Automatic Synthesis of Computation Interference Constraints for Relative Timing Verification

Yang Xu and Kenneth S. Stevens
Electrical and Computer Engineering Department
University of Utah
{*yxu , kstevens*}*@ece.utah.edu*

***Abstract*— Asynchronous sequential circuit or protocol design requires formal verification to ensure correct behavior under all operating conditions. However, most asynchronous circuits or protocols cannot be proven conformant to a specification without adding timing assumptions. Relative Timing (RT) is an approach to model and verify circuits and protocols that require timing assumptions to operate correctly. The process of creating path-based RT constraints has previously been done by hand with the aid of a formal verification engine. This time consuming and error prone method vastly restricts the application of RT and the capability to implement circuits and protocols. This paper describes an algorithm for automatic generation of RT constraints based on signal traces generated from a formal verification (FV) engine that supports relative timing constraints. This algorithm has been implemented in a CAD tool called Automatic Relative Timing Identifier based on Signal Traces (ARTIST) which has been embedded into the FV engine. A set of asynchronous and clocked designs and protocols have been verified and proven to be hazard-free with the RT constraints generated by ARTIST which would have taken months to perform by hand. A comparison of RT constraints between hand-generated and ARTIST generated constraints is also described in terms of efficiency and quality.**

I. Introduction

Timing is an inherent quality and correctness aspect of circuit and protocol design, whether the designs are clocked or asynchronous. Furthermore, very few, if any, designs operate entirely independently of timing – even if the designs and protocols are "delay-insensitive" [1]. Unfortunately, the ability to reason about joint behavior and timing in sequential systems and protocols is a complicated task, often resulting in memory or state space complexity that consists of double exponentials.

Several approaches have been taken to mitigate this complexity. In the world of asynchronous design, *delay-insensitive* (DI) protocols and circuits have been used to *avoid* timing problems. Theoretically these systems operate correctly with arbitrary wire and device delays. In practice, creating DI systems results in circuits much larger, slower, and power hungry than similar timed circuits [2]. Even worse, it is not possible to design complex systems without some timing (resulting in the *quasi delay-insensitive* (QDI) model).

A second approach has been to specify the upper and lower bound on delay between signal events becoming enabled and firing [3], [4], be it a device, wire, or protocol signal. This timing approach results in high state, memory, or computation complexity so significant effort has been made to reduce the complexity of such joint behavior and metric timed verification systems. Another problem with this approach is that it requires the designer to estimate the min-max delay in a reasonable range such that it meets the accurate delay extracted from post-layout parameters. Further, the impact that a change to the delay of a single component has on the correct behavior of a system as a whole cannot be known by an engineer, making design changes (ECO's) nearly impossible to perform without re-running verification.

Another approach has been to apply a *unit delay model* to the devices and wires [5]. This can result in a provably correct system. Sutherland showed that transistor sizing of a circuit can be constrained to yield unit delays to solve this problem [6]. This technique simplifies pre-layout verification but over-constrains the circuits and reduces their potential benefits.

A final method, which we endorse, is to use path based relative timing (RT) constraints [7]. This method constrains the overall delay of two paths from a common point of divergence (POD) to a common point of convergence (POC) to have a specified order of arrival. Unlike the other methods, necessary timing assumptions are made explicit when using relative timing. Such a method is simple for engineers to visualize and understand, and RT is commonly used as part of the performance and timing validation CAD tool flow. Thus an RT-based method allows the designer and CAD tools to specify and reason about timing constraints. Changes (ECO's) are simple to apply if relative path slack information is available. Verification may not need to be repeated, and only paths that pass through the changed components need to have their timing re-evaluated. The rest of this work assumes that circuit and protocol timing is represented as POD to POC relative timing constraints that can be expressed as: $\text{pod} \mapsto \text{poc}_0 \prec \text{poc}_1$ where $\text{poc}_0 \prec \text{poc}_1$ means that poc_0 occurs before poc_1.

Normally a designer will start with a formal specification of a sequential protocol as a petri-net [8] or process language such as CSP [9], [10] or CCS (Calculus for Communicating Systems) [11]. This specification will be synthesized into a circuit realization. Unfortunately, unbounded delay verification of the circuit against the specification nearly always fails. This is because technology mapping into library gates

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

introduced hazards, inverters introduce skew and hazards between a signal and its complement, or the synthesis engine made timing assumptions to optimize the design. For such designs, a set of relative timing constraints can be applied that result in conformance between the implementation and specification.

Timing constraint generation is currently performed by hand with the aid of a verification engine. This hand generation is very time consuming and requires excellent knowledge of the protocol, circuit, and relative timing constraint specifications. Some protocols can take up to four or five hours to create a sufficient set of RT constraints – even for an expert verification engineer. Thus some automated algorithm to generate the relative timing constraints is imperative. One such algorithm is presented here.

