Static Timing Analysis Algorithms for Sequential Cyclic Circuits

Diploma Thesis

By

XIROMERITIS NIKOLAOS

Department of Electrical and Computer Engineering
UNIVERSITY OF THESALLY

A dissertation submitted to the University of Thessaly in accordance with the requirements of the DIPLOMA degree in the department of Electrical and Computer Engineering.

Supervisors
Christos Sotiriou, Associate Professor
Nestor Eumorfopoulos, Assistant Professor

FEBRUARY 2019

Αλγόριθμοι Στατικής Χρονικής Ανάλυσης για Ακολουθιακά Κυκλώματα με Κύκλους

Διπλωματική Εργασία

Ξηρομερίτης Νικόλαος

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας

Επιβλέποντες
Χρήστος Σωτηρίου, Αναπληρωτής Καθηγητής
Νέστωρ Ευμορφόπουλος, Επίκουρος Καθηγητής

Φεβρουάριος 2019

Περίληψη

Στην πτυχιακή αυτή εργασία παρουσιάζεται μια μεθοδολογία Στατικής Χρονικής Ανάλυσης για ασύγχρονα κυκλώματα ελέγχου, ή γενικότερα, κυκλώματα με κύκλους συνδιαστικής λογικής. Η μεθοδολογία αυτή υλοποιήθηκε και ενσωματώθηκε στο εργαλείο **ASP**, ένα εργαλείο φυσικής σχεδίασης, και είναι βασισμένη στις θεμελιώδεις αρχές της Στατικής Χρονικής Ανάλυσης ακολουθιακών κυκλωμάτων. Λαμβάνοντας ως είσοδο την κυκλωματική περιγραφή (**netlist**) και το μοντέλο συμπεριφοράς (**event model**) του ασύγχρονου συστήματος, το εργαλείο μπορεί να υπολογίσει την ελάχιστη επιτρεπτή περίοδο που το χαρακτηρίζει. Πιο συγκεκριμένα, η τιμή της περιόδου υπολογίζεται από τους κύκλους ενός Χρονικού Γράφου Συμβάντων, που περιγράφει τις χρονικές καθυστερήσεις των σημάτων του ασύγχρονου κυκλώματος, και δημιουργείται μέσω του μοντέλου συμπεριφοράς. Οι χρονικές καθυστερήσεις των σημάτων εξάγονται από τα μονοπάτια του κυκλώματος (**netlist**), δεδομένου ότι οι χρόνοι μεταβάσεων των σημάτων (**transition times**) έχουν συγκλίνει στις τιμές ισορροπίας τους λόγω των κύκλων. Αξίζει να αναφερθεί ότι, η προτεινόμενη μεθοδολογία, προκειμένου να υπολογίσει τις καθυστερήσεις των κυκλωματικών μονοπατιών, αντλεί πληροφορίες για την τεχνολογία και τον χρονισμό των στοιχείων, που απαρτίζουν το κύκλωμα, από βιομηχανικές βιβλιοθήκες (αρχεία **LEF** και **LIB**) . Στο τελευταίο κεφάλαιο της πτυχιακής, παρουσιάζονται τα πειραματικά αποτελέσματα της στατικής χρονικής ανάλυσης μέσω της μεθοδολογίας που υλοποιήθηκε, από 23 ασύγχρονα κυκλώματα ελέγχου. Επιπλέον, παρουσιάζοντας τα αντίστοιχα αποτελέσματα μίας βιομηχανικής μηχανής Στατικής Χρονικής Ανάλυσης, δίνεται έμφαση στο γεγονός ότι οι βιομηχανικές μηχανές βγάζουν λανθασμένα αποτελέσματα για ασύγχρονα κυκλώματα λόγω της παρουσίας κυκλικών μονοπατιών.

Abstract

This work presents an Asynchronous Static Timing Analysis (ASTA) methodology for Asynchronous Control Circuits which has been implemented and integrated on an Electronic Design Automation (EDA) tool called ASP. Using the principles of Graph-Based Static Timing Analysis (STA), the developed ASTA engine, taking as inputs the netlist and the event model that describe the connectivity and behaviour of the controller, is able to compute the circuit's minimum period. This is done through a polynomial complexity algorithm that locates the critical cycles (*i.e.* the cycles that define the system's periodicity) of a constructed Event Timing Graph (ETG) based on the event model. However, to calculate the period, the netlist equilibrium slews (or signal transition times) must be computed in order to extract and annotate the path delays onto the ETG. It is worth mentioning that, the proposed ASTA flow uses the technology and cell timing information from industrial file formats such as LEF and LIB. Eventually, we present the experimental results, derived from 23 asynchronous controller benchmarks, and emphasize on the inability of classic STA industrial tools to handle Asynchronous Circuits properly. More specifically, classic STA engines are based on algorithms that are only applicable on Directed Acyclic Graphs (*i.e.* acyclic circuits). Therefore, when given an asynchronous (and thus cyclic) design they perform cycle cutting which leads to the loss of the cut arcs' timing information.

Dedication and acknowledgements

Firstly, I would like to thank Christos Sotiriou for his continuous support and guidance in the completion of this thesis. I would also like to thank from the bottom of my heart my family and friends as they have been the few people who, in difficult times, always stood by my side and in moments of happiness and joy were always there to make them even more joyful.

TABLE OF CONTENTS

LIST OF TABLES

TABLE **Page**

List of Figures

List of Algorithms

CHAPTER 1

INTRODUCTION

With Moore's law coming to an end the need for alternative circuit design materials and/or techniques increases. Asynchronous systems are a candidate and might come under the spotlight in the next few years, as they tend to be faster than sequential systems, due to the lack of clock signals that define the worst-case delay upper bound of the circuit paths. Additionally, such systems are characterised by significantly lower power consumption, compared to sequential systems, because no transistor ever transitions unless it is performing useful computation. Although, due to the complexity of asynchronous systems and the adaptation of the industry in sequential circuits, there are not many people trained in this style of design. Another core issue that is preventing the application of asynchronous techniques in state of the art designs is the lack of Electronic Design Automation (EDA) tools.

The most important EDA tools up to date are Static Timing Analysis (STA) engines while most steps in modern EDA flows are timing-driven or timing-aware. STA engines are responsible for the timing verification of circuits as they can locate possible timing violations (*i.e.* scenarios where signals propagate too slowly or too fast) and provide the designer with timing information that can be used to further improve the circuit's performance or as metrics to other flow-steps of EDA tools, such as timing-driven detailed placement. It is worth mentioning that, the algorithms that compose a STA engine are applicable only on Directed Acyclic Graphs (DAGs), which are used to describe sequential paths. Therefore, such engines are not compatible with asynchronous designs due to the presence of combinational cycles. In more detail, modern STA engines, in an attempt to perform the timing analysis of an asynchronous circuit, will cut the graph arcs that form cycles and thus the timing information of these arcs will be lost. Based on the above, we can conclude that an Asynchronous Static Timing Analysis (ASTA) engine requires a new flow and new algorithms.

The presence of cycles in asynchronous circuits is not the only complexity factor of an ASTA engine. The behavioral model of asynchronous systems is depicted by concurrent event models, such as Free-Choice Petri Nets and Signal Transition Graphs (STGs), that view the circuit signal transitions as events. ASTA heavily relies on such models, because the period of asynchronous designs is extracted from the cycles formed by the events (known as critical cycles). Note that, these cycles are not related to the previously mentioned cycles of the circuit. A further challenge for ASTA is the complexity involved in correlating an asynchronous gate-level netlist to its original event model.

In this thesis we present a general GBA flow for performing STA on asynchronous control circuits. Our flow, inheriting the principles of classic STA, is based on industry-standard formats (*i.e.* `.lib` file formats and Verilog netlists) supporting both worst-case and best-case analysis, and can be distinguished into the following steps:

- Cyclic equilibrium slew (or transition times) computation.
- Event Timing Graph (ETG) construction.
- ETG transition-to-transition (T2T) delay derivation in netlist.
- ETG period and Critical Cycle(s) computation.

In more detail, having the netlist and the event model of an asynchronous controller, we are able to generate a delay annotated STG, also called Event Timing Graph (ETG). The arc weights of this graph correspond to gate-level path delays extracted from the netlist, based on cyclic equilibrium transition times (*i.e.* transition times that need some time to converge to their final value due to circuit combinational loops). After the ETG construction and delay annotation, the controller's period can be computed by Burns' Polynomial Primal Dual Method Algorithm [1]. Note that, in this work we also present multiple extensions to Burns' algorithm, such as floating point precision support, critical cycle(s) extraction and reporting, and event model missing marking insertion. It is worth mentioning that, the ETG construction allows us perform ASTA on various event model types, such as Free-Choice Petri Nets, that were not originally supported by Burns' Period Computation Algorithm.

CHAPTER 2

STATIC TIMING ANALYSIS BACKGROUND

The timing validation of most digital circuits today is performed with Static Timing Analysis (STA) making it the cornerstone of modern industrial Electronic Design Automation (EDA) flows. In this chapter we present the fundamentals and the terminology of STA that are required for developing a timing engine. STA in general is considered to be one of the most important EDA flow steps, as its results are crucial to the design's correctness, and can also provide the designer with timing information used as metrics to other EDA flow steps.

2.1 Static Timing Analysis Definition and Limitations

Static Timing Analysis (STA) is one of the many techniques available to verify the timing of a digital design. An alternate approach of timing verification is timing simulation, which can also verify the functionality of the design [6]. STA does not depend on the data values being applied at the cell input pins.

However, the core algorithms that compose STA engines are only applicable on Directed Acyclic Graphs (DAGs). Therefore the timing verification of circuits with combinational loops is not possible. In more detail, STA engines, in order to handle cyclic circuits, perform cycle cutting which leads to loss of the cut arc timing information. Additionally, the presence of cycles impacts the signal transition times (or slews), of which the final equilibrium values cannot be computed by a classic STA engine.

2.2 Setup and Hold Times

Given a user-defined clock, one of the goals of performing STA on a digital circuit is to verify that no **setup** or **hold** time violations exist. In more detail, a flip-flop's operation requires the logic

value of the input data pin to be stable for a specific period of time before the capturing clock edge. This interval is the setup time (t_{setup}). Additionally, the logic value of the input data pin must also be stable for a specific period of time after the capturing clock edge. This interval is the hold time t_{hold}. Thus, we perform worst-case (max or late) delay timing analysis to validate setup violations and best-case (min or early) delay timing analysis to validate hold time violations. Figure 2.1 shows the setup and hold time intervals relative to the capturing edge of a D Flip-Flop.

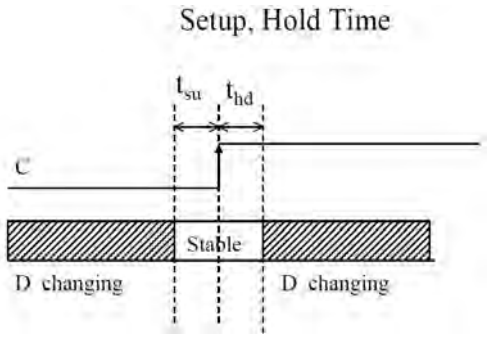


Figure 2.1: Example of a combinational path between two Flip-Flops.

2.3 Timing Arcs and Unateness

Static Timing Analysis works on the concept of **timing paths**. Each timing path starts from a primary input or a sequential component output and ends to a primary output or a sequential component input. These paths consist of **timing arcs**. Given a cell or a net, a timing arc defines the timing relationship between two related pins. Combinational cells such as AND, OR, NOT gates have arcs from each input pin to each output pin. Sequential cells such as Flip-Flops and Latches have arcs from their clock and data pins to the output pin(s). Timing arc startpoints and endpoints are called source pins and sink pins respectively. Figure 2.2 shows the cell and net timing arcs of two connected OR gates.

Timing arcs may be categorised based on their timing sense (or **unateness**) that describes the polarity relationship of the related pins. Timing arcs may be positive unate, negative unate, or non-unate (also known as binate). More specifically:

- **Positive Unate Timing Arc**: An arc is called positive unate, if the source pin rise transition causes a sink pin rise transition (if at all) and vice-versa. Gates such as AND, OR and BUF have positive unate arcs and can be also called positive unate gates. It is worth mentioning that all net arcs are possitive unate arcs.

- **Negative Unate Timing Arc**: An arc is called negative unate, if the source pin rise transition causes a sink pin fall transition (if at all) and vice-versa. Gates such as NAND, NOR and INV have negative unate arcs and can be also called negative unate gates.

- **Non-unate Timing Arc**: An arc is called non-unate, if the source pin rise transition causes a sink pin rise or fall transition (if at all) and vice-versa. Gates like XOR and XNOR have non-unate timing arcs and can be also called non-unate gates.

In practice, most combinational cells are unate (positive or negative), thus their timing arc sense for an output is independent of multiple input state. However various gates, *e.g.* XOR, XNOR, MUX, are binate. Binate gates are state-dependent, as their timing arc behavior, from an input to output, is dependent on the state of other inputs. For example, a XOR gate is positive unate for one input when the other is at Logic 0, but negative unate, when the other input is at Logic 1.

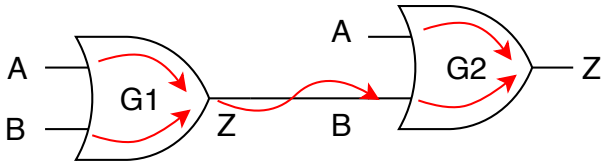


Figure 2.2: Cell and Net Timing Arcs

2.4 Non-Linear Delay Model Look-Up Tables

Static Timing Analysis uses the concept of timing libraries, *e.g.* LIBERTY (`.lib`) format, where combinational and sequential cells are described using Non-Linear Delay Model (NLDM) 2D Look-Up Tables (LUTs). NLDM LUTs describe timing arcs between a gate's input and output pins, where input slew and output capacitance (driving load) values (tr_{in},C_{out}) produce output slew and delay values (tr_{out},d_{out}). More specifically, input slew and output capacitance (or driving load) values are used as LUT indexes. However, in most cases they do not correspond exactly to index values, therefore delay and output slew values are extracted by performing bi-linear interpolation/extrapolation. In NLDM LUTs, binate gates are described using multiple LUTs, one per related input pin. Sequential cell LUTs are similar and also include input to input timing checks.

2.5 Graph-Based and Path-Based Analysis

Two commonly used STA methodologies are **Graph-Based Analysis** (GBA) and **Path-Based Analysis** (PBA). Unless stated, when talking about STA the GBA methodology is assumed by default. In a Graph-Based methodology, while performing a late mode (setup) analysis, *i.e.*

computing the worst case delays and the longest paths, the output slew and delay of a cell is a function of the output capacitance and the **worst** input slew. Accordingly, in a early mode (hold) timing analysis the output delay and slew of a cell are computed assuming the output capacitance and the **best** input slew. For example, in Figure 2.3, assuming a worst case (setup) analysis, the delay of the NAND gate (in blue) will be computed considering the worst case slew, *i.e.* the slew at pin A.

