Using Statecharts for Hardware Description and Synthesis

DORON DRUSINSKY AND DAVID HAREL

Abstract—**Statecharts have been proposed recently as a visual formalism for the behavioral description of complex systems. They extend classical state-diagrams in several ways, while retaining their formality and visual nature. In this paper we first argue that statecharts can be beneficially used as a behavioral hardware description language. We illustrate some of the main features of the approach, including: hierarchical decomposition, multi-level timing specifications and flexible concurrency and synchronization capabilities. We also present a VLSI synthesis methodology, through which layer area and delay periods can be reduced relative to the conventional finite state machines (FSM) synthesis method.**

I. Introduction

Finite state machines (FSM's) have been one of the main formalisms underlying the prevailing approaches to hardware description and synthesis. Two of the major drawbacks of FSM's, however, are their inherent sequentiality and flat, non-hierarchical nature. 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 digital hardware components. These facts seem to be almost universally accepted as indicating the inherent limitations of state-machine descriptions.

Recently, an attempt at overcoming these limitations has been made with the advent of *statecharts* [6], [7], which extend the familiar FSM's in several ways, while retaining both their formality and their visual nature. Statecharts enable modular, hierarchical descriptions of system behavior, catering for multi-level descriptions, concurrency, and state-history. This paper presents the results of an initial assessment of the feasibility of using statecharts in the realm of digital hardware description and synthesis.

Section II provides a brief description of statecharts, and Section III presents an example of their use in describing a relatively complex traffic-light controller. Several of the features relevant to hardware description are then discussed. Section IV contains an overview of the conventional synthesis method for FSM's, and presents the principles of a statechart-based VLSI synthesis methodology. Finally, Section V presents a programmable approach to statechart synthesis.

II. The Statechart Formalism

The *statecharts* method was introduced recently ([6], and see also [7]) as a visual formalism for specifying the behavior of complex reactive systems (see [9], [16].) Like FSM's, 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 of externally observable events, conditions, and outputs.

This basic idea is well known, and is actually a simple combination of the Moore and Mealy definitions of conventional finite state automata. The allowed sequences of interface elements correspond to the language accepted by the automaton. Moreover, such automata come complete with a standard visual rendering, the transition diagram. In its naive form, this classical state transition method has been unsuccessful at 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. Various notions of communicating FSM's have been suggested, but the lack of modularity in such approaches appears to cause users to decompose behavior according to the physical decomposition of their system, thus loosing the advantages of carrying out conceptual behavior specification before the design stage.

The statechart method is rooted in an attempt to revive the natural FSM approach to the specification of systems, by extending it to overcome these difficulties. The extensions apply to the underlying nongraphical formalism, too, but there are advantages in presenting the ideas in terms of the graphical version. Some of the extensions are now briefly described, but the reader is urged to consult [6] 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*.

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 Fig. 3).

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 hand 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

Manuscript received October 6, 1987; revised March 1, 1988 and December 5, 1988. This work was supported in part by grants from Ad-Cad, Ltd., and by the Israel Aircraft Industries. The review of this paper was arranged by Associate Editor M. R. Lightner.

This paper is an expanded version of the work originally presented at the ICCAD-87.

D. Drusinsky was with The Weizmann Institute of Science, Rehovot, Israel 76100. He is now with the CAD Department, Sony Corporation, Atsugi, Japan.

D. Harel is with the Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel 76100.

IEEE Log Number 8826963.

0278-0070/89/0700-0798$01.00 © 1989 IEEE

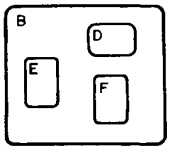


Fig. 1. OR-ing states.

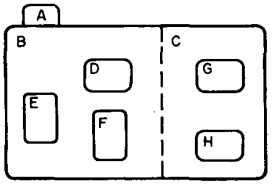


Fig. 2. AND-ing states.

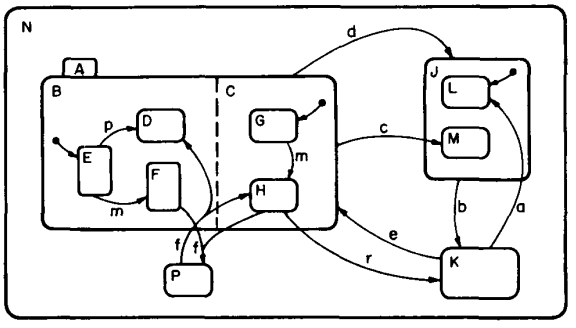


Fig. 3. An output-free statechart.

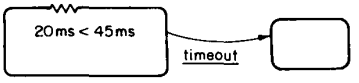


Fig. 4. Upper and lower time bounds for being in a state.

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 [10] and [11].

The formalism presented in [6] 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.

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 by the orthogonality construct.

III. 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. In the figures we have used some abbreviations: the colors red, green, and yellow are simply RD, GR, and YL. A condition followed by ↑ is the event that occurs when the condition changes from false to true; the opposite change is depicted by a ↓. In addition, we abbreviate certain events as follows:

- timeout becomes tm;
- entered(state) becomes ent(state);
- timeout[**pd_main** ∨ **new_car_sec**] becomes γ;
- **new_car_sec**↑ becomes δ.

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 the day ($D/\overline{N} = 1$) 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 ($D/\overline{N} = 0$) 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 (**pdmain** = 1) or a new car has 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 ($A/\overline{M} = 0$). In this mode, whenever a policeman pushes a special button (**police** becomes 1) 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 s of flashing yellows lights and then MAIN receiving the green lights. A hidden CAMERA can be operated by the controller when it is in AUTOMATIC mode only. 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). An ambulance signal can arrive (**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 it returns to its previous operation mode, namely DAY or NIGHT. The controller can receive an ERROR message (**errin** = 1) and then both yellow lights flicker. 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 s in six of the states in LIGHTS, and at most 15 min in a substate of MANUAL). We also use *conditional connectors* (e.g., the entrance to AUTOMATIC, which is dependent upon $D/\overline{N}$).