The first algorithm for automatic generation of RT constraints was proposed by Kim *et. al.* [12]. This algorithm created state sets where failure transitions are enabled to fire. Transitions which exit the state set are required to fire before the transition that produced the error, thus avoiding the timing violation. The Kim algorithm can produce efficient POC constraint sets. However, this algorithm has a few weaknesses. First, the transition sets are generated from local regions of the complete flat graph of the system, which has lost any hierarchical and modular information. The algorithm reported here is based on the hierarchical behavior of local processes and signal sets that directly affect the failure. Second, path based constraints are not generated since the point of divergence is unknown. Without path-based POD/POC constraints, the pre- and post-layout timing cannot be validated through industry standard CAD [13]. This relegates the primary use of that tool to verifying that it may be possible to apply timing to correctly design the circuit. Finally, this algorithm does not support multi-cycle constraints or other more complicated dependencies between signal sets, which is supported through the unrolling and backtracking used here.

The paper is organized as follows. Section II introduces the verification engine, defines computation interference and the basic notions used throughout the paper. Section III describes the formal algorithm of both finding relative orderings and POD backtracking. Section IV gives a simple example to show how the algorithm works. Section V makes a comparison between ARTIST generated constraints and hand generated constraints in terms of efficiency and quality.

II. FV and Computation Interference

A. Formal Verification Engine

This work is applied to a semi-modular[1] [14], [15] verification engine that directly supports several variants of relative timing, including the preferred path-based constraints. The engine used in this work is an explicit state verification engine using a labeled transition system [16].

[1]Such systems do not allow enabled output transitions to be disabled as this could create a glitch in the circuit.

A binary relation $\mathscr{LC} \subseteq \mathscr{P} \times \mathscr{P}$ over agents is a **logic conformation** between implementation I and specification S if $(I,S) \in \mathscr{LC}$ then $\forall\, \alpha \in Act$ and $\forall\, \beta \in \overline{\mathscr{A}} \cup \{\tau\}$ (outputs and τ) and $\forall\, \gamma \in \mathscr{A}$ (inputs)

(i) Whenever $S \xrightarrow{\alpha} S'$ then
$\exists\ I'$ such that $I \stackrel{\widehat{\alpha}}{\Rightarrow} I'$ and $(I',S') \in \mathscr{LC}$

(ii) Whenever $I \xrightarrow{\beta} I'$ then
$\exists\ S'$ such that $S \stackrel{\widehat{\beta}}{\Rightarrow} S'$ and $(I',S') \in \mathscr{LC}$

(iii) Whenever $I \xrightarrow{\gamma} I'$ and $S \stackrel{\gamma}{\Rightarrow}$ then
$\exists\ S'$ such that $S \stackrel{\gamma}{\Rightarrow} S'$ and $(I',S') \in \mathscr{LC}$

Fig. 1. Bisimilar Logic Conformance Relationship

Definition 1 *A labeled transition system,* $(S, T, \{\xrightarrow{t} : t \in T\})$*, consists of a set S of states, a set T of transition labels and a transition relation* $\xrightarrow{t} \subseteq S \times S$ *for each* $t \in T$.

Definition 2 *The labels (or actions) in labeled transition systems are defined as follows,*

- *Input action set* **names** $a \in \mathscr{A}$ *(the set of names* $\mathscr{A}$ *are inputs* $\mathscr{I}$ *)*
- *Output action set* **conames** $\overline{a} \in \overline{\mathscr{A}}$ *(the set of conames* $\overline{\mathscr{A}}$ *are outputs* $\mathscr{O}$*). By convention,* $\overline{\overline{a}} = a$.
- *The set of actions or* **labels** $\mathscr{L} = \mathscr{A} \cup \overline{\mathscr{A}}$
- *The invisible internal action* τ *(tau).* $\tau \notin \mathscr{L}$
- *The actions of a system are:* $Act = \mathscr{L} \cup \tau$
- *The* **sort***(P) of an agent P is its complete set of observable input and output actions.*

The verification engine takes an implementation I, optionally a specification S, and a set of RT constraints C which is initially empty. The specification and implementation are modeled using the process language CCS [11]. The implementation is composed of multiple agents (agents can be logic gates or minimized specifications of protocols) using the parallel composition operator, e.g. $I = (P_1 \mid P_2 \mid \ldots \mid P_n)$.

Hierarchical verification can be performed without a specification. Some designs implement *timed protocols*. For example, asynchronous burst-mode implementations are all timed protocols since they require that the circuit stabilizes before new inputs can be accepted [17], [18]. One can verify the timing constraints of the protocol by composing the minimized specifications in parallel. If one protocol module is not in an accepting state when an input is driven by an associated protocol, computation interference occurs. Thus at the protocol level, relative timing constraints may be required for proper implementation without the need for a system level specification S.

However, the more common verification task to perform is the conformance of an implementation I to its specification S. The verification used here employs bisimulation semantics [19], [20], [21] and is applied to the conformance relation shown in Fig. 1 [16]. Conformance verification is employed between I and S using the set of RT constraints C.

