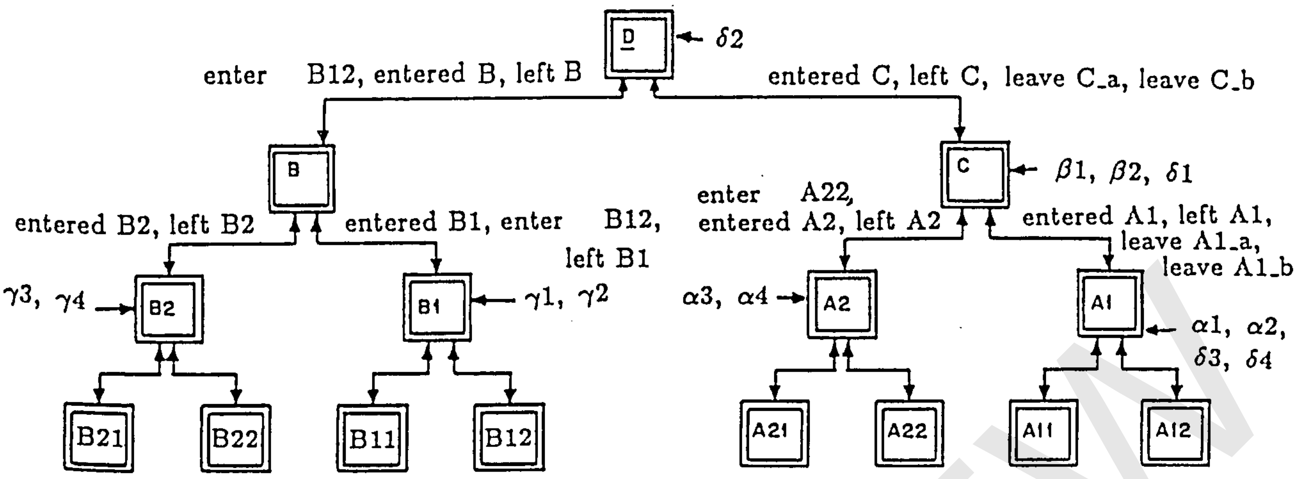

D
δ2
enter B12, entered B, left B
entered C, left C, leave C_a, leave C_b
B
C
β1, β2, δ1
entered B2, left B2
entered B1, enter B12, left B1
enter A22, entered A2, left A2
entered A1, left A1, leave A1_a, leave A1_b
γ3, γ4
B2
B1
γ1, γ2
α3, α4
A2
A1
α1, α2, δ3, δ4
B21
B22
B11
B12
A21
A22
A11
A12

Figure 13. Processor connection scheme for Figures 11,12

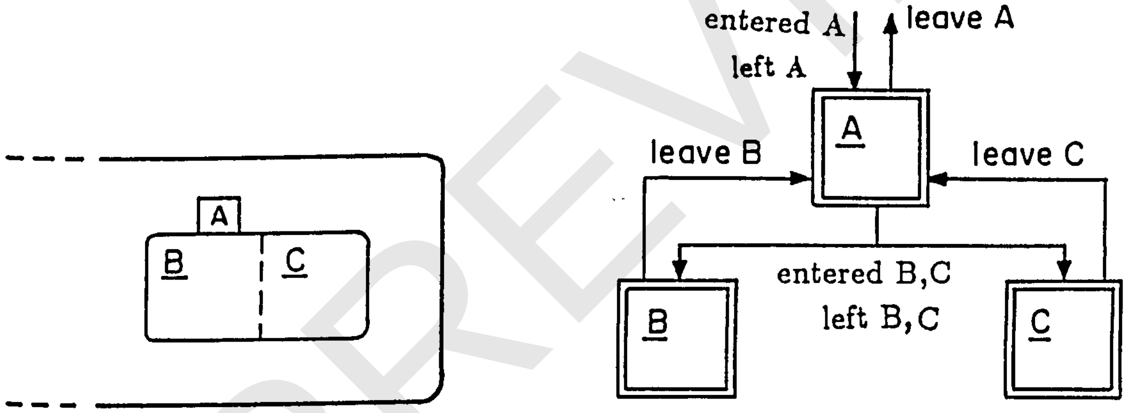


Figure 14. Machine configuration for orthogonal states

There are three immediate possible layouts for the resulting tree-machine. We can layout the tree-machine in $O(n)$ area using the general algorithm of [41]. This layout, however, has the disadvantages of creating a non regular structure and ignoring the I/O wires from the individual machines. Also, in this layout a basic machine and a wire are of the same width, causing considerable waste.

