A State Assignment Procedure for Single-Block Implementation of State Charts
Doron Drusinsky-Yoresh

Abstract—State charts have been investigated recently as a powerful specification language for control structures. They extend classical finite state machines (FSM's) in several ways, catering mainly for hierarchy, concurrency, and communication. We present a novel, simple, single-block implementation scheme for state charts, which uses a single conventional combinational-logic block and a state register. The most attractive feature of the proposed scheme is the absence of communication. It eliminates the need for communicating FSM's owing to an older realization method, and does so without having to account for all state configurations implied by concurrency. We investigate the state encoding conditions for our implementation and suggest an appropriate optimization technique.

I. INTRODUCTION
Finite state machines (FSM's) have constituted one of the main formalisms underlying the prevailing approaches for the description and implementation of hardware control units. Their advantages are mainly their simple semantics as well as their simple and regular implementation scheme, based on a combinational-logic unit and a state register, illustrated in Fig. 1.

Typically, FSM's are pictorially represented by state diagrams. These diagrams, however, are inherently sequential and flat. Recently, an attempt at overcoming these limitations has been made with the advent of state charts, [7], [9], which extend state diagrams to cater for hierarchy, concurrency, and synchronization, while retaining their formality and visual nature. In a recent paper [7], we investigated their use for hardware description and synthesis.

One of the major drawbacks of state charts is the absence of a simple, easy-to-implement implementation method that will be reasonably economical in terms of VLSI resources. In this paper we investigate a novel implementation scheme based on the classical model of Fig. 1, namely, a single combinational-logic block and a state register. In Section III, we investigate the sufficient conditions for this scheme to realize state charts correctly. In Section IV we investigate the state-assignment optimization problem for this implementation scheme.

The only existing synthesis method for state charts in the literature is that of [7]. There, the suggested synthesis technique is asymptotically efficient for very large state charts, where the state chart tree out-degree is assumed to be of constant size with respect to the number of states in the state chart. It is based on a tree of communicating FSM's which is isomorphic to the state chart tree.

The major drawback of this method derives from its multiple-block implementation scheme, where several communicating FSM's, arranged in a tree structure, realize the state chart. The communication and synchronization messages within the FSM tree require considerable care to be implemented correctly. This is especially true for submicron technologies, where communication delays are dominant. Also, the old method is inefficient with respect to area and speed for small and medium sized state charts, where constant factors play an important role, and it is sensitive to other parameters (e.g., the out-degree of the state chart tree). In Section II we present a brief overview of the state chart formalism and the implementation method of [7].

The major contribution of this method is its handy implementation scheme. Having to deal only with a single combinational-logic block and a state register, we managed to make the actual low-level implementation of state charts identical to that of a conventional FSM, thus enabling the use of most CAD tools built for FSM implementation (e.g., PLA optimization techniques). This is done without having to enumerate all state configurations implied by concurrency and without any state blow-up caused by the power of high-level transitions implied by hierarchy.

Other related synthesis methods are FSM synthesis and Petri-net synthesis. FSM synthesis is related as follows. Given a state chart, one can unfold its concurrency and hierarchy and generate an "equivalent FSM," namely, a sequential state machine that will accept (and produce) the same formal language. This is done essentially by considering the state set defined by the Cartesian product of the state sets within the state chart [3]. This process can be automated so that one can consider the following alternative synthesis method for state charts: 1) convert the state chart into an equivalent FSM and 2) synthesize the FSM using existing CAD tools. However, existing results exhibit exponential lower bounds for the conversion step [8]. Consequently, this method becomes useless even when small degrees of concurrency exist in the state chart. For this reason we shall not review conventional FSM synthesis methods.