Verification is performed to prove behavioral and timing correctness. If a failure occurs, a relative timing constraint is generated (manually by the designer or automatically with this algorithm) and added to the RT constraint set C.

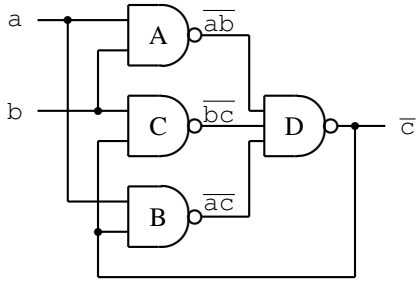

Fig. 2. C-Element implemented with NAND gates

This procedure is iterated until the verification succeeds or the circuit is proven to not conform to the specification. For example, the specification for the C-element circuit of Fig. 2 is: `CE = a.b.`$\overline{\text{c}}$`.CE + b.a.`$\overline{\text{c}}$`.CE`. Starting with an empty RT constraint set C, the circuit fails verification. An verification engineer iterates through verification runs adding the four constraints c+ $\mapsto$ bc- $\prec$ a-, c+ $\mapsto$ bc- $\prec$ b-, c+ $\mapsto$ ac- $\prec$ a-, and c+ $\mapsto$ ac- $\prec$ b-. At this point the verification succeeds and the circuit conforms to the specification.

B. Computation Interference

Computation interference applies between processes in a parallel composition. Each process specifies its local behavior. The definition of a semi-modular [14] 2-input NAND gate used in the C-element of Fig. 2 is shown in Fig. 3.

Computation interference is implemented in the verification engine as an extension to the parallel composition operator of CCS. CCS normally operates using *handshake communication* where labels and colabels from two agents composed in parallel interact to make autonomous communication event τ. Our extensions permit only a single process P_i to drive any output $\alpha \in \overline{\mathscr{A}}$ when agents are composed in parallel, and require all inputs $\beta \in \mathscr{A}$ to concurrently evolve with the output $\overline{\alpha} = \beta \wedge (P_0 \mid P_1 \mid \ldots \mid P_n) \xrightarrow{\tau} (P_0' \mid P_1' \mid \ldots \mid P_n')$ where (i) $P_i \xrightarrow{\alpha} P_i'$ (ii) $\forall P_j \;.\; \beta \in \text{sort}(P_j)$, P_j must be in a state where $P_j \xrightarrow{\beta}$ and $P_j \xrightarrow{\beta} P_j'$ (iii) for all other processes $P_k = P_k'$ [16]. This formalism models hardware implementations, where a single gate will drive an output and that signal will be concurrently observed (assuming zero wire delay[2]) by all gates the output drives. Computation interference is now defined as follows.

Definition 3 dynamic(P_i) *is the set of actions that can occur at agent P_i in the current state for the parallel composition of agents $(P_0 \mid P_1 \mid \ldots \mid P_n)$, such that $\alpha \in$* **dynamic**(P_i) *when*

- $P_i \xrightarrow{\alpha} \wedge\, \alpha \in \overline{\mathscr{A}}$
- $P_j \xrightarrow{\alpha} \wedge\, \alpha \in \overline{\mathscr{A}} \wedge \overline{\alpha} \in$ **sort**(P_i)

Thus dynamic defines the set of actions or signals that interact with process P_i and its composed agents in any given state. The possible behaviors will be equivalent to the corresponding state in the specification of the process (such as the NAND gate of Fig. 3) where some inputs may be removed because the environment will not provide the inputs, and some inputs will be added. The extra inputs that are not specified by the behavior of the process result in computation interference.

[2]Arbitrary wire delay is implemented by including wire fork processes when multiple outputs are driven by a single gate.

```
agent NAND001 = a.NANDa01 + b.NAND0b1;
agent NANDa01 = a.NAND001 + b.NANDab1;
agent NAND0b1 = a.NANDab1 + b.NAND001;
agent NANDab1 =                      'y.NANDab0;
agent NANDab0 = a.NAND0b0 + b.NANDa00;
agent NAND0b0 =             b.NAND000 + 'y.NAND0b1;
agent NANDa00 = a.NAND000             + 'y.NANDa01;
agent NAND000 = a.NANDa00 + b.NAND0b0 + 'y.NAND001;
```

Fig. 3. Semi-modular CCS specification of a 2-input NAND gate with inputs a, b and output $\overline{\text{y}}$ (written 'y in ASCII).

Definition 4 Computation Interference *occurs on signal α in a parallel composition of agents $(P_0 \mid P_1 \mid \ldots \mid P_n)$ when $\alpha \in$* **dynamic**$(P_i) \wedge P_i \xrightarrow{\alpha}$.

Computation interference occurs between two processes composed in parallel when an output signal is enabled to be driven by one process, and the corresponding input action can not be accepted in the current state of a receiving process. This error corresponds to an input "stalling" the transmission of a signal. For example, if we have a system that uses a 2-input NAND gate and the gate is in state `NANDab1` where both inputs a b and the output $\overline{\text{y}}$ are high, the only acceptable action by this process is a transition on the output $\overline{\text{y}}$. If any input were accepted, it would disable the firing of the output, thus violating the semi-modular constraint. Therefore inputs are not allowed in this state, and if any input occurs, a computation interference error results.