Actions can appear along transitions as in Mealy automata (e.g., α or 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 hierarchial descriptions becomes vital as the complexity of the described system grows. Specifying in a hierarchical fashion makes the development process clearer, easier, and more manageable. The hierarchical decomposition that statecharts offer uses a mechanism that condenses information. Thus in Fig. 5, for example, the event **amb** operates in all of NORMAL's substates, by the semantics of statecharts. There is no need to explicitly send such an event to each substate. More generally, the event *b* in Fig. 3, 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

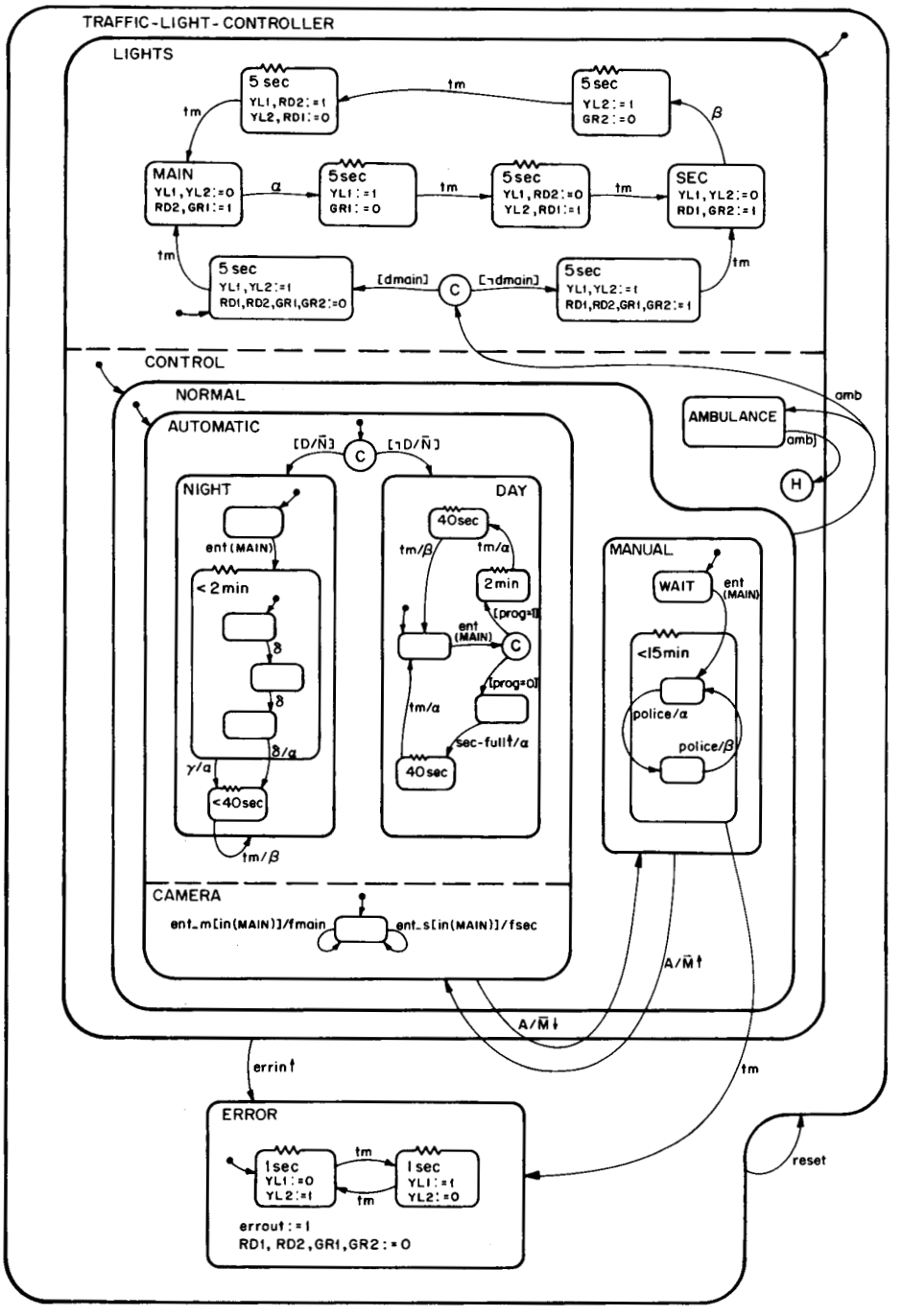


Fig. 5. Statechart for the traffic-light-controller.

description must usually show how it is sent, explicitly, to both L and M, as in Fig. 7.

- *Concurrency Descriptions:*

The ability to describe concurrency is crucial. Statecharts support a natural 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, and in any orthogonal component, this actually admits global timing constraints, too. Fig. 8 illustrates multi-

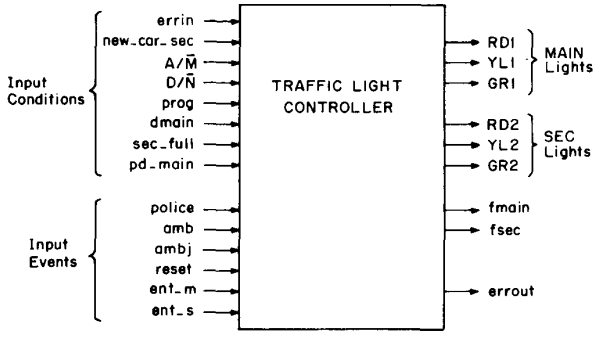


Fig. 6. The I/O interface for the traffic-light-controller.

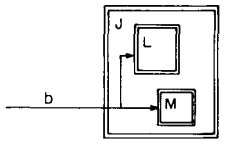


Fig. 7. A hierarchical description that does not condense information.

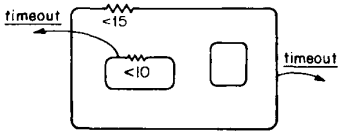


Fig. 8. Multi-level timing constraints.

level timing constructs, combinations of which can yield quite subtle behavior.

