Copyright © 1992, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A UNIFIED SIGNAL TRANSITION GRAPH MODEL FOR ASYNCHRONOUS CONTROL CIRCUIT SYNTHESIS

by

Alexandre Yakovlev, Luciano Lavagno, and
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/78

20 July 1992

COVER PAGE

A UNIFIED SIGNAL TRANSITION GRAPH MODEL FOR ASYNCHRONOUS CONTROL CIRCUIT SYNTHESIS

by

Alexandre Yakovlev, Luciano Lavagno, and
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/78

20 July 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

This work is supported by the University of Colorado subcontract grants BS0016421 and BS0036038.

A UNIFIED SIGNAL TRANSITION GRAPH MODEL FOR ASYNCHRONOUS CONTROL CIRCUIT SYNTHESIS

by

Alexandre Yakovlev, Luciano Lavagno, and
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/78

20 July 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

This work is supported by the University of Colorado subcontract grants BS0016421 and BS0036038.

A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis

Alexandre Yakovlev
Dept. of Computing Science
The University
Newcastle upon Tyne, NE1 7RU
United Kingdom

Luciano Lavagno
Dept. of Electrical Engineering
University of California
Berkeley, CA 94720

Alberto Sangiovanni-Vincentelli
Dept. of Electrical Engineering
University of California
Berkeley, CA 94720

July 20, 1992

Abstract

Characterization of the behavior of an asynchronous system depending on the delay of components and wires is a major task facing designers. Some of these delays are outside the designer's control, and in practice may have to be assumed unbounded. The existing literature offers a number of analysis and specification models, but lacks a unified framework to verify directly if the circuit specification *admits* a correct implementation under these hypotheses.

Our aim is to fill exactly this gap, offering both low-level (analysis-oriented) and high-level (specification-oriented) models for asynchronous circuits and the environment where they operate, together with strong *equivalence* results between the properties at the two levels. One interesting side result is the precise characterization of classical static and dynamic *hazards* in terms of our model. Consequently the designer can check the specification and directly decide if the behavior of *any* implementation will depend, e.g., on the delays of the signals described by such specification.

We also outline a design methodology based on our models, pointing out how they can be used to select appropriate high and low-level models depending on the desired characteristics of the system.

1 Introduction

Formal methods and CAD support for synthesis of asynchronous control circuits have become an important issue in VLSI design, as designers are tackling the most difficult problems of system-level design, such as inter-component interfacing, where asynchronous circuits are inevitable.

The asynchronous circuit designer must face two major problems in his work:

- Specify in a clear and unambiguous way the desired behavior of the system.
- Implement that behavior correctly. The asynchronous circuit behavior depends heavily on the *delay* of the components and the interconnecting wires. Only some of these delays are under the designer's control, and can be used (often with a non-trivial effort) to achieve a correct implementation of the specified behavior. Some delays depend on the *environment*, and/or some signals must travel on long busses, and no reliable assumption can be made on those delays.

The existing literature describes models to solve both these problems *separately*. Namely a number of *high-level specification* techniques for control-oriented asynchronous circuits have recently become available (see, for example, [4, 15, 16], [25, 11, 12]). Among them Signal Transition Graphs (STGs) based on Petri nets as an underlying formalism, have captured wide attention, due to a simple yet powerful mechanism to describe explicitly the major aspects of asynchronous control circuit behavior, such as concurrency, causality and conflict ([20] and [5]). Furthermore, all these models (unlike

older ones, as Flow Tables [22]) allow to specify the system in its interaction with the environment, which is also crucial for control, reactive hardware.

On the other hand, a number of *analysis* models (see [3] for a thorough review) allow the designer to verify, for example, if the circuit will or will not have *hazards* during its operation, or if, depending on the relative magnitude of the delay of two components, it may "hang" forever in an invalid state. In classical, informal terms, a circuit that operates correctly independently of the delays of each component is called *speed-independent*, while one that operates correctly independent of the delays of each interconnecting wire is called *delay-insensitive*.

These circuit models, though, are defined only for circuits built out of components whose output signal behavior can be characterized as a *Boolean function* of a set of input signals. Such class of circuits excludes some very useful components, for example fair arbiters, that cannot be described by such interconnections of Boolean functions. Moreover it does not allow for *behavioral abstraction*, by modeling some component using *non-determinism* rather than explicitly describing its operation in detail. For example, it is much easier to describe a CPU interacting with a bus interface as a device that can non-deterministically read or write, rather than deterministically describe its instruction memory, program counter, etc. Modeling the CPU as alternating between read and write cycles may not be acceptable either, since the interaction between successive, pipelined cycles can be non-trivial.

Moreover there is no known general methodology to decide whether a given STG specification *admits* an implementation that is, for example, hazard-free, or speed-independent, or delay-insensitive. And there is no satisfactory characterization of the above properties if the delays are *pure* (i.e. a translation in time of the input waveform) rather than *inertial* (i.e. short "pulses" are not transmitted). The only effort in this direction, to the best of our knowledge, is the so-called Change Diagram representation, that was shown in [25] to be formally equivalent to hazard-free circuits under the unbounded inertial gate delay model. Change Diagrams, however, are not general enough, in that they can represent concurrency and causality, but not conflict, i.e. they can model only *deterministic* behavior, and as such the description, for example of a bus protocol with different read and write phases is awkward and imprecise, as we informally argued above.