There are two main sources of computation interference when applying relative timing verification to the circuit realization of an asynchronous handshake protocol.

1) An input transition is trying to disable an output transition.
2) A short circuit would occur in a dynamic gate by turning on both pull-up and pull-down networks at the same time.

III. Automating Computation Interference Verification

The flow through ARTIST is similar to the manual flow. ARTIST is implemented as an embedded function call into the RT formal verification engine. It takes an error trace or action sequence and feeds the generated constraints back to the verification engine iteratively until circuit implementation conforms to the specification or it is proven that circuit implementation will never conform to the specification due to a defective implementation or incompatible timing constraints. Instead of analyzing the state graph of the circuit [12], ARTIST focuses on the specific process of an agent at the error position. It greatly reduces the complexity because unrelated concurrent signals are ignored.

Require: Trace Status Table
Ensure: Return a set of RT Constraints
  Find $\alpha_{ci}, \alpha_{en}, P_i$ and P_i^{-1}
  Find **dynamic**(P_i) and **dynamic**(P_i^{-1})
  for all $\alpha \in$ **dynamic**$(P_i^{-1}) \wedge \alpha \neq \alpha_{en}$ **do**
    Find POD signal β of α and α_{en} ;
    Add $\beta \mapsto \alpha \prec \alpha_{en}$ into RT ;
  end for
  for all $\alpha \in$ **dynamic**$(P_i) \wedge \alpha \neq \alpha_{ci}$ **do**
    Find POD signal β of α and α_{ci} ;
    Add $\beta \mapsto \alpha \prec \alpha_{ci}$ into RT ;
  end for
  return RT;

Fig. 4. Top level algorithm for computation interference

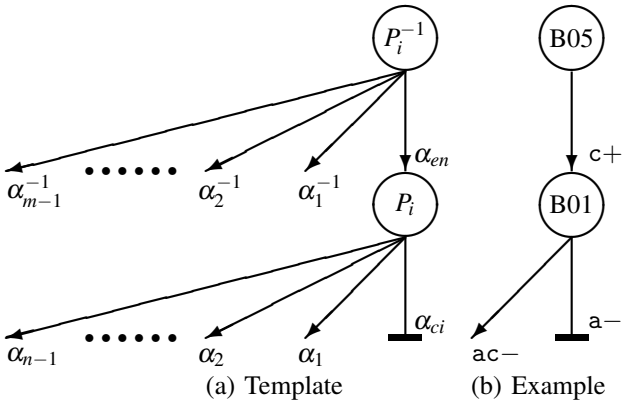

Fig. 5. State Graph

The primary task of the algorithm is to select the correct signals and ordering constraints as the POC, and then backtrack to find their common causal point which is the POD. Fig. 4 shows the top level algorithm for automatic generation of RT constraints. First the information at the error point is obtained based on the trace status tableau such as $\alpha_{ci}, \alpha_{en}, P_i$ and P_i^{-1} and their corresponding **dynamic** sets. Then relative orderings are constructed by combining all the elements in **dynamic**(P_i^{-1}) and α_{en} at higher level and all the elements in **dynamic**(P_i) and α_{ci} at lower level into pairs. The point of divergence is found by backtracking the causalities of each pair.

A. Relative Ordering

Forcing relative signal sequencing at a component or process is achieved by delaying the occurrence of a signal. Constrained signal sequences will prevent a system from entering error states. This can be enforced locally in a design where computation interference occurs.

All the possible signal sequences are provided by the verification engine due to its unbounded device and wire delay model used for verification. The processes or components that are composed to form the implementation update their semi-modular states incrementally based on the signal execution trace from the verification engine. This allows us to create a tableau and template graph. The set of enabled transitions and current process states allow us to generate the template graph of Fig. 5(a) that shows possible transitions of a process where computation interference occurs.

- α_{ci} is the computation interference signal defined in Def. 4.
- the horizontal bar directed from event α_{ci} indicates the failure transition.
- P_i is the state where computation interference occurs.
- $P_i^{-1} \xrightarrow{\alpha_{en}} P_i$ where α_{en} is the transition moves process from P_i^{-1} to P_i.
- **dynamic**$(P_i) = \cup_{i=1\ldots n} \alpha_{n-1} \cup \alpha_{ci}$.
- **dynamic**$(P_i^{-1}) = \cup_{i=1\ldots m} \alpha_{m-1} \cup \alpha_{en}$.