0278-0070/91$01.00 © 1991 IEEE
Petri nets (PN's) have an extensive body of literature (cf. [14]). In their finite resource versions, PN's resemble nondeterministic FSM's, which allow many simultaneous computation paths. PN's, however, do not incorporate hierarchy as part of their syntax. Consequently, there is no known method for emulating state charts on finite-resource PN's without using extensive communication. As pointed out later in this paper, the major difficulty in synthesizing state charts is due to the combination of hierarchy and concurrency. For each feature alone, there exist relatively simple implementation methods. For example, one recent implementation method for PN's [1] uses a 1-hot state assignment (also suggested in [15]) for nondeterministic FSM's, later optimized so that sets of “exclusive” places (i.e., places that are never reached simultaneously) use a logarithmic number of bits. This is essentially what our method does when the state chart incorporates concurrency without hierarchy (i.e., when it is a set of concurrent FSM’s), with the exception that in our case these sets of “exclusive” states are given syntactically by the state chart whereas for PN’s one must find them somehow. When the state chart has no concurrency, the equivalent FSM is not exponential and FSM synthesis tools are indeed relevant.

II. STATE CHARTS, LANGUAGE, DEFINITIONS, AND OLD-IMPLEMENTATION OVERVIEW

The state chart formalism was introduced in [9] as a visual formalism for specifying the behavior of complex reactive systems [7], [10]. We shall not give their formal syntax and semantics, of which several versions exist [3], [11]-[13], but rather review their behavior through the traffic-light controller example of [7]. Fig. 2(a) describes the behavior of a traffic-light controller whose I/O interface is described in Fig. 2(b). There are two sets of lights: one is positioned over the main road and the other is over the secondary road. During the day (Day = 1) the controller operates according to one of two possible programs, whereas during the night it operates according to a special night program. The controller can be operated manually as well (Auto = 0). In this mode whenever a policeman pushes the Police button, the lights alternate. A hidden camera can be operated by the controller only when it is in AUTOMATIC mode. An ambulance signal can arrive (Amb = 1), notifying the controller that an ambulance is approaching the junction. Then the lights are set according to the ambulance’s direction, and all other events are ignored. The controller can receive an error message (Errin = 1), which will cause yellow lights to flicker. Another possibility for an ERROR occurs when the controller operates manually for more than 15 min without a policeman pushing the police button. A reset signal resets the controller to the AUTOMATIC state. Note that this state chart is not given in detail; such details are available in [7], whereas a more elaborate description of the formalism is to be found in [9], and formal syntax and semantics are available in [3] and [11].

In Fig. 2(a), we have exclusive states, which can never be reached simultaneously (e.g., DAY and NIGHT), and orthogonal states, which can be reached simultaneously (e.g., AUTOMATIC and CAMERA). Note that these relations are symmetric and not transitive; e.g., DAY and NIGHT are exclusive and so are AMBULANCE and CAMERA, but DAY and CAMERA are not orthogonal. We have basic states (e.g., AMBULANCE) and superstates (e.g., LIGHTS, CONTROL, NORMAL). We use default entrances (e.g., the entry to AUTOMATIC within NORMAL). We have high-level transitions, such as the Errin transition to ERROR, which is equivalent to drawing all transitions that lead from any configuration within OPERATE to the default state of ERROR. We also have flexible concurrency, where concurrency is described at any hierarchical level, without causing sequential descriptions to become an awkward exception. Hence, in Fig. 2(a) CAMERA is orthogonal to AUTOMATIC, but LIGHTS is orthogonal to both of them, on a much higher level. Sometimes we think in terms of concurrent processes, where several states that are reached simultaneously are considered as concurrent processes. We call such an element a state configuration and denote it as a tuple of states. Note that because of the flexible-concurrency feature, the number of processes in a state configuration is not necessarily fixed. We will see in the sequel that this is the reason for naive state assignments for state charts to be incorrect. Finally, internal communication between concurrent superstates is possible; throughout this paper we shall assume that such communication is operation-
Fig. 3. (a) State chart. (b) Corresponding I/O interface. (c) State chart tree induced by the state chart of (a).

