



The United Nations
University

UNU-IIST

International Institute for
Software Technology

Asynchronous Controller Design

Li Xiaolei and J W Sanders

May 2007

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research **[R]**, Technical **[T]**, Compendia **[C]** or Administrative **[A]**. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director



The United Nations
University

UNU-IIST

**International Institute for
Software Technology**

P.O. Box 3058

Macao

Asynchronous Controller Design

Li Xiaolei and J W Sanders

Abstract

This paper introduces a new method for the synthesis of gate-level asynchronous controllers. The method is based on a user-level specification formalism, the path model, and a simple algebraic tool, the signal calculus, that in combination provide a formalism for the specification and analysis of path-model designs. The path model view of communication is state-based as in other asynchronous finite-state-machine methods; however it focuses on the critical component of state machines, the paths, and ignores other non-critical free states. The signal calculus, a temporal lifting of Boolean logic, helps to formalise path-model specifications algebraically, removing much of the inadequacy of traditional tabular tools like flow tables, with their dependency on table cells that is exponential in input size. The method is demonstrated on two examples.

Li Xiaolei is a DPhil student in the Programming Research Group at the University of Oxford. **Jeff Sanders** is Senior Research Fellow at UNU-IIST, having recently joined from the Programming Research Group at Oxford. His interests lie largely in Formal Methods.

Contents

1	Introduction	1
2	The Design of AFSMs	1
2.1	Hazards	2
2.1.1	Race between secondary signals	2
2.1.2	Race between primary and secondary signals	3
2.2	Traditional synthesis methods	3
2.2.1	Fundamental mode Huffman circuits	3
2.2.2	Extension of Huffman machines	4
2.3	Recent developments	4
3	The signal calculus, SC	6
3.1	Signals	6
3.2	Boolean operations	7
3.3	Delay	7
3.4	Differentiation	8
4	Path-model specifications	8
4.1	Specification	8
4.2	Definition formalised	10
4.3	Specification in SC	13
5	Implementation overview	14
6	Synthesis method	15
6.1	Environment models	16
6.2	Sequential equation	16
6.3	Normal form	17
6.4	Modelling of input transitions	18
6.5	Modelling of state variables	19
6.6	Assignment of symbolic states	20
6.7	Encoding of secondary signals	23
7	Examples	24
7.1	An example of Yun	24
7.2	Example ISEND	25
8	Conclusion	28

1 Introduction

The multitude of asynchronous controllers designed in the last 30 years or so fall roughly into three design styles: (1) translation methods; (2) Petri-net, or graph-based, methods; and (3) asynchronous finite state machines (AFSM) methods. The method presented here falls essentially into (3) [14] and consequently shares many features with other AFSM methods such as those presented in [11, 16]. Its main distinguishing feature is that it is algebraically based, relying on an algebraic formulation of signal calculus [12], a form of temporal Boolean algebra. The paper focuses on introducing the new model and its use in identifying and removing sequential hazards; combinational hazards are not considered.

Section 2 briefly covers related work in the design of AFSM controllers; Section 3 summarises what is required of the signal calculus; Section 4 describes the new user-level specification model; Section 5 discusses implementation styles; Section 6 presents the new synthesis method in a top-down fashion; Section 7 presents two examples to illustrate the method; and Section 8 concludes.

2 The Design of AFSMs

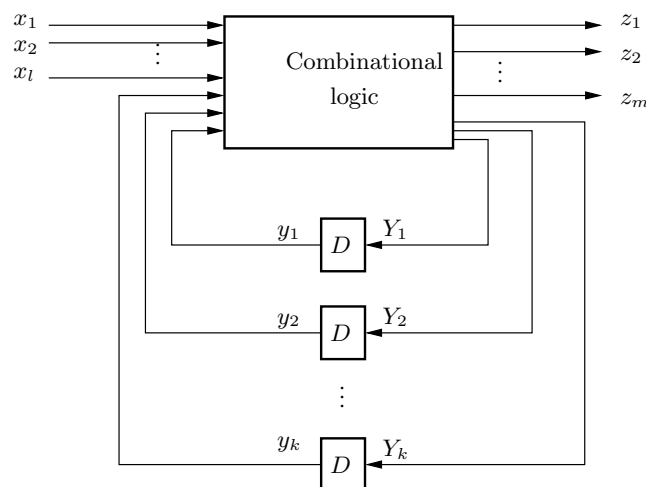


Figure 1: A typical asynchronous sequential circuit.

The traditional approach to the design of controllers, adopted in this paper, views a concurrent system as a finite state machine. In its simplest form, an AFSM is a Huffman machine (see Figure 1) consisting of *primary inputs*, *combinational logic*, *primary outputs* and *feedback state variables* (also called *secondary inputs*) that dynamically capture state [14]. No latches or flip-flops are required since state is stored in feedback loops which may incorporate delay elements. The view of computation in this approach is state-based: a machine is in some state, it receives inputs, generates outputs, and moves to a new state. Such specifications are naturally described

by *flow tables* or *state tables* [14] which define the behaviour of outputs and the next state as a function of the inputs and the current state.

AFSM methods typically follow the same general approach used in synchronous design: the system is initially specified by a flow table which is reduced through *state minimisation*; each row of the reduced flow table is assigned a symbolic state; binary codes are assigned to the symbolic states using *state assignment*; Boolean functions for state variables and outputs are derived from the state table in an enumerative fashion; the Karnaugh map is then employed to manipulate the Boolean functions in order to remove combinational hazards; and finally, the Boolean functions are implemented using simple gates. However, asynchronous machines are more difficult to design following that method because of the absence of a global clock and presence of hazards.

2.1 Hazards

The elimination of all hazards from an asynchronous design is a difficult problem. Many existing design methods do not guarantee freedom from all hazards; others require harsh restrictions on input behaviour (single-input changes only) or restricted implementation style (the use of large, slow inertial delays) to ensure correct operation. Hazards are classified as being combinational or sequential, according to where they occur.

Sequential hazards

In sequential circuits, the hazards introduced by the feedback state variables are called *sequential hazards*; they exist regardless of the correctness of the underlying combinational circuit. There are two types of race condition that can cause sequential hazards: a race between secondary signals and a race between primary and secondary signals.

2.1.1 Race between secondary signals

Secondary signals change as a consequence of input transitions. In traditional AFSMs [6, 14], a *critical race* is said to be present if the final state to be reached after the transition depends on the order in which the state bits change. But due to arbitrary gate and wire delays, there is no way to control which state bit changes first; thus when several state variables change simultaneously, such a circuit may behave unpredictably. Hence critical hazards are obviously undesirable. In recent burst-mode and 3D machines [16, 17, 18], a critical race is referred to as a manifestation of a function hazard during a state burst.

There are two conventional solutions to that problem. The first uses special encodings [14] to assign secondary signals to the internal states of a sequential circuit so that for each state tran-

sition only one secondary variable changes, effectively eliminating the race. The second involves a systematic procedure [14, 8, 13] that uses extra state bits to ensure freedom from the critical-race condition and allow multiple changes of secondary signals to take place simultaneously. For further details see Section 6.

2.1.2 Race between primary and secondary signals

A hazard which results from a race between an input signal and a secondary signal, due to the same input changes at the input of the combinational logic, is known as an *essential hazard*. (The race occurs when the changes on the secondary signals arrive at the input of the combinational logic before it has completely assimilated the input changes.) In early Huffman machines, since single input changes (SICs) are assumed, essential hazards can be avoided by inserting sufficient delays in the feedback paths so that the combinational logic has enough time to stabilise after an SIC and before the secondary signals go through the delays in the feedback loop. In recent burst-mode and 3D machines where multiple input changes (MICs) are allowed, the extra restriction is imposed that no output can change until all the inputs have arrived. This extra restriction is necessary because transitions of an MIC are allowed to occur at any time and in any order; therefore, no feedback delays can guarantee that the secondary signals will not go back until after all the input transitions have occurred. Without that restriction it is possible for a transition to arrive later than the secondary signals. Later it will be shown how circuits are synthesised with this restriction in effect.

2.2 Traditional synthesis methods

To avoid both combinational and sequential hazards, a number of design methodologies have been developed. The circuits developed using these methods operate in different modes.

2.2.1 Fundamental mode Huffman circuits

The earliest asynchronous state machine implementations are *Huffman machines* [14]. In this model, circuits are designed in much the same way as synchronous circuits. The circuit's environment is assumed to ensure SIC, so combinational hazards are easily removed. Delays are added to the feedback path and specialised encoding schemes used to remove the sequential hazards. The final requirement in this mode is that the next external input transition cannot occur until the machine has settled into a stable state, and this restriction is what characterises a *fundamental-mode* circuit.