Due to the unbounded delay used in verification, no one can predict which event occurs before another among multiple concurrent events. Therefore ARTIST returns a set of all combinations of event orderings for each error. Thus any action in **dynamic**(P_i) can be constrained to fire before α_{ci} to avoid computation interference. Likewise, any action in **dynamic**(P_i^{-1}) can fire before α_{en} to avoid computation interference as well because P_i where computation interference occurs becomes unreachable. There may exist more candidate signal sequencing at the higher level beyond P_i^{-1} that could be used to remove computation interference, but this algorithm only use the constraints at the level of P_i and P_i^{-1}. Higher level constraints reduce timing margins and may over-constrain the design which could result in non-conformance to the specification. Note that the constraints returned are mutual exclusive and only one of them is used as the feedback to the verification engine. If a weaker constraint is selected, the cardinality of the final set of RT constraints may be bigger. These constraints also allow choice of the best constraint. The strong and weak aspect of RT constraints is demonstrated in Section V-B.

To find the relative ordering, one must identify and label signals such as α_{ci}, α_{en}, and P_i, P_i^{-1}, **dynamic**(P_i) and **dynamic**(P_i^{-1}) based on their behavior. However, this information cannot be identified solely with an error signal trace passed from the verification engine. Thus a tableau is constructed to include all the necessary information which reflects the changes of each gate's status as the signal trace grows incrementally.

Table I is a status tableau for the C-element circuit of Fig. 2. The signal trace that results in the computation interference error is listed on the bottom row. The signals show the logic level of their transition as either a '+' for a rising transition or '−' for a falling transition. The other rows list a signal and the process that generates that signal. Primary inputs are generated by the spec if provided or are unconstrained. The other signals are process outputs; in this case the outputs of gates A – D in Fig. 2. The full signal set, consisting of primary inputs, primary outputs, and internal signals, is listed in the first column. Subsequent columns are numbered based on the depth of the signal trace. Each of these columns represent all necessary signal status information. This information includes the state of

TABLE I

AN EXAMPLE STATUS TABLE FOR A PARTICULAR ERROR TRACE IN VERIFICATION OF C-ELEMENT. EACH CELL REPRESENTS STATE, #TRANS, *EN* FLAG, *CI* FLAG, RESPECTIVELY.

	0	1	2	3	4	5
a	S00,0,T,F	S01,1,F,F	S02,1,F,F	S02,1,F,F	S00,1,T,F	S01,2,F,F
b	S00,0,T,F	S01,0,T,F	S02,1,F,F	S02,1,F,F	S00,1,T,F	S01,1,T,F
ab	A00,0,F,F	A05,0,F,F	A01,0,T,F	A06,1,F,F	A06,1,F,F	A02,1,T,F
ac	B00,0,F,F	B05,0,F,F	B05,0,F,F	B05,0,F,F	B01,0,T,F	B01,0,T,T
bc	C00,0,F,F	C02,0,F,F	C05,0,F,F	C05,0,F,F	C01,0,T,F	C01,0,T,F
c	D00,0,F,F	D03,0,F,F	D03,0,F,F	D01,0,T,F	D12,1,F,F	D12,1,F,F
T	init	a+	b+	ab-	c+	a-

TABLE II

UNOPTIMIZED RT CONSTRAINTS AND CORRESPONDING TRACES VERSUS HAND-GENERATED CONSTRAINTS FOR C-ELEMENT

ARTIST Generated			Hand Generated	
No.	Trace	RT Constraint	No.	RT Constraint
A1	a b ab c a	c+ $\mapsto$ ac- $\prec$ a-	H1	c+ $\mapsto$ ac- $\prec$ a-
A2	a b ab c b	c+ $\mapsto$ bc- $\prec$ b-	H2	c+ $\mapsto$ bc- $\prec$ b-
A3	a b ab c ac a ac ab c	c+ $\mapsto$ bc- $\prec$ c-	H3	c+ $\mapsto$ bc- $\prec$ a-
A4	a b ab c ac a ac ab bc	c+ $\mapsto$ bc- $\prec$ ab+	H4	c+ $\mapsto$ ac- $\prec$ b-
A5	a b ab c bc b bc ab c	c+ $\mapsto$ ac- $\prec$ c-		
A6	a b ab c bc b bc ab ac	c+ $\mapsto$ ac- $\prec$ ab+		

the module, the number of transitions this signal has made, whether the signal is enabled and ready to fire (*EN* flag), and whether computation interference occurs on this signal as a result of the trace (*CI* flag). Generation of this tableau requires the trace information from the verification engine as well as the behavior of the individual parallel processes comprising the implementation.

All necessary information for the algorithm can be calculated from the tableau. Computation interference occurs in the module where the *CI* flag becomes asserted. This identifies the process that defines the POC. The signal that results in the violation is α_{ci}. It is normally the last signal transition in the trace. The enabling signal α_{en} is found by observing the causality indicated by the signal enabled flag (c+). P_i and P_i^{-1} are associated with α_{en} (B05 and B01 for P_i^{-1} and P_i respectively in this example). **dynamic**(P_i) and **dynamic**(P_i^{-1}) can be derived by searching enabled inputs and outputs of the agent at P_i and P_i^{-1}.