Furthermore the classical definition of a "valid" Signal Transition Graph specification is unnecessarily restrictive, as [28] showed by presenting some useful, correctly implementable behaviors that cannot be described using the constrained STGs used by Chu in [5]. For example Chu required the Petri net underlying the STG to be *safe, live and free-choice*, in order to ease the STG analysis/synthesis task. This requirement is not part of the STG definition *per se*, and has nothing to do with a deeper characterization of the STG behavior as, say, speed-independent or delay-insensitive.

In this paper we approach the problems mentioned above in the most general way, in the following steps.

- Give a general, low-level model of the *structure* and *behavior* of an asynchronous structure (where with the term "asynchronous structure", or sometimes "asynchronous system", we mean an interconnection of basic components that may be more complex than standard logic gates). This model, called Asynchronous Control Structure (ACS), allows multi-output components, non-determinism, etc. The *structure* of the ACS is a labeled, directed graph, while its *behavior* is described by a state-transition-like representation, that describes the events that can occur in every state, and the corresponding next state of the system (Arc-Labeled Transition System, ALTS). We need a structural model because fundamental aspects of asynchronous design, such as delays, are associated with the structural components of the system.

- Describe how a *special* case of ACS, where each component has one output and is described by a Boolean function, corresponds to the classical model of an asynchronous circuit. The corresponding ALTS behavior specification now is determined by those Boolean functions changing the values of the circuit outputs in response to input and output signal transitions.

- Relate the local and global properties of the ALTS of a circuit with known low-level properties of the circuit, such as hazards, speed-independent operation, etc., both under inertial delays and pure delays. In order to establish formally this correspondence, we will have to introduce some auxiliary formalisms that capture the "history" of the circuit, beside its "current state", and show how this "history" relates to significant properties of the state-based ALTS description

- Give a general *high-level* model of the *behavior* of an asynchronous system (the associated structure will be described using the same graph-like representation as in the low-level model). This model, the Signal Transition Graph, will not have unnecessary restrictions superimposed, to allow us to prove the correspondence between low-level ALTS properties (and hence circuit properties) and high-level STG properties.

At this point the designer can use the framework to verify if a specification meets some circuit-level requirements, or, conversely, given a set of circuit-level properties, what class of specifications needs to be used.

ACS	Asynchronous Control Structure	Structural model of asynchronous systems
BACS	Binary Asynchronous Control Structure	Binary version of ACS
TS	Transition System	Uninterpreted state-transition based behavioral model
ALTS	Arc-Labeled Transition System	TS with transitions interpreted as signal value changes
STD	State-Transition Diagram	ALTS with binary-labeled states
CD	Cumulative Diagram	Cumulative history of transitions in the system
ALC	Asynchronous Logic Circuit	Structural/behavioral model of asynchronous circuits
PD	Pure Delay	All input changes are transmitted to the output
ID	Inertial Delay	Pulses shorter than the delay magnitude are not transmitted
PN	Petri net	Uninterpreted event-based behavioral model
STG	Signal Transition Graph	PN with transitions interpreted as signal value changes

Table 1: Principal abbreviations used in the paper

Note that the paper is not concerned with the details of how each component will be implemented in a specific technology. The main concern is to analyze properties that are common to every implementation of the specified behavior, using a model that is general enough to abstract various different implementation techniques, but detailed enough to have practical relevance. Such component implementation issues are dealt with elsewhere (see, for example, [8], [13], [1]).

The paper is organized as follows. Section 2 defines the low-level structural and behavioral model of asynchronous systems, called Asynchronous Control Structure and Arc-Labeled Transition System, together with the related trace and partial order models. Section 3 describes Asynchronous Logic Circuits, a special cases of Asynchronous Control Structures, and relates properties of the two, underlining the effects of the inertial/pure delay model dichotomy. Section 4 defines Signal Transition Graphs as interpreted Petri nets and describes the problem of their implementation in Asynchronous Logic Circuits. Section 5 presents a classification of Signal Transition Graphs according to the corresponding Asynchronous Logic Circuit properties. Section 6 compares the Change Diagram model proposed in [25] with the STG model. Section 7 outlines a design methodology based on our models. Section 8 concludes the paper.

To help the reader remember the numerous abbreviations used throughout the paper, we have collected them in Table 1, together with a brief summary of their meaning.

2 A Low-level Structural and Behavioral Model for Asynchronous Systems

This section introduces a low-level, *state-transition-based*, model of asynchronous systems. It has two *components*: a *structural* component called Asynchronous Control Structure (ACS) and an associated *behavioral* component to describe its evolution in time, the Arc-Labeled Transition System (ALTS).

The combination of the two (ACS and ALTS) is somewhat similar[1] to a network of interacting asynchronous Finite State Machines (the structure, describing who communicates with whom) together with a State Table describing the behavior of the entire system, where each state is the product of the states of each machine, and transitions correspond to allowed change of values on the interconnecting signals.

The properties of the model are characterized using the concept of Cumulative Diagram, that records the history of changes of each signal in the Asynchronous Control Structure. We then give an example of the power of our model using an asynchronous fair arbiter, that would be impossible to describe using "standard", Boolean-function based, models of asynchronous circuits.