2.2.2 Extension of Huffman machines

The assumptions made in Huffman machines, while making logic design easy, significantly increase the cycle time and restrict concurrency. As a result, there have been several attempts to remove restrictions on Huffman machines for better performance.

A MIC machine [4] allows several inputs to change concurrently. Once the inputs change, no further inputs may change until the machine has stabilised. This approach allows greater concurrency, but its use in a concurrent environment is limited since there it may be necessary for input changes to be nearly simultaneous.

An *unrestricted-input change* (UIC) machine [15] allows arbitrary input changes, provided that no single input changes more than once in some given time interval. This design is not currently practical because it requires the use of large inertial delays and has not been proven to avoid metastability problems.

Another method, described by Hollaar [5], uses detailed knowledge of the implementation strategy to allow new transitions to arrive earlier than the fundamental-mode assumption allows. Although this method may ease much of the fundamental restrictions in some circuits, the original basic implementation strategy is faulty, in that not all hazards can be removed, and it is expensive both in terms of space and time.

2.3 Recent developments

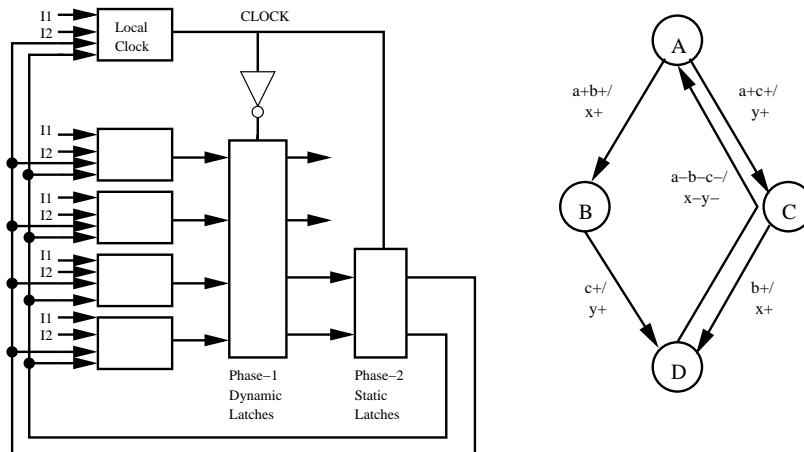


Figure 2: Circuit schematic for burst-mode circuits (left), burst-mode specification (right).

Much of the recent work on asynchronous state machines has centred on *burst-mode machines* developed by Nowick, Yun, and Dill [9, 10, 11] based on earlier work at HP laboratories by Davis, Stevens, and Coats [2]. These machines are similar to Huffman machines, but have a

local synchronisation unit that acts like a clock on the machine's latches or flip-flops. The clock is aperiodic, being generated as required by the given computations.

As shown in Figure 2 (right), burst-mode circuits are specified via a standard state-machine, where each arc is labelled using a non-empty set of inputs (an *input burst*), and a set (which may be empty) of outputs (an *output burst*). When in a given state, only the inputs specified on one of the arcs leaving this state can occur, in any order at any time, and the machine does not react until the entire input burst has occurred. For example, in Figure 2 (right), from state A to B the allowed input burst is $a+b+$, where the symbols $+$ and $-$ mean rising and falling respectively; therefore, inputs a and b can rise in any order at any time. The machine then fires the specified output burst, $x+$ in this case, and enters the specified next state. New inputs are allowed only after the system has reacted completely to the previous input burst. Hence, burst-mode systems still require the fundamental-mode assumption, but only between transitions for different input bursts. It is also a requirement that no input burst can be a subset of another input burst leaving the same state; thus the machine can unambiguously determine when a complete input burst has occurred and react accordingly.

As described by Nowick and Dill [9, 10] burst-mode circuits can be implemented using the circuit shown in Figure 2 (left), in which a clock is generated locally in each state machine and is independent of local clocks in other modules. The major claim of this implementation is that by having a local clock, many of the hazards encountered by normal Huffman circuits are avoided and standard synchronous state assignment techniques can be employed. For example, since the local clock guarantees that no state bit can fire until all input changes have occurred, essential hazards are avoided.

However, not all hazards can be ignored in burst-mode machines. In all transitions, the outputs are generated directly in response to the inputs, and the local clock offers no protection. Therefore, redundant *cubes* (defined in [11]), which are necessary in Huffman circuits, are also required for the output logic. The local clock generation logic may also contain similar hazards. Although this signal is not seen directly by the environment, a hazard on the clock lines could cause the state to change partially or completely when no change is intended.

To use standard encoding schemes, special hardware has to be added to the local clock logic to ensure that all AND products involved in the previous state are disabled before the next state bits arrive [10]. This eliminates the hazards caused by multiple state bit transitions, at the expense of more complicated internal timing constraints. Consequently, critical races are avoided and synchronous state encoding schemes can be used, potentially with a significant decrease in the required number of state bits.

Burst-mode circuits can also be implemented using techniques similar to those of Huffman circuits [16, 17, 18]. Since the burst-mode specification allows only outputs to change after an entire input burst, and since transitions from the next input burst are not allowed to arrive until the circuit has finished reacting to the previous burst, there are no unavoidable hazards in the circuit. Only special state encodings and delays on the feedback loops are required to avoid sequential hazards.

A major attraction of state machines is that this synthesis method performs global optimisation. The synthesis method allows one the freedom to choose among many possible state reductions, state assignments and logic implementations, which would be difficult or impossible to do with local transformations. Designing and implementing efficient gate-level hazard-free circuits in this method is also easier than in translation and graph-based methods.

However, according to Chu [1], this design approach is “very difficult to use, especially for synthesising circuits with many input variables”. One important reason, true also of earlier approaches and recent methods, is the exponential dependency of the number of entries in the flow table on the number of input signals. Because an asynchronous state machine needs continuously to sense the changes in the input and produce changes at the output and the state variables, an asynchronous implementation requires the listing of all input combinations in the flow table, resulting in an exponential increase in the number of entries. Big tables are difficult to handle because each cell, whether it contributes to the final implementation or not, needs to be examined to determine the relation between the inputs, outputs and state variables in order to derive the final expressions. Neither early Huffman machines nor more recent burst-mode and 3D machines can avoid this problem because flow tables play a key role in their implementation.

To solve that problem, a new synthesis method is provided here. Known as the *path-model* method and based on a new view of state machines it obviates flow tables, thus simplifying the design process, by using instead an algebraic formalism. Equations similar to those in Boolean algebra are used for specification and are manipulated by laws of lifted Boolean algebra to generate the final sequential equations which are free of sequential hazards and so can be implemented by simple gates after combinational hazards are removed.

3 The signal calculus, SC

The *signal calculus*, SC, [12] forms a temporal logic whose modalities capture delay and signal increase and decrease. The latter two modalities can thus be thought of as discrete rising and falling derivatives respectively. Most of the Boolean operators and laws can be lifted pointwise to signals without any change.

3.1 Signals

In SC, time is modelled using the integers \mathbb{Z} . Negative integers represent the past, 0 represents the present and positive integers represent the future. A signal is a Boolean valued time-dependent function. Writing \mathbb{B} for the type of Booleans, the type of all signals is therefore defined to consist of all functions from integers to Booleans $\mathbb{S} \triangleq \mathbb{Z} \rightarrow \mathbb{B}$.

For example the Heaviside function is represented as a signal that was low in the past, is high

at the present time and will stay high in the future:

$$heavi : \mathbb{S}, \quad heavi(t) \triangleq (t \geq 0).$$

A clock having period 2, which is high now and which alternates, is a signal:

$$clock : \mathbb{S}, \quad clock(t) \triangleq \overline{t \bmod 2},$$

where $(t \bmod 2)$ denotes the integer remainder of t after division by 2.

3.2 Boolean operations

SC extends Boolean algebra by lifting pointwise each Boolean binary operator \otimes (like conjunction (written as juxtaposition), disjunction (written $+$), implication (written \Rightarrow), *etc*) to signals:

$$\otimes : (\mathbb{S} \times \mathbb{S}) \rightarrow \mathbb{S}, \quad (s \otimes s')(t) \triangleq s(t) \otimes s'(t)$$

and similarly for the unary operator of negation

$$\forall t : \mathbb{Z} \cdot \bar{s}(t) \triangleq \overline{s(t)}$$

and the nullary operators *false* and *true*.