B. POD Backtracking

The POD/POC pair specifies the paths in a race between two events. Once the POC has been defined, the POD can be identified. The algorithm defines the POD by backtracking the causality of the two events selected in the POC identification. In this case α_{ci} (a−) and α_{en} (c+) are used. The trace status tableau provides an easy way to identify the causal relationship between signal transitions in the trace by observing the *EN* flags of the signals. By default, ARTIST returns the last common causal signal transition as the POD. To facilitate pre- and post-layout timing validation of these constraints, a feature that supports user-specified POD is added.

IV. EXAMPLE

A simple example is used to demonstrate how the algorithm works on a C-Element implemented with three 2-input NAND and one 3-input NAND gates shown in Fig. 2.

A computation interference error trace {a, b, ab, c, a} is returned from verification engine. Note that CCS equates high and low logic levels when their behavior is identical, as is the case with the specification. There is no separate identity given to a signal logic level or high or low going transition. While this reduces the state space and makes this a more generic tool, it also makes it harder to equate to circuit state. Thus logic levels are added to the trace in the bottommost row in Table I. The verification engine and the rest of the tableau do not model logic levels. Instead, they track the number of transitions each signal has made. Given the initial signal state, the actual logic level can be easily calculated.

In this trace the final signal a− results in a computation interference error. This can be determined from state {B01, 0, T, T} for signal ac in the rightmost column of Table I. Following are the algorithmic parameters derived from this trace:

- POC: the gate with computation interference: ac
- α_{ci} = a− and α_{en} = c+
- P_i^{-1} = B05 (NANDa01 in Fig. 3)
- P_i = B01 (NANDab1 in Fig. 3)
- **dynamic**(P_i^{-1}) = {c+}
- **dynamic**(P_i) = {ac−, a−}

Fig. 5(b) shows the state graph of this trace example. The only relative ordering to avoid computation interference in this case is ac− $\prec$ a−.

Then backtracking is employed on the above two transitions to find their POD by identifying causalities in the table, which occur based on the signals that assert the *EN* flag.

- b+ $\Rightarrow$ ab− $\Rightarrow$ c+ $\Rightarrow$ ac−
- b+ $\Rightarrow$ ab− $\Rightarrow$ c+ $\Rightarrow$ a−

The full causal paths of the relative ordering signals are listed above. The last common causal signal c+ is the POD. The full causal path list is included in order to better support timing driven synthesis, place and route, and validation algorithms in tools such as Design Compiler and PrimeTime. These algorithms work on DAGs and use a 'clock' pin for timing references. For asynchronous circuits, one of the handshake signals (such as *req*) will typically be defined as the 'clock'. If the user defines one of the primary inputs to the module as the 'clock' pin, the algorithm will continue backtracking to that pin rather than choose the nearest point of divergence.

Iterating the above procedures creates the RT constraint set, which is shown along with the corresponding traces in Table II.

V. COMPARISON

The RT constraints for this example generated by hand and by ARTIST are compared in terms of efficiency and quality. The hand-generated RT constraints that are based on designer's intuition are shown in Table II in the rightmost column.

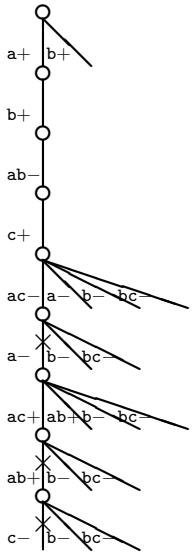

Fig. 6. Partial state graph to represent a particular trace

A. *Efficiency*

ARTIST greatly reduces the design and verification time. A set of over 100 4-phase latch protocols created through concurrency reduction [22] were verified by ARTIST. The result of a few selected protocols from this set is listed in Table III. The program was run on a workstation configured with Intel® Xeon™ 3.20GHz CPU and 2GB memory. The average number of RT constraints for a protocol generated by ARTIST is 10 and the average runtime is 0.15 seconds.

B. *Quality*

The number of RT constraints is a primary quality metric. It largely determines work load of pre- and post-layout timing validation through static timing analysis engines. In Table II the number of hand-generated RT constraints is 2 fewer than that of ARTIST.

The set of RT constraints generated by ARTIST can be optimized into a smaller size by removing redundant constraints. Take a close look at constraints A3 and A4 generated by ARTIST in Table II. Note that constraint A4 covers constraint A3. In Fig. 6, a partial state graph is drawn to represent the special trace path of constraints A3 and A4. For simplicity, all other states are omitted. Constraint A3 truncates the subgraph from c− that is indicated by the bottom $\times$. Constraint A4 makes subgraph from ab+ unreachable, which means that constraint A3 will never be applied for state reduction. Likewise, constraint A5 can be removed as well. Thus a total of 4 RT constraints from ARTIST are sufficient make the implementation conform to the specification - which has the same set size as hand-generated constraints.