- *Synchronization Methods:*

Fig. 5 contains several user-specified synchronization mechanisms, such as the event $A/\overline{M}\downarrow$, which causes a transition from AUTOMATIC to MANUAL. One should notice how the arrow crosses both the NORMAL and CONTROL boundary lines, thus causing the system to exit LIGHTS also and then immediately reenter it through the default entrance. The side-effect is to synchronize LIGHTS to its default state upon the $A/\overline{M}\downarrow$ event. Fig. 9 illustrates some similar possibilities: α synchronizes all orthogonal states into their default substrates; β synchronizes B into $B1$ while moving from $A1$ to $A2$; C is synchronized into $C2$ as A enters $A2$ (i.e., $A2$ is essentially used as a common state); and D is synchronized into $D2$ as C enters $C2$ (i.e., x is used as a common variable).

- *Fault Specification:*

A discussion of HDL's with respect to fault modeling can be found in [17]. Since the event-based descriptions in statecharts can be highly hierarchical, any event, including those thought of as faults, can be represented at any desired level in the hierarchy. This alleviates the need for a special effort of "planting" the test for the event in the low-level 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 diagrammatic systems, covering hierarchy, concurrency, timing aspects, and synchronization.

The main apparent disadvantage of the statechart approach is that it separates control from data. Statecharts are 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 [8]), utilizes statecharts for control and, in addition, supports a graphical language of hierarchical data-flow nature, called *activity-charts*, which is integrated with the statecharts, for the data and functional aspects of the system under description.

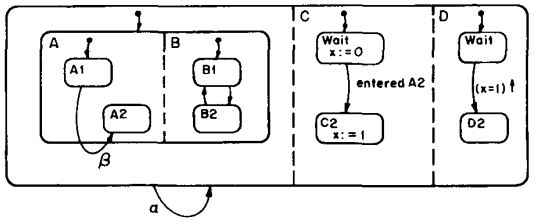


Fig. 9. Various synchronization mechanisms.

IV. The Statechart Synthesis Methodology[1]

Note: Throughout this section and the next, n is taken to be the number of states in a statechart, i.e., the size of the state tree, and d is a bound, assumed to be enforced, on the outdegree of the tree, i.e., on the maximal number of immediate substates of a state.

As discussed above, statecharts are an extension of the Moore and Mealy variants of FSM's. These have typical implementations using PLA's (programmable logic arrays) for the specification of the combinatorial logic (Fig. 10). Here, an n-state FSM is represented by its (at most) $O(n^2)$ state transitions,[2] 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 [14]) 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. Folding and partitioning techniques are often used to overcome this area blowup. However, the problem of applying such techniques after-the-fact is *NP*-Complete [5], [18]. Our methodology can be viewed as recommending that such a partitioning be carried out during the specification phase, using the designers' knowledge of the problem to generate an efficient product.

The basic idea of our synthesis methodology is to trade the concept of single machine implementing an FSM for a tree of interconnected machines implementing a statechart. Each state, at each nonatomic level of the statechart hierarchy, is represented by a machine implementing the FSM corresponding to its substates on the next immediate level. Thus for the statechart of Fig. 11, for example, fifteen small 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

[1]The methods presented in this section and the next are part of U.S. patent 4 799 141, dated January 17, 1989, granted to the authors via Yeda Research and Development Corp., Ltd.

[2]If the FSM has $k > 1$ input lines, the number of state transitions grows even larger. However, for simplicity we shall be concerned only with the asymptotic growth of the FSM implementation as a function of n.

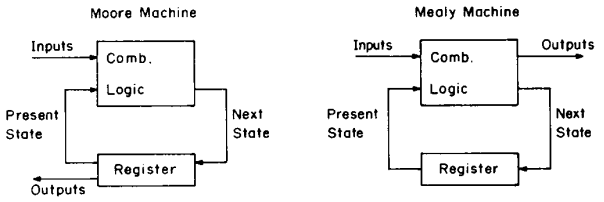


Fig. 10. Two implementations of the FSM model.

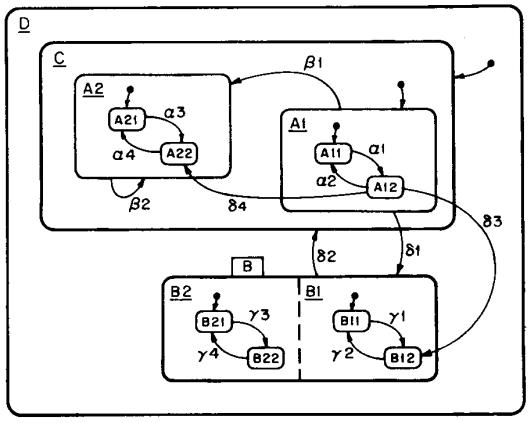


Fig. 11. A statechart example.

of the **entered** signals, and notify the lower level machines to move into their *Idle* state. The **leave** signals are created by the lower level states, to notify their predecessors about their termination (e.g., **A1** notifies ***C*** in Figs. 11 and 12). Similarly, the ***enter*** *X* signal is created by a high-level state when one of its substates (that is not the default) is required to start operating in the *X* state (e.g., ***C*** notifies **A2** in Figs. 11 and 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***.

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 (and by our convention each of these is of limited size), its cost is bounded.

Three possible layouts for the resulting tree-machine come to mind. We can use area $O(n)$ via the general algorithm of [13]. 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 better layout can be obtained using the configureable techniques of [13]. This 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 wire can really 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, and so on. The only remaining decision is where to put the "solder dots" that determine the actual connections. The method of [13] uses the fact that any finite tree with n vertices and bounded outdegree can be partitioned 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; the edges in the cut-set will appear as horizontal lines in the layout.

A even better layout, with area $O(n \cdot \log n)$, can be achieved using the 1-separator theorem for trees [13], and is illustrated in Fig. 16 for the same example. Using the theorem, the tree is bisected into two sets, each consisting of between $n/4$ and $3n/4$ vertices, by removing at most a constant number of edges. Each set is then laid out recursively along the baseline, and the removed edges are placed horizontally above.