A consequence of lifting is that SC inherits the laws of Boolean algebra. Signals s and s' are equal as functions if and only if they are pointwise equivalent:

$$s = s' \triangleq \forall t : \mathbb{Z} \cdot s(t) \Leftrightarrow s'(t).$$

3.3 Delay

To model the race that underlies potential hazards, SC contains a modality for delay: the retiming operator δ takes an integer n (expressed as a subscript) and a signal s and returns a signal which, at time t , behaves like s at time $t-n$. With the usual convention that functional application binds to the left, that is defined

$$\delta : \mathbb{Z} \rightarrow (\mathbb{S} \rightarrow \mathbb{S}), \quad \delta_n(s)(t) \triangleq s(t-n).$$

The delay function with unit delay, δ_1 , is abbreviated to δ ; and the application of the delay function is assumed to bind more tightly than that of the lifted logical operators.

Several laws are required later:

$$\begin{aligned}\delta_n(s \otimes s') &= \delta_n(s) \otimes \delta_n(s') \\ \delta_n(\delta_m(s)) &= \delta_{n+m}(s) = \delta_m(\delta_n(s)) \\ \delta_0(s) &= s \\ (\delta_n(s) = s') &\equiv (s = \delta_{-n}(s')).\end{aligned}$$

3.4 Differentiation

Signal change is captured by the modalities that represent signal rise and signal fall: the discrete ‘slope’ of the signal. For signal s its rising derivative is defined to be high at time t iff s is high at time t and low at time $t-1$

$$D^+ : \mathbb{S} \rightarrow \mathbb{S}, \quad D^+(s) \triangleq \overline{\delta(s)} s.$$

Similarly, the falling derivative is

$$D^- : \mathbb{S} \rightarrow \mathbb{S}, \quad D^-(s) \triangleq \delta(s) \overline{s}.$$

Two useful laws of differentiation are:

$$D^+(\overline{s}) = D^-(s), \quad D^-(\overline{s}) = D^+(s).$$

4 Path-model specifications

This section contains a new user-level specification formalism, known as the path model. Since every legal path-model specification is implementable, the path model constitutes a design tool rather than a mere abstract modelling tool.

4.1 Specification

As in traditional AFSM methods, the view of communication is state-based: a machine in a certain state, receives inputs, generates outputs, and moves to a new state. What is different is that the states are organised into paths to facilitate the generation of hazard-free equations.

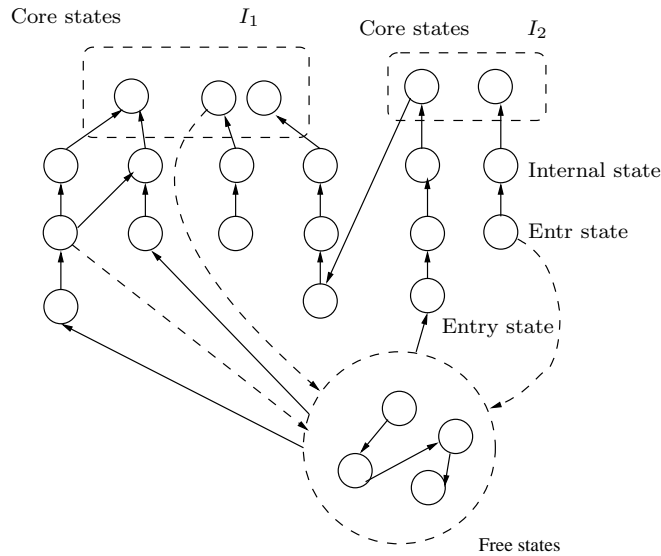


Figure 3: A path-model specification.

In a sequential system, the output of a state depends not only on its current input but also on its input history; thus the same input may be mapped to different outputs. In the path model, a state for which the input can be mapped to different outputs in different states is called a *core state* and its input is called a *core input*. A core input can be mapped to more than one core state while a core state has a unique core input. To determine whether a state is core and if so, its output, the device must refer to the input history (we will assume for now that the device can record and refer to the input history). A path of a core state s is defined to be a shortest sequence of states, with s as last element and which the device can traverse in order from the first element to the last through a sequence of input transitions, whose corresponding input sequence (unambiguously) determines the output of s . It is required for every path to consist of a finite number of states, have a unique entry state (the first state of a path) and a unique final state (the core state itself).

States that lie on a path but are not core are called *bounded*; states that are not on any path are referred to as *free*; and states between the first and the last are referred to as *internal*. The inputs of a non-core state are mapped to unique outputs: there is a function from inputs and outputs.

Each core state must, by definition, have at least one *conflicting* state: a state having the same input but different output. Two paths containing conflicting states are themselves said to be conflicting. The conflicting relation partitions states and paths into groups called *state groups* and *path groups* respectively. Two conflicting paths have different input paths because otherwise the outputs in the respective conflicting states would be the same. Different core states may have the same input and output but these states and their paths are said not to be in the same group. An asynchronous sequential circuit is hence modelled by one or more state groups and path groups with or without free states.

Figure 3 shows a typical path-model specification, in which circles in solid lines represent states, arrows between states represent transitions, and enclosed areas in dashed lines group conflicting core states and free states respectively. For clarity, arrows from bounded or core states to free states are dashed, and inputs and outputs of each state are omitted except for the two state groups which have input I_1 and I_2 . Core states, free states, some of the internal states and entry states are also identified. As illustrated, the structure of a path-model specification can be complicated: paths can share states, a core state of a path can be a bounded state of another, one core state can have more than one path, and so forth. As a general rule, a specification in the path model must have more than one core state and at least one other bounded state; however, it can have no free state. Without at least one core state and one path the design problem and hence its specification are combinational; but there need be no free states.

4.2 Definition formalised

Formally, a path-model specification is an 8-tuple, $P = (V, E, I, O, v_0, in, out, his)$, where: V is a finite set of states; $E \subseteq V \times V$ is a set of transitions; $I = \{x_1, \dots, x_m\}$ is a set of inputs (binary input variables); $O = \{z_1, \dots, z_n\}$ is a set of outputs (binary output variables); $v_0 \in V$ is a (unique) start state; and in , out and his are labelling functions used to define each state as follows. Writing P^m for the set of minterms on m inputs and letting the value of input variable x_i in minterm m be denoted m_i , (a minterm is also called an *input state* of the function) the function $in : V \rightarrow P^m$ defines the values of the m inputs, the function $out : V \rightarrow P^n$ defines the values of the n outputs, and the function $his : V \rightarrow \mathbb{P}(seq V)$ defines a set of state sequences that represent the input history for each state.

Comparing that definition with the conventional AFSM definition, each state is seen to have a new attribute: a set of state sequences, each of whose elements represents a path (excluding the core state). Clearly each core state has at least one such sequence, but each non-core state has none. This new attribute is the most important component of the path-model specification and forms the basis of our synthesis method.

The concepts of entry states, bounded states and core states can be derived from that new attribute. The set of core states $Core : \mathbb{P} V$ is defined

$$Core \triangleq \{s : V \mid his(s) \neq \emptyset\}.$$

The set of free states $Free : \mathbb{P} V$ is defined (where the two rows are implicitly conjoined)

$$Free \triangleq \left\{ s : V \mid \left(\begin{array}{l} \forall v : V \cdot \neg \exists n : \mathbb{N} \cdot \exists q : his(v) \cdot q[n] = s \\ his(s) = \emptyset \end{array} \right) \right\}.$$

The set of bounded states $Bound : \mathbb{P}V$ is defined

$$Bound \triangleq \{s : V \mid \exists v : Core \cdot \exists n : \mathbb{N} \cdot \exists q : his(v) \cdot (n < \#q \Rightarrow q[n] = s)\}.$$

And the set of entry states $Entry \subseteq Bound$ is defined

$$Entry \triangleq \{s : Bound \mid \exists v : Core \cdot \exists q : his(v) \cdot q[1] = s\}.$$

Evidently $Free$ is disjoint from every other state set, indicating that a free state cannot be bounded or core. Set intersections that do not involve $Free$ can be non-empty, indicating that a bounded state can be a core state, and *vice versa*. A group of conflicting states with input I can be defined as a set of states:

$$Con_I \triangleq \{s : Core \mid in(s) = I\}.$$

One invariant for the set Con_I is that no two members can share an output:

$$\forall w, v : Con_I \cdot w \neq v \Rightarrow out(w) \neq out(v).$$

Therefore, if two core states have the same input and output they cannot be in one state group. A group of conflicting paths with input I is defined to be a set of sequence sets:

$$ConP_I = \{his(is) \mid is \in Con_I\}.$$

The function to find the input paths of a state s is $his_input : V \rightarrow \mathbb{P}(seq P^m)$ where

$$his_input(s) = state_input(his(s))$$

and the function $state_input : \mathbb{P}(seq V) \rightarrow \mathbb{P}(seq P^m)$ is defined

$$\begin{aligned} state_input(\{\}) &\triangleq \{\} \\ state_input(\{q\} \cup s) &\triangleq \{seq_input(q)\} \cup state_input(s) \end{aligned}$$

where the function $seq_input : seq V \rightarrow seq P^m$ is defined, using \frown for sequence catenation,

$$\begin{aligned} seq_input(\langle \rangle) &\triangleq \langle \rangle \\ seq_input(\langle x \rangle \frown s) &\triangleq \langle in(x) \rangle \frown seq_input(s). \end{aligned}$$

A direct result of this definition is

$$\forall s : Core \cdot \forall q : his(s) \cdot \exists t : his_input(s) \cdot \left(\begin{array}{l} \#q = \#t \\ \forall i : \mathbb{N} \cdot i \leq \#q \Rightarrow in(q[i]) = t[i] \end{array} \right).$$

We need the second Lemma below, but the first helps to prove it.