In Path-Based Analysis, we take into consideration the actual slew of the encountered arcs while traversing **any particular timing path**. For instance, given the path from FF1 to FF2 shown in Figure 2.3, the input slews of each gate, that are used to compute the output slews and delays, are: pin A slew for the OR gate; pin B slew for the NAND gate; pin B slew for the XOR gate; pin A slew for the inverted AND gate. Note that in PBA, for a specific path, the best-case and worst-case slews are extracted from the same gate pins. The pros and cons of each STA methodology are:

- Graph Based Analysis:
  - Pros:
    1. Polynomial algorithmic complexity.
  - Cons:
    1. Pessimistic, due to the slews.
    2. Not every path is sensitisable.
- Path Based Analysis:
  - Pros:
    1. Accurate.
    2. No pessimism.
  - Cons:
    1. Exponential complexity, as the number of paths is usually exponential to the number of nodes.

2.6 Timing Propagation

One key feature of STA is delay and transition propagation across multiple input combinational cell timing arcs. The complete set of timing arcs compose the timing graph. Delay propagation depends on the STA mode (max or min), corresponding to maximum or minimum timing arc delay respectively. For transition (or slew) propagation, the worst (max) and best (min) slew timing

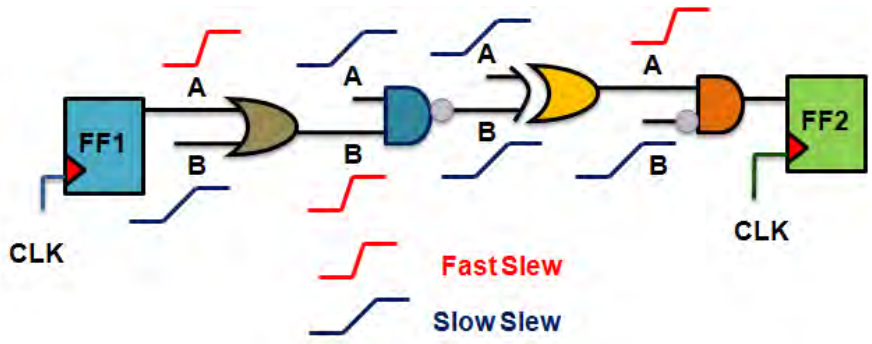


Figure 2.3: Example of a combinational path between two Flip-Flops.

arcs are propagated independently from timing arc delays. Therefore, at each cell output pin, the slew and delay timing arcs are independent.

For a combinational path, starting from its startpoint which can be either a primary input or a flip-flop output pin, we quantify the instant that a signal reaches a pin of a circuit element as the **Arrival Time** (at). Arrival times are computed by adding the delays across the path. For a specific pin, in late mode analysis, the arrival time will be equal to the maximum sum of predecessor arrival time and timing arc delay. Accordingly, in early mode analysis, arrival times are computed by taking the minimum predecessor arrival time and arc delay sum. Given the pins u and v, where u is a predecessor of v, and $d_{u \to v}$ is the timing arc delay, the mathematical definition of arrival times is:

$$at_v^{early} = \begin{cases} 0, & \text{if } v \text{ is a path startpoint} \\ min\{\, at_u^{early} + d_{u \to v}^{early} \,\} \mid \exists\ u \to v \end{cases}$$

$$at_v^{late} = \begin{cases} 0, & \text{if } v \text{ is a path startpoint} \\ max\{\, at_u^{late} + d_{u \to v}^{late} \,\} \mid \exists\ u \to v \end{cases}$$

In the same manner, starting from path endpoints and traversing backwards, we compute the **Required Arrival Times** (rat) for each pin, that quantify the bounds of actual arrival times. More specifically, in late mode analysis, the required arrival time of a pin is equal to the minimum required arrival time of its successor pins subtracted by the arc delay. The application of the min operation is necessary, as in this way we are constraining late mode actual arrival times to the least upper bound (LUB). Early mode required arrival times are computed by taking the maximum successor arrival times subtracted by the timing arc delay. In this way, we compute the greatest lower bound (GLB) for the early mode actual arrival times. Given pins u and v, where u is a predecessor of v, and $d_{u \to v}$ is the timing arc delay, the mathematical definition of required arrival times is:

$$rat_u^{early} = \begin{cases} t_{hold}, & \text{if } v \text{ is a path endpoint} \\ max\{\, rat_v^{early} - d_{u \to v}^{early} \,\} \mid \exists\ u \to v \end{cases}$$

$$rat_u^{late} = \begin{cases} T_{clk} - t_{setup}, & \text{if } v \text{ is a path endpoint} \\ min\{\, rat_v^{late} - d_{u \to v}^{late} \,\} \mid \exists\ u \to v \end{cases}$$

To guarantee proper circuit operation, for every pin v in the timing graph, the following conditions must hold:

$$at_v^{early} \geq rat_v^{early}$$

$$at_v^{late} \leq rat_v^{late}$$

The quantification of how well these timing constraints are met is defined by the notion of **slack**. In early mode, slack represents the distance from arrival time lower bound while in late mode the distance from the upper bound. In late mode analysis, the path endpoints' slack is a useful indicator of how much we are allowed to reduce the user-defined clock period value. Mathematically, for a pin v:

$$slack_v^{early} = at_v^{early} - rat_v^{early}$$

$$slack_v^{late} = rat_v^{late} - at_v^{late}$$

It is worth mentioning that timing propagation is performed only for valid transitions, following the unateness of the cells participating in the path. For instance, in the previously presented Figure 2.2, knowing that OR gates are positive unate, if we consider that $G1$ arc $B \to Z$ has the worst delay with pin B rising, then while performing late mode STA, $G1$ pin Z and $G2$ pin B will also rise (or stay risen). Note that, when performing STA for a given path, the startpoint transition (rise or fall) is unknown. Therefore, STA must be performed for both cases, as different startpoint transitions give us different results.

CHAPTER 3

PERFORMANCE ANALYSIS OF ASYNCHRONOUS CIRCUITS

Asynchronous (or self-timed) circuits are digital logic circuits not synchronized to a clock. The synchronization and communication of such circuits is mainly implemented with handshake protocols, whereas for synchronous circuits, changes of edge signal values are triggered by a repetitive clock pulse. Asynchronous circuits have the potential to be faster and may also have advantages in power consumption. However, due to their complexity, there are not many people trained in this field and there is a lack of dedicated EDA tools. Although, with Moore's law coming to an end, the importance and need of asynchronous systems in the industry increases. One of the most important and challenging problems in this field that needs to be solved is the static timing analysis of such systems.

3.1 Event Models

3.1.1 Petri Nets

A Petri net, or Place/Transition Net (PTnet) [9], [10] is one of the most popular mathematical models used for the representation and the analysis of asynchronous circuits. In general, a Petrinet in its most basic form is nothing more than a directed graph that consists of nodes *(places and transitions)* and *arcs*. Arcs connect places to transitions and *vice versa*, but are not allowed between nodes of the same type. Graphically, transition nodes (*i.e.* events that may occur) are visualized as bars and place nodes (*i.e.* conditions) as circles. Additionally, places might contain a discrete number of markings called tokens.

A transition is considered enabled, when there is at least one token in all of its input places, *i.e.* the places from which an arc runs to the transition. An enabled transition might fire, which means that it will consume exactly one token from every input place and will increment the

tokens of each output place. In Figure 3.1(a) the displayed transition is ready to fire, as each input has at least one token. In Figure 3.1(b) the transition has fired consuming the tokens from its input places and incrementing the tokens of the output places. The Petri net execution is **nondeterministic** because, when multiple transitions are enabled at the same time they can fire in any order. Since firing is nondeterministic, and multiple tokens may be present anywhere in the Petri net (even in the same place), Petri nets are well suited for modeling the **concurrent** behavior of distributed systems.

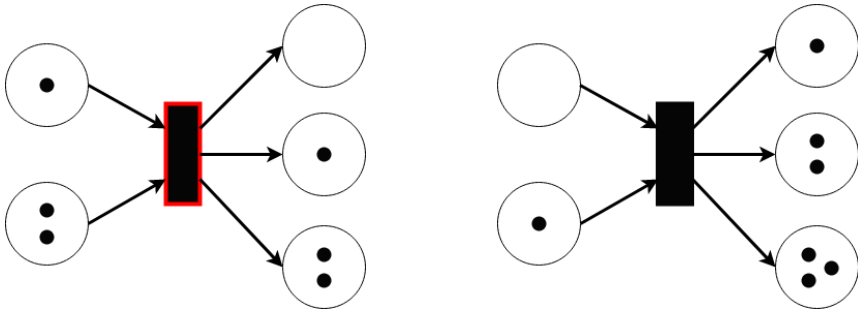


FIGURE 3.1. (a) The transition is ready to fire. (b) The transition has fired.

3.1.1.1 Petri Net Mathematical Definition

Formally, a petrinet is a 3-tuple $PN = (N, M, W)$ where:

1. $N = (P, T, F)$ where:
   a) P and T are disjoint finite sets of places and transitions respectively.
   b) $F \subset (P \times T) \cup (T \times P)$ is a set of arcs
2. $M : P \rightarrow Z$ is a place multiset, where Z is a countable set ($\{1,2,...,k\}$). M describes the marking of the petrinet, as each element's index corresponds to a place and each element's value to the number of tokens at that place.
3. $W : F \rightarrow Z$ is an arc multiset, so that the count (or weight) for each arc is a measure of the arc **multiplicity**. In other words the arc weight describes the number of tokens subtracted from the input place of a firing transition or the number of tokens added to the output place of the firing transition (depending on whether the arc connects a place to a transition or a transition to a place).

In general, the firing rule of a transition can be characterised by subtracting a number of tokens from its input places equal to the multiplicity of the respective input arcs and accumulating a new number of tokens at the output places equal to the multiplicity of the respective output arcs.

However, in most cases arcs have their multiplicity set to 1, which makes F identical to W. In Figure 3.1 all arcs have weight equal to 1, as the firing transition consumes one token from each input place, and adds one token to every output place. An alternative definition of Petri nets is the following 4-tuple:

$$PN = (S, T, W, M_0) \text{ or}$$

$$PN = (S, T, F, M_0) \text{ if } W(x, y) = 1 \wedge (x, y) \in (S \times T) \cup (T \times S)$$

where $(S, T, W) = N$ is a Petri net graph and M_0 is the initial marking. From this point on we assume that every Petri net's arcs have weight equal to 1.

3.1.1.2 Petri Net Classes

Multiple types of Petri nets can be obtained by restricting their formalism. Each restriction leads us to a new Petri net class. Figure 3.2 displays the Petri net types graphically which are briefly described below:

- In a **State Machine** (SM) every transition has one incoming arc, and one outgoing arc, and all markings have exactly one token.
- In a **Marked Graph (MG)** every place has one incoming and one outgoing arc.
- In a **Free-Choice PTnet** (FC) every arc from a place to a transition is either the only arc from the place or the only arc to that transition.
- **Extended Free-Choice PTnet** (EFC) is a Petri net that can be transformed into a FC.
- In an **Asymmetric Choice PTnet** (AC), concurrency and conflict (*i.e.* non-determinism) may occur, but not symmetrically

If the transitions of a Marked Graph correspond to the signal transitions of a specific circuit, then we have a **Signal Transition Graph** (STG). Additionally, the places of a MG or STG can be removed as the marking information can be modelled as labels to transition-to-transition (T2T) arcs. Figure 3.3 depicts the STG of an asynchronous controller. Each signal has two possible transitions (rise or fall).

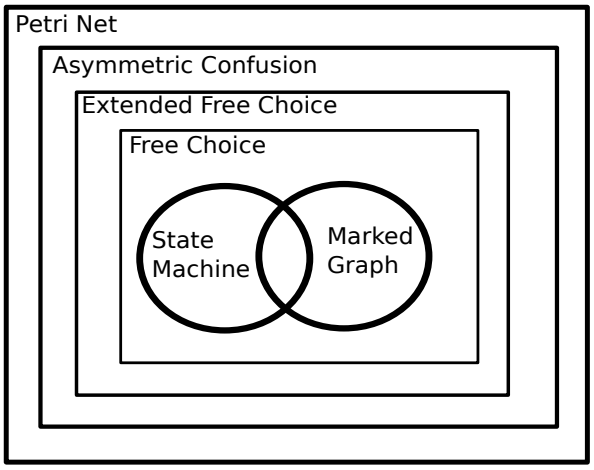


Figure 3.2: Petrinet types graphically.

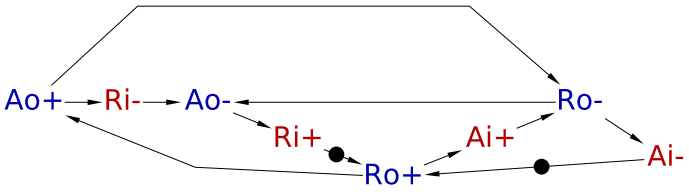


Figure 3.3: `half` Asynchronous Benchmark Signal Transition Graph (STG). Red (or blue) colored transitions correspond to the primary inputs (or outputs) of the controller.

3.1.1.3 Behavioral Properties

Petri nets are characterized by the following three behavioral properties:

- **Reachability**: Given a Petri net (N, M_0) and a marking M, the reachability problem for Petri nets is to decide whether $M \in R(N)$, where $R(N)$ is the set of all possible valid marking configurations. In other words, a marking configuration M is called reachable iff there is at least one transition-firing sequence that can lead us to M starting from M_0.
- **Boundedness**: A Petri net is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0. A 1-bounded Petri nai is also called safe.
- **Liveness**: A transition $t \in T$ is live if from every reachable marking M there is a marking M' such that t is enabled. A Petri net is live (or deadlock-free) if every transition $t \in T$ is live. Petri net can be described as having different degrees of liveness $L1 - L4$. A petri net (N, M_0) is called L_k-live iff all of its transitions are L_k-live, where a transition is:

1. L_0-live (dead) if it can never fire.
2. L_1-live, when it can fire at least once in some firing sequences.
3. L_2-live, when it can fire k times in some firing sequences (for any k).
4. L_3-live, when it can fire infinitely often in some firing sequences.
5. L_4-live (live), if it may always fire, *i.e.* it is L_1-live in every reachable marking.

3.1.2 Event Rule Systems and Event Graphs

Event-Rule Systems (ER Systems) is another mathematical model that can be used for representing asynchronous circuits. This model is suitable for the determination of the circuit's timing performance due to its event occurrence constraints and has been adopted by Burns [1] in his work regarding the performance analysis of asynchronous designs.

A **general** Event-Rule System is a pair $< E, R >$ where:
E is a set of events, and
R is a set of rules defining timed constrains between events.
Each $r \in R$ is written $s \xrightarrow{a} t$, where
$s \in E$ is the source of r,
$t \in E$ is the target of r, and
$a \in [0, +\infty)$ is the delay of r.

A **Repetitive** Event-Rule System $< E', R' >$ is the ER System where events periodically reoccur. In more detail, each repetitive ER System is associated with a **period** value, which describes the amount of time that is required for an event u to occur again.

