Causal Path Identification for Timed and Sequential Circuits

Mackenzie J. Wibbels, *Student Member, IEEE* Kenneth S. Stevens, *Senior Member, IEEE*

***Abstract*—**
Self timed and pulse mode circuit modules are often implemented using combinational logic with feedback, or require timing constraints for the circuit to function correctly. Identifying causal paths through these circuit modules is a fundamental requirement to enable automatic system level timing closure, performance analysis, and timing driven optimizations when using static timing analysis algorithms. A methodology and algorithm for identifying causal paths in cyclic sequential circuits under arbitrary timing models are presented. The algorithm identifies causal paths in a sequential circuit implemented with standard cell gates. The algorithm is demonstrated by identifying causal paths for over 100 asynchronous finite state machines and timed circuits designed using a variety of timing models. Cyclic paths, pulse generating paths, and unbounded length self-enabling loops are identified inside these circuit blocks. Some of the causal paths in these designs have never been previously identified.

***Index Terms*—false paths, causal paths, static timing analysis, asynchronous circuits, self timed circuits, relative timing**

I. Introduction

STATIC timing analysis (STA) algorithms are the primary means for performing timing closure in most fabricated integrated circuit designs. STA has been optimized to accurately evaluate path delays. Static timing analysis algorithms have small run times, variation and noise aware properties, and are directly integrated into timing aware design optimization algorithms such as logic synthesis, place and route, and performance evaluation.

Due to these advantages we want to extend the set of circuits for which static timing analysis algorithms can be employed to include automatic optimization and validation of sequential and timed circuits.

Timed circuits can employ desirable glitches, such as the pulse generating circuit in Fig. 1. In other circuits, glitches can be fatal. Sequential circuit design is commonly implemented using standard logic gates with feedback, resulting in combinational loops. The feedback paths normally hold the sequential circuit state. The sequential circuit example of Fig. 2 contains the two combinational loops [c ac] and [c bc]. Incorrect timing and glitches on feedback logic in sequential circuits can result in the circuit stabilizing in an incorrect state [1], [2].

Circuit and system timing closure and validation consist of two aspects: identifying the correct timing paths, and using STA to accurately model delay through those paths. This paper focuses on a solution to the first aspect: identifying causal paths through a sequential or timed circuit.

STA algorithms efficiently bound the critical circuit delay through combinational logic by performing the topological sort algorithm, which returns the longest netlist delay path in the circuit [3]. Topological sort requires that timing paths do not contain cycles. Correctly applying STA to validate the timing paths of a circuit or system that may include cyclic paths is not addressed here.

Intuitively, a causal path is analogous to a Rube Goldberg machine or a set of dominoes stacked on end where an initial event (e.g. a domino falling over) will cause a sequence of events in a chain reaction that ultimately cause the desired terminal event. Each action *must* directly cause subsequent events for it to be causal.

The objective of this work is to identify, under formally correct circuit operation, the complete set of sequences of causal events (starting with the initial falling domino) which *cause* the desired final signal transition (the terminal falling domino). Causal event chains must be traced across multiple circuit states and circuit netlist gates in order to identify sequences of causal events between the desired source and destination signal events.

The contribution of this paper is the *causal path algorithm* that identifies a complete and correct set of causal paths through sequential and timed circuit modules built from standard library cells. The approach works for any timing model, gate function and sizing, and placement of standard cells. Once identified, these *causal paths* will be used to enable algorithms which employ static timing analysis for accurate path delay modeling including timing driven synthesis, place and route, and system level timing validation.

II. Background

Sequential circuits are usually implemented using combinational logic gates with feedback, and are commonly used for data storage. These circuits are deployed in nearly every integrated circuit (IC) design. Flip-flops and latches are well known sequential building blocks that fall into this circuit class.

Sequential logic can also be used to implement asynchronous finite state machines (AFSMs) and control logic. The circuit shown in Fig. 2 is the AFSM that implements the pausable silicon oscillator used to clock micropipeline circuits under handshake control [4]. Architectures based on AFSMs that implement handshake protocols include, among others, asynchronous, self timed, and relative timed design. Timed and sequential control blocks dictate the behavior and timing

The authors are in the Electrical and Computer Engineering department at the University of Utah. This material is based on work supported by Granite Mountain Technologies. Kenneth S. Stevens declares financial interest in Granite Mountain Technologies which commercializes high performance low power integrated circuits.

of a circuit in these design approaches. We use the term AFSM to refer to both sequential and timed circuit blocks throughout the rest of this paper.

A. Related Work

The goal of causal path identification is to enumerate the complete set of timing paths which must be evaluated to perform timing validation and system level timing closure of an AFSM circuit. The most similar work related to causal path identification is the false path problem, which uses circuit causality to improve STA timing bounds of combinational circuits. The false path problem and its relationship with AFSM timing will be discussed further in Section II-B.

Timing properties of AFSM circuits have been studied using a number of methods and models including linear programming [5], [6], cycle ratio algorithms [7], [8], Max-Plus Algebra [9], [10], and Timed Automata [11].

These methods have been employed to report AFSM timing properties including time separation between events [5], [6], performance and periodicity properties [9], [12], [13], [14], data dependent delay [15], and cyclic pipeline delay as a function of data occupancy [16], [17], [18]. These approaches jointly evaluate AFSM logic and timing, which allows path based analysis to be circumvented and critical delays to be evaluated directly from circuit behavioral models. As a result, these methods must either restrict module instance timing by using hard-macro approaches [19], [20], [14] or must be evaluated on each AFSM timing configuration caused by placement and physical design. Additionally, these methods typically employ simulation [21], [22], SPICE [23], or custom timing analysis environments [23], [24] to analyze timing delays rather than commercial static timing analysis tools.

Path based approaches which evaluate AFSM timing using STA tools have also been developed [25], [26], [27]. These approaches leverage STA tools which enable improved flexibility to technology advancements in path delay and variation analysis. A fundamental problem with applying STA tools to AFSM timing analysis is in identifying the correct sets of timing paths to evaluate. Timing path identification for AFSM modules is currently manually performed by circuit designers, which is both time consuming and error prone. Thus an algorithm to identify causal path sets between timing endpoints is necessary to enable automatic static timing analysis of AFSMs by EDA tools.

B. False and Causal Path Analysis Sensitization Criteria

False path analysis improves the accuracy of STA algorithms by identifying the longest delay path for which a signal transition can propagate. Netlist paths that do not propagate signal transitions are defined as *false paths*. Netlist paths that can propagate a transition are defined as *true paths*. Identifying the set of false paths is known as the *false path problem* [28]. The false path problem determines the longest stabilization delay for sensitizable paths and therefore ignores glitches.

Both false path and causal path identification employ circuit causality to identify netlist paths. False path analysis addresses two problems. (1) The path sensitization problem, which identifies netlist paths that propagate signal transitions. (2) The critical path problem, which reports the longest sensitizable path within the circuit netlist [29].

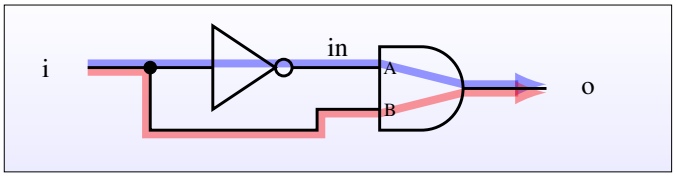

Fig. 1. Simple pulse generator circuit. The causal i↑→o↑ path has a red overlay, and the causal i↑→o↓ path is shown in blue.

A number of sensitization criteria have been developed [30], [31], [32]. A summary of the challenges and merits of different sensitization criteria is presented in [28]. Multi-cycle false path algorithms have also been developed for sequential circuits, where the algorithms assume static state bits which remain stable during an input vector [33]. One well known sensitization criteria is the *exact* criteria, which is defined as follows [29]:

Definition 1. *The exact sensitization criteria for a true path tp requires for a valid input vector v_i, tp exhibit two conditions for all pins f_i driving gate g_i in the path.*

1) *If f_i is a controlling value which stabilizes at t_i, all other inputs on g_i are either non-controlling or controlling values which stabilize at time t_j where $t_j >= t_i$.*
2) *If f_i is a non-controlling value which stabilizes at t_i, all other inputs f_j must be non-controlling and stabilize at time t_j where $t_j <= t_i$.*

A controlling value is an input which independently determines the output of a gate. For example, the controlling value for an AND gate is a low signal (logic 0) which independently determines that the output of the AND gate will also be low.

The causal path algorithm also employs a similar sensitization criterion, but contains key differences. The above exact criteria assumes acyclic netlist paths and reports critical stabilization time, which ignores signal pulses. Conversely, in order to identify causal paths for AFSM circuits, the causal paths algorithm must support both cyclic paths with transient state bits and identify pulse transitions along logic paths. Additionally, one of the objectives of the causal path algorithm is to validate timed behavioral models for any circuit delay configuration realized after physical design. Therefore, the returned causal path set must contain critical delay paths for all conformant delay configurations.