Lemma 1. *Two conflicting states do not have a common input path*

$$\forall w, v : Con_I \cdot w \neq v \Rightarrow his_input(w) \neq his_input(v).$$

Proof. According to the definition of path, in a state group the output of a core state is uniquely determined by its input paths, *i.e.*,

$$\forall w, v : Con_I \cdot his_input(w) = his_input(v) \Rightarrow out(w) = out(v).$$

However, as shown earlier, two conflicting states cannot have the same output:

$$\forall w, v : Con_I \cdot w \neq v \Rightarrow out(w) \neq out(v).$$

The result follows. □

Lemma 2. $\forall w, v : Con_I \cdot w \neq v \Rightarrow \neg \exists p : seq P^m \cdot his_input(w) = p \frown his_input(v)$

Proof. Suppose, by way of contradiction, that there exists an input sequence p such that $his_input(w) = p \frown his_input(v)$. Then according to the properties of paths, two different outputs can both be assumed in the same state. Suppose that the machine has gone through the input sequence $p \frown his_input(v)$ to reach state u . Then since $his_input(v)$ is an input path, $u = v$ and $out(u) = out(v)$. However, since $his_input(w) = p \frown his_input(v)$, it is also true that $u = w$ and $out(u) = out(w)$. This is the desired contradiction, since by assumption $w \neq v$ and $out(w) \neq out(v)$. □

As has been seen, the path-model specification identifies a further attribute of each state, namely a set of paths. If it is difficult to find the paths, then the feasibility of the path-model methodology is in doubt. However, since a sequential circuit is characterised by its history (of input/state transitions), such paths tend to be explicit or easy to find, whilst the complexity of identifying paths tends to increase in proportion to the complexity of identifying states and drawing flow tables as in classical methods. As pointed out by Yun [16], the expressiveness of a design style can only be determined empirically, by examining design examples in the target application area. For all the examples we have implemented, the specifications in the path model are normally easier to find and construct than with conventional methods (using tables or graphs). The

examples to be presented later thus justify the view that the complexity associated with the identification of paths is unlikely to be a concern in path-model synthesis.

4.3 Specification in SC

In the path-model specification for a state s , a member of $his(s)$ is equivalent to a path of s excluding s itself. The data type *sequence* has been chosen to model paths because its members are totally ordered, reflecting the unidirectional property of paths and the timing orders of transitions between path states. The laws and operations associated with sequences thus facilitate reasoning about paths. For the same reason, each sequence can be represented equally well using an SC expression with the delay operator δ .

Some notation from propositional logic is needed. If m is a minterm of some logical expression, $product(m)$ denotes the corresponding product term. For example, with $m = 0011$ in variables a, b, c, d , we have $product(m) = \bar{a}\bar{b}cd$. It is convenient to lift the concept ‘product term’ from Booleans to signals. Thus $\bar{a}\bar{b}cd$ also represents the product of four signals.

Now, for a core state $s : Core$, if $q \in his_input(s)$ and $n = \#q$ then an equivalent SC expression for the sequence q is captured by a variable sc_input_s where

$$sc_input_s = \delta_n(product(q[1]))\delta_{n-1}(product(q[2])) \dots \delta_1(product(q[n])).$$

The subscripts of the operator δ , like the sequence index, indicate only the order between different signals and not physical time. For a non-core state s , since $his_input(s) = \emptyset$ the variables sc_input_s evaluate to the constant *true* (because an empty sequence set means that the input path does not matter for this state). Therefore, since the input path and the current input uniquely determine the current output, the relation is

$$sc_input_s(product(in(s))) \Rightarrow product(out(s)).$$

There, the implication holds pointwise between signals. The LHS represents a complete input path for state s , the RHS is the expected output of s , and the implication describes the relation between input, input history and output in state s for one path.

That specification, together with similar specifications for all other paths of s , must be implemented in the circuit to-be-synthesised. In order to find an implementation for the whole path-model specification, a synthesis method must identify such implications for all states and solve them because they specify the expected behaviour of the circuit. Note that when sc_input_s reduces to *true* for a non-core state s , this implication expresses the expected combinational relationship between the input and output. The implication is not an equivalence because in a given state there might be other states in which other conditions also imply the same output,

the RHS. Only when all conditions that imply the RHS are identified can an equivalence for the output be obtained.

That implication can be seen as an algebraic equivalent of the enumerative process in flow tables, where conditions implying the same result are found in all table cells before they are put together into a sum-of-products equation.

5 Implementation overview

As illustrated in Figure 1, path-model circuits are implemented using techniques similar to those of Huffman circuits; *i.e.*, the hardware implementation is a combinational network with the outputs of the network fed back as inputs. There are no explicit storage elements such as latches, flip-flops or C-elements, and feedback loops are used to store the state of the machine.

The state of the machine, as in Huffman-mode state machines [6], is generally stored only in internal state variables to simplify the synthesis process. 3D implementations [16, 17, 18] also use primary outputs to store the state of the machine to minimise the number of internal state variables. However, when the outputs and state variables are both used to store states, the encoding of the state variables becomes difficult because critical races involve not only state variables but also primary outputs. When multiple outputs and state variables change simultaneously and are fed back to the input of the combinational logic, it becomes difficult to guarantee that the transitions are critical-race free.

One solution, known as a Type II 3D machine [16, 17, 18], is to separate the output burst and the state burst in each cycle: when the machine detects that all transitions of the input burst have appeared, it generates an output burst; a state burst immediately follows the output burst, completing the 3-phase cycle. Obviously, the cycle times of Type II machines are therefore longer than Huffman machines which require 2-phase cycles. The logic synthesis is also complicated for the output burst of Type II machines because the critical races incurred by them must also be identified and removed. In comparison, for Huffman machines output transitions can simply be ignored during the synthesis process because they are not the input of the combinational logic.

Therefore, this paper generally uses only internal state variables to store states in order to simplify the synthesis process and focus on the introduction of the path model and SC. However, as an exception, when it can be guaranteed that at most one state or output signal changes for each transition, states can also be stored in primary outputs. Therefore, no critical races are possible for this kind of machine, regardless of where the states are stored. For example, in Section 7, a toggle is designed in this way since its states can be stored in its two primary outputs and at most one of them can change during any state transition.

A path model implementation is formally defined to be a 4-tuple (I, O, S, f) in which

- I is a non-empty set of primary input symbols;
- O is a non-empty set of primary output symbols;
- S is a (possibly empty) finite set of internal state variable symbols; and
- $f : I \times O \times S \rightarrow O \times S$ is a next-state function.

The combinational network shown in Figure 1 implements the next-state function f which, as discussed above, can refer to outputs as sources of state values.

Since MICs are allowed, one would expect the implementation to have the same problems that motivate the SIC restriction in Huffman circuits. However, as illustrated in the next section since the outputs or the secondary signals are allowed to change only after all inputs have arrived, and since transitions from the next input burst are not allowed to arrive until the circuit has finished reacting to the previous burst, there are no unavoidable hazards in the circuit. However special state encodings, such as one-hot encodings and delays on the feedback lines, are required to avoid critical and essential hazards.