The elements of E' are formally called **transitions** and the *events* of $E = E' \times \mathbb{N}$ are now represented as $< u, i >$ tuples, where $u \in E'$ is the event's transition, and $i \in \mathbb{N}$ is a number called **occurrence index**, a variable denoting how many times a transition has occurred. It is worth mentioning that, there is an infinite number of *events*, due to the occurrence index, but a finite number of *transitions*.

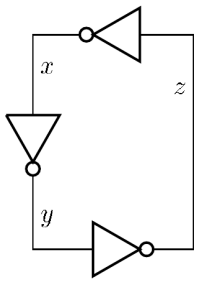
The elements of R' ($r' \in R'$) are now written:

$$r' = < u,\, i - \epsilon > \xrightarrow{a} < v,\, i > \quad or \quad r' = u \xrightarrow{(a,\epsilon)} v, \quad or \quad r' = u \xrightarrow{(a - \epsilon p)} v, \quad \text{where}$$

$a \in [0, +\infty)$ is the delay of r',
$u \in E'$ and $v \in E'$ is the source and target of r' respectively,
$i \in [\max(0, \epsilon),\ +\infty) \in \mathbb{N}$ is the **occurrence index** of the events u and v, and
$p \in [0, +\infty)$ is the **period** that describes the repetition of the events in E'.
$\epsilon \in \mathbb{N}$ is the **occurrence-index offset** of r' which indicates repetition every ϵ period cycles.

Pseudo-repetitive ER Systems is a broader class of ER Systems and a combination of the aforementioned *general* and *repetitive* classes. A *pseudo-repetitive* ER System consists of a finite set of *initial events*, an infinite set of *repeated events*, a finite set of *initial transitions*, a finite set of *repeated transitions*, a finite set of *initial rules* and a finite set of *repeated rules*. As there are only finitely many initial events, after a period of time, only the repeated events occur.

Example: Figure 3.4 displays a ring oscillator of 3 inverters and its respective ER System. Note that E' has six *transitions* (two *transitions* per inverter), as an inverter's output signal will either rise or fall. In general, the ER system of a n-output boolean function will have 2^n *transitions*. Starting from the transition of x rising from zero to one ($x\uparrow$), y will fall ($y\downarrow$) which leads to z rising ($z\uparrow$). Afterwards, due to the rise of z, x falls which makes y rise and eventually, z *falls*. The transition of z falling ($z\downarrow$) will lead us to the *transition* we started from ($x\uparrow$). Thus, the rule from $z\downarrow$ to $x\uparrow$ will have its occurrence-index offset ϵ set to 1.



$$
\begin{aligned}
E' &= \{x\uparrow, y\uparrow, z\uparrow, x\downarrow, y\downarrow, z\downarrow\} \\
R' &= \{\ \langle z\downarrow, i-1\rangle \overset{\alpha_{x\uparrow}}{\mapsto} \langle x\uparrow, i\rangle, \\
&\quad\ \langle x\uparrow, i\rangle \overset{\alpha_{y\downarrow}}{\mapsto} \langle y\downarrow, i\rangle, \\
&\quad\ \langle y\downarrow, i\rangle \overset{\alpha_{z\uparrow}}{\mapsto} \langle z\uparrow, i\rangle, \\
&\quad\ \langle z\uparrow, i\rangle \overset{\alpha_{x\downarrow}}{\mapsto} \langle x\downarrow, i\rangle, \\
&\quad\ \langle x\downarrow, i\rangle \overset{\alpha_{y\uparrow}}{\mapsto} \langle y\uparrow, i\rangle, \\
&\quad\ \langle y\uparrow, i\rangle \overset{\alpha_{z\downarrow}}{\mapsto} \langle z\downarrow, i\rangle \\
&\ \}\ .
\end{aligned}
$$

FIGURE 3.4. Repetitive ER System of a Ring Oscillator with 3 Inverters.

Each ER System can be represented graphically as a **Collapsed-constraint Graph** (or **Event Graph**). Figure 3.5(a) displays the ring oscillator of the previous example as an event graph. Note that in this Figure the unfolded version of the graph is presented. A more compact event graph notation is shown in Figure 3.5(b) in which the edge connecting $z\downarrow$ with $x\uparrow$ has label $a_{x\uparrow} - p$, where $a_{x\uparrow}$ is the delay between the edge's events and p is the period of the event graph. More specifically the notation $z\downarrow \xrightarrow{a_{x\downarrow}-p} x\uparrow$ corresponds to the rule $r = <z\downarrow, i-1> \xrightarrow{a} <x\uparrow, i>$ of the equivalent ER system and denotes the periodicity of the system.

In general, the edges of an event graph are associated with labels in the notation $a - \epsilon p$ or $(a, \epsilon p)$, where ϵ is the edge occurrence index offset. In most designs, the maximum value of ϵ is equal to 1 and is visualized as an edge marking (similar to STG markings or MG tokens) instead of being present at edge labels. To avoid creating confusion, it is worth mentioning that the use of such labeling is only applicable when all of the events share the same period. A more in-depth look will be presented in the next subsection.

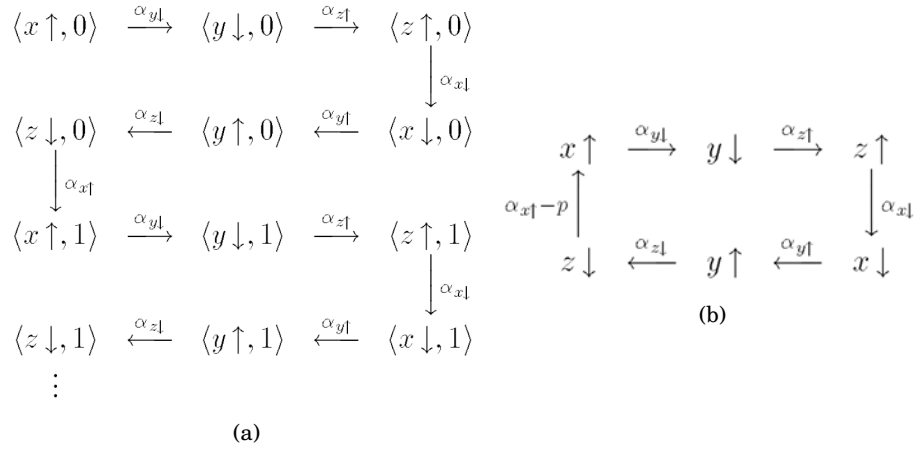


Figure 3.5: (a) Ring Oscillator Unfolded Collapsed-Constraint Graph (b) Ring Oscillator Collapsed-Constraint Graph

3.1.3 Event Graph Constraints

The core constraints of the collapsed-constraint graph is that an event may only occur when all of its predecessor events have occurred. Consider $\bar{t}(u,i)$ the *timing simulation* function which indicates the i'th ($i \geq 0$) occurrence of the event u. The function can be expressed as:

$$\bar{t}(u,i) = x_u + p_u i, \text{ where}$$

p_u is the period of the event's occurrence and
x_u is the **occurrence offset** of u (*i.e.* the amount of time needed for the event to occur after the start of a new period cycle)

Let the events u and v and the rule $r =< u, i-\epsilon > \xrightarrow{a} < v, i >$. In order for v to occur for a given index (i), the event u must have already occurred for the same i value. Taking advantage of the timing simulation function $\bar{t}$ definition the events' constraint can be mathematically written as:

$$\bar{t}(v,\, i) \geq \bar{t}(u,\, i-\epsilon) + a, \text{ for each i} \geq max(0,\epsilon) \Rightarrow$$

$$x_v + p_v i \geq x_u + p_u(i-\epsilon) + a \Rightarrow$$

$$x_v \geq x_u - p_u\epsilon + a + (p_u - p_v)i$$

$$x_v \geq x_u - p_u\epsilon + a + (p_u - p_v)\, max(0,\epsilon) \tag{3.1}$$

If $p_u > p_v$, the constraint equation will never be satisfied for all i values, since i can be arbitrarily large. Thus, as a part of the constraint we also consider that $p_v \leq p_u$.

Going back the ring oscillator's event graph, presented in the previous subsection (Figure 3.5), we observe that the period p is the same across all nodes. In general, all nodes in the same

strongly-connected component of the event graph have the same period, as there is a path between all pairs of nodes, *i.e.* each pair of nodes exists in at least one event graph cycle. In this case, the previously computed constraint Equation 3.1, given that the nodes now belong to the same strongly-connected component, will be transformed to:

$$x_v \geq x_u + a - \epsilon p \tag{3.2}$$

From now on, unless stated, each event graph is assumed to be strongly connected, as most asynchronous design event models are. Therefore, all nodes share the same period and the constraint between two connected nodes ($u \rightarrow v$) is given by Equation 3.2.

3.2 Event Graph Minimum Period Computation

3.2.1 Event Graph Period

As aforementioned, an event graph can be associated with a period value, as long as the event constraints of Equation 3.2 are not violated. Computing the minimum period poses a big challenge, as this value is strictly defined by the maximum delay/occurrence-index-offset sum ratio, derived from the graph cycles' edges. More specifically, given an event graph $G = (E', R')$, let $\overrightarrow{a}$ and $\overrightarrow{\epsilon}$ being vectors containing the event graph's edge delays and edge occurrence indexes respectively:

$$\overrightarrow{a} = (a_0, a_1, a_2, ..., a_{|R'|-1}), \quad \overrightarrow{\epsilon} = (\epsilon_0, \epsilon_1, \epsilon_2, ..., \epsilon_{|R'|-1})$$

Let U_k being a vector of zeros and ones where each index corresponds to a specific edge. Variable k can take values from 0 to $2^{|R'|-1}$ as U_k can contain any combination of edges.

$$U_k = (e_0, e_1, e_2, ..., e_{|R'|-1}), \quad \text{for } e_i = \{0, 1\}$$

Note that, the multiplication of U_k with $\overrightarrow{a}$ or $\overrightarrow{\epsilon}$ gives us the sum of the delays or occurrence indexes of the edges present in U_k (*i.e.* elements with value 1). Vectors that contain only edges that form an event graph cycle are called *cycle vectors*. Based on the definitions above, the minimum period value can be mathematically defined as:

$$p = max\left\{ \frac{U_k^T \overrightarrow{a}}{U_k^T \overrightarrow{e}} \;\middle|\; U_k \text{ is a cycle vector} \right\} \tag{3.3}$$

if $U_k^T \overrightarrow{e} > 0$ for all cycle vectors U_k

Example: Consider the event graph shown in figure 3.6 where in fig 3.6(a), we see the standard labeling and in fig 3.6(b) the labels denote the numbered arcs (or the indexing of the vectors).

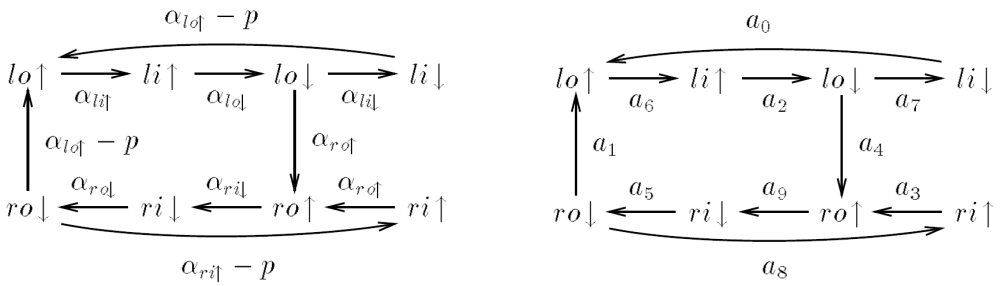


Figure 3.6: (a) Event-graph with standard labeling (b) Event-graph with numbered arcs

The three cycles of the graph can be presented by the cycle-vector matrix:

$$U^T = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Given the equation 3.2 the period is computed:

$$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} p \geq \begin{pmatrix} a_{lo\uparrow} + a_{lo\downarrow} + a_{ro\uparrow} + a_{ro\downarrow} + a_{li\uparrow} + a_{ri\downarrow} \\ a_{lo\uparrow} + a_{lo\downarrow} + a_{li\uparrow} + a_{li\downarrow} \\ a_{ro\uparrow} + a_{ro\downarrow} + a_{ri\uparrow} + a_{ri\downarrow} \end{pmatrix} = \begin{pmatrix} a_{cycle0} \\ a_{cycle1} \\ a_{cycle2} \end{pmatrix}$$

Thus, the minimum p is $max(a_{cycle0}, a_{cycle1}, a_{cycle2})$.

3.2.2 Burns' Primal-Dual Method Algorithm

One way to compute the event graph's minimum period value is by exhaustively exploring all graph cycles and computing the max delay/occurrence-index sum ratio. Although, this method is computational too expensive as the number of cycles in in an arbitrary graph can be exponential in the number of its arcs.

For this reason, various methodologies have been proposed in literature that aim to compute the event model period in non-exponential complexity. In more detail, Ramamoorty and Ho proposed a matrix based algorithm [13], whereas Maggot proposed and LP-based approach [11]. Burns [1] has presented both an LP based algorithm as well as a Primal-Dual Method Algorithm [12]. The algorithm is iterative and can compute the minimum period of a given event graph $G' = (E', R')$ in polynomial time complexity (see algorithm 1). The algorithm is constructed by applying the primal-dual method, a general technique for constructing special-case algorithms for solving linear programs. In the next section, we will have an in-depth look at Burn's algorithm and the intuition behind each step through examples, as well as a complete view of its applications in real asynchronous systems.

1. Form the initial feasible solution $x^{(0)}$, $p^{(0)}$

   a) Temporarily remove all arcs with $\epsilon > 0$ in G'

   b) Topologically sort the resulting acyclic graph

   c) Using the topological order, set

$$x_v^{(0)} = \begin{cases} 0, & \text{if } v \text{ is a root} \\ max\left\{ x_u^{(0)} + a \;\middle|\; \langle u,\, i-\epsilon\rangle \xrightarrow{a} \langle v,\, i\rangle \in R \;\wedge\; \epsilon \le 0 \right\}, & \text{otherwise} \end{cases}$$

   d) Set

$$p^{(0)} = max\left\{ \frac{x_v^{(0)} - x_u^{(0)} - a}{-\epsilon} \;\middle|\; \langle u,\, i-\epsilon\rangle \xrightarrow{a} \langle v,\, i\rangle \in R \;\wedge\; \epsilon > 0 \right\}$$

2. For each arc $\langle u,\, i-\epsilon\rangle \xrightarrow{a} \langle v,\, i\rangle$, if $x_v^{(k)} = x_u^{(k)} + a - \epsilon p^{(k)}$, then the arc is marked as a *critical arc*.