Several timing problems can occur in the example of Figs. 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**, for example, 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 [6], [10] 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 (e.g., entering **A21** after ***C*** has moved from **A1** to **A2** in Fig. 11), or when lower level states are terminated by a high-level transition (e.g., the transition $\beta 1$ causing exit from **A11**, **A12**, and **A1**). Such timing problems become especially acute in an example such as that of Fig. 17, where the time it takes for ***D*** to get the message that it should be left might be so large as to cause a situation where ***D*** is still active after ***E*** has become active.

Such problems can be overcome in several ways. For example, we can implement each individual FSM as a combination of a Mealy machine and a Moore machine as in Fig. 18. Actually, this will be a Moore machine with asynchronous **leave**, **entered**, **enter**, and **left** signals that will propagate up or down the hierarchy using asynchronous logic rippling. It can be guaranteed that no spikes (that is, non-valid output signals, created because of several non-synchronized input signal transitions) will occur. This solution can solve the timing problems at the price of cutting down the clock frequency, so that such asynchronous signals will be able to propagate up or down the whole hierarchy in one cycle.

Now, in general, $\Omega(\sqrt{n}/\log n)$ is a lower bound for any layout technique (see [15]), meaning that there exists a constant c for which every layout will have an edge of length at least $c \cdot \sqrt{n}/\log n$. (Note that an $n \cdot \log n$ area layout is optimal for trees in which all nodes are required to be layed out on the perimeter of a convex region [1].) For the two latter layout techniques we have suggested, the distance between two nodes is at most $O(n)$. The big O constants are influenced, of course, by the maximum outdegree d allowed in the statechart, which determines both the maximal size of an elementary machine and the maximal cut-set in both the recursive procedures. Consequently, the clock period for the preceding layout techniques is $O(n \cdot \log n)$ for a balanced statechart, because of the $O(n)$ distance between nodes, and the $O(\log n)$ levels of hierarchy. For an unbalanced statechart the clock period will be at most $O(n^2)$. We should remark that the special history operator of [6] can be implemented using the local state storage devices to hold return addresses.

This statechart-based layout methodology thus offers a number of advantages, including reduced area and a natural implementation scheme for the concurrency and synchronization capabilities of statecharts. Some other apparent advantages of the approach (with either of the two layouts) are the following: (1) the input and the output wires to the control unit do not need routing inside the layout because the machines are on the perimeter; (2) the individual PLA's can be laid out in a single block in a rather regular fashion, similar to the way conventional PLA's are layed out, demanding no special solutions for the ground, voltage and clock wires; (3) each individual machine is small and fast, and the connection part consists only of wires, thus its area is not measured in machine units but rather in the usual λ unit; (4) the internal communication of [6], [10] can be layed out in the area beneath the machines, and needs no special routing inside the machine-tree layout.

We have thus seen that an n-state statechart of bounded degree can be layed out in area $O(n \cdot \log n)$, whereas the layout of an n-state FSM might result in $O(n^2)$ area. It is important to realize that, in addition, statecharts are considerably more succinct than FSM's, in fact exponentially more succinct, so that this difference is all the