This redundancy occurred due to the iteration order of the computation interference errors. The traces for RT constraints A3 and A4 are the same except for α_{ci}. At that time, both α_{ci} signals are enabled and can fire at either order. The verification engine arbitrarily chooses one of them. This nondeterminism motivates a future work on an optimization algorithm to minimize the set of RT constraints. One can also add a simple algorithm to test if any constraint is redundant. After the implementation conforms to the specification, each constraint can be removed and the model checking is performed again.

The final set of RT constraints is $\{$A1, A2, A4, A6$\}$. Now that we have the same number of RT constraints for both ARTIST generated and hand generated, the working load on timing validation is the same. A1 and A2 are the same as H1 and H2. Fig. 6 allows one to compare constraints A4 and H3. Constraint H3 c+ $\mapsto$ bc- $\prec$ a- cuts the graph at a−, two transitions after the c+ transition. However, A4 doesn't cut the graph until transition ac+. Thus the hand generated constraint has less slack and this is stronger than the automatically generated constraint from ARTIST. This shows that there can be different sets of RT constraints solutions, either stronger or weaker. Stronger constraints may result in compact set of RT constraints but reduces slack and may over-constrain the design and induce failure; weaker constraints guarantees the correctness of the design under RT constraints but increases the burden of pre- and post-layout timing validation. Further investigation is required to determine if there is a significant run-time difference for the timing driven synthesis, place and route, and validation tools for differing RT constraint set strengths.

VI. Conclusion

Formal verification is the core of template based asynchronous design methodology. Relative timing is a method of integrating the verification of timing constraints on an untimed protocol or circuit implementation and converting them into a format that can be used with traditional CAD tools [13]. The manual generation of RT constraints is time consuming and error-prone which reduces the wide adoption of asynchronous design.

This paper presents an algorithm for the automatic generation of RT constraints. This algorithm supports RT constraints expressed as a point of divergence to point of convergence pair. This representation explicitly represents the causal paths in a race between two events. The algorithm resolves all computation interference errors, which result when a disabled input can fire. Over 100 asynchronous controller have been verified by automatically generating sets of RT constraints that guarantee the circuits function correctly in both the behavior and timing domains.

This algorithm also supports the feature of allowing for user-specified input signal for mapping 'clock' pin in pre- and post-layout timing validation. A comparison has been made between ARTIST generated and hand generated constraints in terms of efficiency and quality. It is obvious that one push of the button of ARTIST is much more efficient than hand generation which requires the verification engineer to have good knowledge on the specific circuit implementation. In the result table, the average runtime for a single RT constraint generated by ARTIST is 0.015 second which will take hours or even days by hand. Currently ARTIST may

create more RT constraint than the hand generated sets, but some of them can be merged based on redundancy and the strength of the constraints.

VII. Acknowledgments

This work was supported by grant 1424.001 from Semiconductor Research Corporation.

References

[1] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design – A Systems Perspective*. Kluwer Academic Publishers, 2001.

[2] K. S. Stevens, "Energy and performance models for clocked and asynchronous communication," in *International Symposium on Asynchronous Circuits and Systems*. IEEE, May 2003, pp. 56–66.

[3] C. J. Myers and T. H.-Y. Meng, "Synthesis of timed asynchronous circuits," in *Proceedings of the International Conference on Computer Design (ICCD)*, Oct. 1995, pp. 279–282.

[4] C. J. Myers, "Computer-aided synthesis and verification of gate-level timed circuits," Ph.D. dissertation, Dept. of Elec. Eng., Stanford University, Oct. 1995.

[5] R. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers*, vol. C-33, no. 2, pp. 160–177, Feb. 1984.

[6] I. E. Sutherland and J. K. Lexau, "Designing fast asynchronous circuits," in *7th International Symposium on Asynchronous Circuits and Systems*, Mar. 2001, pp. 184–193.

[7] 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.

[8] J. Peterson, *Petri Net Theory and Modeling of Systems*. Prentice Hall, 1981.

[9] C. A. R. Hoare, *Communicating Sequential Processes*. London: Prentice Hall International, 1985.

[10] ——, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, August 1978.

[11] R. Milner, *Communication and Concurrency*, ser. Computer Science. London: Prentice Hall International, 1989.

[12] H. Kim, P. A. Beerel, and K. S. Stevens, "Relative timing based verification of timed circuits and systems," in *8th International Symposium on Asynchronous Circuits and Systems*, Apr. 2002, pp. 115–126.

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

[14] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," in *Proceedings of an International Symposium on the Theory of Switching*. Harvard University Press, Apr. 1959, pp. 204–243.

[15] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli, "A unified signal transition graph model for asynchronous control circuit synthesis," in *International Conference on Computer-Aided Design (ICCAD)*. IEEE Computer Society Press, Nov 1992, pp. 104–111.

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

[17] K. Y. Yun and D. L. Dill, "Automatic Synthesis of Extended Burst-Mode Circuits: Part I (Specification and Hazard-Free Implementation)," *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 2, pp. 101–117, Feb. 1999.