Fig. 4. FSM for A

illustrated in Fig. 3(b), with four input channels, each of which receives \( v_i \) (meaning \( v_i = 1 \)) or \( \neg v_i \) (\( v_i = 0 \)), \( i = 1, 2, 3, 4 \), and three output channels, each of which produces \( u_i \) or \( \neg u_i \), \( i = 1, 2, 3 \). Note how the state chart syntax induces an AND/OR tree, where hierarchy and concurrency are replaced with OR and AND relationships, respectively, as illustrated in Fig. 3(c). This state chart tree is frequently used for the definition of various important relations, one of which is the least common ancestor (LCA) relation, where the LCA of \( S_8 \) and \( S_9 \) is \( E \), the LCA of \( S_9 \) and \( S_{13} \) is \( D \), and the LCA of \( S_8 \) and \( S_{13} \) is \( C \). Now, two states are formally defined as exclusive if their LCA is an OR state, and as orthogonal otherwise.

Intuitively, the semantics of state charts can be understood as follows. The state chart computation visits state configurations, which are elements of a Cartesian product. For example, starting at \( S_1 \) (as indicated by the default transition), when input \( v_1 \) is received, the next state configuration will be \((S_1, S_2)\), and the output event \( u_1 \) is produced. Next, when the pair of inputs \((v_2, v_3)\) is received, the following state configuration is \((S_3, S_5)\), and the triple \((u_1, u_2, u_3)\) of output events is produced. The transitions \( S_1 \rightarrow S_3 \) and \( S_2 \rightarrow S_5 \) are said to be orthogonal. Hence, computation takes place in the form of global transitions between state configurations, global transitions which are composed of one or more orthogonal state chart transitions, each of which either encapsulates one or more transitions from basic states, owing to hierarchy, or is (recursively) a global transition.

Clearly, these examples illustrate, on the one hand, the tremendous flexibility within the language and, on the other hand, the difficulty of emulating this behavior in a simple way. Currently, more than one version of formal semantics exists for the formalism [3], [11]-[13], differing mainly in the notion of time and communication. Hence, our implementation scheme will not be based on formal semantics but rather on the intuitive behavior of the main features of state charts.

The synthesis methodology of [7] maps the state chart tree onto an isomorphic tree of communicating FSM's, one FSM for each superstate. Such an implementation of the state chart of Fig. 3 is illustrated in Fig. 4. Each FSM implements a single superstate and its immediate substates. Hence, FSM \( D_i \) implements an FSM with two states, \( E \) and \( S_{11} \), and an extra idle state, whereas FSM \( E \) implements an FSM with four states, \( S_4, S_5, S_{10}, \) and idle (see [7]).
Later the FSM tree is laid out using well-known layout techniques for trees. Hence, this synthesis methodology is efficient when \( d \), the state chart tree out-degree, is constant with respect to \( n \), the number of states within the state chart. This, of course, is true only if \( n \) is extremely large. Also, special communication signals are generated for transitions that cross state boundaries, such as the transition from \( S_5 \) to \( S_6 \). Naturally, such communication overhead degrades performance significantly. The biggest drawback of this implementation scheme, however, is its complexity. (The FSM tree requires special attention for the correct implementation of the inter-FSM communication and synchronization, which is a nontrivial design problem.)

III. THE SUGGESTED SINGLE-BLOCK IMPLEMENTATION

In order to reduce the variety of possible transitions within a state chart, we use the abstraction step illustrated in Fig. 5; namely, we replace high-level transitions with a primitive form of concurrency. Both state charts in Fig. 5 are equivalent; in the original state chart the \( \alpha, \beta \) transitions take place from either \( A_1 \) or \( A_2 \), whereas in the transformed one they take place concurrently with the activity within \( A \), which has the same final effect. This is true no matter how high the states are within the hierarchy; we simply add an orthogonal state to the superstate \( A \) (the original source of the high-level transition), which consists of precisely one basic substate \( A' \), and change the corresponding high-level transition so as to depart from \( A' \). Also, transitions will always be used with basic states as their target states (e.g., instead of \( D \rightarrow D \) we will use \( D \rightarrow (S_5, S_1) \)). These two transformations are done for every transition in the state chart. Fig. 6 is the basic structure of the transformed state chart equivalent to that of Fig. 3. Hereafter, we shall refer only to the transformed state chart, and we will omit the prime.