3. If the graph of critical arcs is cyclic or $p^{(k)} = 0$, then set $\hat{p}^{(k)} = 0$ and exit with the optimal solution $x(k)$, $p(k)$. Else,

   a) Topologically sort the graph of critical arcs.

   b) Set $\hat{p}^{(k)} = 1$

   c) Using the topological order, set

$$\hat{x}_v^{(k)} = \begin{cases} 0, & \text{if } v \text{ is a root} \\ min\left\{ \hat{x}_u^{(k)} - \epsilon\hat{p}^{(k)} \;\middle|\; \langle u,\, i-\epsilon\rangle \xrightarrow{a} \langle v,\, i\rangle \text{ is critical} \right\}, & \text{otherwise} \end{cases}$$

   d) Set

$$\theta^{(k)} = min\left\{ \frac{x_v^{(k)} - x_u^{(k)} + \epsilon p^{(k)} - a}{\hat{x}_v^{(k)} - \hat{x}_u^{(k)} + \epsilon\hat{p}^{(k)}} \;\middle|\; \hat{x}_v^{(k)} - \hat{x}_u^{(k)} + \epsilon\hat{p}^{(k)} > 0 \right\}$$

   e) Set

$$x^{(k+1)} = x^{(k)} - \theta^{(k)}\hat{x}^{(k)}, \qquad p^{(k+1)} = p^{(k)} - \theta^{(k)}\hat{p}^{(k)}$$

4. Increment k and go to step two.

Algorithm 1: Burns' Algorithm, a polynomial-complexity algorithm which finds the cycle period of a given event-graph

CHAPTER 4

BURNS' ALGORITHM ANALYSIS

In this chapter, we take a deeper look at Burns' algorithm by explaining the intuition behind each of its steps through detailed examples. We elaborate on the period relaxation and how it impacts the occurrence offsets of event graph nodes. The fact that no cycles are explored throughout the period computation process, is key for the polynomial complexity of the algorithm.

4.1 Computing the Initial Feasible Solution

In this section we present a complete example on Burns' algorithm, as shown in Figure 4.5. Let the event graph shown in Fig 4.5(a). The **first step** of the algorithm is to form the initial feasible solution $x^{(0)}$, $p^{(0)}$. All arcs with positive occurrence index offsets ($\epsilon > 0$) are temporarily removed in order to compute the longest path to each node using the topological sort order of the resulting acyclic graph. Recall that:

$$x_v^{(0)} = \begin{cases} 0, & \text{if } v \text{ is a root} \\ max\left\{ x_u^{(0)} + a \;\middle|\; \langle u, i-\epsilon\rangle \xrightarrow{a} \langle v, i\rangle \in R \;\wedge\; \epsilon \leq 0 \right\}, & \text{otherwise} \end{cases}$$

In Figure 4.5(b), the initial occurrence offsets (x-values) are computed for each node. After removing positive occurrence index offset edges, one of the valid topological orderings is: {B, C, D, E, F, G, H, I, J, K, A}. Due to this ordering, when we visit a node v to compute its x-value, we know that the x-values of its predecessors have already been computed as they are positioned before v in the topological sort list. For instance $x_h^{(0)}$ will be equal to $max(x_g^{(0)} + 1, x_c^{(0)} + 1)$ where $x_g^{(0)}$ and $x_c^{(0)}$ have already been computed as nodes G and C are before B topologically.

With the initial event graph occurrence offsets, we may compute the initial feasible period value. The period is derived from arcs with positive occurrence index offsets $\epsilon > 0$, which makes sense, as these arcs define the periodicity of the system forming cycles. In more detail, let nodes u

and v, where $< u, i-\epsilon > \xrightarrow{a} < v, i >$ (or alternatively $u \xrightarrow{a-\epsilon p} v$) and $\epsilon > 0$. The event graph constraint $x_v \geq x_u + a - \epsilon p$ which was presented in the previous section can easily lead us to the formula of the period:

$$\begin{aligned} x_v &\geq x_u + a - \epsilon p \Rightarrow \\ -\epsilon p &\leq x_v - x_u - a \Rightarrow \\ p &\geq \frac{x_v - x_u - a}{-\epsilon}, \text{ for } \epsilon > 0 \end{aligned} \tag{4.1}$$

Equation 4.1, can be viewed as the event graph period constraint, when all event occurrence offsets are known. For each pair of connected nodes (u, v), this constraint introduces a new local lower bound to the period which is only relevant to the connection of the aforementioned pair. Therefore, the minimum acceptable period value is the greatest lower bound of the constraints. Revisiting Burns' algorithm, the initial period is indeed equal to:

$$p^{(0)} = max\left\{ \frac{x_v^{(0)} - x_u^{(0)} - a}{-\epsilon} \;\middle|\; \langle u,\, i-\epsilon \rangle \xrightarrow{a} \langle v,\, i \rangle \in R \;\wedge\; \epsilon > 0 \right\}$$

Going back to our example, there are five edges with positive occurrence index offsets where each one introduces a new period lower bound. The period will be assigned to the maximum computed value. For instance, from edge $A \rightarrow B$ we have $p_1 = (x_b^{(0)} - x_a^{(0)} - a_{ab}) / (-\epsilon_{ab}) = 7$, from $A \rightarrow F$ we have $p_5 = 11$ *e.t.c.* Eventually, $p^{(0)} = max(p1,\, p2,\, p3\,, p4\,,\, p5) = 11$ which makes sense as the longest path to node A is 6 time units and in order to reach F we need 5 additional time units.

4.2 Critical-Arc Graph Formation and Termination Condition

The **second step** of the algorithm is the formation of the *critical-arc graph*. For each arc $\langle u,\, i-\epsilon \rangle \xrightarrow{a} \langle v,\, i \rangle$, if $x_v^{(k)} = x_u^{(k)} + a - \epsilon p^{(k)}$ the arc is considered *critical*. This means that x_v and x_u cannot take lower and higher values respectively, for a specific period. In other words, event v occurs after u delayed exactly by the edge'sa time units. Note that a node can be present in multiple critical arcs. Figure 4.5(c) shows the critical-arc graph of our example, with greyed-out arcs being the non-critical ones. For instance, if we take edge $A \rightarrow F$ we have, $x_f = x_a + a_{af} - \epsilon_{af} p$, or $0 = 6 + 5 - 11$. Bear in mind that, in most cases, arcs that have their occurrence index offset equal to zero (for example $G \rightarrow H$) are all critical. Although, this cannot be generalized as a critical arc might become non-critical at later iterations of the algorithm.

In the **third step**, we initially check whether the critical-arc graph contains at least one cycle. If this is the case, the algorithm terminates with the optimal solution. Recall Equation 3.3 which states that the minimum period is equal to the maximum delay/occurrence-index-offset ratio of the graph cycles. The formation of a cycle in the critical-arc graph indicates that the maximum $(a \,/\, \epsilon)$ ratio has been achieved and any further period relaxation will violate the event graph

constraints. It is important to clarify that, if we increase the x-value of a node belonging to a critical cycle, we also increase the x-values of the remaining cycle nodes by the same amount. For the constraint $x_v \geq x_u + a - \epsilon p$ of two connected nodes ($u \rightarrow v$), it might be a common mistake to think that we can increase x_v indefinitely without causing violations. However, if v belongs to a critical cycle, it participates in another critical arc as its startpoint, which means that there will be a second constraint that will not allow us to increase its offset x.

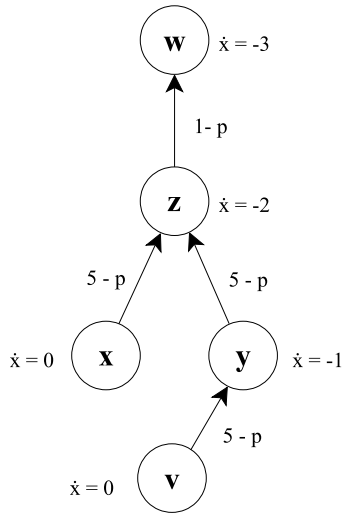
4.3 $\hat{x}$-Values Intuition

After the formation of the acyclic critical-arc graph, a topological sort is performed, and a new variable $\hat{p}$ is set to 1. This new variable has little to no significance in the algorithm, as it is used as a flag to indicate if the critical-arc graph is acyclic with nonzero period. Using the topological sort order of the acyclic graph the $\hat{x}$-values are computed for each node. Recall the formula:

$$\hat{x}_v^{(k)} = \begin{cases} 0, & \text{if } v \text{ is a root} \\ min\left\{ \hat{x}_u^{(k)} - \epsilon\hat{p}^{(k)} \;\middle|\; \langle u, i-\epsilon\rangle \xrightarrow{a} \langle v, i\rangle \text{ is critical} \right\}, & \text{otherwise} \end{cases}$$

Intuitively, the $\hat{x}$-value of a node represents the critical arc period distance from the furthest root node. Having in mind that $\hat{p}$ is equal to 1 and occurrence-index offsets (ϵ) are greater or equal to zero, all $\hat{x}$ values will be non-positive. The application of the min operation to non-positive numbers is similar to the application of the max operation to positive numbers. In other words, $\hat{x}$ values represent the ϵ-value longest path to each node. Note that the computation is performed on the critical-arc graph.

Example: Consider the critical arc-graph shown in Figure 4.1. The topological sort ordering of the nodes is $\{v, x, y, z, w,\}$. Initially, roots v and x have their $\hat{x}$ set to 0. Afterwards, node y will be assigned with $\hat{x}$ equal to -1 as the ϵ-distance of the furthest root node (v) is 1. In the same manner, node z will have its $\hat{x}$ value set to -2 as the furthest root node is v, with a period distance of 2. Eventually, node w will have $\hat{x}$ equal to -3 as its distance from the furthest root is 3.



Topological Order: $\{v, x, y, z. w\}$

$\hat{x}_v = 0$

$\hat{x}_x = 0$

$\hat{x}_y = \hat{x}_v - \epsilon\hat{p} \; = \; 0 - 1 \; = \; -1$

$\hat{x}_z; = min\{\hat{x}_x - \epsilon_{xz}\hat{p},\, \hat{x}_y - \epsilon_{yz}\hat{p}\} = min\{-1,\, -2\} = -2$

$\hat{x}_w = \hat{x}_z - \epsilon_{zw}\hat{p} \; = \; -3$

FIGURE 4.1. A critical-arc graph with the $\hat{x}$ values annotated

Figure 4.5(d) displays the annotated $\hat{x}$ values on the critical-arc graph of our complete example. Notice that, all $\hat{x}$ values except $\hat{x}_f$ are equal to 0 as the critical-arc graph, at this point, has only one edge with positive occurrence-index offset ($A \xrightarrow{5-p} F$).

4.4 Intuition behind θ value

The next step of Burn's Algorithm (step 3d) is to compute the value θ which can be viewed as the minimum excess period delay. It also represents the amount of the period reduction at the end of each iteration. The computation of θ is performed accross the whole graph. To fully understand the meaning of θ we must take a look at its denominator and numerator separately. The value of θ is determined by the following formula:

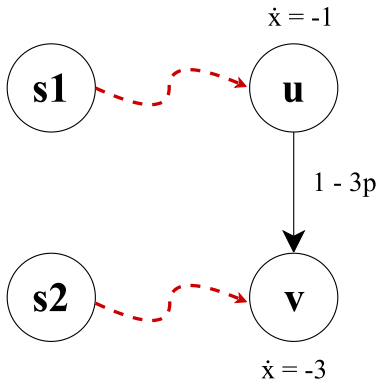
$$\theta^{(k)} = min\left\{ \frac{x_v^{(k)} - x_u^{(k)} + \epsilon p^{(k)} - a}{\hat{x}_v^{(k)} - \hat{x}_u^{(k)} + \epsilon\hat{p}^{(k)}} \;\middle|\; \hat{x}_v^{(k)} - \hat{x}_u^{(k)} + \epsilon\hat{p}^{(k)} > 0 \right\}, \quad \text{where } u \xrightarrow{a-\epsilon p} v$$

The **numerator** of this formula may be viewed as the *excess delay* between u and v, or in other words, it shows how much delay we must add to edge $u \rightarrow v$ in order for it to be critical. Note that, if the arc connecting u to v is critical, the nominator of θ will be equal to zero, as the occurrence offset x_v is at its minimum possible value and strictly defined by x_u.

Regarding the **denominator**, the difference $\hat{x}_v^{(k)} - \hat{x}_u^{(k)}$ represents how many more period cycles node v needs in order to occur, in relation to u, starting from the furthest root node in the critical-arc graph. Product $\epsilon\hat{p}^{(k)}$ is equal to the occurrence index offset of the edge (ϵ), as $\hat{p}^{(k)}$ is set to 1, and displays the period distance between u and v through their arc. A more

clear way of viewing the denominator is by examining the difference $(\hat{x}_u^{(k)} - \epsilon\hat{p})$. This value represents the furthest root period distance of v $(\hat{x}'_v)$, if edge $u \to v$ was critical. Therefore, the denominator $\hat{x}_v^{(k)} - (\hat{x}_u^{(k)} - \epsilon\hat{p}^{(k)})$ shows how much more period distance would v have, if arc $u \to v$ was critical. The meaning of the denominator will be more clear with the following example:

Example: Consider u and v, where $u \xrightarrow{1-3p} v$. As shown in Figure 4.2, the $\hat{x}$-value of u has been computed to -1, by the longest critical-arc path starting from root $s1$, and the $\hat{x}$-value of v is set to -3, due to the critical-arc path starting from root node $s2$. The denominator of θ is equal to 1 which means that, for the current **current period**, if event v was to occur through edge $u \to v$, its period distance would be bigger by 1.



The period distance of v's occurrence through edge $u \to v$ is equal to:

$$\hat{x}_u^{(k)} - \epsilon\hat{p}^{(k)} = -4$$

The *theta* denominator is equal to:

$$\hat{x}_v^{(k)} - (\hat{x}_u^{(k)} - \epsilon\hat{p}^{(k)}) = -3 - (-4) = 1$$

FIGURE 4.2.

Lemma 4.1 There are no valid θ values extracted from critical arcs as they produce non-positive theta denominators.

Proof. Let nodes $\{u1, u2, ..., un\}$ being the predecessors of v where each arc $ui \to v$ is critical. Starting from the formula of the $\hat{x}$-values:

$$\hat{x}_v^{(k)} = min\{\hat{x}_{ui} - \epsilon_{uiv}\hat{p}\}, \quad \text{for } i \in \{1, n\}$$

Without loss of generality, let $\hat{x}_{u1}^{(k)} - \epsilon_{u1v}\hat{p}^{(k)}$ being the minimum value. Therefore:

$$\hat{x}_v^{(k)} = \hat{x}_{u1}^{(k)} - \epsilon_{u1v}\hat{p}^{(k)} \Rightarrow \hat{x}_v^{(k)} - \hat{x}_{u1}^{(k)} + \epsilon_{uiv}\hat{p}^{(k)} = 0 \tag{4.2}$$

$$\hat{x}_{u1}^{(k)} - \epsilon_{u1v}\hat{p}^{(k)} < \hat{x}_{ui}^{(k)} - \epsilon_{uiv}\hat{p}^{(k)} \Rightarrow -\hat{x}_{u1}^{(k)} + \epsilon_{u1v}\hat{p}^{(k)} > -\hat{x}_{ui}^{(k)} + \epsilon_{uiv}\hat{p}^{(k)}, \quad \text{for } i \neq 1 \tag{4.3}$$

Adding $\hat{x}_v^{(k)}$ to equation 4.3 we construct the denominator of θ:

$$\hat{x}_v^{(k)} - \hat{x}_{u1}^{(k)} + \epsilon_{u1v}\hat{p}^{(k)} > \hat{x}_v^{(k)} - \hat{x}_{ui}^{(k)} + \epsilon_{uiv}\hat{p}^{(k)} \overset{(4.2)}{\Longrightarrow} \hat{x}_v^{(k)} - \hat{x}_{ui}^{(k)} + \epsilon_{uiv}\hat{p}^{(k)} < 0$$

■

After examining the intuition behind the θ numerator and denominator separately, we can now state that the θ value of a specific edge can be viewed as excess delay, *i.e.* the amount of delay needed for the edge to become critical, evenly distributed among the excess edge cycles. A reduction to the period by that amount will cause the edge to become critical. Thus, in order to avoid creating constraint violations, the chosen global θ value must be the minimum of individual *theta*s.