To achieve these goals, the algorithm must operate on a behavioral model which contains the complete reachable behavior due to all delay configurations which conform to the circuit specification. The timed specification accepted by this algorithm is a state graph.

Informally, a causal path is a sequence of transitions along a path in a circuit for which each transition enables the next transition in the path to fire. Once a signal in the path is enabled to fire, it can not be disabled by a side input or the causal sequence of transitions ends.

The causal path algorithm computes changes in a gate's

output stability using the Boolean difference equation shown in Eqn. 1 [34].

$$\frac{df}{df_x} = f_x \oplus f_{\overline{x}} \tag{1}$$

The Boolean difference, shown in Eqn. 1, identifies the sensitivity of a Boolean function f to a change in signal x. The Boolean difference performs functional decomposition of f by evaluation $f(x == 0) = f_{\overline{x}}$ and $f(x == 1) = f_x$ and the exclusive-or operator determines if a transition on signal x will generate or suppress the output of $f(x)$.

An example of differences between causal and false paths can be illustrated with Fig. 1. The causal path algorithm for paths from i↑ to o↑ returns the causal path [i↑ o↑]. Under the exact criteria of Definition 1 the i↑ transition causes a non-controlling high transition at pin B of the AND gate. However, the path [i↑ in↓] generates a controlling falling input at pin A of the AND gate, making [i↑ o↑] a false path. For falling transitions on i, [i↓ o↓] generates the earliest controlling input and would be a true path under Definition 1. However, based on the timed specified behavior of the pulse circuit in Eqn. 2, this transition will never occur as o must already be low before i falls.

III. Causal Path Identification Flow

Fig. 3 identifies the methodology used to generate causal paths. Five inputs are required for this flow: the circuit netlist mapped to gates in a standard cell library, the Boolean function for each gate, a timing model, a behavioral specification of the circuit, and a set of timing endpoints to be traced. Four of the inputs are passed to a formal verification engine that is used to create the reachability graph that conforms to the specification under the provided timing model. The reachability graph is called the *specification state graph* or SSG in this work.

The circuit netlist, Boolean gate model, specification state graph, and timing endpoints are inputs to the causal path algorithm. A set of causal paths between each timing endpoint pair is produced. The causal path set, along with conformance constraints generated by the verification engine, is used for timing driven design optimization and system level timing closure.

The causal path algorithm presented in this paper is agnostic to the behavior and timing models that are used to describe the circuit. This is achieved by passing the reachability graph of the timed behavior of the circuit netlist to the algorithm. The specification state graph is a behavioral model that embeds timing in the graph structure, which allows the causal path algorithm to support circuit design methodologies employing a variety of timing models.

We briefly describe each of the inputs and the verification flow used to generate the specification state graph provided to the causal paths algorithm.

A. Timing Bounds, Models, and Methodologies

Asynchronous finite state machines are designed and verified under varying degrees of timing assumptions. Different methodologies may independently or jointly model the delay

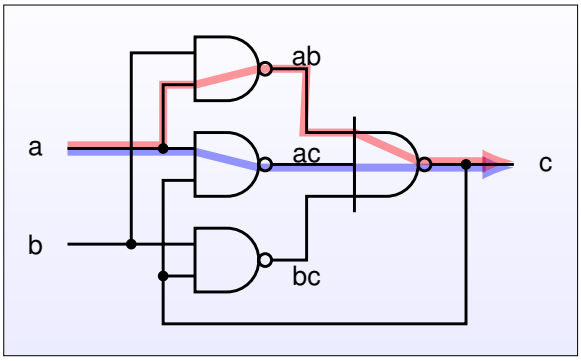

Fig. 2. C-element design implementing function ab + ac + bc. The causal a↑→c↑ path has a red overlay, and the causal a↓→c↓ path is shown in blue. There are two state holding feedback cycles, [ac c] and [bc c].

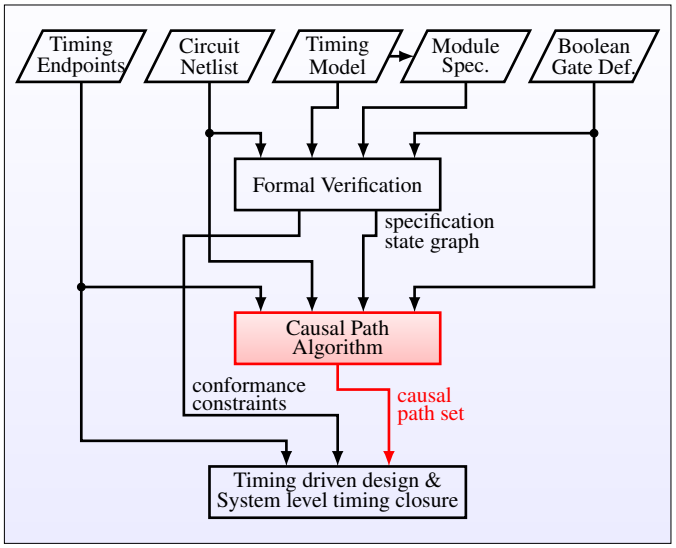

Fig. 3. Inputs required for the causal paths algorithm to generate path sets for design optimization and system level timing closure.

of active devices (e.g. logic gates) and passive devices (e.g. wires and capacitors). Module level timing assumptions can be employed, such as setup or hold times. Delay bounds can be specified using bounded or unbounded delays, and gates can use an inertial, transport, or semimodular timing model.

Delay bounds used in verification engines specify the minimum and maximum delay through a circuit. The *unbounded* delay model employs a zero delay lower bound and unbounded upper delay. The *bounded* delay model has either a minimum delay greater than zero or a fixed delay for the upper bound. Static timing analysis algorithms employ the bounded delay model. For example, a NAND gate in a specific implementation could have a delay between 8 and 13 ps.

The size of the specification state graph (SSG) of a circuit being verified and the effort required to perform system level timing closure directly depends on the delay bounds and timing model. The looser the timing bounds and model, the greater the concurrency and larger the SSG.

All timing constraints applied to a circuit must hold in the final implementation to guarantee that the SSG faithfully models the timed behavior of the circuit. For example, when

using the bounded delay model, each gate must be verified as part of the system level timing closure to ensure that it meets the bounds used in generating the reachability graph. If the timing model defines a particular gate to have a maximum delay of 57 ps, but in the design, the gate has a worst case delay of 68 ps, the SSG could be an incorrect model. Therefore, we assume that each timing constraint applied to a circuit will be verified as part of system level timing closure. These are identified as the *conformance constraints* in Fig. 3.

Most AFSMs employ unbounded delays using either the inertial or semimodular timing model. However, numerous methodologies are commonly used. The delay-insensitive model applies unbounded delays to both gates and wires. The speed-independent model employs unbounded gate delays and zero wire delays. This model is usually sufficient for AFSM modules where the causal path algorithm is applied. The fundamental mode assumption and burst-mode design are methodologies that require all internal gates of an AFSM to stabilize before a new input transition can occur [35], [36]. The relative timing design methodology specifies the relative arrival time of related race paths in a circuit from a common timing reference [37].

All examples in this paper employ unbounded speed-independent inertial delays with a semimodular gate model unless otherwise specified. The relative timing methodology is used to specify path based timing constraints that are required to have the implementation conform to the specification.

B. Timed Formal Specification and SSG Generation

Timing and behavior are directly related in circuits. Circuit specifications formally identify reachable transitions that can occur on the primary inputs and outputs of a module. Input and output behavior can be specified based on circuit timing. Timing is required to specify the behavior of the pulse generating circuit of Fig. 1. The specific behavior of internal nets and signals is not part of a formal specification because they are not directly observable by connected modules. Through the verification process, the reachability of every internal net in the circuit netlist is generated. This is the *specification state graph* in Fig. 3.

The specification is required to identify the timed sequential behavior of an AFSM. This specification can directly restrict the reachability of the specification state graph. The specifications for the pulse generator circuit of Fig. 1 and C-element of Fig. 2 are shown in Eqn. 2 and 3 respectively.

$$\mathsf{agent\ PULSE} = \mathsf{i}\uparrow.{}'\mathsf{o}\uparrow.{}'\mathsf{o}\downarrow.\mathsf{i}\downarrow.\mathsf{PULSE}; \tag{2}$$

$$\mathsf{agent\ CE} = \mathsf{a}.\mathsf{b}.{}'\mathsf{c}.\mathsf{CE} + \mathsf{b}.\mathsf{a}.{}'\mathsf{c}.\mathsf{CE}; \tag{3}$$

These specifications are formally defined using the Calculus of Communicating Systems (CCS) where the '.' operator is sequential execution and the '+' operator is nondeterministic choice [38]. Quoted signals are outputs in this model. An up or down arrow can be appended to an event to identify it as a rising or falling transition (and is ignored by CCS). The pulse circuit behavior is specified such that after a rising input, a high pulse will occur on the output before the input lowers. The C-element specification in Eqn. 3 states that when both inputs rise, the output rises, and when both inputs fall, the output falls. CCS employs interleaving concurrency, as can be seen with the choice of a.b or b.a, both of which produce output c.