Consequently, there is now only one general type of state chart transition, namely, a quadruple \((X, Y, \alpha, \beta)\), where \( X \) and \( Y \) are tuples of basic states, representing the transition’s source and target, respectively, \( \alpha \) is the transition’s label (input), and \( \beta \) is the output produced when this transition is traversed. The transition’s source and target are sub-tuples of a state configuration (in our example, \( (E, S_3) \) is a sub-tuple of \( (E, S_3, S_1) \)).

Now, consider the FSM emulation method of Fig. 1. Here, there is an isomorphism between the FSM state set and the code words stored in the state register. Also, the FSM transition function is emulated by the combinational logic. Obviously, if we try to emulate a state chart in the same way, we might need an enormous combinational-logic block to emulate an exponential number of global transitions between state configurations induced by concurrency. Hence, we shall emulate all state chart transitions in the combinational-logic block in a way which will preserve the global behavior of the state chart. Our implementation scheme is illustrated in Fig. 7. The combinational logic implements the original state chart transitions in the conventional way; a point on the LSI grid which represents a transition (i.e., the value at this point is 1 iff the transition is traversed) is called a term. For example, in the PLA of Fig. 8(a), rows 1 through 24 represent 24 terms.

Based on Fig. 7, a state chart implementation consists in implementing all terms for all state chart transitions, in a way that is similar to the conventional FSM implementation. For a transition \((X, Y, \alpha, \beta)\), the source block contains a conjunction of the code words that represent the basic states within \( X \) (and of \( \alpha \), the inputs).

For a given basic state (or a given output), the target block contains a union of all terms that represent transitions whose target tuples contain this basic state.

State assignment is the crucial step within this implementation if one wants to implement both hierarchy and concurrency correctly. We shall further elaborate this point through a sequence of three abstractions. First, consider the extremely restricted case where state charts are restricted to be FSM’s only (i.e., no concurrency or hierarchy is permitted). In this case, any \( \log n \) bit unambiguous code assignment suffices for a correct implementation of the model of Fig. 1. Next, Assume that state charts are permitted to have concurrency, but no hierarchy. In other words, consider

---

All methods presented in the following sections are protected by a pending patent.
only state charts that consist precisely of m concurrent FSM's. In this case we can simply derive a state assignment for each FSM individually, and then for every state configuration, concatenate the appropriate individual code words into one wider code word. Note that in this case, an unambiguous individual state assignment for each FSM will suffice. Finally, we end our abstraction process in the general case where both hierarchy and concurrency exist. Now, not every such individual state assignment is correct. Consider a state assignment where 0 and 1 are assigned to \( Y_0 \) to represent \( S_2 \) and \( S_3 \), respectively, whereas 00, 01, and 10 are assigned to \( Y_1 \) and \( Y_2 \) to represent \( S_4 \), \( S_{15} \), and \( S_{17} \), respectively, in Fig. 6. Now, when the transition \( S_4 \rightarrow (S_5, S_6, B) \) is traversed, \( Y_1 \) and \( Y_2 \) are not affected by the target block, and might be assigned an erroneous code. In fact, when a PLA is used, this code will be 00, so \( S_{17} \) will become part of the next configuration. Hence, as discussed in the introduction, the main cause of difficulty is that because of hierarchy, state chart concurrency is flexible, so one cannot naively assign a fixed portion of a code word to each process. For this reason, state chart state assignment is not a naive extension of an FSM state assignment, as in the restricted case discussed earlier.

We shall distinguish between state assignments and configuration assignments. A configuration assignment is a code that represents state configurations, whereas a state assignment is a code that maps basic states to binary strings. Our goal is to find a state assignment such that the induced \( n_0 \)-bit configuration assignment will be unambiguous. We define the \( \cup \) operator to extend the binary "or" operator with \( \forall a \in \{0, 1, \text{tri} \} \): \( a \cup \text{tri} = a \). Now, for a well-defined state assignment, we require the following state chart state assignment conditions:

1) There exists a 1–1 function \( \rho: \) basic states \( \rightarrow \{0, 1, \text{tri} \}^{n_0} \) triangular strings, where \( \text{tri} \) in coordinate \( j \) means that state variable \( Y_j \) does not depend on the current basic state; \( \rho \) is considered the state chart state assignment.