Going back to our complete example in Figure 4.5(d), we have the following valid theta values:

Edge $A \rightarrow B$: $\theta 1 = \frac{x_b - x_a + \epsilon_{ab} p - a_{ab}}{\hat{x}_b - \hat{x}_a + \epsilon_{ab} \hat{p}} = \frac{0-6+11-1}{0-0+1} = 4$

Edge $A \rightarrow C$: $\theta 2 = \frac{x_c - x_a + \epsilon_{ac} p - a_{ac}}{\hat{x}_c - \hat{x}_a + \epsilon_{ac} \hat{p}} = \frac{0-6+11-2}{0-0+1} = 3$

Edge $A \rightarrow D$: $\theta 3 = \frac{x_d - x_a + \epsilon_{ad} p - a_{ad}}{\hat{x}_d - \hat{x}_a + \epsilon_{ad} \hat{p}} = \frac{0-6+11-3}{0-0+1} = 2$

Edge $A \rightarrow E$: $\theta 4 = \frac{x_e - x_a + \epsilon_{ae} p - a_{ae}}{\hat{x}_e - \hat{x}_a + \epsilon_{ae} \hat{p}} = \frac{0-6+11-4}{0-0+1} = 1$

Edge $F \rightarrow K$: $\theta 5 = \frac{x_k - x_f + \epsilon_{fk} p - a_{fk}}{\hat{x}_k - \hat{x}_f + \epsilon_{fk} \hat{p}} = \frac{5-0+0-1}{0-(-1)+0} = 4$

Therefore: $\theta = min\{\theta 1, \theta 2, \theta 3, \theta 4, \theta 5\} = 1$

4.5 Updating occurrence offsets x and period p

The **last step** in Burns' algorithm before proceeding to its next iteration, is the period and occurrence offset update. Having computed the θ value, the period is reduced by that amount. For instance, the period of the example shown in Figure 4.5 will be reduced from 11 to 10, as θ has been set to 1. The occurrence offsets of the nodes are updated accordingly:

$$x^{(k+1)} = x^{(k)} - \theta^{(k)} \hat{x}^{(k)}, \qquad p^{(k+1)} = p^{(k)} - \theta^{(k)} \hat{p}^{(k)}$$

Note that nodes with $\hat{x}$ value equal to 0 will have no changes in their occurrence offsets as they are not affected by period updates. These nodes often belong to paths where no edges with non-zero occurrence index offsets exist. On the other hand, nodes with $\hat{x}$ values smaller than -1, will have their occurrence offsets incremented by θ multiple times ($|\hat{x}|\theta$). For example, a node which has a period cycle distance of 2 from the furthest root, will have its occurrence offset incremented two times by θ. It is a common misconception to assume that the updated occurrence offsets always impact the actual occurrence of the events. Occurrence offsets are no more than time offsets, relative to the current period value. A visual explanation of the previous statement is provided by the following example:

Example: Figure 4.3 shows an example of how the occurrence offsets of nodes u, v, w, y and z update after the period reduction. Above the nodes, their $\hat{x}$ value is displayed, which indicates the period distance from u affecting the x values. For instance, we know that the offset x_z will be incremented 2 times by $\theta = 1$ as there are two period intervals between z and u ($|\hat{x}_z| = 2$). Note that, after the period update, the events still occur the same time as before.

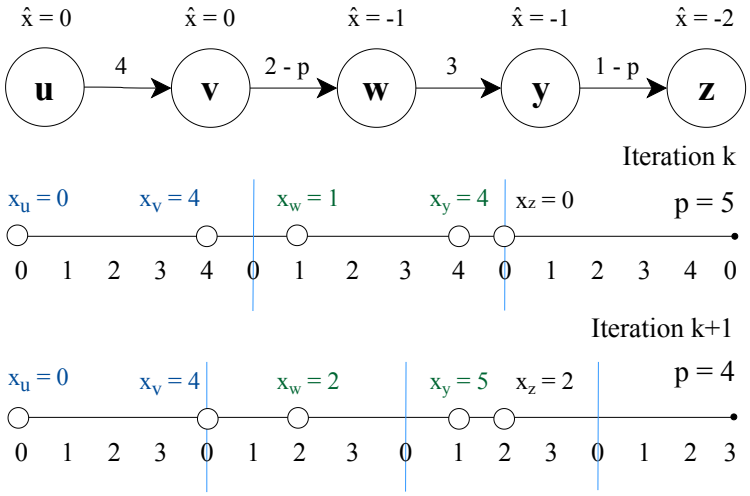


FIGURE 4.3. Visual representation of occurrence offset values (x) between subsequent iterations. The difference between the old and new period value gives us θ, thus $\theta = 5-4 = 1$.

Lemma 4.2 The fact that occurrence offsets are updated to be relative to the new period value means that formula $x^{(k+1)} = x^{(k)} - \theta^{(k)}\hat{p}^{(k)}$ can be derived from $p^{(k+1)} = p^{(k)} - \theta^{(k)}\hat{p}^{(k)}$.

Proof. It is obvious that the occurrence of the nodes that have no critical arcs in their in-degree should not be affected by the period update. For the rest of the nodes, starting from the formula of $\hat{x}$ we have:

$$\hat{x}_v^{(k)} = min\{\hat{x}_u^{(k)} - \epsilon\hat{p}^{(k)}\} \Rightarrow \hat{x}_v^{(k)} \leq \hat{x}_u^{(k)} - \epsilon\hat{p}^{(k)} \Rightarrow -\epsilon\hat{p}^{(k)} \geq \hat{x}_v^{(k)} - \hat{x}_u^{(k)} \tag{4.4}$$

From the period update formula:

$$p^{(k+1)} = p^{(k)} - \theta\hat{p}^{(k)} \Rightarrow p^{(k)} = p^{(k+1)} + \theta\hat{p}^{(k)} \tag{4.5}$$

Taking the event graph constraint equation:

$$x_v^{(k)} \geq x_u^{(k)} + a - \epsilon p^{(k)} \overset{(4.5)}{\Longrightarrow} x_v^{(k)} \geq x_u^{(k)} + a - \epsilon\,(p^{(k+1)} + \theta\hat{p}^{(k)}) \Rightarrow$$

$$x_v^{(k)} \geq x_u^{(k)} + a - \epsilon p^{(k+1)} - \theta\epsilon\hat{p}^{(k)} \overset{(4.4)}{\geq} x_u^{(k)} + a - \epsilon p^{(k+1)} + \theta\,(\hat{x}_v^{(k)} - \hat{x}_u^{(k)}) \Rightarrow$$

$$x_v^{(k)} \geq x_u^{(k)} + a - \epsilon p^{(k+1)} + \theta\hat{x}_v^{(k)} - \theta\hat{x}_u^{(k)} \Rightarrow$$

$$(x_v^{(k)} - \theta\hat{x}_v^{(k)}) \geq (x_u^{(k)} - \theta\hat{x}_u^{(k)}) + a - \epsilon p^{(k+1)} \tag{4.6}$$

Equation 4.6 indicates that, if we update $x^{(k+1)}$ values to $x^{(k)} - \theta\hat{x}^{(k)}$, there will be no constraints violated. Therefore, the occurrence offset update formula is: $x^{(k+1)} = x^{(k)} - \theta\hat{x}^{(k)}$. It is worth mentioning that this formula also provides valid x values to the nodes that have no critical arcs in their in-degree, as for these nodes $\hat{x}^{(k)} = 0$. ■

4.6 Continuing with Burns' Example

In the previous sections, we delved into the intuition behind each step of the algorithm, presenting an example at the end of each step (Figures 4.5(a) to 4.5(d)). The last step of each iteration updates the period as well as the occurrence offsets for the next iteration.

Figure 4.6 displays the next iterations of Burns' Algorithm example. Notice that each iteration introduces a new critical arc in the event graph. More specifically, as shown in fig 4.6(a), the period p has been reduced at the end of the previous iteration to 10, making arc $A \rightarrow E$ critical. Given the annotated x and $\hat{x}$ values, θ is computed equal to 1. Figure 4.5(c) shows the critical-arc graph after the second period reduction ($p = 9$) introducing a new critical arc $A \rightarrow D$. Eventually, all arcs are marked as critical, and critical cycles are formed (Figure 4.6(d)), leading to the termination of the algorithm, with optimal period equal to 7 time units. Figure 4.4 summarises Burns' Primal Dual Method Algorithm by presenting a brief description of its variables and steps.

Variables:

x:	occurrence offset
a:	edge delay
ϵ:	occurrence index offset
p:	event model period
$\hat{x}$:	max root period distance
$\hat{p}$:	flag that shows if period can be reduced
θ:	period reduction amount

Constraint: $x_v \geq x_u + a - \epsilon p$, for each $u \rightarrow v$

Burns' Algorithm
- Compute initial x values
- Compute initial period
- *L:* Construct critical arc graph
- **if** critical arc graph is cyclic, **then** exit
- Compute $\hat{x}$ values
- Compute θ
- Reduce period and x values
- Go to L

FIGURE 4.4. Burns' Primal Dual Method Algorithm Summary

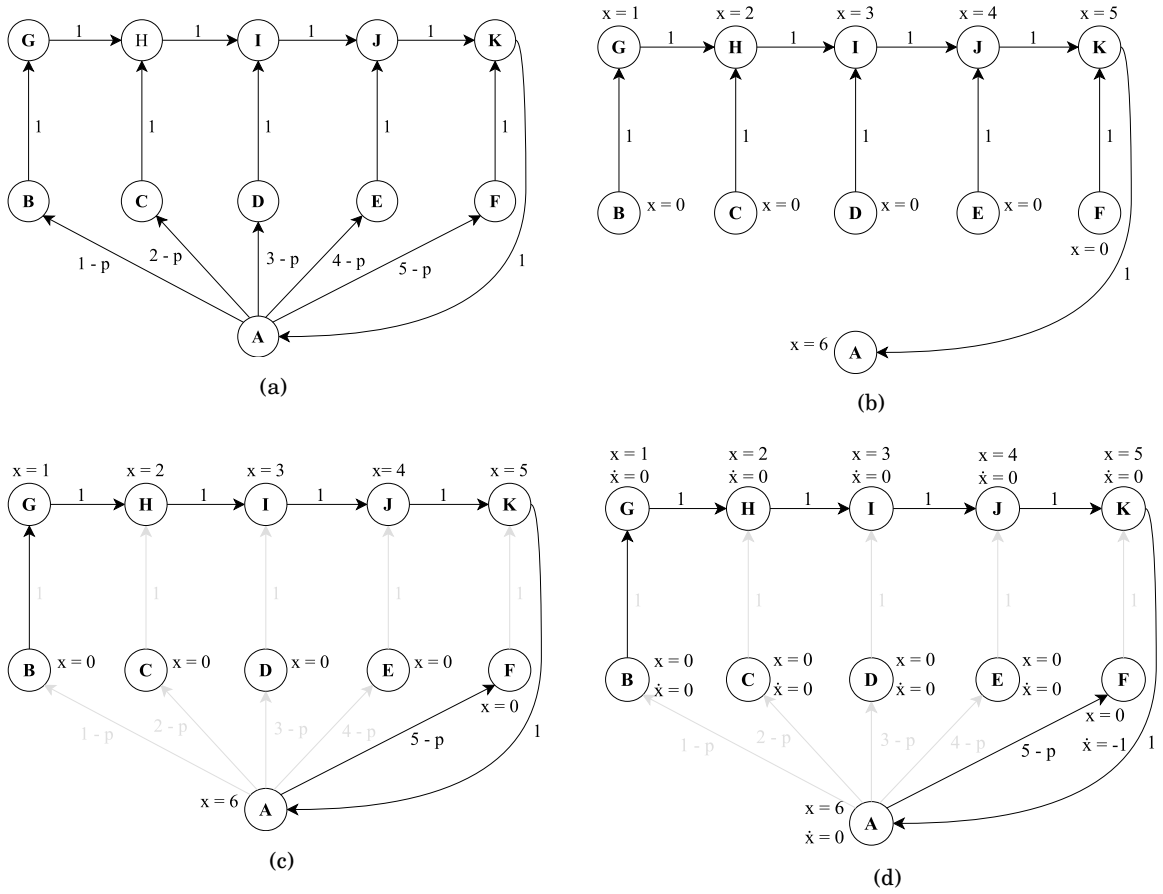


FIGURE 4.5. (a) The initial event graph before performing Burns' algorithm. (b) The event graph without the positive-occurrence-index arcs. Event occurrence offsets x have been initialized and the period has been computed to be equal to 11. (c) The critical-arc graph of the first iteration. Arcs with grey color are non-critical. (d) The event graph with the $\hat{x}$ values annotated.