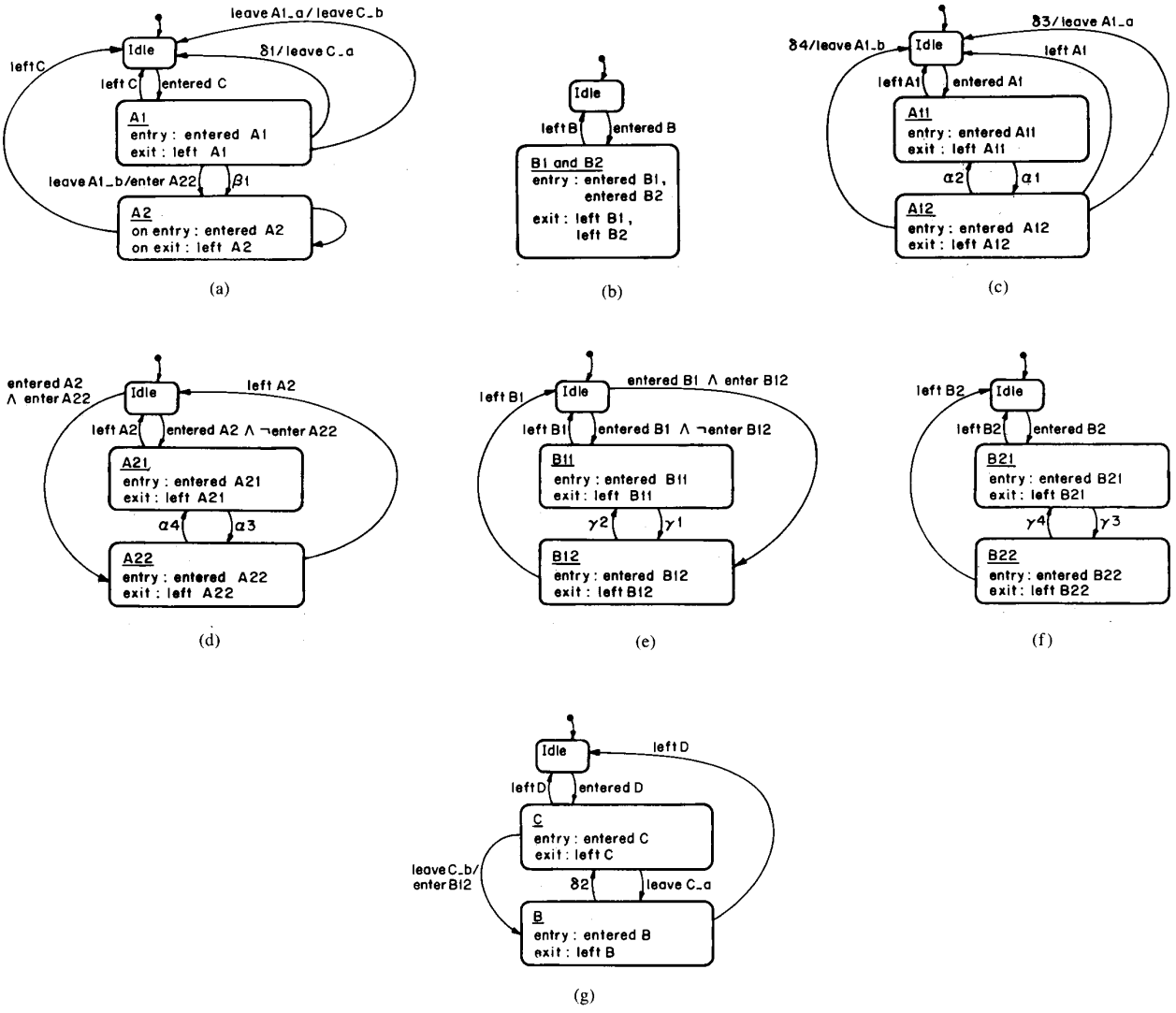


Fig. 12. The seven FSM's for Fig. 11. (a) FSM for *C*. (b) FSM for *B*. This FSM transfers the "enter *B12*" signal directly from input to output. (c) FSM for *A1*. (d) FSM for *A2*. (e) FSM for *B1*. (f) FSM for *B2*. (g) FSM for *D*.

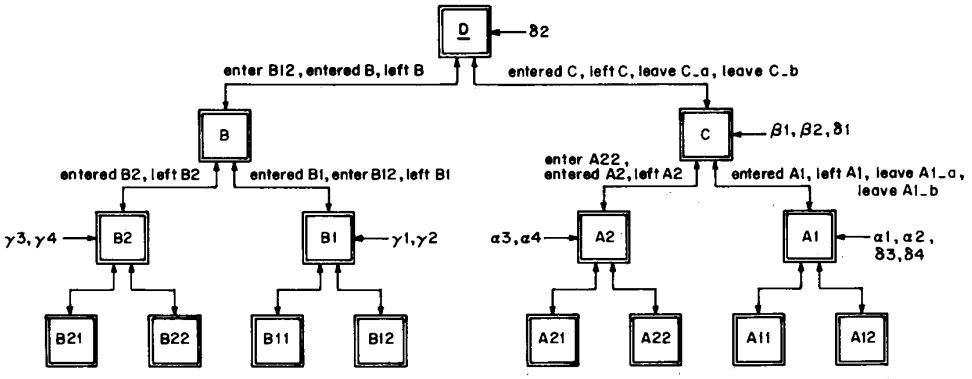


Fig. 13. Processor connection schemes for Figs. 11 and 12.

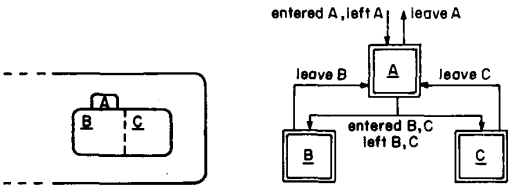


Fig. 14. Machine configuration for orthogonal states.

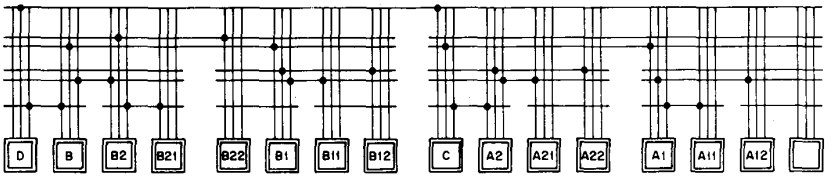


Fig. 15. Layout of processors in Fig. 13.

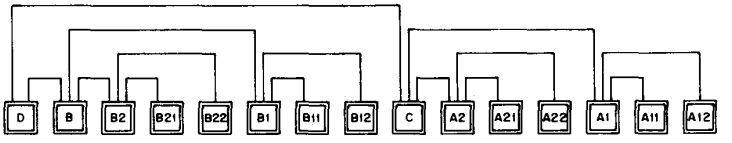


Fig. 16. A slightly better layout of the processors in Fig. 13.

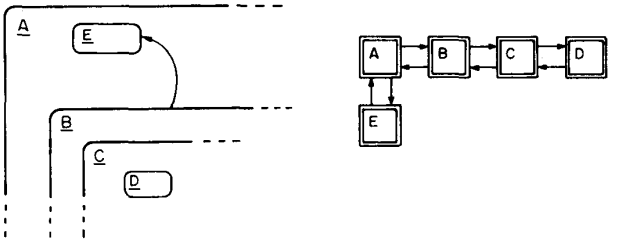


Fig. 17. An example of a timing problem.

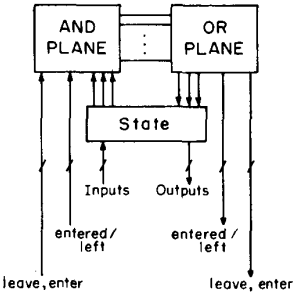


Fig. 18. A PLA implementation of a Mealy/Moore machine.

more significant. Specifically, in [2] it is shown that an n-state nondeterministic statechart can describe a problem for which the smallest deterministic[3] FSM has at least 2^{2^n} states, and an n-state deterministic sequential statechart (i.e., a statechart without orthogonality) with history states, can describe a problem for which the smallest deterministic FSM has at least 2^n states. Such succinct descriptions exploit the orthogonality and history-state features of statecharts, and our design methodology is tailored to suit these features, with the area-complexity of the layout depending only on the fact that the statechart is of bounded degree.

In [2], [3] we have investigated the theoretical background of statecharts, including a comparison of their descriptive complexity with other well-known regular-event formalisms, such as deterministic, nondeterministic and alternating finite automata, (condition/event) Petri-nets and regular expressions. Regular expressions have been suggested in [18] as a description language for VLSI, and a synthesis method is included therein. It should be noticed that, whereas all of these formalisms have the same expressive power (they all recognize the regular sets), there are significant discrepancies in their descriptive complexity. Specifically, regular expressions are, in the worst case, exponentially less succinct than deterministic FSM's [4] which are similarly exponentially less succinct than deterministic statecharts. Also, regular expressions do not enjoy many of the convenient features mentioned earlier.