[18] S. M. Nowick, "Automatic synthesis of burst-mode asynchronous controllers," Ph.D. dissertation, Stanford University, Department of Computer Science, 1993.

[19] R. Paige and R. Tarjan, "Three partition refinement algorithms," *SIAM Journal of Computation*, vol. 16, no. 6, pp. 973–989, 1987.

[20] J.-C. Fernandez, "An implementation of an efficient algorithm for bisimulation equivalence," *Science of Computer Programming*, vol. 13, pp. 219 – 236, 1990.

[21] J.-C. Fernandez and L. Mounier, ""On the Fly" Verification of Behavioral Equivalences and Preorders," in *Proceedings of CAV'91*, ser. LNCS, K. G. Larsen and A. Skou, Eds., no. 575, 1991, pp. 181–191.

[22] G. Birtwistle and K. S. Stevens, "The family of 4-phase latch protocols," in *14th International Symposium on Asynchronous Circuits and Systems*. IEEE, April 2008, pp. 71–82.

TABLE III

Verification result for protocol, number of RT constraints generated (unoptimized) from ARTIST, number of states of specification, and for implementation.

No.	Name	#Const-raints	ARTIST Runtime	FVEngine Runtime	#Spec States	#Impl States
1	L2112_R2222	9	0.128	0.850	19	113
2	L3223_R0020	2	0.061	0.527	21	104
3	L2112_R2022	7	0.071	56.658	21	395
4	L3223_R2044	16	0.221	1.644	13	95
5	L2222_R2242	26	0.438	1.492	15	124
6	L1111_R0044	12	0.165	1.699	21	335
7	L2222_R0020	10	0.116	0.959	23	145
8	L2112_R2264	2	0.007	0.063	13	26
9	L2002_R2262	3	0.037	0.187	17	49
10	L1001_R2262	13	0.177	4.422	19	114
11	L3333_R0042	16	0.185	1.393	15	136
12	L3333_R0020	24	0.344	2.526	19	177
13	L3333_R0000	29	0.609	4.816	21	326
14	L3223_R2042	15	0.243	1.042	15	143
15	L3223_R2022	9	0.124	0.506	17	106
16	L3223_R0042	16	0.197	2.325	17	210
17	L3223_R0022	14	0.188	1.424	19	150
18	L3223_R0000	12	0.201	3.475	23	275
19	L3113_R2242	4	0.034	0.199	15	52
20	L3113_R2222	4	0.046	0.233	17	70
21	L3113_R2042	8	0.107	0.723	9	158
22	L3113_R0040	6	0.096	4.079	21	261
23	L3113_R0022	6	0.119	2.979	11	318
24	L3003_R2042	19	0.314	13.952	19	390
25	L3003_R0022	15	0.268	17.619	23	352
26	L2222_R2022	10	0.137	1.136	19	106
27	L2222_R0040	9	0.106	0.633	21	131
28	L2222_R0022	6	0.050	0.319	21	80
29	L2112_R2042	12	0.209	1.822	19	344
30	L2112_R0042	22	0.349	15.833	21	1251
31	L2112_R0020	21	0.227	18.869	25	426
32	L2002_R2022	12	0.158	3.119	23	351
33	L1111_R2042	9	0.131	2.993	21	280
34	L1111_R0022	4	0.060	0.583	25	136
35	L1001_R2042	4	0.048	0.452	23	291
36	L3333_R0044	2	0.015	0.138	13	52
37	L3113_R2044	3	0.028	0.289	15	68
38	L3113_R0044	1	0.010	0.112	17	65
39	L2002_R2222	4	0.070	0.464	21	85
40	L2222_R2222	5	0.032	0.223	17	106
41	L3113_R2022	10	0.126	1.667	19	220
42	L3113_R0042	7	0.100	4.465	19	272
43	L0000_R2242	25	0.931	44.525	23	1152
44	L0000_R2244	6	0.088	0.454	21	125
45	L0000_R2262	12	0.197	2.359	21	340
46	L0000_R4044	4	0.049	7.069	21	515
47	L0000_R4264	18	0.226	1.061	17	173
48	L1001_R2242	6	0.090	0.851	21	203
49	L1001_R2244	12	0.210	1.363	19	200
50	L1001_R4264	7	0.043	0.378	15	127
51	L1111_R2044	11	0.090	0.773	19	130
52	L1111_R2222	7	0.111	0.644	21	135
53	L1111_R2242	4	0.042	0.341	19	91
54	L1111_R2262	5	0.048	0.427	17	79
55	L1111_R2264	4	0.057	0.289	15	56
56	L2002_R2244	4	0.036	0.171	9	49
57	L2002_R2264	4	0.038	0.168	15	45
58	L2002_R4244	4	0.042	0.171	15	50
59	L2112_R2244	4	0.034	0.173	15	52
60	L2112_R2262	16	0.216	2.301	15	137
61	L2222_R0044	6	0.085	0.501	17	169
62	L2222_R2262	12	0.109	0.751	13	90
	Average	10	0.150	3.930	18	207