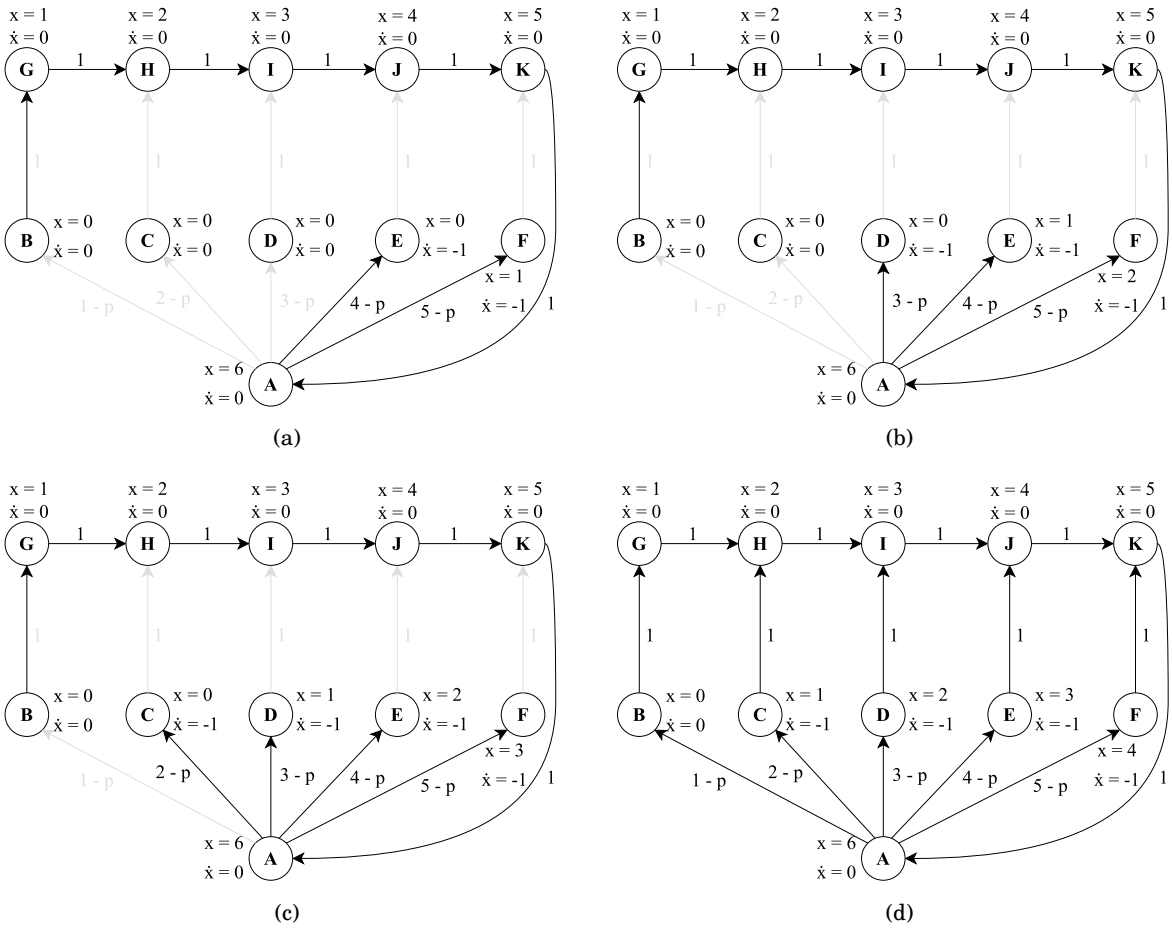


FIGURE 4.6. (a) Second iteration (p = 10) (b) Third iteration (p = 9) (c) Fourth iteration (p = 8) (d) Fifth iteration (p = 7). The algorithm terminates as critical cycles have formed. Note that the delay/occurrence-index-offset ratio of each cycle is equal to 7.

CHAPTER 5

PETRINET TIMING SIMULATION

As presented in previous sections, Burns' algorithm is applied on **Collapsed-constraint Graphs** (event graphs) which correspond to STGs and actually stem from Event Rule Systems. However, if the timing model is described by a Petrinet or a Free-Choice Petrinet, a graph transformation is necessary in order to create an event graph compatible with Burns' Algorithm. Note that an event graph can be also viewed as a **Marked Graph** (*i.e.* a petrinet where each place has exactly one incoming arc and one outgoing arc), with the difference that place tokens/markings are included in the transition-to-transition edge weights as occurence index offsets (ϵ).

5.1 Initial Petrinet to Event Graph Transformation

The main issue that makes Burns' algorithm not applicable to petrinets is the presence of place nodes in the graph. The procedure of removing such nodes is straightfoward but might impact the correctness of the event graph. For each place, each predecessor transition is connected directly to every successor transition. If the place has a token then each transition-to-transition edge will have its occurrence index offset (ϵ) set to 1. Note that, each place might be associated with more than one predecessor/successor transition which is usually the case for systems described by Free-Choise Petrinets. In figure 5.1 the aforementioned transformation is displayed.

It is worth mentioning that the transformation might alter the graph to the point where new traps and sinks are introduced. According to commoner's theorem [3], a free-choice system [8] is live if and only if every proper siphon includes an initially marked trap. The same applies to marked graphs as they are a Free-Choice Petrinet sub-type. Therefore, the resulting event graph (viewed as a marked graph) must be live and 1-bounded. In other words, we must guarantee that each graph cycle contains at least one edge with non-zero occurrence index offset (ϵ).

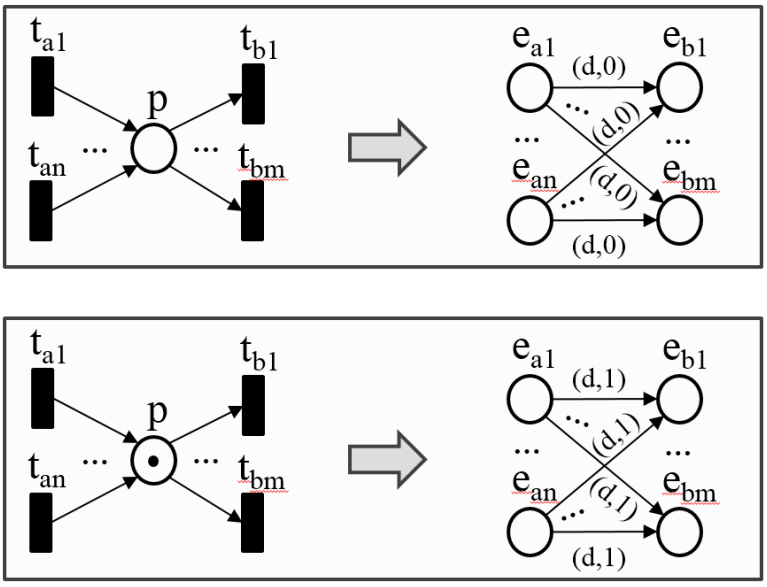


Figure 5.1: This figure shows the initial transformation where all of the predecessors and successors of a given place are connected with each other. Each transition-to-transition edge is associated with a delay ($a = d$) and the occurrence index offset (ϵ) which can be either one on zero depending on whether the place has a token or not.

The initial PTnet marking plays a crucial role to the correctness of the event graph after the transformation. Figure 5.2(c) shows the event graph derived from the PTnet in fig. 5.2(a). Transitions $t0$ and $t1$ have an arc towards $t2$ as they are connected with the same place in the initial PTnet. Arc $t3 \rightarrow t0$ inherits the place's marking which translates to an occurrence index offset equal to 1. However, the generated event graph contains a cycle ($\{t2, t1\}$) without positive occurrence index offsets, thus Burns' Algorithm will fail as, after the removal of arcs with positive occurrence index offsets, it considers the graph acyclic.

Figure 5.2(b) displays an alternative PTnet initial marking. In this case, the event graph after the transformation (fig. 5.2(d)) will have two marked edges ($t0 \rightarrow t2$ and $t1 \rightarrow t2$) with each cycle having at least one arc with positive occurrence index offset (ϵ). At this point it is obvious that propagating tokens to the places with the highest degree will lead to more marked edges in the final event graph. However, this should not be considered a solution as there might still be cycles with no marked edges. Additionally, this might also lead us to scenarios where cycles contain multiple edges with positive ϵ-values, possibly affecting the final period.

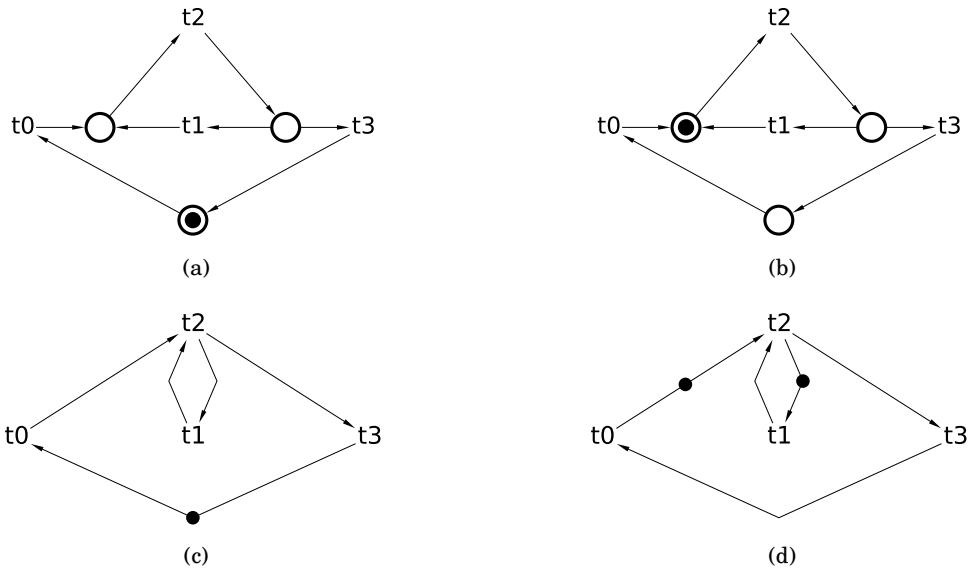


FIGURE 5.2. This figure shows how, different PTnet initial token placements affect the correctness of the generated event graphs. The event graph in (c) is constructed from PTnet (a) and the one in (d) from PTnet (b). The different token placement in (b) will lead to a marked edge at each graph cycle.

5.2 Ensuring the Event Graph's Correctness

In this section we will show the transformation step that takes place after the removal of places and leads us to an event graph compatible with Burns' algorithm. Let the PTnet shown in figure 5.3(a). The place removal step of the transformation will lead us to the event graph in fig. 5.3(b) where two cycles have no marked edges ($\{t1,t2\}$, $\{t4,t5\}$). Note that, propagating the token to another place does not solve the issue as we will still have one unmarked cycle. For instance, if we mark the place connecting $\{t0,t2\}$ to $\{t1,t3\}$ in the initial PTnet, the resulting event graph will have cycle $\{t4,t5\}$ completely unmarked. Additionally, the cycle consisting of $\{t0,t1,t2,t3,t6\}$ will have two marked edges ($t0 \rightarrow t1$ and $t2 \rightarrow t3$) as the new marked place in the PTnet will set all of the related transition-to-transition arcs' occurrence index offsets to 1.

After removing the PTnet places, in order to mark the remaining unmarked event graph cycles without exhaustively exploring them, we disable all edges with positive occurrence index offsets and perform a Depth-First-Search (DFS) traversal on the resulting graph. This traversal allows us to extract the back-edges (or feedback-edges) of a graph at runtime in $O(|V|+|E|)$ time complexity. Such edges are always part of at least one cycle, thus, when located, we set their occurrence index offset (ϵ) to 1. It is worth mentioning that the DFS traversal does not locate all cycles but only one per back-edge as each edge might be associated with multiple cycles. Figure 5.3(c) shows the final event graph after performing a DFS traversal on fig 5.3(b) and marking the back-edges.

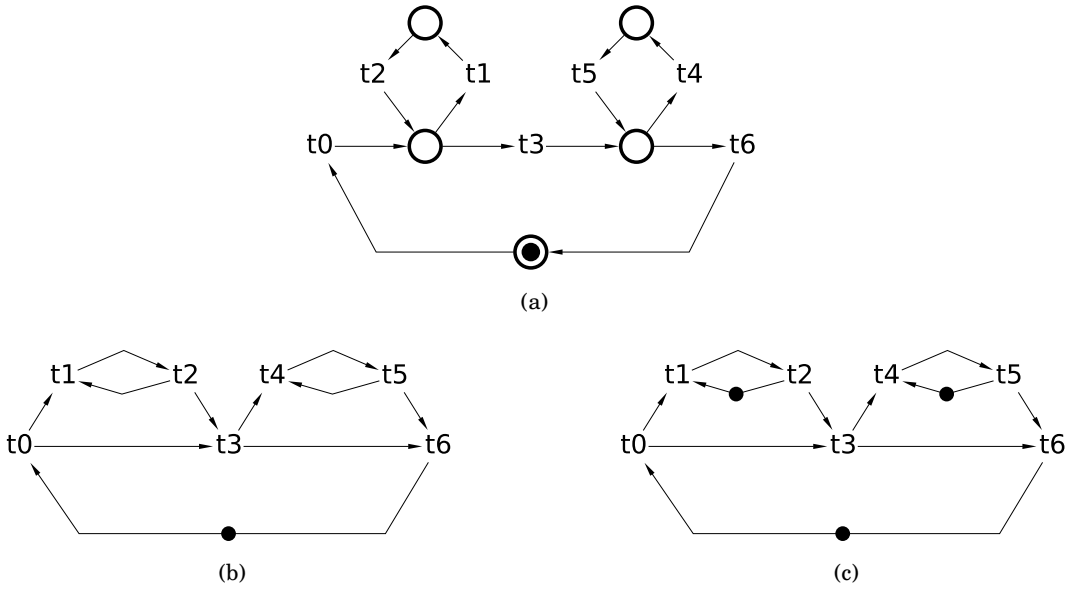


FIGURE 5.3. Complete Petrinet-to-Eventgraph Transformation. (a) The initial PTnet. (b) The resulting event graph after the PTnet's places removal (c) The final event graph after performing a DFS traversal and marking the unmarked back-edges. The DFS is performed after the initially marked edges are temporarily removed.

5.3 Complete Petrinet to Event Graph Transformation

The complete graph transformation is presented in algorithm 2 below. The algorithm takes a PTnet as an input and outputs the equivalent event graph (or STG) that is compatible with Burns' Algorithm. Note that the term **ETG (Event Timing Graph)** is used throughout the pseudo-code. This term is used to emphasize that the resultant event graph does not describe the behaviour a system (like PTnets) but the timing relationship between its transitions. It is a common misconception to assume that a marked edge in an event graph is similar to a marked place in a PTnet. Edges in event graphs correspond to timing arcs and markings define the system's periodicity. In more detail, lines 3-6 show that STGs can be converted to event graph with a simple removal of places. Lines 6-16 show the first step of the transformation where, for a given place, each predecessor transition is connected to every successor transition with their arc inheriting the place's marking. Line 17 disables the marked arcs of the generated ETG in order to perform a DFS traversal and extract the graph's back-edges. Eventually, in lines 22-26 the back-edges are marked and the final ETG is generated.

It is clear up to this point that the initial FCPTnet marking impacts the liveness of the constructed ETG. The same stands for boundedness. There are certain FCPTnet markings that can violate the boundedness of the resultant ETG, meaning that the ETG is k-bounded with $k > 1$ while the original FCPTnet was 1-bounded. This usually occurs when the initial petrinet token is placed on a *choice* or *return-from-choice* place (or in general a place with more than

one successor or predecessor transition). In this special case, the marking of the ETG must be recomputed. To achieve this without searching for a new petrinet initial marking equivalent to the original, we remove all markings and rerun Algorithm 2. The DFS traversal will mark all of the back-edges which guarantees us a live and 1-bounded ETG.