2.1 Asynchronous Control Structures

The notion of Asynchronous Control Structure (ACS) is a generalization of the "interconnection structure" of an asynchronous control circuit. It removes the usual structural limitation (used, e.g. by [17] or [22]) that each component has exactly one output signal. Thus an ACS structure can represent an arbitrary interconnection of modules, with the only restriction that no two modules can drive a single signal[2]. The *behavior* of this interconnection of modules will be described using an Arc-Labeled Transition System, as shown in Section 2.2.

[1]This analogy should not be taken literally, and is only given to help the reader understand the general idea of the approach.

[2]I.e. no wired-or or wired-and constructs are allowed, but note that at this level of abstraction they can still be modeled using discrete gates.

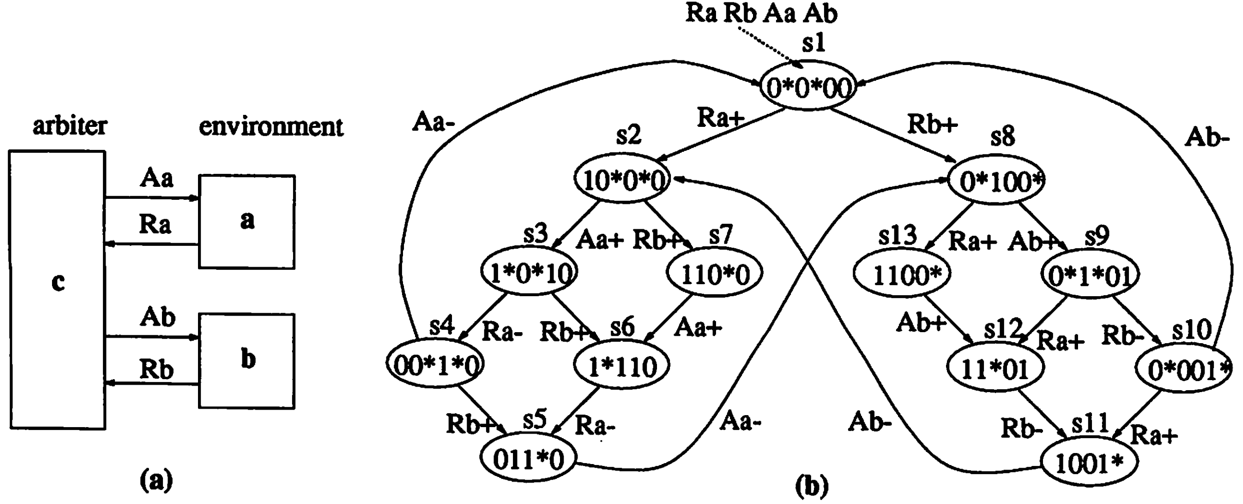

Figure 1: A Binary Asynchronous Control Structure and its State Transition Diagram

Formally, an Asynchronous (Discrete) Control Structure (ACS) is a directed graph $\langle V, H, Y, \rho \rangle$, where V is a finite set of *nodes*, associated with the abstract discrete components of the ACS, $H, H \subseteq V \times V$ is a finite set of *arcs*, standing for the *interconnections* between the components, $Y = \{y_1, ..., y_n\}$ is a finite set of finite-state *variables*, or *signals*, and $\rho : H \rightarrow Y$ is a *labelling* (total) function, associating every arc with a variable. Any two arcs labelled with the same variable must have the same source node (i.e. they represent a branching interconnection), so formally $\forall (v_1, v_2), (v_1', v_2') \in H$ we must have $\rho(v_1, v_2) = \rho(v_1', v_2') \Rightarrow v_1 = v_1'$.

We also denote the sets of input and output *interconnections* for a component v as: $I^H(v) = \{(v', v) \in H\}$, and $O^H(v) = \{(v, v') \in H\}$, respectively. The sets of input and output *signals*, or simply inputs and outputs, for a component are denoted: $I^Y(v) = \{y : \rho^{-1}(y) \in I^H\}$, and $O^Y(v) = \{y : \rho^{-1}(y) \in O^H\}$, respectively.

A simple example of an ACS is described in Figure 1.(a). Here $V = \{a, b, c\}$, $H = \{(c, a), (a, c), (c, b), (b, c)\}$, $Y = \{R_a, R_b, A_a, A_b\}$, and $\rho(c, a) = A_a$, $\rho(a, c) = R_a$, $\rho(c, b) = A_b$, $\rho(b, c) = R_b$. Furthermore $I^H(c) = \{(a, c), (b, c)\}$, $I^Y(c) = \{R_a, R_b\}$, and so on.

For every variable $y \in Y$, $S(y) = \{y^0, y^1, ..., y^k\}$ is called the set of *variable values*, or states.

An ACS $\langle V, H, Y, \rho \rangle$ is called a *Binary Asynchronous Control Structure* (BACS) if $\forall y : S(y) = \{0, 1\}$. Hence, for a BACS, the set of allowed changes can be denoted as $Y \times \{+, -\}$, where "+" stands for a signal change from 0 to 1, and "-" for a signal change from 1 to 0. The *behavior* of a BACS is defined by a *binary transition system*, called state transition diagram, which is introduced in the following section.

2.2 Transition Systems and State Transition Diagrams

This section describes how the interconnected components of an ACS *behave* in time, that is how the variables associated with them change, using some key concepts from [10].