2) There exists a 1–1 function \( \tau: \) state configurations \( \rightarrow \{0, 1\}^{n_0} \); this is the (conceptual) configuration assignment. Hereafter, \( \tau((S_1, \ldots, S_j)) = \rho(S_1) \cup \cdots \cup \rho(S_j) \).

The first condition describes the nonambiguous state assignment we are looking for, whereas the second condition guarantees that the configuration assignment is nonambiguous and (because it per-
mits binary strings only) guarantees that code words, in their entire length, describe legal configurations in a precise way. Hence, these conditions ensure that inductively, as computation proceeds, the binary code words of length \( n_b \) represent, in a unique and precise way, only legal state configurations. It is the responsibility of the combinational logic to implement the induction correctly with the correct implementation of terms as described earlier. Stated informally, a well-defined state assignment \( \rho \) is such that when code words of elements of a state configuration are summed (\( \cup \)), the resulting configuration assignment \( r \) is well defined (i.e., binary) and unambiguous. Hence, in the previous example, the 3-bit prefixes of \( p(S_i) \), \( p(S_j) \), and \( p(B) \), are \( 000, 000 \), and \( 000 \), respectively; thus the 3-bit prefix of \( r(S_i, S_j, B) \) is \( 000 \) which is not well defined.

A trivial well-defined state assignment resembles the well-known FSM 1-hot state assignment. Here, \( n_b = n \), where \( n \) is the number of basic states in the state chart (e.g., in Fig. 6, \( n = 21 \)). We assign to \( \rho(S_i) \) a tuple of \( n_b \) ternary symbols, where 1 is assigned to the \( i \)th variable, 0 is assigned to the \( j \)th variable for every \( j \neq i \) such that \( S_i \) and \( S_j \) are exclusive, and \( 0 \) is assigned to every other symbol. For our example, the 1-hot state assignment for \( S_{11} \) and \( S_{12} \) are, respectively,

\[
0000000000000000000000000
\]

(where we consider \( B, C, D, E \) to be \( S_{16}, S_{19}, S_{35} \), and \( S_{21} \), respectively).

**Theorem 1:** The 1-hot state assignment satisfies the state chart state assignment conditions.

**Sketch of Proof:** Clearly \( \rho \) is 1–1. That \( r \) maps configurations to \([0,1]^m\) follows from the fact that for every configuration \( c = S_i, \ldots, S_j \) and every \( i \leq n_b \), the \( i \)th symbol of the code word \( \rho(S_i) \) for some \( S_j \), \( j \leq k \), is not 0. This is because either \( S_j \) is in \( c \) or it is exclusive to some such \( S_j \) in \( c \); otherwise the configuration is not "complete" (some concurrent process is not described in \( c \)).

Consider any two different configurations \( c_1 \) and \( c_2 \). Clearly, there must be two exclusive basic states \( S_i \) in \( c_1 \) and \( S_j \) in \( c_2 \); hence \( r(c_1) \) and \( r(c_2) \) differ in the \( i \)th and \( j \)th symbols. Thus, \( r \) is 1–1. Q.E.D.

**IV. EFFICIENT STATE ASSIGNMENTS**

As with conventional FSM state assignment, where the total implementation area is optimized (e.g., [2]), we wish to optimize the total area consumed by our implementation. This includes two main aspects: minimizing the number of terms and minimizing the number of state variables. Term minimization is outside the scope of this paper mainly because it seems to be tightly coupled with state chart minimization, which has not yet been solved.\(^3\)