The specifications for the inverter and AND gate used in the implementation for the pulse generating circuit in Fig. 1 are shown in Eqn. 4 and 5. The Boolean behavior of these circuits are defined with an inertial timing model and are used to create the untimed reachability graph in Fig. 4. The behavior of the inertial inverter is symmetric for rising and falling transitions. The inertial AND gate is not symmetric, so the state is identified in the agent name.

$$\mathsf{agent\ IINV} = \mathsf{a}.(\mathsf{a.IINV} + {}'\mathsf{b.IINV}); \tag{4}$$

$$\mathsf{agent\ IAND000} = \mathsf{a.IANDa00} + \mathsf{b.IAND0b0}; \tag{5}$$

$$\mathsf{agent\ IANDa00} = \mathsf{a.IAND000} + \mathsf{b.IANDab0};$$

$$\mathsf{agent\ IAND0b0} = \mathsf{a.IANDab0} + \mathsf{b.IAND000};$$

$$\mathsf{agent\ IANDab0} = \mathsf{a.IAND0b0} + \mathsf{b.IANDa00} + {}'\mathsf{c.IANDab1};$$

$$\mathsf{agent\ IANDab1} = \mathsf{a.IAND0b1} + \mathsf{b.IANDa01};$$

$$\mathsf{agent\ IAND0b1} = \mathsf{a.IANDab1} + \mathsf{b.IAND001} + {}'\mathsf{c.IAND0b0};$$

$$\mathsf{agent\ IANDa01} = \mathsf{a.IAND001} + \mathsf{b.IANDab1} + {}'\mathsf{c.IANDa00};$$

$$\mathsf{agent\ IAND001} = \mathsf{a.IANDa01} + \mathsf{b.IAND0b1} + {}'\mathsf{c.IAND000};$$

A specification that imposes no restrictions on the input and output behavior of a circuit netlist while applying the delay-insensitive model will produce the full untimed reachable behavior of the circuit. This produces a reachability graph containing 32 states for the pulse generator circuit of Fig. 1 when using inertial gate delays. Under the speed-independent model, the untimed reachability graph contains 8 states and is shown in Fig. 4.

Races and hazards are properties of the circuit netlist. The only means to *remove* a race or hazard from an AFSM is

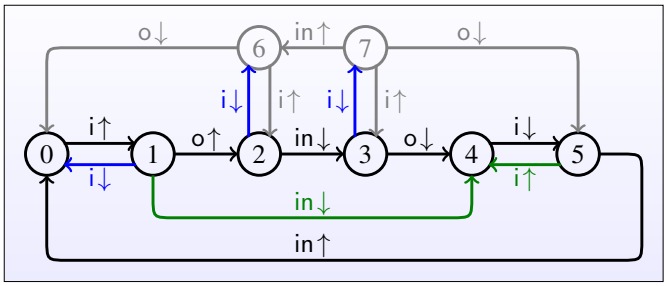

Fig. 4. Untimed speed-independent reachability graph for the circuit of Fig. 1.

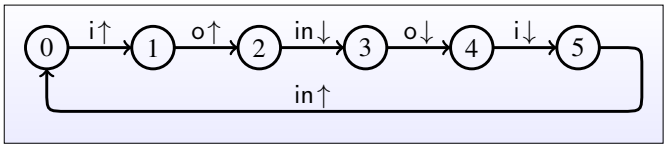

Fig. 5. Specification state graph (SSG) for the pulse circuit of Fig. 1 created by removing arcs from the speed-independent reachability graph of the circuit netlist in Fig. 4 to make it conform to the specification in Eqn. 2.

to change the circuit netlist. However, timing does play a role in the *manifestation* of hazards and races. Thus a second approach to addressing races and hazards is to ensure that hazards become *unreachable* due to circuit timing. This is done by guaranteeing through timing that the races resolve in such a way that the racing signals always occur in a specific order. (The setup and hold times of a Flip-Flop are defined to properly resolve races in its netlist.)

The untimed reachability graph in Fig. 4 includes the races that are a property of the pulse generating circuit netlist. Two race conditions can prevent the output o↑ from occurring. Observe that in state 1 in Fig. 4, the input i has asserted. If the input lowers before the AND gate can fire output o, no output occurs and the circuit returns to state 0. If the inverter switches in↓ faster than the NAND gate switches o↑, then the output will likewise not occur, and the circuit moves to state 4.

A circuit can be made to conform to a specification by constraining the timing of a circuit to ensure that races and hazards in the netlist resolve in a particular order. This is achieved through verification by removing transitions in the untimed behavior of the circuit which cause the circuit to violate the specification. A graph with pruned reachability that is conformant to the specification produces the specification state graph (SSG) used by the causal path algorithm.

The specification state graph for the pulse generating circuit that conforms to the specification in Eqn. 2 is shown in Fig. 5. Two reachable states (6 and 7) and five illegal transitions (three blue and two green) are removed from the untimed reachability graph in Fig. 4.

The blue arcs in Fig. 4 are removed because the specification dictates that the environment will not perform these input behaviors. This makes states 6 and 7, along with the gray arcs, unreachable. The green arcs are removed to make the circuit behavior conform to the specification, ensuring that an output pulse is always generated for every rising input transition. This produces the specification state graph shown in Fig. 5.

Applying the following relative timing constraints to the circuit make it conform to the specification, and are used to create the SSG. The relative timing constraint of Eqn. 6 prunes the three blue arcs. This constraint states that the delay from i↑ to o↓ must be less than the delay from i↑ to i↓. Eqn. 7 prunes the green transition from state 1 to state 4, and Eqn. 8 removes the blue arc from state 5 to state 4.

$$\mathsf{i}\uparrow \mapsto \mathsf{o}\downarrow \prec \mathsf{i}\downarrow \tag{6}$$

$$\mathsf{i}\uparrow \mapsto \mathsf{o}\uparrow \prec \mathsf{in}\downarrow \tag{7}$$

$$\mathsf{i}\downarrow \mapsto \mathsf{in}\uparrow \prec \mathsf{i}\uparrow \tag{8}$$

Note that all three of the relative timing constraints required to generate the specification state graph must be proven correct in system level timing closure. The causal paths between these nets must be identified to validate timing of these constraints.

Since the specification of the pulse generating circuit is performed using an unbounded timing model, an additional user-defined constraint would typically be applied to a pulse mode circuit to constrain the pulse width. Delay engineering can be applied to the circuit to ensure it always generates a pulse of width w or greater. This occurs if the maximum delay through the red path [i↑ o↑] in Fig. 1 is w time units less than the minimum delay through the blue path [i↑ in↓ o↓]. This is expressed as a relative timing constraint in Eqn. 9.

$$\mathsf{i}\uparrow \mapsto \mathsf{o}\uparrow + w \prec \mathsf{o}\downarrow \tag{9}$$

The same approach is used to create the SSG for the C-element in Fig. 2 for the specification in Eqn. 3. Note that in the C-element there are two minimal length paths in the netlist between net a and c: [a ab c] and [a ac c]. Only one of these two reconvergent fanout paths is causal for c↑. A low voltage on c is controlling for the ac NAND gate, which disables the [a ac c] path on rising a transitions due to the logical sequential behavior of the circuit. Thus in the circuit state where the output (and feedback signal) c is low, the only possible causal path through the circuit is [a ab c].

In a like manner timing can restrict reachable circuit behavior. Both a↑ and b↑ are enabling signals for ab↓. Thus, in both minimum and maximum delay paths, the signal that arrives last will be causal for ab↓. Assume the C-element is placed in an environment where a↑ is always guaranteed to occur before b↑. The specification for this C-element is now expressed by Eqn. 10. Under this timing model, there is *no* causal a↑→c↑ path. Path [a ab c] is no longer causal because a↑ always arrives when b is low, so it never enables ab↓.

$$\begin{aligned} &\mathsf{agent\ TC} = \mathsf{a}\uparrow.\mathsf{b}\uparrow.{}'\mathsf{c}\uparrow.\mathsf{C1}; \\ &\mathsf{agent\ C1} = \mathsf{a}\downarrow.\mathsf{b}\downarrow.{}'\mathsf{c}\downarrow.\mathsf{TC} + \mathsf{b}\downarrow.\mathsf{a}\downarrow.{}'\mathsf{c}\downarrow.\mathsf{TC}; \end{aligned} \tag{10}$$

We automatically generated the specification state graph and associated timing constraints for the example circuits discussed in Sec. V. Semimodular representations for each gate are connected and composed in parallel to generate a formal representation of the circuit. Each design is verified with a conformance relationship using bisimulation semantics [39]. The set of relative timing constraints are automatically generated [40].