A Transition System (TS) is a pair $\langle S, E \rangle$, where S is a set of *states*, and $E, E \subseteq S \times S$, is a set of *transitions*. Note that we do not restrict S and E to be finite. The directed graph representation of a TS is as usual: states are vertices and transitions are arcs. For example, in Figure 1.(b) $S = \{s_1, \ldots, s_{13}\}$ and $E = \{(s_1, s_2), (s_1, s_8) \ldots\}$. We denote $(s_1, s_2) \in E$ by $s_1 E s_2$.

An Arc-Labelled Transition System (ALTS) is a quadruple $\langle S, E, A, \delta \rangle$, where $\langle S, E \rangle$ is a TS, A is a finite *alphabet of actions* and $\delta : E \rightarrow A$ is a (total) labelling function, which assigns each transition a single *action name* in A. Each action name represents a *change of value* of a variable in the associated ACS, and each (possibly infinite) path along the graph represents a valid sequence of such changes in time. Thus the ALTS describes the complete allowed behavior of the associated ACS. For example, in Figure 1 we have $A = \{R_a^+, R_a^-, A_a^+, A_a^-, R_b^+, R_b^-, A_b^+, A_b^-\}$, where we use A_a^+ to denote the change of signal A_a from 0 to 1, and A_a^- to denote the change from 1 to 0. Furthermore $\delta(s_1, s_2) = R_a^+$, $\delta(s_2, s_7) = R_b^+$, and so on.

For a BACS with a set of variables Y ($|Y| = n$) we define a Binary (encoded) Transition System, or State Transition Diagram (STD), $\langle S, E, \lambda \rangle$, where $\langle S, E \rangle$ is a TS and $\lambda : S \rightarrow \{0, 1\}^n$ is a (total) labelling function such that each state is encoded with a *binary vector* consisting of the values of Boolean variables. The i-th component of the vector associated with

each state s is denoted as $\lambda(s)_i$, but for simplicity, unless it creates confusion, we generally use the simpler notation s_i. An STD is called *contradictory* if λ is not injective. Hence for a non-contradictory STD we can identify the state with its binary label.

For every STD arc, connecting a pair of states s and s', we allow s and s' to differ in one and only one component, say the i-th. This component variable, y_i, is called *excited* in state s and its value s_i is marked with a "*" in s. Since there can be several outgoing arcs from each state, a number of variables can be excited in it. The variables that are not excited in a state are called *stable* in it. We assume that transitions between the states can have *arbitrary but finite delays*, and that these delays are associated with the delays of the components in the modeled BACS (similar to the *gate delay model* in asynchronous circuits, Section 3.2). We call an STD *initialized* if it has an explicit initial state. For example, in Figure 1.(b) $Y = \{R_a, A_a, R_b, A_b\}$, and $\lambda(s_1) = 0000$, $\lambda(s_5) = 1000$ and so on. Furthermore, R_a and R_b are excited and A_a and A_b are stable in s_1.

Note that every STD can be also interpreted as an Arc-Labelled Transition System, with the following labelling (consistent, since exactly one variable changes in every arc of an STD):

$$\forall e = (s, s') \in E : \delta(e) = \begin{cases} y_i^+ & \text{if } s_i = 0 \text{ and } s'_i = 1 \\ y_i^- & \text{if } s_i = 1 \text{ and } s'_i = 0 \end{cases}$$

The following important property of any STD comes directly from its definition:

Property 2.1 *No state in an* STD *can have two outgoing transitions labelled with the same variable but with different signs.*

We can now examine more in detail the meaning of Figure 1. It represents the interconnection of an *arbiter* and two other components (the arbiter's *environment*), that independently of each other may request access to a single resource, with signals, R_a and R_b. The arbiter grants access with A_a and A_b (which are mutually exclusive).

Note that this BACS/STD pair specifies a *fair* behavior, because if the arbiter receives a request at one input, say R_a, while it is processing a previous request from R_b, then it must, after finishing the transaction for R_b, respond to R_a before it can react to a new request from R_b again. Our abstract arbiter is capable of distinguishing the order in which the two, possibly concurrent, requests arrive at its inputs, by going to two different states (s_7 and s_{13}), labelled with the same vector 1100 (hence the STD is contradictory).

2.2.1 .Reachability and Unique Action Relations

Intuitively, a state s_2 of a TS $\langle S, E\rangle$ is reachable from a state s_1 if there exists a directed path from s_1 to s_2. More formally, the *direct reachability* relation is simply given by the set E. For any pair of states $s, s' \in S$, the state s' is called reachable from s if there is a finite length (including zero length) sequence of transitions leading from s to s'. Therefore reachability is given by reflexive and transitive closure of E, i.e. E^*. In the example of Figure 1.(b) all states are mutually reachable.

Similarly for any ALTS $\langle S, E, A, \delta\rangle$ we can define the *reachability through a sequence of actions*. Specifically, for direct reachability through action $a, a \in A$, we have $sE(a)s'$ if sEs' and $\delta(s, s') = a$. For example in Figure 1.(b) we have $s_1E(R_a^+)s_2$. For general reachability through a sequence of actions, $sE(\alpha)s'$ would imply that there is a finite sequence of action names $\alpha \in A^*, \alpha = a_1, a_2, ..., a_m$ such that $sE(a_1)s^1, s^1E(a_2)s^2, ..., s^{m-1}E(a_m)s'$. We can sometimes use the notion of an allowed sequence from a state, i.e. α is *allowed from* s if $\exists s'$ such that $sE(\alpha)s'$. So R_a^+, A_a^+, R_b^+ is allowed in s_1, but it is not allowed in s_2 in Figure 1.(b)

Note also that among the various arcs labelled with the same action in Figure 1.(b), some of them actually represent exactly the same "event". For example, arcs (s_2, s_3) and (s_7, s_6) both represent the same event, the arbiter acknowledging request R_a^+ from the environment. Now we make this intuitive idea more formal, because it will become important when we relate *state-based* models, such as the State Transition Diagram, with *event-based* models, such as the Signal Transition Graph, where the notion of *unique occurrence of an event* is explicit.