6 Synthesis method

Now that the implementation style has been explained, the synthesis method can be presented in a top-down fashion. In the path model, although several conflicting core states have the same input, their outputs can be differentiated by their different paths. The general methodology of synthesis in the path model is therefore to distinguish conflicting paths for conflicting core states in a hazard-free manner. This requires each path to be uniquely labelled, from its entry state to the final core state, and sequential hazards to be removed in the process. Although each $s : Con_I$ may have a unique input history ($his_input(s)$), state variables are still used in the synthesis in the same way as they are used in other AFSM methods to label the paths, as it can be seen later that the input history expressed by SC is not implementable. The state variables are used in the path model as follows: when each entry state is entered, a unique state code is assigned and this uniqueness is maintained in the state variables as long as the state transitions follow the path. Whenever a transition deviates from the path, the state variables are reset to clear this uniqueness, and if the uniqueness is maintained until the final core state is reached, then according to the state variables, the machine knows what core state it is in and what output should be assumed. The synthesis procedure consists of the following steps:

- Formal specification of the design problem. This step involves the identification of stable states, transient states and transitions between states, which are then categorised into core states, bounded state and free states, and organised into paths to construct a path-model specification. Relations between inputs and outputs in each state are identified and expressed as implications in SC.

- Assignment of symbolic states. Symbolic state variables are assigned uniquely to label paths so that different core and bounded states can be differentiated wherever it is required.
- Encoding of symbolic states. A critical-race-free encoding is performed so that the state symbols assigned in the previous step are encoded in binary state variables.
- Application of the state variables to implications. After the state variables are available, they are used to replace the input history expressed in SC in the original implications.
- Formation of equivalence equations. Finally, an enumeration is followed to combine all implication relations into an implementable equation for each output or state variable.

Of course, for the resulting equations to be free of combinational hazards, they need to be post-processed using relevant techniques which are overlooked in this paper. Next, the details of each step of the synthesis procedure will be presented.

6.1 Environment models

Since the results produced by synthesis methods are only as good as the model used to approximate the physical circuits, the models used must first be understood. If the circuit model used by a synthesis method is too optimistic, then the synthesised circuits may be incorrect, even though the circuits may function correctly according to the model. On the other hand, if the circuit model is too pessimistic, then the synthesised circuits may be very robust, but suboptimal.

The environment of a circuit is said to operate in *multiple-input change fundamental mode* if it changes a specified set of inputs of the circuit and waits for a specified set of outputs to change, and for the circuit to stabilise before it changes the next set of inputs. The synthesis method presented in this paper (as discussed in Section 2.3) allows *burst-mode* transitions [11], a variation of the MIC fundamental mode operation where the environment is allowed to change certain inputs that do not cause the outputs of the circuit to change before the circuit has stabilised. In other words, in a burst-mode transition, a function may change only after every input in the burst has changed. Defined formally, for a combinational function f a burst-mode input transition from input state A to B is an input transition where for every input state C that is after A but strictly preceding B , $f(A) = f(C)$; the ordering on states is obtained by representing a MIC in terms of a succession of SICs (see [11]). Evidently a burst-mode transition is free of function hazards, and the conditions required to eliminate logic hazards are also simpler.

6.2 Sequential equation

The goal of any AFSM synthesis is to find out the conditions under which a secondary or output signal is turned on and maintained on (the conditions under which the same secondary or output

signal is turned off and maintained off are implicit by contraposition). Expressed in SC, for each signal S it is necessary to solve the equation

$$S \equiv D^+(S) + \delta(S)S.$$

Both of the dynamic conditions on the right-hand side of the equation need to be solved: operators d and D^+ indicate that, as required by the implementation style in Figure 1, only the last state can be used to determine the current value of the signal (because the state stored in the feedback loop is refreshed in every cycle). As discussed earlier, in each state equivalent conditions describing the relations between inputs, outputs and state variables are used to represent these two conditions in order to obtain an implementable equation. If the ‘turn-on’ condition, $D^+(S)$, is not completely represented, then the final implementation will not turn on when it should; if the ‘maintain-on’ condition, $\delta(S)S$, is not completely represented then the final implementation will turn off when it should not. Only for a necessary and sufficient condition satisfying both will the implementation behave as specified. The equations in this form for the outputs and secondary signals are called *sequential equations*. The correctness of this equation is straightforward as, according to the laws of SC, the right-hand side can be simplified to S . An equivalent complementary equation is

$$S \equiv \overline{D^-(S) + \delta(\bar{S})\bar{S}}$$

which can be used in the same way to calculate S . The equation that is chosen depends on which is more efficient for different design specifications.

6.3 Normal form

We already know that the history of inputs can be conveniently expressed by SC using its concept of timing. For example, for a C-element with inputs a, b and output x , when $\bar{a}b\delta(ab)$ is true the output is x ; when $\bar{a}b\delta(\bar{a}\bar{b})$ is true the output is \bar{x} . In both cases, although the present inputs are both $\bar{a}b$, the input history distinguishes between the two states. However, as mentioned earlier, equations in this form are not directly implementable with the implementation style illustrated in Figure 1. In that style, inputs of a state are immediately overwritten by inputs of the next, *i.e.*, previous inputs are not saved. Therefore, expressions containing previous inputs such as $\delta(ab)$ cannot be directly mapped to any gates, modules or structures in the implementation. Instead, in this style, only the state signals are stored in the feedback loop. When the inputs of a new state arrive, the secondary signals at the input of the combinational logic are still in the previous state. Therefore, an implementable equation can use only current inputs and secondary signals of the last state to calculate the values of secondary signals and the outputs of the present state.

Note that, in the path model implementation the primary outputs are also used to store the state of the machine whenever possible in order to minimise the number of internal state variables.

They can also be referred to as state variables when used in that way. Therefore implementable equations can be expressed in SC as

$$\begin{aligned} o &= f(I, \delta(S), \delta(O)), \\ s &= g(I, \delta(S), \delta(O)), \end{aligned}$$

where f and g are functions, I , S and O are sets of inputs, secondary signals and outputs respectively, and o and s are a secondary signal and an output signal respectively.

However, it is not always the case that secondary signals can only be used in this way. Very often, when some output or secondary signals are isolated from the changes of primary inputs, they can only use the current values of secondary signals and outputs in their SC expressions. Therefore, in summary, a valid SC equation in normal form can take one of the following forms:

$$\begin{aligned} o &= f(I, \delta(S), \delta(O), S', O'), \\ s &= g(I, \delta(S), \delta(O), S', O'), \end{aligned}$$

where O' does not contain o and S' does not contain s because clearly a signal cannot use its present value to calculate itself. Equations in this form are said to be in *normal form* because they use only state variables stored in the feedback loop and can be immediately mapped to circuit implementation without further processing. Therefore y_i in Figure 1 can be replaced by $\delta(Y_i)$ and the same result can be obtained. From the implementation perspective, the current value of a signal means a direct connection from the circuit output and the previous value of a signal means the circuit output that goes through the feedback loop before it is connected.

In order to construct a path-model specification, SC must be able to model and support reasoning about its basic components: the states and transitions between states.

6.4 Modelling of input transitions

Modelling of SICs is trivial: with the present input and the input of the previous state both being known, it is sufficient to apply operator δ to the previous input and conjoin the two.

However, since the path model also allows burst-mode MIC transitions, they too must be modelled. In a given state, inputs in an input burst can occur in any order and the device will not react until the entire input burst has occurred, at which point it fires the specified output burst and enters the specified next state. Hence, the way to model an MIC is to decompose it into sub SICs and maintain the required properties for each SIC. For example, to model the burst-mode specification $a+b+/x+y+$ where a, b are inputs and x, y are outputs, the path-model

specification uses two implications for the outputs:

$$\begin{aligned}\delta_2(\bar{a}\bar{b})\delta(a\bar{b})ab &\Rightarrow D^+(X)D^+(Y) \\ \delta_2(\bar{a}\bar{b})\delta(\bar{a}b)ab &\Rightarrow D^+(X)D^+(Y).\end{aligned}$$

These two implications indicate that after $\bar{a}\bar{b}$ changes to ab in any order, both signals X and Y rise; but the device does not respond to any single transition on either a or b . Inputs $a\bar{b}$ and $\bar{a}b$ are inputs of two intermediate states created when an MIC is decomposed into several SICs. Therefore outputs and state variables of the intermediate states must be the same as their preceding states to ensure that the machine is essential-hazard free. To guarantee that the state does not change in the intermediate states either, two implications suffice:

$$\begin{aligned}\delta(\bar{a}\bar{b})a\bar{b} &\Rightarrow A\delta(A) \\ \delta(\bar{a}\bar{b})\bar{a}b &\Rightarrow A\delta(A)\end{aligned}$$

assuming that the preceding state code is A . Whether A should change or not when the input finally changes to ab depends on the assignment of state symbols as introduced later in this section.