IV. The Causal Path Algorithm

This paper implements an algorithm that finds causal paths in a circuit. A causal path between two nets in a circuit is defined as follows.

Definition 2. *A causal path is defined between a source and destination net in a circuit netlist $\langle n_s, n_d \rangle$ as a sequence of transitions on nets $[n_s, \ldots, n_d]$ where the following conditions hold:*

1) *Each net, starting with the source net n_s will enable a gate g_i to switch its output net n_i.*
2) *Side channel inputs to each gate g_i are allowed to transition according to the timed behavior of the circuit as long as they do not disable the transition on n_i.*
3) *When the enabled net n_i makes a transition, it must enable another gate g_j to make a transition on net n_j.*
4) *A causal path is found when net n_i, previously enabled to fire in a causal transition sequence, enables gate g_d to fire net n_d and net n_d subsequently makes a transition according to the timed behavior of the circuit.*

Four inputs are required for the causal path algorithm (CPA) as shown in Fig. 3. (1) A directed graph of the implementation

netlist of the module (IG). (2) A specification state graph (SSG) of the logical and timing behavior of the module that includes transitions on all internal edges in the implementation. (3) The Boolean behavior of each node in IG. (4) A set of source and destination edges which may include signal polarity. The specification state graph SSG must identify the Boolean logic value for each net in the module for each state. Nodes in IG map to Boolean functions that are typically implemented as combinational gates in a standard cell library. Edges map to nets in the circuit implementation. The logic level of each net and Boolean function of each gate are used to identify when gate outputs are enabled or disabled using the Boolean difference function.

The causal path algorithm (CPA) creates sets of causal paths between a source net transition e_s and a destination net transition e_d in a circuit module. Rising (+) or falling (-) transition directions can be specified for the nets, e.g. $\langle e_s\text{+}, e_d\text{-}\rangle$. Multiple causal path sets can exist for a source and destination pair. This occurs when the state of a circuit enables or disables potentially causal paths in an AFSM.

The causal path algorithm only reports causal paths that exist inside the design module. Cyclic paths often exist in architectures employing AFSMs where an output from the module eventually excites an input to the module. For example, many of the circuits evaluated in this paper drive handshake communication channels, where an output request signal will result in the environment responding with an acknowledge signal. Gates and causality of external paths outside of the AFSM are not known, and may not be defined until design instantiation.

A. Algorithm Overview

A fundamental aspect of the causal path algorithm is the concept of a enabling and disabling transitions. An edge is *enabled* to fire under the Boolean difference equation: *iff* the value of the edge is not equal to the edge's Boolean function. Some timing models allow edges that are enabled to fire to be disabled before they make a transition. This disabling behavior must be tracked in the causal paths algorithm because disabled transitions break causal transition chains.

One of the most common timing models is the inertial delay model, because timed circuit behavior can be fully represented with Boolean logic. Definition 3 defines the disabling function for the inertial delay model.

Definition 3. *An edge e driven by gate function f is* **disabled** *from firing by signal x when $f_x \neq e \wedge f_x \oplus f_{\overline{x}}$*

Enabled edges will only make transitions based on the timed behavior identified in the specification state graph (SSG). Thus the netlist and Boolean logic of a circuit define enabling and disabling properties of an edge, but concurrency and firing relationships are defined by the SSG.

The causal path algorithm performs a forward search on the SSG starting from all states where the source net can transition in the identified polarity. For example, the initial search state set is $\{0\}$ for the timing endpoints $\langle$i+,o-$\rangle$ in Fig. 5. The destination search nodes include all states with the desired destination transition as an incoming edge (e.g. state set $\{4\}$ for $\langle$i+,o-$\rangle$ in Fig. 5).

A set of causal transition vectors is maintained for each state in the specification state graph. Each causal transition vector represents a subsequence of transitions reached during the search for a potential causal path. Each net in the vector, excluding the last element, represents a causal net transition that has occurred. The last element in each path vector represents a net transition enabled to fire. For the SSG in Fig. 5 in state 1, the transition vector set is {[i+ o+],[i+ in-]}, because i+ enables both o+ and in- to fire in the circuit of Fig. 1

If an edge of the specification state graph is enabled to fire, that edge is added to all causal path segments for which the path propagation condition holds. If the destination transition is satisfied, the path is added to the solution set and not copied to the current state. For timing endpoints $\langle$i+,o-$\rangle$, when state 4 is reached the solution [i+ in- o-] is added to the result set, and the causal path set is now empty, terminating the search for further solutions.

B. Algorithm Data Structures

The implementation graph $IG = \{N, E, T\}$ defines the circuit implementation of the AFSM. It is the circuit netlist shown in Fig. 1, 2, 6, 8, and 10. The IG contains a set $n_i \in N$ of nodes with an associated Boolean function (usually representing standard cell gates in a circuit netlist), a set $e_i \in E$ of edges, which represent nets in the circuit netlist, and a transition relation $t_i \in T$ where t_i is the tuple $(n_i \times e_k \times n_j)$ identifying a transition from node n_i to node n_j on edge e_k. Function $GATEFN(n_i)$ returns the Boolean expression for the gate of node n_i mapped to edges in E.

The specification state graph $SSG = \{S, E, T\}$ defines the timed behavior of the AFSM. These are the graphs or graph segments shown in Fig. 5 and 7. It contains a set $s_i \in S$ of nodes, a set $e_i \in E$ of edges, and a transition relation $t_i \in T$ where t_i is the tuple $(s_i \times e_k \times s_j)$ identifying a transition from node s_i to node s_j on edge e_k. The Boolean logic level for all edges is defined for all nodes in SSG. Function $LVL(s_i, e_j) \mapsto \{0, 1\}$ returns the logic level for edge e_j in node s_i.

The edge set E in IG and SSG are equivalent because they model the same AFSM. The causal path algorithm reports all causal paths between a source edge $e_s \in E$ and destination edge $e_d \in E$. The search is refined to identify rising and falling transitions.

The search state ST maintains the algorithm's results during exploration of the SSG. The search state is defined as $ST = \{EX, DST, CN, CVS, SP, R\}$ where $n_i \in EX$ is a set of nodes to explore, $n_i \in DST$ is a set of destination nodes, $e_i \in CN$ is a set of nets (called causal nets) between the source edge e_s and destination edge e_d in IG. Each state in SSG maps to a causal transition vector set $CVS_i \in CVS$, initialized to the empty set, where each causal transition sequence in $v_i \in CVS_i$ is a vector of causal net edges $e_i \in CN$. SP is a set of cyclic enabling path segments found in the circuit, and R is a set of completed causal paths between the target and destination net.

C. Example

Most of the interesting causal paths in an AFSM are due to reconvergent fanout and cycles. Fig. 6 is a circuit example demonstrating reconvergent fanout.

Three specification state graph segments based on different timing assumptions are presented in Fig. 7 for the circuit of Fig. 6. These examples show the calculated causal transition vector set for each state. Each causal transition vector set identifies the causal transition sequence from some source edge e_s to the current edge.

Assume that the algorithm is searching for causal paths from B+→F+ in the circuit of Fig. 6, and that $\forall e_i \in E, LVL(0, e_i) \mapsto 0$ in state 0 for all three SSG examples in Fig. 7. Therefore, nets A, B, C, D, and E must all rise to create a causal path to net F. Nets D and E may or may not be causal based on the timed behavior of the specification state graph. If D always occurs before E, D will not be causal and vice versa.

The causal path algorithm first walks the implementation graph IG netlist and generates the set of causal nets CN between the timing endpoints. In this example, for the endpoints ⟨B+,F+⟩ the CN set is {B D E F}. The algorithm next finds all transitions with an edge equal to the source edge e_s with the correct polarity (B+), places the initial node of each transition in the set to be explored EX, and creates a causal transition vector set with a transition vector containing the source edge e_s ({[B]}) for each of these nodes.

The search for the destination transition (F+) begins by searching all edge transitions from each node in EX, building a vector of nets from the causal transition vector set CVS. The search continues until all enabled causal transitions are propagated or a transition is taken on the destination net e_d with the proper polarity (F+).

The SSG of Fig. 7(a) starts with $EX = \{1\}$ where all nets are zero except for net A. The B+ transition from node 1 is evaluated, and the gate function $GATEFN(n_i)$ is calculated for all destination nodes for net B in graph IG (gates D and E). If the gate is enabled to fire, and the output net of the gate is a member of the causal net set CN, the net is appended to all paths in the causal transition vector set CVS. CVS is then used to continue the causal path search at the destination node.

The search continues until all causal transition vectors are propagated. A causal path is found when a transition from the current state matches the destination net e_d with the correct polarity. At this point the transition vectors in the CVS for the node are copied to the result R and an empty CVS is copied to the destination node of the transition, terminating the search along this path.