For an ALTS $\langle S, E, A, \delta\rangle$, we define a pairwise relation $\sim^1$ on the set E of arcs as $(s_1, s'_1) \sim^1 (s_2, s'_2)$ if $\delta(s_1, s'_1) = \delta(s_2, s'_2)$ and $s'_1 \neq s_2$ and s_1Es_2 (i.e. there exists an arc $(s_1, s_2) \in E$). Let $\sim$ be the equivalence relation formed by the reflexive, symmetric and transitive closure of $\sim^1$. We call $\sim$ the *unique action relation*. We can easily see that Unique-Action Relation partitions the set E into a set of Unique-Action Relation-*classes*, $[E]^A$. Each such class, $[e]^a$, is called an *action*. The set of actions with the same name, a, is called the *action set* of the name a and denoted as $[E]^a$. This notion will be useful later, when we shall associate the transitions of an STG with the transitions of the corresponding STD.

For example, in Figure 1.(b) we have $(s_2, s_7) \sim^1 (s_3, s_6)$, and $(s_3, s_6) \sim^1 (s_4, s_5)$, hence, by transitivity, $(s_2, s_7) \sim (s_4, s_5)$. Also $(s_1, s_8) \sim^1 (s_2, s_7)$. Then $[e]^{R_b^+} = \{(s_1, s_8), (s_2, s_7), (s_3, s_6), (s_4, s_5)\}$. In this very simple case, each action set has a single element, $[E]^{R_b^+} = \{[e]^{R_b^+}\}$ and so on.

For an action $[e]^a$, the set, always forming a connected subgraph, of states which are the sources for the transition arcs in $[e]^a$ is called *excitation region* for action $[e]^a$. So in Figure 1.(b) the excitation region of $[e]^{R_b^+}$ is $\{s_1, s_2, s_3, s_4\}$, and they correspond to the states where the label bit for R_b has value 0 and is tagged with "*".

2.2.2 Interleaving Semantics of Concurrent Actions

Throughout this paper we assume that the actions associated with a set of arcs outgoing from the same state can be performed in the modeled system *concurrently*, i.e. independently of each other. See for example R_a^+ and R_b^+ in s_1 in Figure 1.(b), which are "produced" by different and independent components. Since our model is *entirely asynchronous*, we must assume that the changes of corresponding variables can occur *in time in any order*.

Our low-level behavioral model, on the other hand, requires that *a single variable changes for every transition*. We then choose to model such concurrency by *interleaving*, i.e. considering all possible alternative chain orderings compliant with the partial order between possibly concurrent actions (in Figure 1.(b) this corresponds to paths s_1, s_2, s_7 and s_1, s_8, s_{13}). Such a modeling is convenient yet sometimes problematic, because it hides the semantic distinction between true concurrency and "shuffled" alternative selections. This distinction can be made explicit only in models with explicit causality notions, and we postpone it until Section 4, where we will consider Signal Transition Graphs.

2.2.3 Properties of Transition Systems and State Transition Diagrams

In this section we analyze a set of behavioral properties of Transition Systems and State Transition Diagrams that we will show later been connected with corresponding, important properties of asynchronous systems. For example, the property of *confluence* below is closely connected to the requirement that the "long term behavior" of the system must not be influenced by the relative magnitude of the delay of two components. No matter who "wins the race", we must still be able to reach the same state in the future. Similarly, *local confluence* will be shown to be related to the classical concept of *static hazards* in a circuit.

Following [10], we call an ALTS $\langle S, E, A, \delta \rangle$:

- *confluent*, if $\forall s, s1, s2 \in S$, if sE^*s1[3] and sE^*s2, then $\exists s3 \in S$ such that $s1E^*s3$ and $s2E^*s3$.
- *locally confluent*, if $\forall s, s1, s2 \in S$, if $sEs1$[4] and $sEs2$, where $s1 \neq s2$, then $\exists s3 \in S$ such that $s1Es3$ and $s2Es3$. If such $s3$ is unique, then the ALTS is called *uniquely locally confluent*.

So, Figure 1.(b) is confluent (all pairs of states can reach any state), but *not locally confluent*, due to s_1, s_2 and s_8 (s_1Es_2, s_1Es_8, but there is no common immediate successor of s_2 and s_8).

Keller, in [10], gave three *sufficient* conditions for local confluence (and hence confluence) of an ALTS. An ALTS is:

- *deterministic*, if $\forall s, s1, s2 \in S$ and $\forall a \in A$, if $sE(a)s1$ and $sE(a)s2$, then $s1 = s2$ (i.e. for each action there can be only one outgoing transition from a state that is labeled with it).
- *commutative*, if $\forall s \in S$ and $\forall a, b \in A$, if ab and ba are allowed in s, then $\exists s'$ such that $sE(ab)s'$ and $sE(ba)s'$ (i.e. if the effect of interleaving two transitions both allowed in a state and not mutually exclusive is the same).
- *persistent*, if $\forall s \in S$ and $\forall a, b \in A, a \neq b$, if a and b are allowed in s, then ab is allowed in s (i.e. if no transition can disable another one).