Hence, we shall concentrate on the minimum state-encoding problem, where a state assignment is sought that is of minimum width (i.e., \( n_b \) is minimum). This state assignment should be constrained to satisfy the state assignment conditions described earlier.\(^3\) We do not know how to solve this problem in general. Instead, we shall investigate one approach, the *exclusivity encoding state assignment*. Generally speaking, it is identical to the 1-hot assignment, except that sets of basic states, called exclusivity sets, are coded with a logarithmic number of bits. Formally, we split the set of basic states into \( m \) pairwise disjoint exclusivity sets \( R_1, \ldots, R_m \), where each such set consists of states that are pairwise exclusive. For the example of Fig. 6, we can divide the basic sets into \( \{ S_1, S_2, S_3, S_4, S_5, S_6, S_7 \} \) and \( \{ S_8, S_9, S_{10}, S_{11}, S_{12}, S_{13}, S_{14} \} \), \( \{ S_{15}, S_{16}, S_{17} \} \), \( \{ B, E \} \), \( \{ D \} \), and \( \{ C \} \). Now we encode all states that belong to a common set \( R_i \) with \( \log |R_i| \) bits instead of the \( |R_i| \) bits used by the 1-hot assignment. This reduction is possible because all states in each such set \( R_i \) (i.e., 1, \ldots, \( n_b \)), are pairwise exclusive. These groups of bits are then concatenated to form a state configuration code word. Hence, in the above example, we use 4, 3, 2, 1, 0, and 0 bits, respectively. However, a subtle point now emerges. Consider, in the above example, states \( S_{13} \) and \( S_2 \). They are exclusive, and in the above decomposition they belong to different exclusivity sets. As a result, when the present configuration is \( S_{13}, S_2 \), the pair of bits representing the exclusivity set \( \{ S_{13}, S_{12}, S_{121} \} \) should represent a dummy state so that, say, \( S_{13}, S_a, S_{121} \) will not be represented. Formally, for the two exclusivity sets \( R_1 \) and \( R_2 \), when there is a basic state in \( R_i \) which is part of a state configuration \( c \) in which no basic state of \( R_j \) is included, then \( R_j \) needs a dummy state to ensure its exclusivity from other elements of \( c \). Hence, when the present configuration is \( S_{13}, S_2 \) the pair of bits representing \( \{ S_{13}, S_{12}, S_{121} \} \) should hold the code for a dummy state.

Once such exclusivity is guaranteed, \( r \) becomes 1–1, and we can conclude.

**Theorem 2:** The exclusivity encoding state assignment method satisfies the state chart state assignment conditions.

The *exclusivity encoding optimization problem* (EEOPl) is the appropriate optimization problem, namely, the problem of splitting the basic states into pairwise disjoint exclusivity classes \( R_1, \ldots, R_m \), such that

\[
\sum_{i=1}^{m} | \log |R_i| | \text{ is minimum, taking into account the need for dummy states as well.}
\]

We have investigated a simpler

\[
\sum_{i=1}^{m} \log |R_i|
\]

where the need for dummy states is not considered. Note that the solutions for both problems do not necessarily unify. For example, the decomposition given above is EEOPl optimal for our example, although not EEOPl optimal. This is evident from the state assignment in Fig. 8(a), where we have moved \( S_1 \) and \( S_2 \) from the first set to the second and forth sets, respectively, and thus saved one state variable.

We say a state chart is \( m \)-concurrent if \( m \) is the maximal dimension consumed by any configuration. For example, the state chart of Fig. 6 is 6-concurrent. The dimension \( m \) is defined recursively over the state chart tree as

- for a basic-state \( S, m(S) = 1 \);
- for an \( n \) state \( S \) with substates \( S_1, \ldots, S_n, m(S) = \max \{ m(S_1), \ldots, m(S_n) \} \);
- for an \( m \) state \( S \) with substates \( S_1, \ldots, S_m, m(S) = \sum_{i=1}^{m} \log |R_i| \).

The following lemmas reveal two important properties of EEOPl.

**Lemma 3:** An optimal solution for EEOPl for an \( m \)-concurrent state chart consists of \( m \) exclusivity sets.

\(^3\) In [5] there is a minimization theorem for hierarchical FSM's that are state charts without concurrency.

\(^4\) It might be the case that a shorter state assignment exists for a sequential machine that enumerates all possible state configurations explicitly.
Sketch of Proof: Clearly, at least \( m \) exclusivity sets are required, one for each basic state in the configuration that has \( m \) processes. If more than \( m \) sets are used, then one set \( R \) is redundant in the sense that for every basic state \( S \in R \) there is an exclusivity set \( R_i \) to which \( S \) can be appended (otherwise the system is not \( m \)-concurrent but has a higher degree of concurrency, thus reducing the total cost).