Algorithm 2: Construct Event Timing Graph

```
1  Input: FCPTnet P = (N, M), where N = (S, T, F)
2  Output: STG N' = (T', F'), with marking M'
3  if (P is STG) then
4  |   N', M' = make_STG(P, M); // P is STG, simply remove places //
5  |   return;
6  end
7  for (each place p in S) do
8  |   for (each transition u = •p in T) do
9  |   |   for (each transition v = p• in T) do
10 |   |   |   new_edge = add_ETG_edge(u, v, F'); // Add ETG edge //
11 |   |   |   if (has_token(p)) then
12 |   |   |   |   mark_STG_edge(new_edge, M'); // Mark edge //
13 |   |   |   end
14 |   |   end
15 |   end
16 end
17 disable_marked_STG_edges(F', M'); // Disable marked ETG edges //
18
19 // get all ETG backedges //
20 backedges_set = DFS_get_backedges(F', M');
21
22 for (each edge e in backedges_set) do
23 |   if (is_marked(e) == 0) then
24 |   |   mark_STG_edge(e, M'); // Mark ETG backedge //
25 |   end
26 end
```

CHAPTER 6

COMPLETE ASYNCHRONOUS STA FLOW

In this chapter, we present a complete Asynchronous Static Timing Analysis (ASTA) methodology that can be used to perform the timing verification of asynchronous controllers. Taking as inputs the event model and the netlist that describe the controller, the proposed flow extracts the transition-to-transition (T2T) delays from the netlist and annotates them onto the event graph in order to compute the minimum period. Additionally, several extensions to Burns' period computation algorithm have been implemented, such as double precision support, missing marking insertion and critical cycle reporting. The ASTA engine has been imported in a tool called ASP and was developed with the contribution of Stavros Simoglou who implemented the equilibrium slews computation and the ETG arc delay derivation algorithms.

6.1 Asynchronous STA Flow Overview

The complete flow of the ASTA engine is shown in figure 6.1. In order to perform the timing verification of an asynchronous controller, the designer must provide the engine with, the controller netlist, the event model, the technology timing library as well as additional technology information regarding the components used. A brief description of the required files and the related file formats is presented below:

- The **netlist** is given in Verilog format (`.v`) and describes the connectivity of the electronic circuit, i.e. the connections between the components that compose the design.
- The **event model** specification is given in Petrify [7] File Format (`.g`), which can describe any type of Petri net, or in Graphviz [5] file format (`.dot`) which is a graph description language.

- The **technology timing library**, given in Liberty File Format (`.lib`), contains the timing information of the components used in the design as well as the NLDM LUTs, originally presented in Section **??**.
- **Library Exchange Format** [2] (`.lef`), is a specification for representing the physical layout of an integrated circuit in an ASCII format. It includes design rules and abstract information about the cells.

In the following sections we will present in more detail each step of the flow, starting from the equilibrium slew (or transition times) computation and proceeding with the delay-annotated Event Timing Graph (ETG) until eventually the minimum period computation.

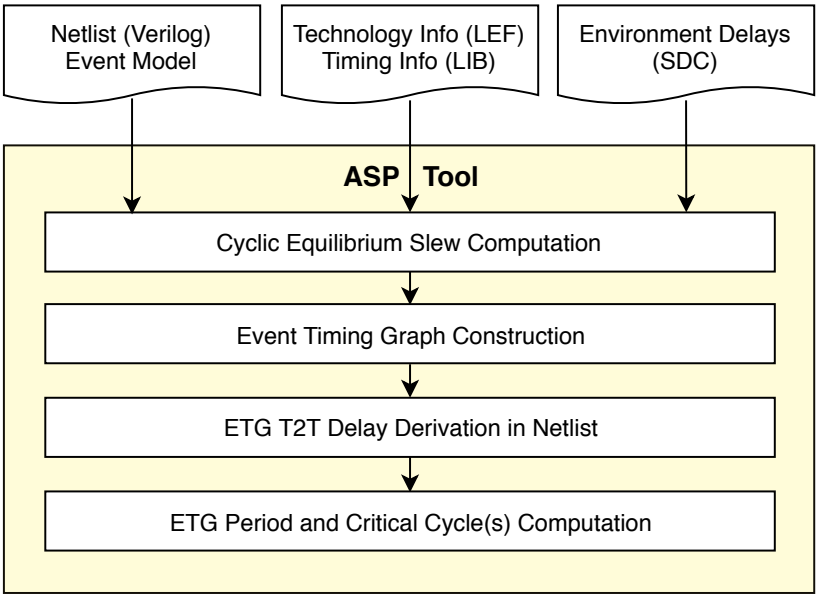


FIGURE 6.1. Asynchronous STA flow.

6.2 Computing Worst Case Slews across Cycles

The first step of the ASTA flow is the computation of cyclic equilibrium slews (or transition times) for all circuit gate pins. In the same manner as classic STA, in late-mode and early-mode analysis we compute the worst-case and best-case slews respectively. In more detail, the presence of cycles makes this process iterative which stops until all slews have reached a steady state across the entire cyclic netlist. It is worth mentioning that this flow step does not worsen the complexity of our engine as, in general, the steady state is reached after the first few iterations.

The slew propagation starts from Primary Input (PI) pins, or a random pin if there are no PIs, and continues through the timing arcs of the circuit. Note that, cycle back-edges impact the

transition times of already visited nodes. The process continues until the slew difference of each pin between subsequent propagation iterations becomes close to zero. Figure 6.2 shows a ring oscillator with the equilibrium slew values annotated. Pin `INV2/ZN`, being connected to `NAND1/A2`, indirectly impacts the slews of the rest gate pins in the same cycle.

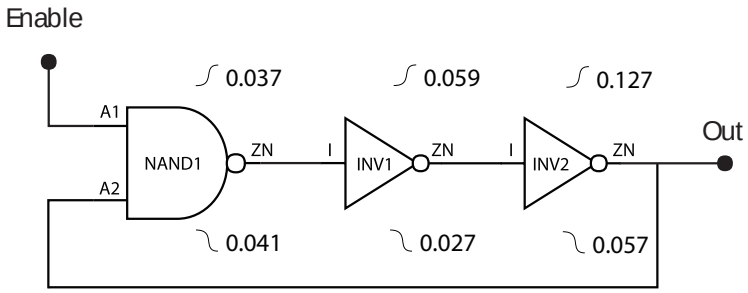


FIGURE 6.2. Three-Stage Ring Oscillator with enable and annotated Cyclic Equilibrium rise and fall Slews.

Note that, most industrial STA engines perform cycle cutting, when at least one combinational loop is encountered in the circuit. Therefore, the disabled arc will have 0 slew and will not impact other gate pin slews, due to the absence of cycles in the resulting gate pin graph. This leads to over-optimistic results which are also timing errors.

Algorithm 3 displays the aforementioned procedure, taking as input the cyclic gate pin timing graph and the slew threshold (`slew_delta`), *i.e.* the minimum allowable slew difference between subsequent iterations. The 4-tuple graph notation $G = (V, E, Load(V), Slew(V)$ is used to denote that each node (or gate pin) in V is characterised by its load (capacitance) and slew (transition time). Initially, as shown in lines 4-11, all slews are initialised to 0 and all PI gate pins are added to a First-In-First-Out (FIFO) queue. In lines 12-19 we extract the first available pin from the queue and compute its new slew value before eventually comparing it with the value of the previous iteration. When pin slews from subsequent computations are not identical (their difference is less than `slew_delta`), the pin's slew is updated and its successors are added into the FIFO, as the update impacts them (lines 21-27). Note that, this algorithm does not require the event model and is performed solely on the netlist.

Algorithm 3: Cyclic Equilibrium Slew Computation

Input: Gate Pin Timing Graph $G = (V, E, Load(V), Slew(V))$, Slew Threshold $slew\_delta$
Output: Slew Annotated Timing Gate Pin Graph G
$Slew(V) = 0$; // Initialise Gate Pin Pin Slews to 0 //
$Q = \emptyset$; // FIFO Queue Q //
for *(**each** Primary Input Gate Pin pi in V)* **do**
    $set\_slew(pi, pi\_slew\_constraint)$; // Set PI Slew (user-defined) //

    for *(**each** Successor Gate Pin s in pi →)* **do**
        $enqueue(Q, s)$; // Add pi Successor Gate Pins to Queue //
    end
end
while *(Q != ∅)* **do**
    $q = dequeue(Q)$; // next Gate Pin for Slew Computation //

    $slew = calculate\_slew(q, get\_slew(\rightarrow q), get\_load(q \rightarrow))$;
    // Compute q Slew, based on input Slew and output Load //
    if *(| slew - get_slew(q) | < slew_delta)* **then**
        continue; // Continue to Another Gate Pin from Queue //
    end;

    $prev\_slew = get\_slew(q)$;
    $new\_slew = max(prev\_slew, slew)$;
    $set\_slew(q, new\_slew)$; // Update Slew Value //

    for *(**each** Successor Gate Pin s in q →)* **do**
        $enqueue(Q, s)$; // Add q Successor Gate Pins to Queue //
    end
end

6.3 Event Timing Graph Construction and Delay Annotation

After computing the equilibrium slew values, we construct the Event Timing Graph (ETG) in order to annotate the signal transition-to-transition (T2T) delays. The ETG construction has been presented in Chapter 5 Algorithm 2, where all places are removed and the timing arcs between transitions are created. Recall that, an ETG is nothing more than an annotated STG with T2T delays, identical to Burns' collapsed constraint graph [1]. Therefore, if the event model is already an STG there is no conversion required.

Figure 6.3 shows the ETG construction of the `alloc-outbound` benchmark. The initially given Petri net, shown in Fig. 6.3(a), contains one *choice* and one *return-from-choice* place. The ETG in Fig. 6.3(b), after the PTnet place removal, is not live and therefore not compatible with Burns' Algorithm, due to the creation of a new cycle with no marked edges. As shown in chapter 5, in order to have a live ETG, we disable all marked arcs and perform a DFS traversal to mark the located back-edges. Fig. 6.3(c) displays the final `alloc-outbound` ETG after the back-edge marking insertion. The temporary removal of arc `req+` → `ack-` caused the DFS to start from `req+` and locate the back-edge `busctl+` → `ackctl+`. In other scenarios we might have no marked arcs to disable before the traversal. In this case, the DFS will start from the first located root or a random node when there are no roots.

With the ETG constructed and the equilibrium slews computed, the T2T delays can be computed and annotated. Similar to classic STA, their computation is based on the gates that

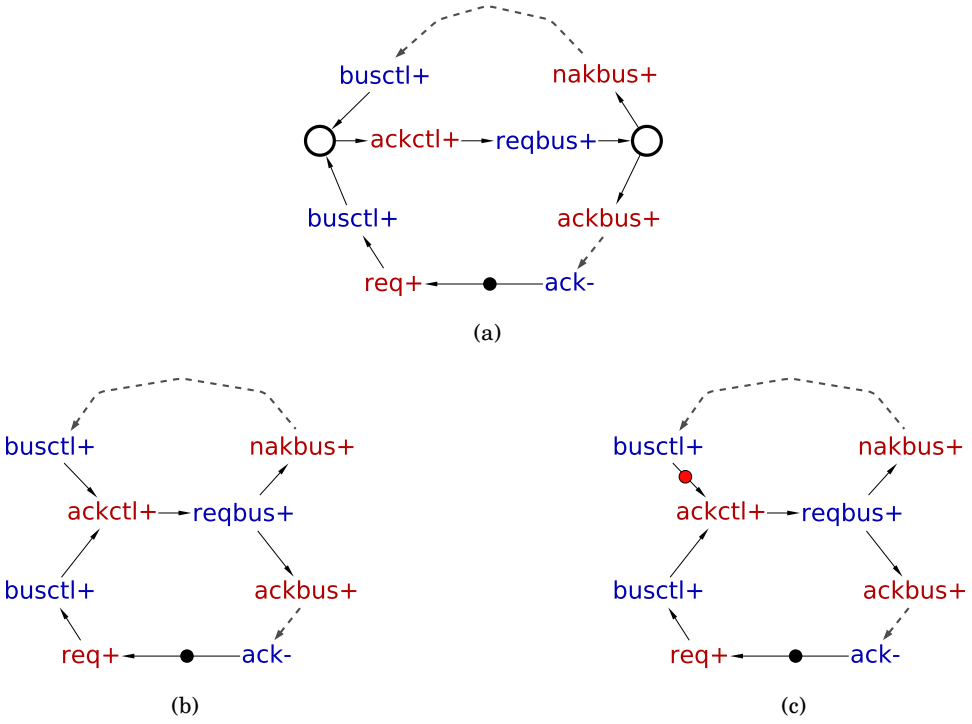


FIGURE 6.3. Conversion of a FCPTnet to a live ETG. (a) `alloc-outbound` benchmark FCPTnet. (b) The initial non-live ETG after the place removal. (c) The live ETG after marking the unmarked back-edges.

compose the transition-to-transition paths and their timing library information. The procedure will be more clear through the example shown in Figure 6.4. In Fig. 6.4(a), there are two paths starting from primary input (PI) `b` that lead to primary output (PO) `y`. The red coloured path contains one negative unate gate, thus for a rising transition of `b`, `y` will fall and for a falling transition of `b`, `y` will rise. Therefore, the delay of this path describes the delays of `b+` → `y-` and `b-` → `y+` T2T arcs. Although, in the ETG depicted in Fig. 6.4(b), only `b-` → `y+` T2T arc is present. In general ETGs might not contain all possible T2T arcs, as in reality some cannot be sensitisable. Analogously, the blue coloured path contains two negative unate gates, therefore for a rising transition of `b`, `y` will rise and *vice versa*. This path provides us the delay for `b+` → `y+` ETG T2T arc. It is worth mentioning that, there might be multiple paths matching to a specific ETG T2T arc. In this case, we annotate the worst-case or best-case delay, depending on whether we are performing max or min analysis. Another scenario that was not presented in Figure 6.4, is the handling of binate gates during the delay extraction. In paths with binate gates we always take the worst-case (for max analysis) or the best-case (for min analysis) delay regardless of side input transitions (rise or fall). This adds pessimism, as the resultant path may not be sensitisable. Note that, PO-to-PI and PI-to-PI arc delays cannot be extracted from the netlist, as there are no such paths, and are derived from the environment or, at this point, defined by the designer.