It was proven in [10] that if an ALTS satisfy all these conditions together, then it is both Locally Confluent and Confluent. The definition of STD implies that if an STD satisfies these conditions, then it is uniquely locally confluent.

The ALTS in Figure 1.(b) is deterministic and persistent, but not commutative (due to s_1, R_a^+ and R_b^+ again). So, being confluent, it shows that Keller's conditions are only *sufficient*.

Now, even though our state-based model has no "direct" idea of *causality* between actions, we can still locally verify if some action has "a unique set of predecessors", that can somehow be identified with its causes. Hence we define the property of *strict causality* of an ALTS, which, as the Unique Action Relation, will become more clear when we introduce our event-based model, where such causality is explicit.

---

[3] I.e. s_1 is reachable from s.

[4] I.e. there is an arc $(s, s_1) \in E$.

Let $\mathcal{S} = \langle S, E, A, \delta \rangle$ be an ALTS and let $[E]^A$ be its set of actions. Let $S([e]^a)$ denote the excitation region for action $[e]^a \in [E]^A$. Let $\pi(s_1, s_2)$ be a directed path of states between s_1 and s_2.

An ALTS is called *strictly causal for action* $[e]^a$ *and state* $s \in S$ if:

- $\forall s_1, s_2 \in S([e]^a), s_1 \neq s_2$, such that $\exists \pi(s, s_1), \pi(s, s_2)$, with $\pi(s, s_1) \cap S([e]^a) = \emptyset$ and $\pi(s, s_2) \cap S([e]^a) = \emptyset$ (i.e. s_1 is the first state in $\pi(s, s_1)$ where $[e]^a$ is excited, and similarly for s_2),
  - $\exists s_3 \in S([e]^a)$ (possibly coincident with s_1 or s_2) such that:
    - $\pi(s, s_3) \cap S([e]^a) = \emptyset$ and
    - $\exists \pi(s3, s1)$ such that $\pi(s3, s1) \subseteq S([e]^a)$.

  I.e. s_1 and s_2 have a common "ancestor", through states where $[e]^a$ is also excited, which is a successor of s and where $[e]^a$ is also excited for the first time.

An ALTS is called *strictly causal* if it is strictly causal for all actions $[E]^A$ in and all states in S.

This definition means, informally, that each excitation region of each action has a single "top" state (or a "cycle" of such states, as in the example below), where it becomes excited for the first time, and all other states in the region (which is connected by definition) are successors of it through paths *within the region.* So actions leading into this "top" state (or cycle) can be informally identified with its causes.

On the other hand, if the ALTS is not strictly causal, it means that some action has "many alternative ways" of becoming allowed.

The ALTS in Figure 1.(b) is strictly causal, because, for example, for action $[e]^{R_a^+}$ the states in its excitation region $\{s_1, s_2, s_9, s_{10}\}$ form a cycle. So, for example, from state s_4 we can reach both s_1 and s_8 (through s_5), and in this case the third state in the definition coincides with s_1, whence s_8 is reachable without leaving the excitation region. Similarly for all other triples of states and actions.

Finally, when analyzing the behavior of an ALTS we are interested in checking if we have a point where future behaviors diverge completely. Such behavior is, in general, not desirable, and hence the *liveness* of an ALTS is important to check. We define it only for *finite* ALTS. For a *finite* ALTS, with the reachability relation E^* between states, we define the mutual reachability relation between any two states, $s, s' \in S$ if both sE^*s' and $s'E^*s$ hold for them. This is an equivalence relation, so it gives rise to a set of equivalence classes. Built for a given initial state, these classes form a partial order induced by the reachability relation. The *maximal* classes in this partial order are called *final classes* (i.e. once we enter one of these classes, we can never leave it).

A *finite* ALTS is *live* if it forms a *single equivalence class* for any initial state. Such a TS is represented by *a strongly connected graph.* In a live ALTS, for every state $s \in S$ and every action name $a \in A$, there exists a state $s' \in S$, reachable from s, in which a is allowed. The ALTS in Figure 1.(b) is obviously live.

2.3 Trace Models

For an ACS defined by an ALTS we can define another representation, called Trace Structure, or Trace Model (see [23])[5], of its behavior. This representation will be needed in Section 5.3, because delay-insensitive circuits were defined in the literature using Trace Models, so in order to define delay-insensitivity within our framework we must relate Trace Models with Arc-Labelled Transition Systems.

A Trace Model representation of the behavior (described by an Arc-Labeled Transition System) of a structure (described by a Binary Asynchronous Control Structure) is a pair $\langle A, \Sigma \rangle$, where $\Sigma \subseteq A^*$ is a *prefix-closed* set of *traces*, or strings of actions. This model is defined with respect to a given initial state, and represents the execution *history* of the ALTS. Each trace then stands for a (possibly infinite) sequence of actions that can be performed on the variables of the ACS. The set of traces in Σ contains those traces that are allowed by the behavioral specification.

For a BACS with an associated initialized STD and a set of variables Y, we can also think about a Binary Trace Model, which is a pair $\langle Y, \Sigma \rangle$, where $\Sigma \subseteq (Y \times \{+, -\})^*$ is a prefix-closed set of traces, or strings of signal changes as allowed by the STD.