After MICs are decomposed into SICs, a specification can be modelled and reasoned about just like a traditional Huffman machine, while simultaneously ensuring that conditions associated with the original MICs are satisfied, *i.e.*, outputs and state variables do not change in the intermediate states.

6.5 Modelling of state variables

As the path-model specification still uses state variables to record input history and represent states, correctly modelling them in state transitions and states themselves is essential.

First, they must be properly modelled for transitions between states. If we define a state symbol as a vector of state variables, a transition from state s (with state symbol S) to state s' (with state symbol S') is modelled as

$$product(in(s'))\delta(S) \Rightarrow S'.$$

When input signals associated with the next state start to arrive, the state symbol S , subject to change to S' , must be prefixed by operator δ (in order to act as the previous state of the current state). Hence, the input history recorded in S and the current input uniquely determine the current state symbol. The output of a core state is also determined in this way, but it is also determined by the current state symbol.

Second, state variables must be properly modelled in both stable states and transient states. A stable state is a state in which the circuit is stable when no inputs change. Formally, for stable state s , if S is the state symbol, the behaviour of state variables is modelled

$$\text{product}(in(s)) \delta(S) \Rightarrow S.$$

Here the operator δ represents a dummy transition since in a stable state all signals are static. The modelling of this dummy transition is necessary because without this product to hold the value of S it is subject to change as soon as it is fed back to the input of the combinational logic, and consequently state s becomes a transient state as will be introduced next.

A *transient state*, unlike a stable state, moves to the next state when the input of the current state does not change. It is formalised:

$$\text{product}(in(s)) \delta(S) \Rightarrow S'.$$

This specification is similar to that for a transition, except that the input is from state s instead of s' , indicating that $in(s) = in(s')$ and now the transition occurs even if no input changes. Comparing this with the specification of a stable state, the modelling of the dummy transition of the latter is critical in differentiating between the two. Consecutive transient states are not needed and hence are not allowed in the path model: a transient state can move only to a stable state.

6.6 Assignment of symbolic states

This section introduces methods for the assignment of state symbols. As with conventional methods, each state in the original path-model specification needs to be assigned a state symbol in order to transform the specification in SC into normal form. The algorithms for encoding the state symbols with state variables are presented in the next section. The equivalent step in traditional AFSM methods is the reduction of primitive flow tables [14].

The transformation from an initial SC specification to normal form requires the history inputs to be replaced with state symbols (because they are not implementable). Specifically, operators δ_n ($n > 0$) on inputs must be removed from the original path-model specification and to achieve this state symbols must be derived from the specification so that they can be used to pass history information between states. To find optimal or near-optimal assignments, especially when the structure of the path-model specification is complicated, we use the classical algorithm from [14], at the same time taking the requirements associated with MICs into account. These techniques have also been used recently in 3D machines [17, 18] either directly or for comparison with similar algorithms. However, all existing methods are partially enumerative and heuristic, and there is no known simple, efficient algorithm for solving the problem in its most general form [14].

The algorithm [14] requires consistent states to be assigned the same state symbol in such a way that no inconsistency can be obtained through any sequence of implications. (A set Q is said to imply a set R if R is the set of next-state entries for Q under some particular input.) After all consistencies are identified, maximal consistencies are constructed and used to find a minimal closed cover.

The first and the key step in that algorithm is to find all inconsistent pairs; here is a simple method. First, in the path model, output inconsistent states are conflicting states as defined in Section 4, and vice versa. Writing \sim for the binary relation of consistency and \approx for its negation, inconsistency, if $a \approx b$ and a, b are output inconsistent, in the path model we have

$$a \approx b \equiv \left(\begin{array}{l} a \in Con_I \\ b \in Con_I \end{array} \right),$$

i.e., a, b are output inconsistent if and only if they are conflicting core states. Therefore, the identification of output inconsistent states is trivial in the path model: states in different state groups are output consistent.

To identify other inconsistent pairs, however, all allowed transitions, including transitions on paths, transitions from states on paths to free states and transitions between free states, and so forth, need to be taken into consideration in order to determine if two output-consistent states can imply two conflicting states. First, we prove a lemma for free states.

Lemma 3. *A free state is consistent with any other state.*

Proof. For a free state s to be inconsistent with another state t , there must exist an input sequence that can settle the machine into two conflicting states when started in s and t respectively. When starting from s , a free state, a whole path, say p , must be traversed to reach its core state; when starting from t , (part of) another path, say q , must be traversed to reach its core state. Because p and q are conflicting paths which are traversed under the same input sequence, one of three conditions holds:

1. the input sequence of path p equals that of path q ;
2. there is an input sequence r which, when followed by the input sequence of path p , yields the input sequence of path q ; or
3. there is an input sequence s which, when followed by the input sequence of path q , yields the input sequence of path p .

To state those conditions, recall that P^m denotes the set of minterms on m inputs and that the

value of input variable x_i in minterm m is denoted m_i :

$$\begin{aligned} & \text{state_input}(p) = \text{state_input}(q); \\ \exists r : \text{seq}(P^m) \cdot r \frown \text{state_input}(p) &= \text{state_input}(q); \\ \exists s : \text{seq}(P^m) \cdot \text{state_input}(p) &= s \frown \text{state_input}(q). \end{aligned}$$

However, according to Lemmas 1 and 2, none of those can hold. Hence, a free state cannot be inconsistent with any state. \square

The following lemma facilitates the identification of inconsistent pairs that are output consistent. It says that two non-core bounded states, w, v , are inconsistent if and only if there are two conflicting core states whose respective conflicting paths contain w and v respectively. Furthermore, the segments from w and v to their respective core states have the same number of states and the same sequence of input transitions.

Lemma 4. *If $w, v \in \text{Bound} \setminus \text{Core}$ then*

$$\begin{aligned} & w \approx v \\ & \equiv \\ & \exists s_1, s_2 : \text{Con}_I \cdot \exists q_1 : \text{his}(s_1) \cdot \exists q_2 : \text{his}(s_2) \cdot \exists m_1, m_2 : \mathbb{N} \cdot \\ & \left(\begin{array}{l} m_1 < \#q_1 \\ m_2 < \#q_2 \end{array} \right) \Rightarrow \left(\begin{array}{l} w = q_1[m_1] \\ v = q_2[m_2] \\ \#q_1 - m_1 = \#q_2 - m_2 \\ \forall i : (m_1, \#q_1] \cdot \forall j : (m_2, \#q_2] \cdot \text{in}(q_1[i]) = \text{in}(q_2[j]) \end{array} \right). \end{aligned}$$

Proof. The right-hand side obviously implies the left-hand side according to the definition of compatibility [14]. To prove that the left-hand side implies the right-hand side, we need to show that if $w \approx v$ there exist two conflicting paths which contain w and v respectively. Whether the machine starts from w or v , it can only move along the current path, under the same sequence of input transitions, until it finally reaches the respective core state. To prove this, we need to prove that for two non-core bounded states to be inconsistent, the machine can never deviate from the current path under any sequence of input transitions, when starting from either of them. Since only conflicting states are output inconsistent, the core states of the two paths must be conflicting, which indicates that the two bounded states must be on two conflicting paths.

As mentioned before the machine can move to either a free state, an entry state, or the next state on the current path when starting from a bounded state. First, however, it cannot move to a free state if the original bounded state is conflicting with another bounded state. This is because that would require a free state to be inconsistent with another state. Second, the machine cannot move to an entry state because that would require a complete path to be traversed by the machine. However, seen in the proof of Lemma 3, such a complete path does not exist because its length cannot be equal to, greater than, or smaller than the length of its conflicting path.

Therefore, in order for two bounded states to be inconsistent, every transition in the sequence can only move the machine to the next state on the same path until the core state is reached. The implication from the left-hand side to the right-hand side is therefore also proved. \square

Lemma 4 provides a convenient way to identify inconsistent pairs in the path model: every two conflicting paths are scanned simultaneously from the core states retrospectively towards the entry states and every pair of states at the same level (as determined by the number of states between this state and the core state) on the two paths are compared until the first pair that has different inputs is encountered. All pairs that are scanned, including the last pair that has different inputs, are inconsistent pairs. This process also covers output incompatibility, so it can be used to determine all inconsistent pairs in a path-model specification. As introduced earlier, all these inconsistent pairs are then merged to form MCs which are used to find a minimal closed covering. We refer to [14] for more details of the heuristics for solving these problems.