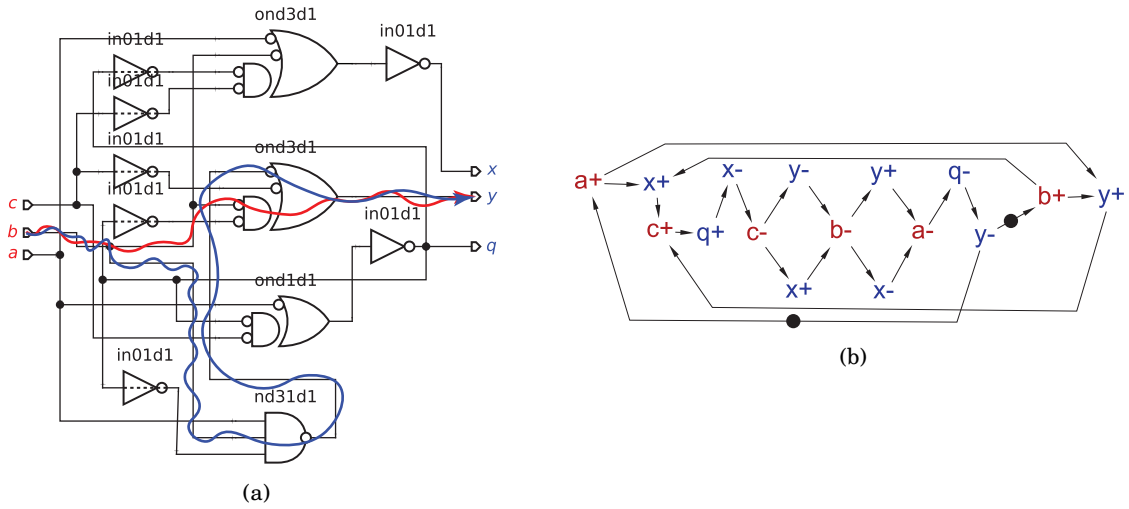


Figure 6.4: (a) The unfolded collapsed-constraint graph of the ring oscillator (b) The collapsed-constraint graph of the ring oscillator

Algorithm 4 presents the delay extraction and annotation procedure. Taking as input the Gate Pin Timing Graph $G = (V, E)$ and the Event Timing Graph (ETG) $N' = (T', F')$, with T' being the set of transitions and F' the set of T2T arcs, the algorithm performs the delay annotation after their extraction from the netlist. More specifically, for each T2T arc, in lines 5-6 we extract the source and destination transitions which are used to identify the respective netlist gate pins (lines 8-10). In line 13 we extract the acyclic paths between the extracted gate pins. Possible internal loops are ignored, as they were only useful in the equilibrium slew computation. Eventually, lines 15-19 compute the GBA STA critical path delay, based on the converged slew values. Similar to classic STA, the delay computation is performed on transitions that exist following the unateness of the cells. Note that the use of max in line 17, indicates that we are performing a worst-case analysis.

6.4 Computing Period and Reporting Critical Cycles

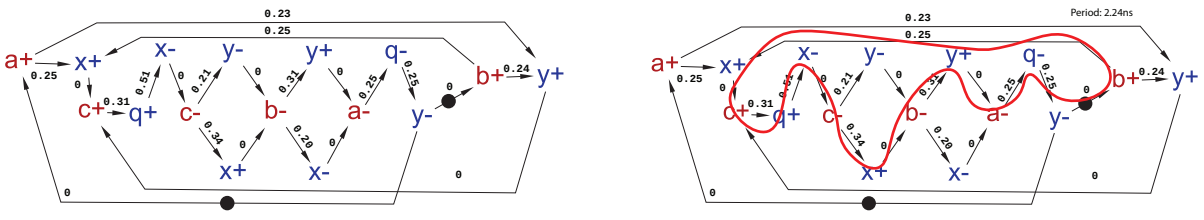
In the final step of our flow, we provide the delay-annotated ETG to Burns' Primal-Dual Method Algorithm. As stated in previous chapters, Burns' Algorithm computes the minimum allowable period of asynchronous designs described by the provided collapsed constraint graph (or in our case, ETG). The period is equal to the minimum delay/markings ratio of the graph cycles and its computation is performed by iterative relaxation, which makes the whole process polynomial in complexity. Recall that, exhaustively exploring all cycles is not feasible, as the number of cycles in an arbitrary graph can be exponential in the number of its arcs.

Algorithm 4: Annotate ETG Delays

```
1  Input: Gatepin Timing Graph G = (V,E), Event Timing Graph N' = (T',F')
2  Output: Delay(E'), for each edge (i,j) of N' ∈ F'
3  for (each edge e in F') do
4      max_delay = −∞;
5      s = source(e); // Get Source node s of Edge e //
6      d = destination(e); // Get Destination node d of Edge e //
7
8      // Gate pins of ETG transitions are unique //
9      gs = get_gate_pins(G,s); // Get Gate Pin of Source //
10     gd = get_gate_pins(G,d); // Get Gate Pin of Destination //
11
12     // Examine all Paths from gate pin gs to gate pin gd //
13     paths = get_acyclic_paths(G,gs,gd);
14
15     for (each path p in paths) do
16         path_delay = compute_path_delay(p);
17         max_delay = max(max_delay,path_delay);
18     end
19     set_delay(e,max_delay);
20 end
```

Our Burns' Algorithm implementation includes multiple improvements and extensions. The first step of the algorithm has been extended to insert possible missing markings, if the graph after the marked-edge removal is still cyclic. This process is done through a DFS traversal and is similar to lines 17-26 of Algorithm 2 shown in Chapter 5. Additionally, the algorithm has been tuned to support floating precision numbers, due to the annotated ETG delays. Another extension allows the designer to view critical arc cycles after Burns' Algorithm execution. This is achieved by performing a DFS traversal on the critical-arc graph where, with the help of a stack, the paths that form cycles are reported whenever a back-edge is found. It is worth mentioning that the DFS traversal does locate all cycles as multiple cycles might have common back-edges.

Figure 6.5 displays the `nowick` benchmark ETG with the T2T delays (fig. 6.5(a)) and critical cycle annotated (fig. 6.5(b)). Note that, all output-to-input and input-to-input arcs have 0 delay, as such paths do not exist in the netlist (Fig. 6.4(a)). The computed period is equal to $2.24ns$ and can be verified by dividing the sum of the critical cycle delays by the number of critical cycle markings (in our case we have only one marked arc: `y-` → `b+`



(a) (b)

FIGURE 6.5. `nowick` Benchmark ETG with Delays and Critical Cycle annotated.

6.5 Experimental Results

Our ASTA flow has been tested using a set of 23 asynchronous control circuit benchmarks, which have been synthesised and technology mapped by Petrify Tool [7]. The timing library used is a 0.25 μm standard-cell library (`.lib`). Our engine is also capable of performing classic STA and we have validated that it produces the same results as other industrial STA tools. The tests have been run with no RC wire delays. However, this can be possible by using technology-library wireload tables or by computing the lumped or distributed RC Elmore wire delays [4].

Table 6.1 shows the results of all 23 benchmarks. Recall that, classic STA engines cut combinational circuit loops which leads to missing timing information and over-optimism in the results. Additionally, the fact that classic STA is not applicable on cyclic circuits, makes the signal slews over-optimistic for every asynchronous circuit by default, as there is no equilibrium slew computation performed. In the fist three columns of table 6.1 we present the benchmarks, as well as the number of cells and gate pins that constitute them. The next six columns consist of our ASTA engine results and the next six present the Industrial STA Tool results. In the first two columns of our results, we show the delay of the critical cycles (*i.e.* the benchmarks' period) and the number of ETG nodes (*i.e.* transitions) that form the first critical cycle extracted (as multiple might exist). In the fourth and fifth columns of our results, we display the mean delay and slew values of the critical cycle arcs, and in the last two, the delay and slew deviation across all timing arcs through their 3σ values. On the other hand, the first two columns of the industrial STA engine results, contain the critical path delays after the cycle cutting, as well as the number of arcs cut. Analogously to our tool's results, the rest of the columns show and delay and slew mean values of the critical path arcs, as well as the derivation of all timing arcs ($3\sigma values$). Note that, there is no correlation between the critical cycles of our tool and the critical paths of the industrial STA engine. However we display the values to reflect on their difference and emphasise on how over-optimistic classic STA engines are on cyclic circuits. The last column illustrates the percentage difference between the critical cycle delays or our tool and the critical path delays of the industrial STA engine. The average difference is approximately 50% meaning that the industrial tool is 50% more optimistic.

One considerable benchmark is `chu150`, as its netlist is acyclic and there are no cut arcs by the industrial tool. However, as seen in the benchmark's ETG shown in figure 6.4(b), the critical cycle contains multiple Output-to-Input arcs, *i.e.* arcs that do not correspond to any netlist path (*e.g.* `x-` $\rightarrow$ `y-`). In other words the cycles of `chu150` are between circuit signals and the environment. Another benchmark worth mentioning is `mr0`, as it poses the smallest delay percentage difference between ASTA and industrial STA (16.69%). Here the industrial tool happens to cut `mr0` arcs, which present minimum slew impact. Additionally, the cut arcs are not critical, thus the benchmark's period is not much affected.

Table 6.1: ASTA Experimental Results and Comparison to Industrial STA Tool

Design	**Cells**	**Gate Pins**	**ASP Tool** (this work)						**Industrial STA Tool**						%Δ **Delay**
			Critical Cycle		**Critical Cycle Arcs**		**Delay** 3σ (ns)	**Slew** 3σ (ns)	**Critical Path Delay** (ns)	**Cut Arcs**	**Delay Mean** (ns)	**Slew Mean** (ns)	**Delay** 3σ (ns)	**Slew** 3σ (ns)	
			Delay (ns)	**ETG Nodes**	**Delay Mean** (ns)	**Slew Mean** (ns)									
dff	2	14	**1.845**	16	0.115	0.091	0.392	0.229	**0.477**	2	0.238	0.130	0.148	0.179	**74.14%**
ring	3	11	**0.557**	6	0.278	0.092	0.048	0.148	**0.247**	1	0.082	0.058	0.219	0.179	**55.55%**
ebergen	5	22	**2.931**	12	0.244	0.122	0.542	0.400	**1.258**	2	0.251	0.197	0.582	0.403	**57.07%**
c3dec2	9	34	**1.865**	6	0.310	0.060	0.750	0.127	**1.060**	4	0.132	0.141	0.185	0.223	**43.14%**
half	9	32	**1.784**	5	0.356	0.087	0.812	0.089	**1.050**	5	0.150	0.130	0.343	0.264	**41.16%**
chu150	10	35	**2.196**	10	0.219	0.057	0.652	0.098	**0.874**	0	0.145	0.086	0.156	0.059	**60.17%**
vbe5b	10	37	**2.707**	8	0.338	0.054	1.013	0.119	**1.275**	1	0.141	0.095	0.193	0.118	**52.88%**
nowick	11	40	**2.239**	12	0.186	0.058	0.534	0.178	**0.818**	1	0.163	0.107	0.198	0.181	**63.44%**
vbe5c	11	37	**1.455**	9	0.161	0.048	0.373	0.079	**0.984**	2	0.123	0.101	0.166	0.202	**32.39%**
ccc_p	13	45	**2.381**	15	0.158	0.112	0.371	0.243	**1.249**	3	0.156	0.135	0.225	0.293	**47.51%**
sbuf-read-ctl	13	46	**2.364**	13	0.181	0.044	0.535	0.108	**1.071**	2	0.153	0.132	0.387	0.326	**54.70%**
seq4	14	46	**3.332**	16	0.208	0.017	1.023	0.051	**1.086**	2	0.155	0.103	0.195	0.180	**67.41%**
mp-forward-pkt	15	50	**1.767**	10	0.176	0.041	0.494	0.141	**0.791**	1	0.131	0.086	0.228	0.203	**55.22%**
alloc-outbound	15	52	**1.781**	14	0.127	0.037	0.388	0.113	**0.755**	2	0.151	0.095	0.117	0.148	**57.62%**
chu133	15	48	**2.315**	11	0.210	0.089	0.426	0.274	**1.342**	3	0.134	0.110	0.205	0.283	**42.03%**
converta	16	55	**3.586**	12	0.298	0.135	0.703	0.347	**1.494**	6	0.213	0.191	0.287	0.421	**58.34%**
nak-pa	18	60	**2.076**	14	0.148	0.033	0.550	0.081	**0.965**	1	0.193	0.147	0.554	0.416	**53.52%**
sbuf-ram-write	20	69	**3.577**	16	0.223	0.088	0.647	0.395	**1.831**	2	0.166	0.113	0.250	0.219	**48.80%**
ram-read-sbuf	23	75	**3.140**	16	0.196	0.059	0.686	0.288	**1.419**	1	0.177	0.138	0.465	0.377	**54.81%**
trimos-send	27	90	**5.695**	8	0.711	0.231	0.628	0.427	**3.360**	14	0.186	0.139	0.313	0.259	**41.00%**
mmu	29	91	**5.415**	8	0.676	0.061	2.819	0.187	**3.361**	11	0.168	0.130	0.334	0.269	**37.93%**
mr1	34	105	**6.665**	11	0.605	0.072	2.843	0.218	**3.058**	13	0.218	0.189	0.499	0.449	**54.11%**
mr0	56	178	**6.545**	17	0.385	0.118	0.866	0.262	**5.453**	21	0.194	0.141	0.321	0.261	**16.69%**
AVERAGE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**50.85%**

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

It is clear that the necessity of an Asynchronous Static Timing Analysis (ASTA) engine is crucial to the application and future of asynchronous systems. The proposed methodology, performs ASTA of asynchronous controllers in polynomial time complexity by utilising both the event model (PTnet, STG *e.t.c*) as well as the Verilog netlist. Our flow is fully automated, as long as all necessary files are provided, and computes the equilibrium slews across all cycles before proceeding to the extraction and annotation of the delays onto the constructed Event Timing Graph (ETG). The final step of our flow includes the design's period computation, which is done through Burns' Polynomial Primal-Dual Algorithm, and the critical cycles' extraction which is optional. Additionally, we have presented a way to convert multiple event models, such as Free-Choice Petrinets (FCPTnets), into an ETG that is compatible with Burns' Algorithm.

As future work we consider the complete handling of Bundled-Data circuits where the design is separated into Control (asynchronous) and Datapath (synchronous) sections. Furthermore, the application of Burns' algorithm can be extended by formulating more problems into the collapsed-constraint graph notation. An application worth considering is the STA of Latch designs, which can be easily performed through Burns' algorithm as the signal arrival and departure times of latches can be viewed as events.

BIBLIOGRAPHY

[1] S. M. BURNS, *Performance analysis and optimization of asynchronous circuits*, (1991).

[2] CADENCE DESIGN SYSTEMS, INC., *LEF/DEF Language Reference*, 2009.

[3] F. COMMONER, *Deadlocks in Petri Nets*, Applied Data Research Inc., Tech. Rep., (1972).

[4] W. C. ELMORE, *The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers*, Journal of Applied Physics, 19 (1948), pp. 55–63.

[5] EMDEN R. GANSNER AND STEPHEN C. NORTH, *An open graph visualization system and its applications to software engineering*, SOFTWARE - PRACTICE AND EXPERIENCE, 30 (2000), pp. 1203–1233.

[6] J. BHASKER, RAKESH CHADHA, *Static Timing Analysis for Nanometer Designs - A Practical Approach*, Springer, 2009.

[7] J. CORTDELL, M. KISHINEVSKY, A. KONDRATYEV, L. LAVAGNO AND A. YAKOVLEV, *Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers*, tech. rep., Universitat Polit'ecnica de Catalunya, 1996.

[8] J. DESEL, JAVIER ESPARZA, *Free Choice Petri Nets*, Cambridge Tracts in Theoretical Computer Science 40, 1996.

[9] JAMES L. PETERSON, *Petri nets*, ACM Computing Surveys, (1977).

[10] ——, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

[11] J. MAGGOT, *Performance Evaluation of Concurrent Systems using Petri nets*, "Information Processing Letters ", 18 (1984), pp. 7–13.

[12] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial optimization: Algorithms and complexity*, Prentice-Hall, Inc., Englewood Cliffs, NJ, (1982).

[13] C. V. RAMAMOORTHY AND G. S. HO, *Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets*, IEEE Trans. Softw. Eng., 6 (1980), pp. 440–449.