In the example in Figure 1, initialized in state s_1, we have $A = \{R_a^+, R_a^-, A_a^+, A_a^-, R_b^+, R_b^-, A_b^+, A_b^-\}$, and $\Sigma = \{\epsilon, R_a^+, R_b^+, R_a^+R_b^+, R_b^+R_a^+, R_a^+A_a^+, R_b^+A_b^+, R_a^+A_a^+R_b^+, \ldots\}$ (here ϵ stands for the empty trace).

[5] We are forced to introduce this term here, as a synonym of the more common "Trace Structure", only to avoid confusion with the abbreviation of the term "Transition System" (TS).

The following property of the Binary Trace Model generated from an STD is the result of Property 2.1 and of the definition of STD.

Property 2.2 *In a Binary Trace Model $\langle Y, \Sigma \rangle$, for every trace in Σ and any variable $y \in Y$, all the occurrences of y have alternating signs, i.e. between any two consecutive changes of the same sign there is at least one opposite change.*

2.4 Cumulative Diagrams

In order to characterize classes of behaviors of asynchronous systems, we need the concept of *history of the execution* of a state-based specification. The complete history of the system is represented by a set of traces, where each trace records *exactly* the order of occurrences of actions. The state of the Arc-Labeled Transition System, on the other hand, describes only the *final result* of such execution. In this section, following [17], we will describe a model to describe this history, where only the *number of occurrences* of each action is recorded, called a *Cumulative Diagram* (CD). Hence this representation will be of intermediate "precision" between a Trace Model and an ALTS.

The Trace Model description of the operation of an initialized ALTS contains all the traces of the ALTS starting from its initial state. The mechanism of trace generation induces a natural mapping between traces and sets of states, where each trace maps to the set of states where the ALTS may be at the end of its generation. Note this mapping is *functional* if the ALTS is *deterministic*. In this case, the state in which the ALTS arrives for a given trace with respect to an initial state is uniquely determined through the reachability relation.

On the other hand, for *any* ALTS (not just deterministic ones) we can think about another mapping from the set of traces. This mapping defines, for every trace $\alpha \in \Sigma$, a *multiset* of action names μ, with the multiplicity of each name $a \in A$, $\mu(a)$, equal to the number of occurrences of a in α. A multiset obtained in this way is called a *cumulative state*. It is convenient to represent a cumulative state by a *vector* of natural numbers with dimension $|A|$.

We can easily see from the above definition that a cumulative state defines a class of equivalence between traces of the ALTS which are simple permutations of each other (note that not every permutation may be a valid trace). Let $[\alpha]$ be such equivalence class for trace α. Every trace $\beta \in [\alpha]$ brings the ALTS to the same cumulative state μ. Therefore we can identify this μ with $[\alpha]$. Now it should be clear that for every deterministic, commutative and persistent ALTS, where all the traces in $[\alpha]$ bring the ALTS in the same state, there exists a *functional mapping* between cumulative states and ALTS states.

The set of cumulative states generated by an ALTS through its Trace Model model (denoted by $[\Sigma]$) is a partial order. This order is a subset of the natural integer vector ordering and is built upon the *prefix order* between traces up to permutations. Formally, $[\alpha] \sqsubseteq [\beta]$ if $\forall \alpha \in [\alpha] \exists \beta \in [\beta]$ such that α is a prefix of β.

The partial order of cumulative states, built as above, is called the Cumulative Diagram (CD) (more precisely, we will use its Hasse diagram, where the reflexive and transitive edges have been removed).

For example, Figure 2 contains an initial fragment of the CD for the ALTS described in Figure 1. Note that the CD model *cannot describe the local divergence* after traces $R_a + R_b+$ and $R_b + R_a+$. In fact both traces lead to the same cumulative state 1100, that corresponds to states s_7 and s_{13} in Figure 1.(b). This also illustrates that the mapping between CD states and STD states in general is a relation, not a function.

The above definitions are easily adapted to the case of a BACS and its STD. By Property 2.1, we can change the notion of cumulative state and build the CD for a set of variable names Y rather than their changes $Y \times \{+, -\}$. This modified version of CD is isomorphic to the original version because *all the changes of the same variable are linearly ordered* (see also Property 2.2).

Now we have all the necessary information to define the notion of speed-independent and semi-modular behavior, which are crucial (as we will see in Section 3.6) for more practical purposes, such as the analysis and synthesis of asynchronous control circuits.

2.5 Speed-independence and Semi-modularity

The intuitive notion of speed-independent behavior can be more formally described using *confluence*, that ensures that the "long term" behavior of the modeled asynchronous system does not depend on the winner of a race between concurrent transitions. In this section we give a set of *alternative* definitions of a set of ALTS properties. These properties will be shown to correspond to:

- interesting circuit properties, such as the absence of hazards in Section 3, and
- high-level specification properties in Section 5.