6.7 Encoding of secondary signals

Encoding is the task of assigning binary state variables to state symbols. In synchronous design that is simple and no special requirement is needed except for the minimisation of transitions of state variables to reduce power dissipation. In asynchronous design, however, encoding must be achieved free of critical hazards.

There are two types of race in a sequential circuit: a race between primary and secondary signals and a race between secondary signals. While the former can be avoided by burst-mode transitions and the insertion of delays in the feedback loop, the latter, known as critical races, need to be resolved by state encoding. Specifically, when one state changes to the next, only one secondary signal can change or the secondary signals should change in a way that does not introduce a race. Many heuristic techniques have been proposed [1] for this problem.

It has been shown [14] that a universal one-shot state encoding, with a Hamming distance of 1 between any two state codes, exists. Critical races are automatically avoided since every transition involves at most one state bit; however, multiple state variables may be required to be *true* for each state, complicating the combinational logic.

One-hot encodings in which each state code has exactly one associated state variable *true*, require two signal transitions for each state transition, but simplify the associated logic. However, a one-hot encoding scheme may not be optimally efficient since it requires a new input to be delayed long enough for three trips through the combinational logic and two trips through the feedback loop. Synthesis with path-model devices is flexible in the choice of encoding schemes; there is no general rule indicating which one is better. For the examples presented in the next section, both one-hot and one-shot encodings are used as appropriate.

7 Examples

This section contains two circuits designed to illustrate the path-model method.

7.1 An example of Yun

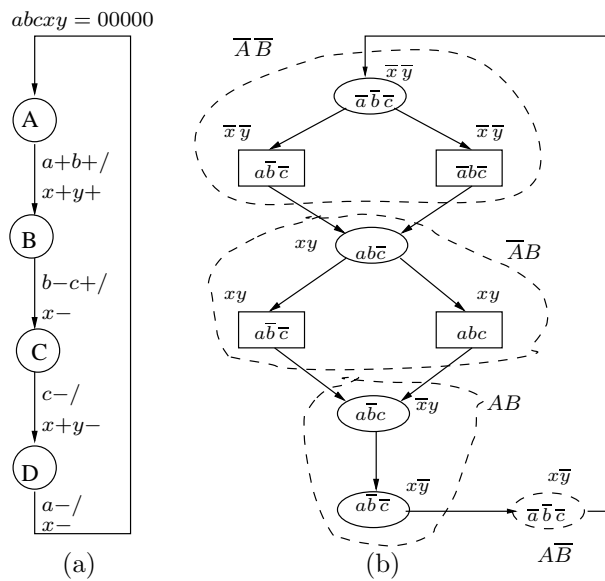


Figure 4: A simple MIC example.

First, a simple example used by Yun [17] to demonstrate the synthesis of 3D machines is used here to illustrate the basic problems associated with MICs in the synthesis process. Figures 4(a) and (b) show the original burst-mode specification and the equivalent state diagram respectively. In 4(b) the original four states are represented by ellipses and intermediate states during MICs are represented by rectangles, with output strings written outside and inputs inside. Recall this invariant of the path-model specification: outputs and state variables of intermediate states should be the same as their individual preceding states. In this specification, the core input is $a\bar{b}\bar{c}$, for which three different outputs can be generated in three different core states on three paths. Representing the three paths using SC expressions:

$$\begin{aligned} \bar{a}\bar{b}\bar{c}\delta(\bar{a}\bar{b}\bar{c}) &\Rightarrow \bar{x}\bar{y} \\ \bar{a}\bar{b}\bar{c}\delta(ab\bar{c}) &\Rightarrow xy \\ \bar{a}\bar{b}\bar{c}\delta(a\bar{b}c) &\Rightarrow x\bar{y}. \end{aligned}$$

According to our method, the three core states, as well as the three bounded states which are also entry states, are not consistent because on input $a\bar{b}\bar{c}$ they produce different outputs. All

other states are consistent free states; however, they cannot be assigned the same state symbol because of the invariant that intermediate states should be assigned the same state symbol as their preceding states. Therefore, the final assignment scheme uses three state symbols to cover the top three shapes, the middle three shapes and the bottom two ellipses respectively. The one-shot encoding scheme was selected to encode the three state symbols to $\overline{A}\overline{B}$, $\overline{A}B$, AB using two state variables, A, B . A transient state with input $\overline{a}\overline{b}\overline{c}$, output $x\overline{y}$ and state $\overline{A}\overline{B}$ is needed between states D and A to avoid the critical race since both state variables change in that transition. The implications for output x are therefore derived from Figure 4(b)

$$\begin{aligned} ab\overline{c} &\Rightarrow x \\ abc &\Rightarrow x \\ \overline{a}\overline{b}\overline{c}\delta(\overline{A}\overline{B}) &\Rightarrow x\delta(x) \\ \overline{a}\overline{b}\overline{c}\delta(AB) &\Rightarrow D^+(x) + x\delta(x) \\ \overline{a}\overline{b}\overline{c}\delta(AB) &\Rightarrow x\delta(x). \end{aligned}$$

The first two implications describe the combinational relations in the free states, the next two reflect relations in the core states, and the last one is obtained from the transient state. These implications are then combined to generate the required equivalence relations which are finally simplified to the following equation for output x :

$$x = ab + \overline{a}\overline{b}\overline{c}\delta(B) + \overline{b}\overline{c}\delta(AB).$$

Similarly equations for output y and state variables can be obtained:

$$\begin{aligned} y &= ab + \overline{a}\overline{b}\delta(\overline{A}\overline{B}) \\ A &= \overline{a}\overline{b}c + \overline{b}\overline{c}\delta(AB) \\ B &= ab + \overline{a}\overline{b}\delta(B). \end{aligned}$$

The original solution [16] uses only one state variable but since output x is also used to store the state, the total number of state variables is two. After the original burst-mode specification is expanded to include intermediate states, the total number of states becomes eight. In this case, if the outputs are also used to store state, then four more states are needed to model the output bursts, complicating the synthesis process. Of course, it also has the inherent problem of requiring longer cycle time.

7.2 Example ISEND

Finally we demonstrate a more complex example called ISEND, depicted in Figure 5. It has been used as a benchmark to demonstrate synthesis methods for 3D devices [16, 17], and has been

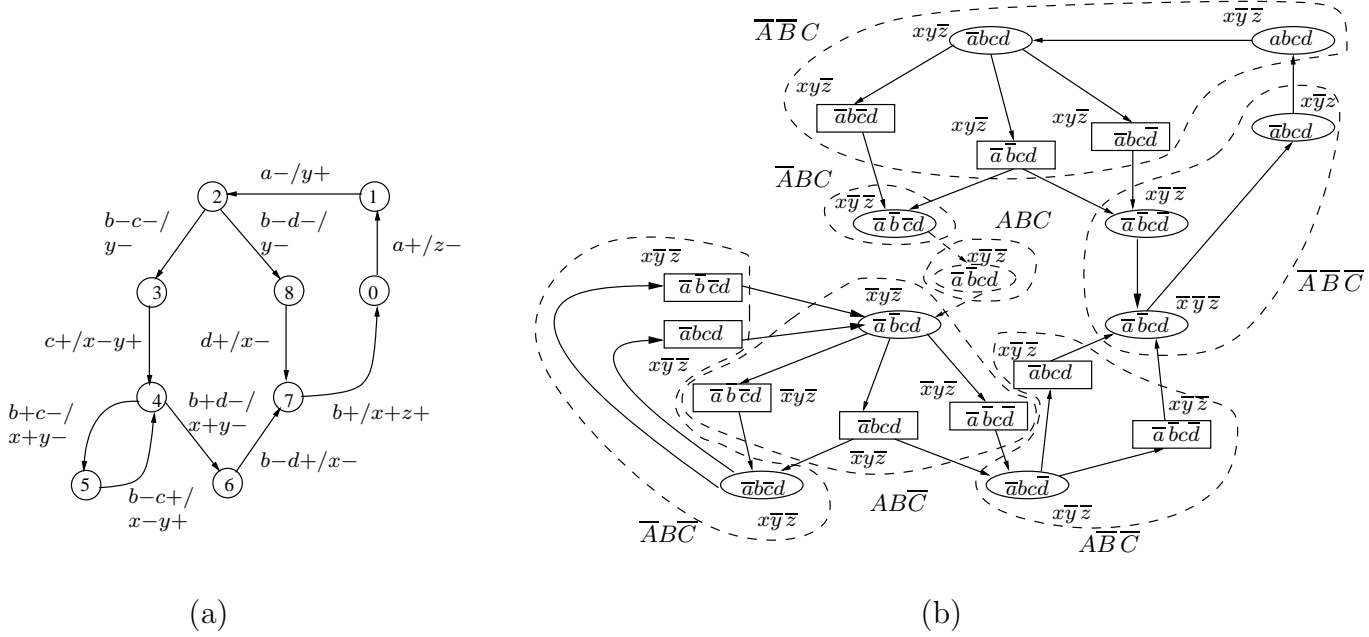


Figure 5: ISEND. (a) Burst-mode specification. (b) State diagram.

recently re-expressed in DISP and synthesised with di2pn and Petrify by Kapoor and Josephs [7].