The search of the SSG in Fig. 7(a) proceeds by adding the enabled causal net D $\in CN$ on transition $1 \xrightarrow{B+} 2$ and F $\in CN$ on transition $5 \xrightarrow{D+} 6$. When F+ occurs in transition $6 \xrightarrow{F+} 7$, the search terminates with the causal path set {[B D F]}. Only net D is causal because the timing in the SSG of Fig. 7(a) specifies that the B+ transition always occurs before C+.

The SSG of Fig. 7(b) has no causal paths. Net B+ only enables causal net D to transition, but timing of this specification indicates that transition D+ always occurs before E+.

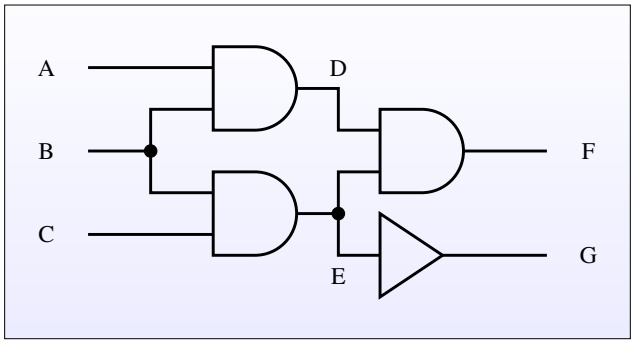

Fig. 6. Reconvergence Example

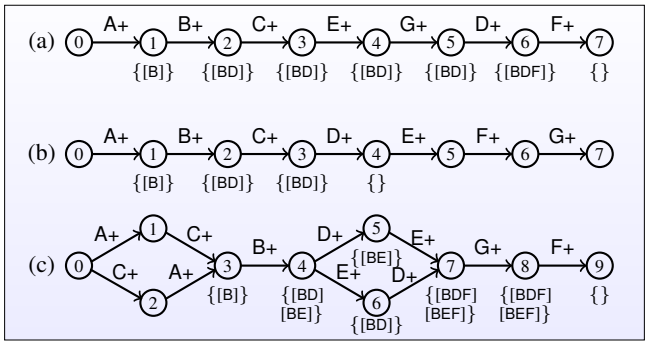

Fig. 7. Three specification state graphs snippets for the circuit of Fig. 6

The SSG in Fig. 7(c) has more concurrency, allowing nets A and C to fire concurrently before B. Thus on the $3 \xrightarrow{B+} 4$ transition the causal transition vector set for node 4 has two paths because gates D and E are both sensitized by B+. For each of the transitions out of state 4, one of these paths is removed because the associated transition does not enable another signal in the causal net set CN. Each of the reconvergent transitions from $5 \xrightarrow{E+} 7$ and $6 \xrightarrow{D+} 7$ add their updated CVS elements because these transitions enable CN member net F to transition. The $8 \xrightarrow{F+} 9$ transition moves both causal transition vectors to the result set R, as they are now causal paths between the identified timing endpoints.

D. Implementation

Pseudocode for the causal path algorithm is shown in Algorithm 1. Lines 2–6 initialize the data structures. Causal nets CN are those nets that can be traversed to connect the source net and the destination net in the implementation graph IG. These are the only nets to be included in the causal paths. The specification state graph SSG is searched to find all nodes that have a transition on edge e_s with the appropriate polarity pol_s. All such states map to an initial causal transition vector set CVS_i for state i consisting of the single net e_s. Destination nodes are those which accept a transition on the destination edge e_d with the identified polarity.

The algorithm then performs a search on each of the state nodes in EX until the set is empty, indicating that all causal paths have been found. In each iteration, line 8 removes a state node n_i from EX. The algorithm then searches across all transitions emanating from this state. Lines 10–13 collect

Algorithm 1 Causal Path Algorithm

```
1: procedure CAUSALPATHS(IG, SSG, gateFns, e_s, e_d, pol_s, pol_d)
2:     R = SP = {};                                          ▷ Initialize results
3:     CN = GetCausalNets(IG, e_s, e_d);
4:     EX = GetSourceNodes(SSG, e_s, pol_s);
5:     ∀n_i ∈ EX, CVS_i = {[e_s]};  ▷ Create causal transition set for start states
6:     DST = GetDestNodes(SSG, e_d, pol_d);
7:     while EX != {} do
8:         n_i = removeFirst(EX);                  ▷ Get next state, remove from EX
9:         for each (n_i, e_k, n_j) ∈ outputTrans(SSG, n_i) do
10:            enabSig = CN ∩ getEnabledNets(IG, SSG, gateFns, e_k, n_i);
11:            disabSig = CN ∩ getDisabledNets(IG, SSG, gateFns, e_k, n_i);
12:            enabP = getEnabledPaths(e_k, enabSig, CVS_i);
13:            disabP = getDisabledPaths(e_k, disabSig, CVS_i);
14:            CVS'_i = CVS_i
15:            if e_k == e_d ∧ n_j ∈ DST then                ▷ Causal path found
16:                R = R ∪ senP;
17:            else          ▷ Otherwise append causal edges to enabled subpaths
18:                ∀v_x ∈ enabP, ∀e_x ∈ enabSig, CVS'_i ∪ v_x.e_x;
19:            CVS'_i = CVS'_i − enabP;                  ▷ Remove enabled subpaths
20:            CVS'_i = CVS'_i − disabP;                 ▷ Remove disabled subpaths
21:            its = getCyclicEnablingPaths(CVS'_i);
22:            CVS'_i = CVS'_i − its;                    ▷ Remove cyclic enabled paths
23:            SP = SP ∪ its;                            ▷ Save cyclic enabled paths
24:            CVS_j = CVS_j ∪ CVS'_i                    ▷ Propagate causal vectors
25:            if CVS'_i ≠ {} then                       ▷ Only search new paths
26:                EX = EX ∪ n_j;
27:    return R, SP;
```

and produce sets nets and paths. All nets enabled and disabled by the transition edge e_k are calculated. Enabled paths (or path sets) are those causal vectors in a path set with a final edge that equals the passed edge e_k. Disabled paths are those whose final edge are disabled by the passed edge e_k.

A solution is found when the transition moves to a node in the destination state set DST and the transition's edge equals the destination edge. The paths enabled by this edge are copied into the solution set. Otherwise path propagation (line 18) is performed by appending the enabled signals to each enabled path and adding these to the current causal transition vector set. After adding propagated paths, the causal path set is modified by removing disabled paths and enabled paths that do not enable a new edge from the temporary transition vector set CVS'_i. After removing non-propagated paths, CVS'_i will only contain propagated paths and paths that are neither enabled nor disabled.

The causal transition vector set CVS'_i is then searched for cyclic-enabling paths. A cyclic-enabling path is found when path vector $v_x \in CVS'_i$ contains an enabled edge transition of the same polarity more than once. These cyclic-enabling paths are removed from the causal transition vector set CVS'_i to allow termination, and added to the cyclic-enabling path structure SP. Paths in CVS'_i are then added to CVS_j (line 24). If there are new causal paths to search in CVS'_i, then the next transition node is added to the set of nodes to be searched (line 26).

E. Complexity

AFSMs can have an unbounded number of causal paths due to the existence of combinational cycles. Unbounded length causal paths may occur when circuits have self-enabling cyclic causal vectors. Generating each unique unbounded causal vector would result in unbounded run time. Therefore, one of the termination conditions for the causal path algorithm is the detection of cyclic self-enabling paths.

Even when ignoring unbounded path sets, the complexity of calculating causal paths is quite high. Therefore, causal path generation is performed once for each AFSM module as a characterization step. Efficient STA algorithms are applied at design time to ensure that the AFSM meets correctness and performance requirements imposed by each system architecture where the module is instantiated.

The main loop of the causal path algorithm performs a modified depth-first search which requires a time complexity of $\mathcal{O}(S + E)$ where S and E represent vertexes and edges of the SSG [41]. However, unlike the depth-first search algorithm, which terminates when the search arrives at a previously visited node, the causal path algorithm has two termination conditions which depend on causal vector set CVS_i.

The first termination condition detects self-enabling cycles. The second termination condition requires that each node s_i in the SSG is explored for all causal transition vectors arriving at s_i. The causal path algorithm generates an exhaustive set of causal paths which, due to reconvergent fanout, may cause $|CVS_i|$ to grow exponentially with respect to the number of states in the graph. It is important to note that $|CVS_i|$ depends on the causal behavior of the circuit specified in the specification state graph, not the complexity of the circuit implementation graph (IG). The SSG normally contains many more nodes that the circuit graph.

Due to the above termination conditions, the number of explorations for each state s_i is bound by the number of unique causal segments and cycles. This is expressed as $\mathcal{O}(|CVS_i|)$ where CVS_i is the number of causal transition vectors.

Combining the exploration bounds with complexity of the depth-first search algorithm, the number of iterations of the while loop can be bound by $\mathcal{O}((S + T) * MAX(|CVS_i|))$, which shows that the algorithm achieves polynomial complexity with respect to the size of the SSG but may grow exponentially with respect to the circuit size due to the number of causal path vectors in CVS_i.