A prefered layout can be obtained using the configurable techniques of [41]. The layout is of area $O(n \cdot \log^2(n))$, and is illustrated in Fig. 15 for the example of Fig. 13. All vertices (machines or processors) are lined up on the baseline and their connections run vertically. Parallel to the baseline are $O(\log^2(n))$ horizontal wires (each such virtual wire can be a multitude of entered, left, enter and leave wires). The top $O(\log(n))$ wires run all the way across the layout, the next

$O(\log(n))$ wires are broken halfway, etc. The only remaining decision is where to put the 'solder dots' that determine the actual connections. The method of [41] uses the fact that any finite tree with n vertices and bounded out-degree can be bisected into two sets of $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ vertices by cutting at most $O(\log(n))$ edges. Once the cut-set is determined, the two sets of vertices can be laid out recursively and combined; placing the edges in the cut set as the horizontal lines in the layout.

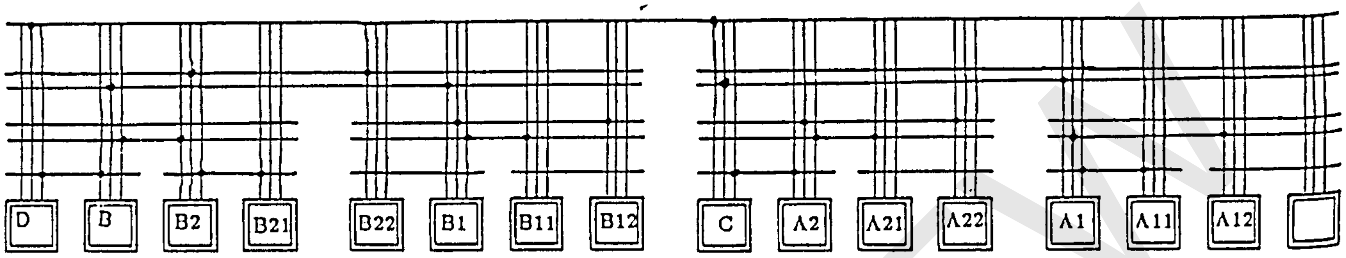


Figure 15. Layout of processors for Figure 13

A better layout, with area $O(n \cdot \log(n))$, can be achieved using the 1-separator theorem for trees [41], and is illustrated in Fig. 16 (for the same example). Using the theorem, the tree is bisected into two sets, each consisting of between 1/4 n and 3/4 n vertices, by removing a constant number of edges. Each set is then laid out recursively along the baseline, and the removed edges are placed horizontally above.

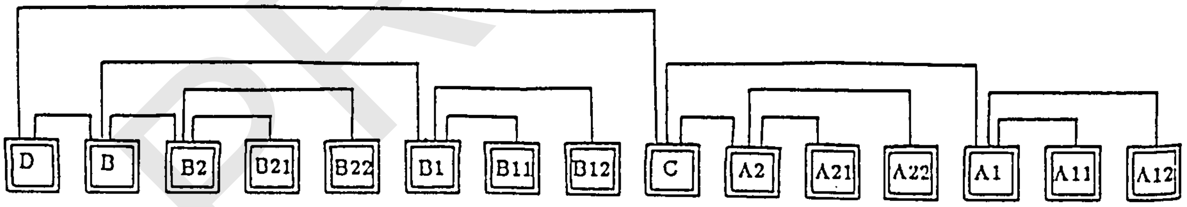


Figure 16. A slightly better layout of the processors in Figure 13

Several timing problems can occur in the example of Figures 11-13. The natural choice is to implement each individual FSM using a Moore machine as was actually done in Fig 13. In this way, it might take several clock cycles for the event $\delta 3$ at A12 to propagate up to D's machine and cause the transition to state B12, a delay which is in conflict with the formal semantics of statecharts [24,28] and also with our intuition that such a transition should be instantaneous in a synchronous system. Similar time delays will occur when a lower level machine enters its first state after the parent has already been entered (such as entering A21 after C has moved from A1 to A2, in Fig. 11), or when lower level states