Figures 5(a) and (b) show the original burst-mode specification and the equivalent state diagram. Continuing the conventions from Figure 4, original states are represented with ellipses and intermediate states are represented by rectangles. It can be seen that all states but one are core, the only exception being state 1 (with input $abcd$) which is an entry state of its succeeding state. There is no fundamental difference for the synthesis of this more complicated example and routine procedure can be followed to identify inconsistent pairs, perform the state assignment, encode the state symbols and derive the implications. This specification requires six state symbols for the 19 states in Figure 5, which can be encoded by three state variables under the one-shot encoding scheme. A transient state is inserted between states 3 and 4 to ensure the critical-race-free condition. The implication equations for output x can therefore be derived:

$$\begin{aligned}
 \overline{A} \overline{B} C &\Rightarrow x \\
 \overline{A} B C &\Rightarrow x \\
 A B C &\Rightarrow x \\
 \overline{A} B \overline{C} &\Rightarrow x \\
 A \overline{B} \overline{C} &\Rightarrow x \\
 \bar{a} \bar{b} \bar{c} \bar{d} \delta(\overline{A} \overline{B} C) &\Rightarrow x \delta(x) \\
 \bar{a} \bar{b} \bar{c} \bar{d} \delta(\overline{A} \overline{B} \overline{C}) &\Rightarrow x \delta(x) \\
 \bar{a} b c d \delta(\overline{A} \overline{B} \overline{C}) &\Rightarrow x \delta(x) .
 \end{aligned}$$

The first five implications express the combinational relations between the state variables and output x (because in those encoded states x is always *true*); the last three describe sequential relations in some encoded states where x can be either *true* or *false*. The final equation for x can be simplified to

$$x = \overline{AC} + BC + \overline{AB} + \overline{AB}\overline{C} + \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{A}\overline{B}) + \overline{a}\overline{b}c\overline{d}\delta(\overline{A}\overline{B}\overline{C}).$$

Similarly equations for all other outputs and state variables are

$$\begin{aligned} y &= \overline{ABC} + (\overline{abd} + \overline{acd} + \overline{abc})\delta(\overline{A}\overline{B}\overline{C}) \\ z &= \overline{abcd}\delta(\overline{A}\overline{B}\overline{C}) \\ A &= \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{BC}) + \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{AB}) \\ &\quad + (\overline{a}\overline{b}\overline{d} + \overline{acd} + \overline{a}\overline{bc})\delta(\overline{ABC}) \\ &\quad + \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{AC}) + (\overline{abc} + \overline{acd})\delta(\overline{AB}\overline{C}) \\ B &= \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{AC}) + \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{BC}) \\ &\quad + (\overline{a}\overline{b}\overline{d} + \overline{acd} + \overline{a}\overline{bc})\delta(\overline{ABC}) \\ &\quad + \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{BC}) + (\overline{a}\overline{cd} + \overline{abd})\delta(\overline{ABC}) \\ C &= \overline{abcd}\delta(\overline{A}\overline{B}) + (\overline{abd} + \overline{acd} + \overline{abc})\delta(\overline{A}\overline{B}\overline{C}) \\ &\quad + \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{AC}) + \overline{a}\overline{b}\overline{c}\overline{d}\delta(\overline{ABC}). \end{aligned}$$

The original solution [16] also uses three state variables but uses outputs to store state, so the expressions are simpler. However, the trade-off for smaller area is longer latency: three trips instead of two through the combinational logic are required for each transition.

Several features of the path-model method have been demonstrated with the examples in this section. First, the path model is flexible in the choice of encoding schemes and both one-hot and one-shot have been used in the examples. Second, the path model can easily be used to model a design problem whether what is given is an original problem description or a specification in other models (3D in this case). As has been seen, whether the design problem is simple or complex, paths and core states can be easily identified and expressed as SC expressions. Third, the complexity of the path model synthesis is not affected by the number of inputs (because no flow tables are required) but increases with the number of states that contribute to the calculation of outputs or state variables; any irrelevant states are implicit and ignored. The algebraic nature of SC makes it easy to model manually either small design problems with several specification states or relatively large problems with as many as 19 states. The Boolean-like notations express the logic relations between the signals clearly and concisely, and algebraic laws make it trivial to reason about the original implications to obtain the final expressions. As a result, design problems with as many as six inputs or 19 states can be manually synthesised following a routine procedure. Finally, it has been observed that although flow tables are no longer required, state diagrams are still useful in the model, although in theory they are not necessary, especially for large devices with many states. However, since they do not have explicitly to include every possible input, they are not subject to the same problems as flow tables.

8 Conclusion

This paper describes a user-level specification, the path model, a simple formalism, the signal calculus, a target implementation style, and a synthesis method to transform an asynchronous controller specified in the path model into implementable equations that allow MICs and are sequential-hazard-free. These equations can then be post-processed using classical methods (not presented in this paper of course) to remove combinational hazards for final implementations with simple gates.

Based on a space composed of signals lifted from Boolean values, the methodology formally models circuits as algebraic expressions, and then, using the laws of signals (some of which are lifted Boolean laws) these expressions are simplified, merged and transformed to generate equations in an implementable normal form. As flow tables are no longer needed, this new methodology removes the exponential dependency on the number of inputs that traditional methods exhibit on the number of entries in the flow table. Demonstrated by both a simple and a more complex example, it has been seen that one-dimensional paths and the algebraic nature of SC help to simplify the specification and reasoning. The new method is also flexible in the choice of encoding schemes, a critical part of any AFSM method, and it has been demonstrated that one-hot and one-shot encoding schemes can be used seamlessly with it.

References

- [1] T. A. Chu, *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, MIT, June, 1987.
- [2] A. Davis, B. Coats, K. Stevens. The post office experience: Designing a large asynchronous chip. *Proceedings of the 26th Hawaii International Conference on System Sciences*, 1:409-418, 1993.
- [3] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9(2):90-99, 1965.
- [4] A. D. Friedman and P. R. Menon. Synthesis of asynchronous sequential circuits with multiple-input changes. *IEEE Transactions on Computers*, C-17(6):559-566, June 1968.
- [5] L. A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133-1141, December 1982.
- [6] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, March/April 1954.
- [7] H. K. Kapoor, M. B. Josephs. Decomposition of benchmark circuits for specifications with concurrent outputs. Available at <http://www.bcim.lsbu.ac.uk/ccsv/dac04/benchmarks.pdf>.

-
- [8] C. N. Liu. A state variable assignment method for asynchronous sequential switching circuits. *Journal of the ACM*, 10:209-216, April 1963.
 - [9] S. M. Nowick, D. L. Dill. Automatic synthesis of locally-clocked asynchronous state machines, *Proceedings of ICCAD*, pp.318-321, 1991.
 - [10] S. M. Nowick, D. L. Dill. Synthesis of asynchronous state machines using a local clock. *Proceedings of ICCD*, pp.192-197, 1991.
 - [11] S. M. Nowick. *Automatic synthesis of burst-mode asynchronous controllers*. PhD thesis, Stanford University, Stanford, CA, March, 1993.
 - [12] C. Ratzko and J. W. Sanders. *A calculus of signals*. Technical report TR-6-00, Programming Research Group, OUCL, Oxford University.
 - [13] J. H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15:551-560, August 1966.
 - [14] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, New York, NY, 1969.
 - [15] S. H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, C-20(12):1437-1444, December 1977.
 - [16] K. Y. Yun. *Synthesis of asynchronous controllers for heterogeneous systems*. PhD thesis, Department of Electrical Engineering, Stanford University, 1994.
 - [17] K. Y. Yun and D. L. Dill. Automatic synthesis of extended burst-mode circuits: Part I (Specification and hazard-free implementations). *IEEE TCAD*, 18(2):101-117, February 1999.
 - [18] K. Y. Yun and D. L. Dill. Automatic synthesis of extended burst-mode circuits: Part II (Automatic synthesis). *IEEE TCAD*, 18(2):118-132, February 1999.