V. RESULTS

Results are reported for the execution of the causal path algorithm on over 100 circuit designs using varying timing models and methodologies, including burst-mode, speed-independent, and delay-insensitive designs. Table I shows results for 94 linear pipeline handshake controllers in a protocol family, a network-on-chip router control module [42], [36], a delay insensitive handshake controller [43], a toggle handshake controller, two pulse generator circuits in which the input is allowed to fall after the first output transition (clk-pulse-si-specx) and the input is allowed to fall only after the second output transition (clk-pulse-si-spec), speed-independent and burst-mode C-element designs, and a burst mode handshake controller.

The results for the family of 94 speed-independent handshake controllers have all been aggregated together in Table I to save space. Data for these controllers identify the minimum, median, and maximum ($min/median/max$) values for the set of 94 different protocol modules.

TABLE I
RESULTS FOR 102 DIFFERENT SEQUENTIAL AND TIMED DESIGN MODULES

1 Modules Set	2 No. of Mod.	3 No. of Causal EndPts	4 No. of IG Std Cells	5 No. of IG Nets	6 No. of SSG States	7 No. of SSG State Transitions	8 No. of Circuit Paths	9 No. of Causal Paths	10 No. of Causal Path Nets	11 No. of Cyclic Paths	12 No. Self enabled Paths	13 No. of Searches	14 No. of Explored States	15 Search Time (ms)
4-phase-cont-set	94	5/24/99	3/10/20	6/13/23	13/168/2040	15/378/6472	7/19/104	4/21/62	5/11/23	0/12/53	0/2/15	5/29/187	2/204/61953	1/166/151079
sbuf-send-ctl	1	15	10	14	103	194	21	12	13	0	0	18	6/28/88	4/66/198
wchb	1	6	8	11	48	73	8	6	9	0	0	7	12/25/70	13/22/118
toggle	1	13	9	13	116	223	21	14	11	0	0	13	4/13/53	2/29/160
clk-pulse-si-spec	1	2	2	3	6	6	2	2	3	0	0	2	4/4/4	1/1/1
clk-pulse-si-specx	1	2	2	3	8	10	2	2	3	0	0	2	6/6/6	1/2/2
C-element-si	1	4	2	4	11	14	3	4	4	0	0	4	5/6/6	2/2/2
C-element-bm	1	4	4	6	29	49	4	6	6	0	0	4	5/16/26	1/5/12
bm-controller	1	12	7	9	87	174	16	10	8	2	0	12	56/103/160	47/88/121

The 94 circuits in the first row of Table I are a subset of the 159 specifications in the family of speed-independent linear pipeline protocols using a four-phase return to zero protocol where data is valid on the rising edge of the left request signal [44]. These 94 protocols are those that can be automatically synthesized by the tool Petrify [45], and for which relative timing constraints can also be automatically generated [40]. The amount of concurrency between the input (left) channel (lr, la) and output (right) channel (rr, ra) differs based on the particular protocol.

The designs in the first row and C-element-si are synthesized using Petrify [45]. A set of burst-mode designs are also included in this example set. The toggle was synthesized with 3D [46]. The sbuf-send-ctl, C-element-bm, and bm-controller designs are synthesized with MEAT [36]. The specification state graph (SSG) for each implementation was generated with a formal verification engine using an inertial speed-independent delay model with relative timing constraints and semimodular Boolean gate behavior. The relative timing constraints generated from the proof system are used to prune reachable states of the circuit implementation (as described in Sec. III-B). The same verification engine is also used to prove conformance of the SSG to the specification used to synthesize each module.

The first column of the table lists the module name or set. The second column (No. of Modules) lists the number of designs in the set. The third column (No. of Causal Endpoints) identifies the total input and output endpoint conditions which have causal paths for the modules in the set. The fourth and fifth columns identify the number of standard cells ($|N|$ in IG) and nets ($|E|$ in IG) of the implementation graph for each module. The sixth and seventh columns identify the number of states ($|S|$ in SSG) and transitions ($|T|$ in SSG) of the specification state graph for each module. The eighth column (No. of Circuits paths) reports the number of acyclic paths between a source and destination endpoints for a circuit (IG) in the set. These paths may or may not be causal; this is a measure of complexity of the circuit. The ninth column (No. of Causal Paths) identifies the number of unique causal paths identified for the module. The tenth column (No. of Causal Path Nets) identifies the number of unique signals from the implementation graph ($|E|$ in IG) which are also contained in a causal path transition for any of the evaluated endpoint conditions. The eleventh column (No. of Cyclic Paths) identifies how many of the unique causal paths contain cycles. The twelfth (No. Self enabled Paths) identifies the number of causal unique self-enabling cyclic paths $|SP|$, which could potentially lead to an unbounded length causal path in a design. The thirteenth column (No. of Searches) is the number of unique endpoint condition pairs searched in order to determine causal paths between the edges. The fourteenth column (No. of Explored States) is the number ($min/median/max$) of states ($|S|$ in SSG) explored by the causal path algorithm for an endpoint pair. The last column (Search Time (ms)) is the average time ($min/median/max$) taken for the algorithm to perform an exhaustive causal path search for endpoint pairs in the circuit, including endpoints with no causal paths.

A. Evaluation and examples

All of the AFSM designs are relatively small, ranging from two to 20 standard cells and three to 23 nets. The size of the timed specification state graphs is much larger, and ranges from six to 2,040 states. The size of the causal path sets for the examples range from two for the simple pulse generator circuit to 62 for one of the four phase handshake controllers. One of the handshake controllers has 53 cyclic paths while another has 15 self-enabling loops. The time to calculate all causal paths between timing endpoints ranges from a millisecond to a little less than three minutes.

While all of the 94 controllers implemented in the example set are of the same family, there is considerable variation in the results. Some controllers contain cyclic self-enabling paths, which may dictate the critical circuit delay in circuit performance under certain conditions.

Reconvergent fanout creates complicated and intriguing paths and behaviors in combinational circuits with feedback. Reconvergent fanout is affected by both logic and timing, as shown in the simple example of Sec. IV-C. In the set of 94 controllers, causal paths do not exist between some edge pairs (such as between lr- and la-) due to both concurrency reduction and timing constraints.

Reconvergence can occur with either enabling or controlling signals. For example, the rising inputs to the AND gate F in Fig. 6 requires that both inputs assert before the output can change. If both paths are causal, then both paths must assert at the point of reconvergence.

We use the circuit in Fig. 8 as an example. This example has enough complexity and simplicity to highlight some of the

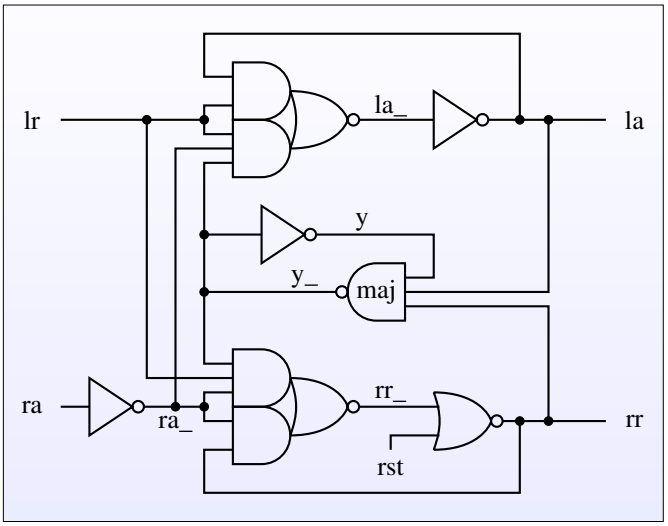

Fig. 8. Circuit implementation of bm-controller in Table I.

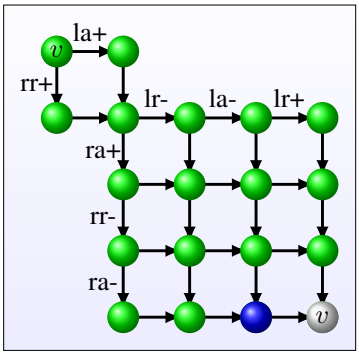

Fig. 9. Specification for the controller in Fig. 8. The blue state is the reset state. The gray state, marked v, is equivalent to the state at the top left corner, also marked v.

TABLE II
CAUSAL PATHS FOR FIG. 8 CIRCUIT

Row	Endpoints	Causal Paths
1	⟨lr+, la+⟩	[lr+ la_- la+]
2	⟨lr+, rr+⟩	[lr+ rr_- rr+]
3	⟨lr-, la-⟩	[lr- la_+ la-]
4	⟨ra+, rr-⟩	[ra+ ra_- rr_+ rr-]
5	⟨ra-, rr+⟩	[ra- ra_+ rr_- rr+]
6	⟨ra-, la+⟩	[ra- ra_+ la_- la+]
7	⟨lr-, la+⟩	[lr- la_+ la- y_+ la_- la+]
8	⟨ra+, rr+⟩	[ra+ ra_- rr_+ rr- y_+ rr_- rr+]
9	⟨ra+, la+⟩	[ra+ ra_- rr_+ rr- y_+ la_- la+]
10	⟨lr-, rr+⟩	[lr- la_+ la- y_+ rr_- rr+]

nuances with causal paths and AFSMs. This circuit has seven gates, nine nets, two causal paths with combinational cycles, and an SSG of 87 states. The 18 state behavioral specification of the circuit is shown in Fig. 9.