V. A Programmable Approach

In this section we would like to suggest the implementing the methodology of Section IV as a programmable machine that executes statecharts. The idea is that after being programmed, the machine results in a hierarchical machine-tree similar to those found in [14]. As in Section IV, each machine implements a superstate in the statechart, and the machines connected beneath it implement its immediate substates. Here, however, the FSM for each superstate will be programmed into memory rather than being hand-wired. Like the original machine-trees of [14], these machines can be conventional von Neumann processors with local memory, data connections between them and local timers for the hierarchical timing notation of statecharts. There will also be a control connection between a machine and its descendants, consisting of the **entered**, **left**, **enter**, and **leave** signals described in Section IV. (A different possibility is to have the machines in the machine-tree be only finite state machines that produce signals to a common processing and data unit, in which case no data collections are needed.)

[3]The conventional synthesis method described at the beginning of the Section IV, suits only deterministic FSM's because it uses a log n sized state register for an n-state FSM. Our methodology, because of its "horizontal" state coding, suits the implementation of nondeterministic statecharts.

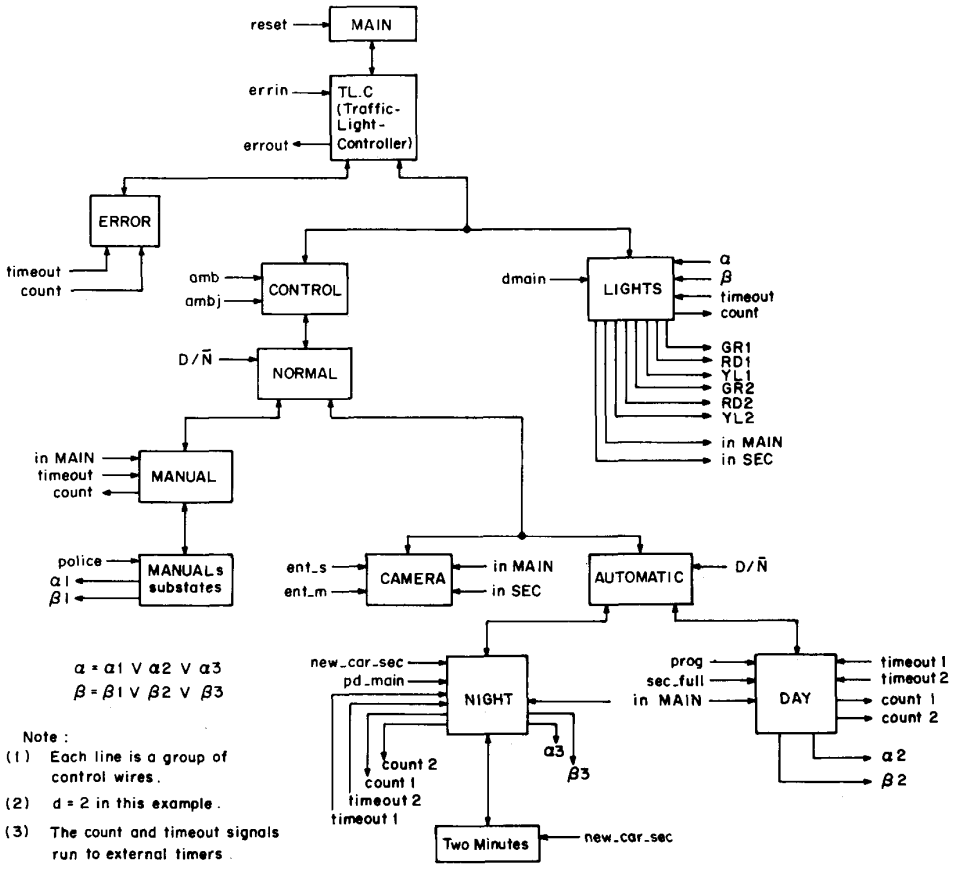


Fig. 19. The machine tree for the traffic-light-controller of Fig. 5.

Since in Section IV the machines are hardwired together, whereas here the presented machine is programmable, programmers might want to define different tree structures. Consequently, the chip's most convenient layout appears to be the $O(n \cdot \log^2 n)$ one discussed in Section IV. This technique enables the manufacturer to create general-purpose chips for which the specific machine structure can be easily determined by adding the appropriate contacts. Dynamic (run-time) machine-structure programming, which will enable the implementation of several statecharts on one machine, can be achieved using switches instead of permanent contacts. Concurrency, in the form of orthogonal states, is achieved by programming the chip so that a number of machines are mapped together as one substate. Hence, the layout technique enables a flexible machine definition including hierarchical relationships, concurrency, and even overlapping states (see [6]), dealt with by connecting a machine to a number of ancestors.

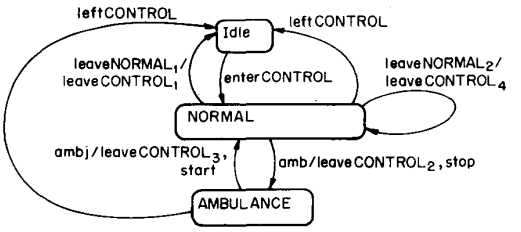
The chip is programmed in three main stages, which we merely outline here. The first consists of programming the machine structure, by defining the exact contacts (or switch-states in the dynamic version). The second stage consists of programming each of the individual machines in a standard way, and is carried out as in a conventional processor. The third and last stage, which we might call *reactive programming*, consists of generating the intra-machine signals. The last two stages are carried out in the following manner: each superstate is programmed as a single flat finite state diagram. In addition, the processor machine-language will have special commands that produce the intra-machine control signals presented in Section IV. These signals run between machines on the layout that was programmed in the first stage. All these programming details can be extracted from the statechart description automatically by a compiler.