Q.E.D.

As a consequence of Lemma 3, given an \( m \)-concurrent state chart, an algorithm for solving EEOP1 consists of finding \( m \), and then finding the \( m \) largest exclusivity sets.

Given a state chart and/or tree \( A \), we define a trace to be a subtree \( A' \) that contains \( A \)'s root, and satisfies the following:

1) If \( S \) is an or state in \( A' \), then all of \( S \)'s substates in \( A \) are its substates in \( A' \).
2) If \( S \) is an and state in \( A' \), then one of \( S \)'s substates in \( A \) is its only substate in \( A' \).

The following lemma follows.

Lemma 4: Given a state chart \( S \), the set of leaves of a trace of \( S \) forms an exclusivity set, and every exclusivity set within \( S \) is a subset of the set of leaves of some trace of \( S \).

Sketch of Proof: First, note that the least common ancestor of any two basic states (i.e., leaves) which belong to a common trace is always an or state, and that the least common ancestor of any two basic states which do not belong to a common trace is always an and state, so the first part of the claim follows. For the second part, given an exclusivity set \( E \), consider the subtree \( T \) of the state chart tree whose set of leaves is precisely \( E \). Clearly, an and state vertex in \( T \) can have only one substate because elements of \( E \) are pairwise exclusive. Hence \( T \) is a subtree of a trace of \( S \), with a common root, so the second claim holds as well. Q.E.D.

Consequently, finding a maximum-cardinality exclusivity set is equivalent to finding a trace with a maximum number of leaves. Such an algorithm is induced by the following recursion for the computation of a trace \( T(S) \), with a maximum number of leaves, \( N(T(S)) \):

- For a basic state \( S \), \( T(S) = S \), and \( N(T(S)) = 1 \).
- For an or state \( S \) with substates \( S_1, \ldots, S_n \), \( T(S) \) is the tree whose root is \( S \) with substates \( S_1, \ldots, S_n \), in which turn are roots of the subtrees \( T(S_1), \ldots, T(S_n) \), respectively; \( N(T(S)) = \sum_{i=1}^{n} N(T(S_i)) \).
- For an and state \( S \) with substates \( S_1, \ldots, S_n \), \( T(S) \) is the tree whose root is \( S \) with one substate \( S_i \) such that the subtree \( T(S_i) \) rooted at \( S \) is the subtree with a maximum \( N(T(S_i)) \); \( N(T(S)) = \max_{i=1}^{n} N(T(S_i)) \).

V. DISCUSSION

Fig. 8(a) is a PLA implementation of the state chart of Fig. 3, according to the decomposition described previously. Fig. 8(b) shows the mapping between state chart transitions in Fig. 3 and the terms of the PLA. The PLA consumes an area of \((16 \times 2 + 15) \times 24 = 1128 \) transistors, and the state register consumes an approximate area of \( 30 \times 12 = 360 \) transistors. Hence the total implementation area is 1488 transistors. The 1-hot implementation consumes a total of \((26 \times 2 + 25) \times 24 + 30 \times 22 = 2508 \) transistors.

An interesting aspect of the implementation has to do with unspecified inputs. Assume, for example, that in Fig. 3 the present configuration is \( (S_5, S_3) \) and that the input \((v_1, v_2, v_3, v_4) \) has been received. Clearly one would expect the next configuration to be \((S_5, S_5) \). All terms of Fig. 8(a), however, will specify \( Y_1 \).

VI. CONCLUSION

We have presented a simple, single-block implementation methodology for state charts, together with an appropriate optimization technique. Our methodology does not require communication and synchronization between FSM's, which makes it easy to implement and verify and allows the use of existing CAD tools (e.g., PLA optimization for the combinational logic generated by this method). Although asymptotically inferior, our scheme is expected to be superior to the previously known method for the synthesis of state charts with a state chart tree out-degree that is not constant with respect to the number of basic states, namely, for small and medium-sized state charts and even some large ones. Also, its efficiency does not depend on the number of edges that cross state boundaries.