The full set of causal paths for this circuit are listed in Table II. The first six rows are the expected minimal input to output causal paths for a burst-mode circuit. The three endpoint pairs starting with an lr transition go through the minimum two gates to reach the specified output transition. Likewise the three endpoint pairs starting with ra transitions go through the minimum three gates. Under the fundamental mode assumption of the burst-mode timing model, these would be all of the valid causal paths.

While the circuit is synthesized using the fundamental mode timing model, the SSG is created using the speed-independent model, which affords more concurrency. While this violates the fundamental mode assumption used to synthesize the circuit, the formal verification has proven the extra concurrency to still be correct and conform to the burst-mode specification. This demonstrates the criticality of the timing model used in determining causal paths, as well as AND/OR causality of the logic. This extra concurrency allows additional internal AFSM causal paths when the environment responds before the internal state of the circuit stabilizes.

Causal paths in rows 7 and 8 are cyclic. These create "pulse generator" cycle paths that are extremely difficult to identify without the causal path algorithm. These paths can control the performance of the circuit. Observe the ⟨lr-, la+⟩ causal path. The la- transition creates a fanout by driving the majority gate and the environment. There is a non-controlling AND based reconvergence in these two paths in the lower AND logic of the la_ AOI gate. If the external la- → lr+ path is faster than the la- → y_+ path, then the fully internal causal cycle [la y_ la_ la] will dictate the minimum low la pulse width of this controller. This complicated (and likely unreachable based on system level timing) cyclic path in the circuit is a true causal pulse generating path in the design. This cyclic pulse generating causal path does not affect correct operation of the circuit, but limits the frequency of the la pulse of the circuit. A similar reconvergent race between internal gates and the environment exists in the ⟨ra+, rr+⟩ causal path, which dictates the minimum duration of the low pulse on rr.

In the ninth row, the rr- transition enables ra- and y_+. When the environmental response on the right handshake channel (rr- → ra-) and the ra inverter are faster than the majority gate asserting y_+, the y_+ event is causal, allowing it to causally assert la. The last row in Table II is the dual of this event, where the la- transition enables both lr+ and y_+. The rest of the path is causal under the unbounded delay model because lr+ can occur before y_+.

Fig. 10 shows one of the circuits from the 4-phase handshake controller set with a self-enabling cyclic causal path. Its specification is shown in Fig. 11. Such paths can result in an unbounded number of causal paths in a design. Petrify uses a speed-independent delay model to synthesize logic. Thus all of the paths in this circuit are legitimate delay paths under the timing model used to synthesize and technology map the circuit.

The full set of causal paths for this circuit are shown in Table III. All paths but one in this circuit change the state variable before asserting the output of the circuit. This helps reduce timing constraints, but produces slower circuits. The first six paths are acyclic, the last five paths (rows 7 through 11) all have cycles.

The last path is a self-enabling cycle. The causal transition sequence [csc0+ csc0_- rr+ la_- la+ rr- rr_+ x5_- csc0+] is an inter-

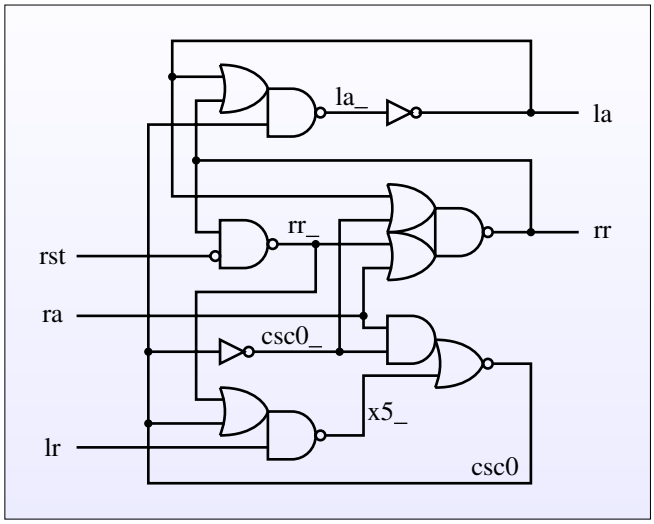

Fig. 10. Handshake controller with self-enabling loop.

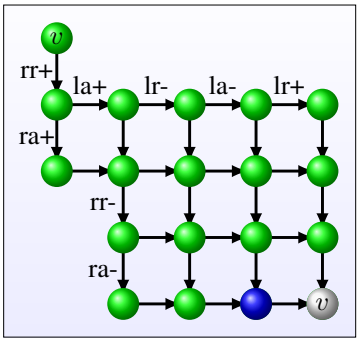

Fig. 11. Specification for the controller in Fig. 10. The blue state is the reset state. The gray state, marked v, is equivalent to the state at the top left corner, also marked v.

TABLE III
CAUSAL PATHS FOR FIG. 10 CIRCUIT

	Endpoints	Causal Paths
1	⟨lr+, la+⟩	[lr+ x5_- csc0+ csc0_- rr+ la_- la+]
2	⟨lr+, rr+⟩	[lr+, x5_-, csc0+, csc0_-, rr+, la_-, la+]
3	⟨lr-, la-⟩	[lr-, x5_+, csc0-, la_+, la-]
4	⟨ra+, rr-⟩	[ra+, rr-]
5	⟨ra-, rr+⟩	[ra-, csc0+, csc0_-, rr+]
6	⟨ra-, la+⟩	[ra-, csc0+, csc0_-, rr+, la_-, la+]
7	⟨lr+, rr-⟩	[lr+, x5_-, csc0+, csc0_-, rr+, la_-, la+, rr-]
8	⟨ra+, rr+⟩	[ra+, rr-, rr_+, x5_-, csc0+, csc0_-, rr+]
9	⟨ra+, la+⟩	[ra+, rr-, rr_+, x5_-, csc0+, csc0_-, rr+, la_-, la+]
10	⟨ra-, rr-⟩	[ra-, csc0+, csc0_-, rr+, la_-, la+, rr-]
11	⟨ra-, csc0++⟩	[ra-, csc0+, csc0_-, rr+, la_-, la+, rr-, rr_+, x5_-, csc0+]

nally oscillating transition sequence which must stabilize for the controller to complete a handshake cycle on both the left and right hand channels. Cyclic self-enabling paths represent unbounded length causal sequences that exist in a controller. When rr+ asserts, it enables ra+ (due to the specification) and la_-, which then enables la+. The la+ transition is causal for rr- when ra+ asserts before la+. The la+ transition now enables two parallel causal sequences, la+ $\rightarrow$ lr+ and la+ $\rightarrow$ rr_+. If the left channel handshake sequence [la+ lr- la- lr+] is faster than the [la+ rr- rr_+] sequence, then x5_- is caused by rr_+ rather than lr+. Another interesting attribute of the self-enabling path is that the path's causality persists across multiple input transition vectors. Unlike synchronous sequential true path analysis which assumes that state bits are stabilized before a clock edge, AFSM modules may depend on previously enabled state bit transitions, which must stabilize before the circuit can perform computations for the next input vector. These paths may result in self-enabling cycles that can have unbounded length causal paths. Self-enabling paths may not be critical under STA evaluation in real circuit implementations, but must be evaluated to ensure that they do not limit performance and may need to be evaluated to ensure max-delay timing constraints are satisfied at a system level.

VI. CONCLUSION

The causal path algorithm presented here is a method of identifying causal paths for sequential finite state machines as well as timed and pulse based logic. The algorithm supports arbitrary timing methodologies and was used to evaluate over 100 designs that are implemented using different synthesis tools and timing formalisms. Examples of non-controlling reconvergence, which enforce a minimum pulse width between input and output transitions are demonstrated. Cyclic causal paths and self-enabling paths, which may result in unbounded length causal paths, are demonstrated in the example set. The results provide insight into the interaction between timing, concurrency, and causality for cyclic sequential circuits. The causal path algorithm developed here provides a step towards automatically applying static timing analysis to sequential finite state machines and pulsed based logic design. Identified causal paths can be used to enable accurate and automatic system level timing analysis, validation, and architecture optimization using static timing analysis algorithms on systems which contain timed logic or sequential controllers with cyclic paths.

ACKNOWLEDGMENT

The authors would like to thank Venkata Nori for his help in preparing circuit examples used in this work.