Fig. 19 is the machine-tree for the example of the traffic-light controller of Fig. 5. Each machine is programmed in a conventional way, using a flat finite state diagram. Fig. 20 consists of two of the appropriate diagrams for Fig. 19. Each machine in our example consists of memory (containing the local finite state program) and control (that fetches the next command from the local memory). In Fig. 19 each communication line represents all control signals discussed in Section IV. Finally, Fig. 21 is the resulting layout, including the specific contact cuts for traffic-light-controller.

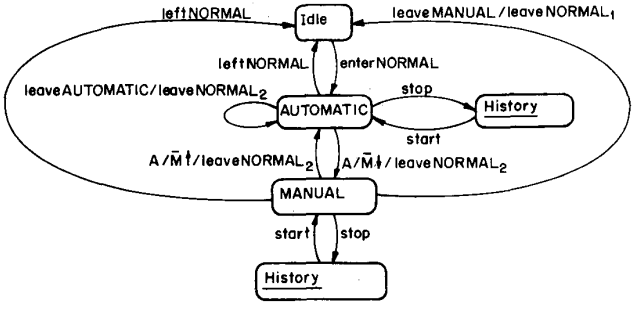
VI. Conclusion

An attempt has been made to show the benefits of using statecharts as a hardware description language and for the synthesis of digital reactive components. The three most important features appear to be information-condensing hierarchy, orthogonality (modeling concurrency), and broadcast communication. It is not clear, however, what design methodology should be used to generate succinct and "well-structured" statecharts. We feel that a well-founded design methodology might be needed to help guarantee that the user produces correct and clear statecharts. This issue requires further investigation.

Another interesting issue not treated here is the possible existence of other applications. It seems that the methodology of Section IV is suitable for synthesizing statecharts into systolic arrays (cf. [12, 13]). Hence, it is especially interesting to understand



FSM for CONTROL



FSM for NORMAL

Fig. 20. Two of the FSM's for Figs. 5 and 19.

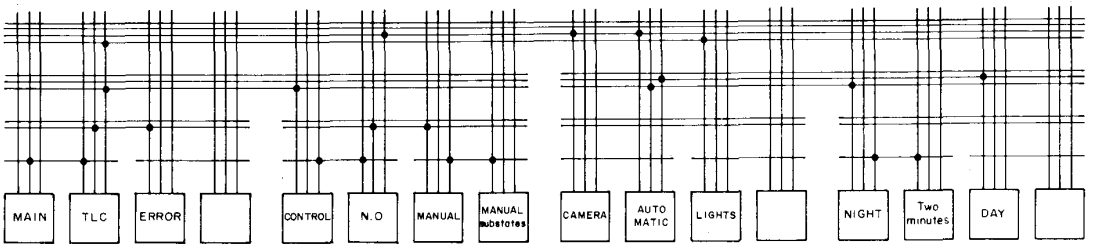


Fig. 21. The regular layout for the traffic-light-controller.

whether and how statecharts can be used as a programmning language for such concurrent systems.

Acknowledgment

The authors would like to thank A. Pnueli, S. Ruhman, A. Caspi, T. Lamdan, and M. Cohen for helpful discussions, and technical personnel from Intel and Motorola (Israel) for their helpful feedback. Y. Barbut did a superb job with the figures.

References

[1] B. P. Brent and H. T. Kung, "On the area of binary tree layouts," *Inf. Proc. Lett.*, vol. 11, pp. 46-48, 1980.

[2] D. Drusinsky, "On synchronized statecharts," Ph.D. thesis, Dep. of Appl. Math. Comp. Sci., The Weizmann Inst. of Sci., 1988.

[3] D. Drusinsky, and D. Harel, "On the power of cooperative concurrency," in *Concurrency '88*, Lecture Notes in Computer Science, vol. 335, pp. 74-103, Springer-Verlag, New York, 1988.

[4] A. Ehrenfeucht and P. Zeiger, "Complexity measures in regular expressions," *J. Comp. Syst. Sci.*, vol. 12, pp. 134-146, 1976.

[5] G. D. Hachtel, "Techniques for programmable logic array folding," in *Proc. 19th Conf. on Design Automation, ACM/IEEE*, pp. 147-155, June 1982.

[6] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Prog.* vol. 8, pp. 231-274, 1987. (Preliminary version appeared in CS84-05, Weizmann Institute of Science, Rehovot, Israel, Feb. 1984.)

[7] —, "On visual formalisms," *Comm. Assoc. Comput. Mach.*, vol. 31, no. 5, pp. 514-530, 1988.

[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring, "STATEMATE: A working environment for the development of complex reactive systems," in *Proc. 10th Int. Conf. on Soft. Eng.* New York: IEEE Press, Apr. 1988. pp. 396-406.

[9] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed., Nato ASI Series, Springer-Verlag, Berlin, 1985, pp. 477-498.

[10] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, "On the formal semantics of statecharts," in *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Ithaca, NY, pp. 54-64, 1987.

[11] C. Huizing, R. Gerth, and W. P. deRoever, "Modelling statecharts behaviour in a fully abstract way," in Proc. 13th Coll. Trees Algebra and Programming (CAAP'88); see also Lecture Notes in Computer Science, (M. Dauchet and M. Nirat, Eds.), vol. 299, Berlin, Germany: Springer-Verlag, pp. 271-294, 1988.

[12] H. T. Kung, "Why systolic architectures?," *Computer*, vol. 15, no. 1, pp. 37-46, 1982.

[13] E. L. Leiserson, *Area-Efficient VLSI Computation*. Cambridge, MA: MIT Press, 1983.

[14] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[15] M. S. Paterson, W. L. Ruzzo, and L. Synder, "Bounds on minimax edge length for complete binary trees," in *Proc. 13th ACM Symp. on Theory of Computing*, pp. 293-299, 1981.

[16] A. Pnueli, "Applications of temporal logic to the specification and

verification of reactive systems: A survey of current trends," in *Current Trends in Concurrency*. de Bakker *et al.* Eds., Lecture Notes in Comp. Sci., vol. 224, Springer-Verlag, Berlin, pp. 510-584, 1986.
[17] A. K. Singh and J. H. Tracey, "Development of comparison features for computer hardware description languages," in *Computer Hardware Description Languages and their Applications*, M. Breuer and R. Hartenstain, Eds. Amsterdam, The Netherlands: North-Holland, 1981, pp. 247-263.
[18] J. D. Ullman, *Computational Aspects of VLSI*. New York: Computer Science, 1984.