REFERENCES

Decision Problems for Interacting Finite State Machines

Doron Drusinsky-Yoresh

Abstract—Given a system of \( n \) interacting finite state machines (FSM’s) and a state configuration, the reachability problem is to examine whether this configuration is reachable within the system. We investigate the complexity of this decision problem and three of its derivatives, namely 1) verifying system determinism, 2) testing for the existence of unspecified inputs to any FSM within the system, and 3) testing for the exclusiveness of two intra-FSM signals. We prove that these problems are all \( \text{PSPACE} \)-complete. We show the effect of these problems on the state assignment process for concurrent systems of interacting FSM’s.

I. INTRODUCTION

Logic synthesis of sequential finite state machines (FSM’s) is a well-developed field of knowledge. There is a massive body of research in this area, originating in the fundamental research of Stearns and Hartmanis [7], [8]. See, for example, [1] and [6].

In this paper we examine logic synthesis of concurrent FSM’s. It is our belief that concurrent FSM’s will be increasingly used in the future, as exemplified by the following three scenarios:

1. Consider a real-time control system; here, the real-time constraints impose hardware concurrency of several, perhaps communicating, sequential controllers where, typically, each controller is modeled as an FSM.
2. Consider a design task which is divided between design groups, where each group designs a designated subsystem. When several such subsystems are individually controlled by a finite state mechanism, the whole system is conceptually controlled by a system of communicating concurrent FSM’s.
3. Finally, consider the process of silicon compilation of a behavioral hardware description language (HDL). Conventionally, such a compilation output is an FSM for a control mechanism that generates the sequencing instructions for the controlled data path. Naturally, with the advent of higher level HDL’s (e.g., VHDL), and for increasingly sophisticated designs, the controlled data path might be inherently concurrent and hierarchical, requiring a network of concurrent FSM’s to control it efficiently. Also, it seems quite natural to expect a concurrent behavioral description to be implemented by many controlling FSM’s, perhaps one for each sequential process in the high level specification.

Hence, it is important to examine existing, predominantly sequential logic synthesis methodologies in the concurrent realm. To date, this has not been thoroughly done.

In this paper we reexamine the well-known state assignment methodology in this context. In Section II, we examine three implicit assumptions made by conventional state assignment tools and review them in the light of concurrency. We describe appropriate decision problems and analyze their complexity in Section III.

II. MOTIVATION: STATE ASSIGNMENTS FOR SEQUENTIAL AND INTERACTING FSM’S

Consider the FSM of Fig. 1(a) and its PLA implementation of Fig. 1(b). This PLA was generated by NOVA [13], a well-known state assignment program, and embodies three typical assumptions made by such a program:

1) Determinism (DET): the FSM is assumed to be deterministic; i.e., for every state and every input there is at most one next state.

2) Unspecified input (UT): if at some state there is an input configuration that causes no next state (it triggers no transition; e.g., \( (q, \alpha, \beta) = (T, F) \) in state \( s_3 \) of Fig. 1(a)), then it is assumed to be an “impossible” input for this state. In other words, it is assumed to be the designer’s responsibility to verify that such an incident does not occur. This enables the state assignment program to exploit the free space for optimization; hence, the PLA of Fig. 1(b) generates \( s_2 \) as the next state in the above case. Note that in the conventional FSM implementation scheme, where a state register is connected to a combinational logic block, the combinational logic always produces a (perhaps erroneous) next state.

3) Input and output orthogonality (IOO): the FSM is assumed to have orthogonal (independent) inputs and outputs, i.e., all combinations of signal values over the IO wires, except those found to be impossible earlier, are possible. Hence, in Fig. 1(a) all four input configurations of \( (\alpha, \beta) = (T, T), (T, F), (F, T), \) and \( (F, F) \), are possible, except for \( (\alpha, \beta) = (F, T) \) in state \( s_3 \). This assumption is self-evident in the single-FSM case, where the IO signals are connected to an unex-