REFERENCES

[1] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York, New York: Wiley-Interscience, 1969.
[2] J. E. Robertson, "Problems In The Physical Realization of Speed Independent Circuits," in *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT)*, Oct 1961, pp. 106–108.
[3] D. E. Kunth, *The Art of Computer Programming*. USA: Addison-Wesley, 1973, vol. 1.
[4] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, Jun 1989.
[5] S. M. Burns, "Performance Analysis and Optimization of Asynchronous Circuits," Ph.D. dissertation, California Institute of Technology, USA, 1992.
[6] N. Xiromeritis, S. Simoglou, C. Sotiriou, and N. Sketopoulos, "Graph-Based STA for Asynchronous Controllers," in *29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, July 2019, pp. 9–16.
[7] A. Dasdan, "Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, pp. 385–418, oct 2004.

[8] R. M. Karp and J. B. Orlin, "Parametric Shortest Path Algorithms with an Application to Cyclic Staffing," *Discrete Applied Mathematics*, vol. 3, no. 1, pp. 37–45, 1981.

[9] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello, "An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems," *IEEE Transactions on Computers*, vol. 44, pp. 1306–1317, nov 1995.

[10] J. Gunawardena, "Timing Analysis of Digital Circuits and the Theory of Min-Max Functions," in *ACM/SIGDA International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, Sept 1993, pp. 0–9.

[11] O. M. A. Pnueli, "Timing Analysis of Asynchronous Circuits using Timed Automata," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 1995, pp. 189–205.

[12] W. Hua and R. Manohar, "Exact Timing Analysis for Asynchronous Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 203–216, 2018.

[13] T. K. Lee, "A General Approach to Performance Analysis and Optimization of Asynchronous Circuits," Ph.D. dissertation, California Institute of Technology, Pasadena, California, USA, 1995.

[14] A. Smirnov and A. Taubin, "Heuristic based throughput analysis and optimization of asynchronous pipelines," in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009, pp. 162–172.

[15] M. Najibi and P. A. Beerel, "Performance Bounds of Asynchronous Circuits with Mode-Based Conditional Behavior," in *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, 2012, pp. 9–16.

[16] C. E. Molnar, I. W. Jones, W. S. Coates, J. K. Lexau, S. M. Fairbanks, and I. E. Sutherland, "Two FIFO Ring Performance Experiments," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 297–307, February 1999.

[17] G. Gill and M. Singh, "Bottleneck Analysis and Alleviation in Pipelined Systems: A Fast Hierarchical Approach," in *15th International Symposium on Asynchronous Circuits and Systems*. IEEE, May 2009, pp. 195–205.

[18] M. R. Greenstreet and K. Steiglitz, "Bubbles Can Make Self-Timed Pipelines Fast," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 2, no. 3, pp. 139–148, 1990.

[19] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous Design Using Commercial HDL Synthesis Tools," in *International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, April 2000, pp. 114–125.

[20] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC Flow for GHz Asynchronous Designs," *IEEE Design and Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[21] C. Brej, "Blame Passing for Analysis and Optimisation," *UK Asynchronous Forum, Newcastle upon Tyne, 2007*, vol. 18, p. 6, 2007.

[22] K. Fazel, L. Li, M. Thornton, R. B. Reese, and C. Traver, "Performance Enhancement in Phased Logic Circuits Using Automatic Slack-matching Buffer Insertion," in *Proceedings of the 14th ACM Great Lakes Symposium on VLSI*. Association for Computing Machinery, 2004, pp. 413–416.

[23] W. Hua, Y. Lu, K. Pingali, and R. Manohar, "Cyclone: a static timing and power engine for asynchronous circuits," in *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE, 2020, pp. 11–19.

[24] A. Winstanley and M. Greenstreet, "Temporal Properties of Self-Timed Rings," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science. Springer, 2001, pp. 140–154.

[25] G. Gimenez, A. Cherkaoui, G. Cogniard, and L. Fesquet, "Static Timing Analysis of Asynchronous Bundled-Data Circuits," in *24th International Symposium on Asynchronous Circuits and Systems*. IEEE, May 2018, pp. 110–118.

[26] K. S. Stevens, Y. Xu, and V. Vij, "Characterization of Asynchronous Templates for Integration into Clocked CAD Flows," in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009, pp. 151–161.

[27] W. Lee, T. Sharma, and K. S. Stevens, "Path Based Timing Validation for Timed Asynchronous Design," in *The 29th International Conference on VLSI Design (VLSID)*. IEEE, Jan 2016, pp. 511–516.

[28] P. C. McGeer and R. K. Brayton, *Integrating Functional and Temporal Domains in Logic Design: The False Path Problem and Its Implications*. USA: Kluwer Academic Publishers, 1991.

[29] H.-C. Chen and D. Hung-Chang, "Path Sensitization in Critical Path Problem," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 2, pp. 196–207, 1993.

[30] D. H. C. Du, S. H. C. Yen, and S. Ghanta, "On the General False Path Problem in Timing Analysis," in *26th ACM/IEEE Design Automation Conference*, 1989, pp. 555–560.

[31] J. Benkoski, E. Vanden Meersch, L. J. Claesen, and H. De Man, "Timing Verification Using Statically Sensitizable Paths," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 10, pp. 10 723–10 784, 1990.

[32] P. C. McGeer and R. K. Brayton, "Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network," in *26th ACM/IEEE Design Automation Conference*, 1989, pp. 561–567.

[33] P. Ashar, S. Dey, and S. Malik, "Exploiting Multi-cycle False Paths in the Performance Optimization of Sequential Logic Circuits," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 14, no. 9, pp. 1067–1075, 1995.

[34] G. Boole and J. F. Moulton, *A Treatise on the Calculus of Finite Differences*. Macmillan and company, 1872.

[35] E. J. McCluskey, "Fundamental Mode and Pulse Mode Operations of Sequential Circuits," in *IFIP Working Conference on Design Methodologies*, 1962, pp. 725–730.

[36] W. S. Coates, A. L. Davis, and K. S. Stevens, "Automatic Synthesis of Fast Compact Self-Timed Control Circuits," in *IFIP Working Conference on Design Methodologies*, April 1993, pp. 193–208.

[37] K. S. Stevens, R. Ginosar, and S. Rotem, "Relative Timing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 11, pp. 129–140, Feb. 2003.

[38] R. Milner, *Communication and Concurrency*, ser. Computer Science, C. Hoare, Ed. London: Prentice Hall International, 1989.

[39] K. S. Stevens, "Practical Verification and Synthesis of Low Latency Asynchronous Systems," Ph.D. dissertation, University of Calgary, Calgary, Alberta, Canada, September 1994.

[40] Y. Xu and K. S. Stevens, "Automatic Synthesis of Computation Interference Constraints for Relative Timing," in *26th International Conference on Computer Design*. IEEE, Oct. 2009, pp. 16–22.

[41] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, Jul. 2009.

[42] K. S. Stevens, S. V. Robison, and A. L. Davis, "The Post Office – Communication Support for Distributed Ensemble Architectures," in *Proceedings of 6th International Conference on Distributed Computing Systems*, May 1986, pp. 160 – 166, best paper award.

[43] A. M. Lines, "Pipelined Asynchronous Circuits," Master's thesis, California Institute of Technology, Pasadena, CA, 1998.

[44] S. Nagasai, K. S. Stevens, and G. Birtwistle, "Concurrency Reduction of Untimed Latch Protocols – Theory and Practice," in *International Symposium on Asynchronous Circuits and Systems*. IEEE, May 2010, pp. 26–37.

[45] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. E80-D, no. 3, pp. 315–325, 1997.

[46] K. Y. Yun, "Synthesis of Asynchronous Controllers for Heterogeneous Systems," Ph.D. dissertation, Stanford University, Aug 1994.

Mackenzie J. Wibbels (S'16) Received his bachelor's degree in computer engineering from the University of Utah, Salt Lake City. He is currently pursuing a Ph.D. degree at the University of Utah while working at Granite Mountain Technologies, which commercializes high performance low power integrated circuits.

His current research interests include asynchronous circuit design, hardware verification and optimization, very large sale integration, and electronic design automation.

Kenneth S. Stevens (S'83-M'84-SM'99) received the B.A. degree in biology, as well as the B.S., M.S., and Ph.D. degrees in computer science from the University of Utah, Salt Lake City, and the University of Calgary, Calgary, AB, Canada. He is currently a Professor with the Department of Electrical and Computer Engineering, University of Utah. He has split time between industry and academia, holding positions with the Fairchild/Schlumberger Laboratory for AI Research, Palo Alto, CA, the Schlumberger Palo Alto Research Laboratory, Palo Alto, and Hewlett Packard Laboratories, Palo Alto, the Air Force Institute of Technology, Dayton, OH, and Intel's Strategic CAD Laboratories, Hillsboro, OR. He holds several patents, is the co-founder of a software and integrated circuit company, and has developed public domain software for the Free Software Foundation. His current research interests include relative timing specifications and analysis, asynchronous circuits, very large scale integration, and the design of software and integrated circuits and systems.