Limitations of Switch Level Analysis for Bridging Faults

ROCHIT RAJSUMAN, YASHWANT K. MALAIYA, MEMBER, IEEE, AND ANURA P. JAYASUMANA, MEMBER, IEEE

Abstract—**Switch level models are widely used for fault analysis of MOS digital circuits. Switch level analysis (SLA) provides significantly more accurate results compared to the gate level models and also avoids the complexities of circuit level analysis. The accuracy of SLA is critically examined, and conditions under which switch level analysis may generate incorrect results are specified. Such conditions may occur when the bulk of a transistor is connected to its source. These conditions are especially applicable under certain types of bridging faults. A simple technique is suggested for accurate switch level modeling under such conditions.**

I. Introduction

In the past, test generation and simulation were conducted exclusively at the gate level. Recently, however, it has been pointed out that the classical stuck-at-fault model does not represent some important failure modes, especially in the case of MOS devices. In a complex gate, the physical nodes do not directly correspond to nodes in an equivalent gate level network [1], [2]. Hence, many physical opens and shorts cannot be satisfactorily represented at the gate level. Gate level fault models, even for simple gates, can become quite complex [2]-[4]. Consideration of failure modes at the switch level or circuit level are alternatives to gate level modeling. Circuit level simulators, such as SPICE, can be used for the study of failure modes but due to the high CPU time requirement, they become impractical even for moderate sized devices. As a consequence, switch level modeling is gaining wide acceptance for fault modeling and test pattern generation of MOS circuits [2]-[7]. The following assumptions are generally used for a simple switch level analysis (SLA).

1) A transistor is an ideal switch. For an n-channel transistor an *H* (definitely recognized high voltage level) at the gate causes it to represent low resistance and an *L* (definitely recognized low voltage level) causes it to represent very high resistance. When the input is not *H* or *L* (i.e., indeterminate), the transistor presents an indeterminate resistance.
2) The resistance of the depletion load transistor in an nMOS gate is much larger than the ON resistance of an enhancement mode transistor, but much less than the OFF resistance of an enhancement mode transistor.
3) A node, when connected to both V_{dd} and ground only through high resistance paths, will retain the previous voltage level (at least for a limited time). A node connected to both V_{dd} and ground through low resistance paths will have an indeterminate voltage level.

It is sometimes possible to resolve an indeterminate situation by assuming a specific resistance ratio for enhancement and depletion type transistors. However, this makes the model more complex. Also, the resistance depends not only on the transistor dimensions, but also on the position of transistor in a network.

In this paper the problem of modeling faults in a two terminal network of n-channel or p-channel transistors is considered. The results obtained are applicable to the *n*-network in nMOS as well as the *p*-network and the *n*-network in CMOS. The results can be used in both voltage testing and current testing [8] environments. We present conditions under which SLA may generate wrong results. In the presence of a bridging fault, unexpected structures may be formed, giving rise to such situations. Techniques for accurate analysis using switch level models are suggested. For simplicity, it is assumed that all the inputs and outputs are accessible and only a single bridging fault exists.

Some terms used in this paper are defined below and illustrated in Fig. 1.

Conductance State of an n-Network (p-Network): An *n*-network (*p*-network) is *on* when it presents very low resistance between the output and the ground (V_{dd}) nodes. It is in the *off* state when it presents a very high resistance between the two nodes.

Internal Node of a Gate: A node in a gate which is neither a transistor input (gate connection) nor the power supply is an internal node. In Fig. 1, internal nodes are marked with lower case letters (*a* to *l*).

Column: A column consists of a set of serially connected MOSFET's. In a column there are no transistors (or sets of transistors) in parallel. In Fig. 1, nine columns are shown, marked 1 to 9.

Parallel Connected Columns (PCC): A structure with more than one column in parallel. Fig. 1 has four PCC's, marked I to IV.

Internal Node of a PCC: Any node in a PCC which is not common to another PCC is defined as an internal node of the PCC. In Fig. 1, nodes *a*, *b*, and *d* are internal nodes of PCC I, *f* is an internal node of PCC II, *g* and *h* are internal nodes of PCC III, and *i*, *j*, and *k* are internal nodes of PCC IV.

Conduction Path: Any path which connects the ground and the output node will be called a conduction path. Fig. 1 has the following conduction paths—*ABCFGH*, *ABCFI*, *DEFI*, *DEFGH*, *JKNOP*, *JKQ*, *LMNOP*, and *LMQ*.

Logical Node: A logical node is a logical input or an output node of a gate. In general all logical nodes are outputs of gates or complex gates.

Deterministically Testable Fault: A fault is deterministically testable if there is at least one vector which will definitely (under the switch level modeling assumptions) cause the logical output to be the complement of the output of fault free circuit.

Equivalence of Switch Level and Circuit Level Analyses: In a MOS network, if SLA generates the same conductance state as that provided by the circuit level analysis, or if the result of the SLA is indeterminate, the results are said to be equivalent. When SLA predicts an indeterminate result, the circuit level analysis often produces a definite result. In this case, SLA cannot be faulted because it does not use specific information about the parameter values. In fact circuit level analysis could sometimes be misleading because

Manuscript received October 28, 1987; revised May 18, 1988, September 1, 1988, and January 11, 1989. This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative and administered through the Office of Naval Research. The review of this paper was arranged by Associate Editor V. K. Agarwal.

R. Rajsuman is with the Department of Computer Engineering, Case Western Reserve University, Cleveland, OH 44106.

Y. K. Malaiya is with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523.

A. P. Jayasumana is with the Department of Electrical Engineering, Colorado State University, Fort Collins, CO 80523.

IEEE Log Number 8926961.

0278-0070/89/0700-0807$01.00 © 1989 IEEE