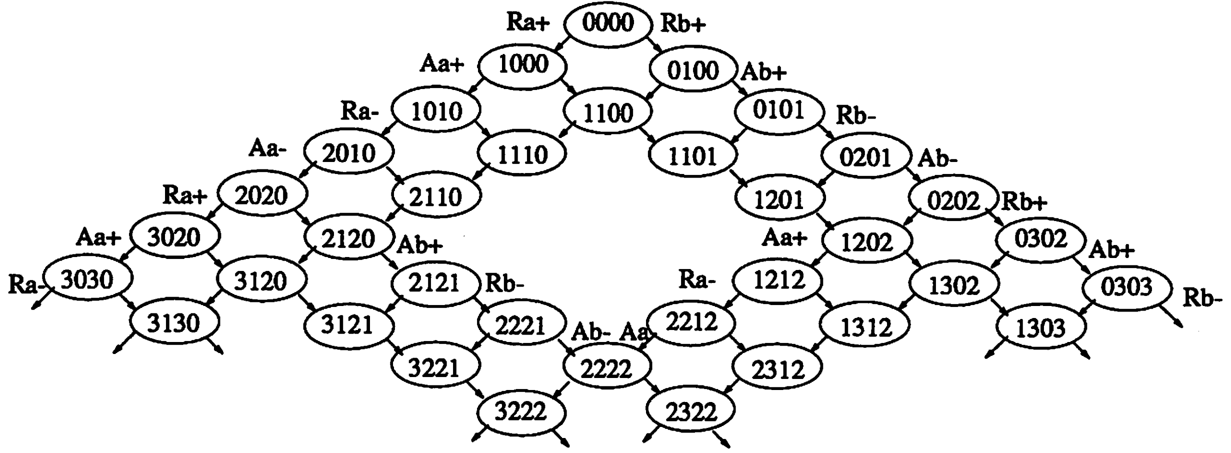

Figure 2: The Cumulative Diagram of Figure 1

So this section provides the desired *bridge* between the two domains.

Let us first recall some definitions from lattice theory. Let C be a partial order. An element $z \in C$ is a *zero* element if for all $c \in C$ we have $z \sqsubseteq c$. An element $c \in C$ is a *greatest lower bound* (g.l.b.) of two elements $c_1, c_2 \in C$, denoted $c = c_1 \sqcap c_2$, if $c \leq c_1, c \leq c_2$ and $c' \leq c_1 \wedge c' \leq c_2 \Rightarrow c' \leq c$. Similarly we can define the least upper bound (l.u.b.) of a pair of elements, denoted $c = c_1 \sqcup c_2$ by replacing $\leq$ with $\geq$. A *lattice* is a partial order where every pair of elements has a g.l.b. and a l.u.b.. A lattice is *distributive* if the g.l.b. and l.u.b. operations are mutually distributive. An element c_1 of a partial order C *covers* another element c_2 of C if $c_2 \leq c_1$ and there is no c_3 such that $c_2 \leq c_3 \leq c_1$. A lattice is *semi-modular* if for every pair of elements c_1 and c_2 that cover a third element c_3 (then obviously $c_3 = c_1 \sqcap c_2$), they are both covered by $c_1 \sqcup c_2$.

- An ALTS (STD) is called *Speed-independent*-1 if it is confluent.
- A *finite* ALTS (STD) is called *Speed-independent*-2 with respect to a state if the CD generated for this state is a *lattice with a zero element*, according to the partial order defined in Section 2.4. The ALTS (STD) is called Speed-independent-2 if it is Speed-independent-2 with respect to every state in S.
- A *finite* ALTS (STD) is called *Speed-independent*-3 ([19]) with respect to a state s if it has a single final equivalence class when initialized in s (Section 2.2.3). The ALTS (STD) is called Speed-independent-3 if it is Speed-independent with respect to every state in S.

For example, the ALTS in Figure 1.(b) satisfies Speed-independent-3, because it is finite and it has a single final equivalence class for every initial state. We can easily see that the TS is confluent, and hence Speed-independent-1. Furthermore its CD, represented in Figure 2, is a lattice (zero is cumulative state 0000), so the STD is Speed-independent-2.

It can be shown that, despite our intuition, the definitions above are not strictly equivalent for a given finite ALTS (STD). Speed-independent-1 is equivalent to Speed-independent-3 in the finite case, while Speed-independent-1 and Speed-independent-2 are equivalent if (but not only if) the given finite ALTS (STD) is *deterministic, commutative and persistent*:

Proposition 2.1

- *A finite* ALTS *(*STD*) is Speed-independent-2 if it is Speed-independent-1 and persistent.*
- *An* ALTS *(*STD*) is Speed-independent-1 if it is Speed-independent-2, deterministic and commutative.*

The example in Figures 1 and 2 shows that our conditions for the equivalence between Speed-independent-1 and Speed-independent-2 are only *sufficient* and *not necessary*, because this ALTS is both Speed-independent-1 and Speed-independent-2 but *not commutative*.

Semi-modularity, that we will relate to hazard-freeness, is a stronger property than speed-independence. Again, two alternative definitions can be formulated.

- An ALTS (STD) is called *Semi-modular*-1 if it is *locally confluent*.

- A *finite* ALTS (STD) is called *Semi-modular-2* with respect to a given state if the CD generated for this state is a *semi-modular lattice with a zero element*, according to the partial order defined in Section 2.4. The ALTS (STD) is Semi-modular-2 if it is Semi-modular-2 for every state in S.

A Proposition analogous to Proposition 2.1 can also be shown to hold about Semi-modular-1 and Semi-modular-2.

The ALTS in Figure 1.(b) is not locally confluent, hence not Semi-modular-1. It is not Semi-modular-2 either, because the corresponding CD in Figure 2 is not semi-modular, due for example to cumulative states 1110 and 1101, that both cover 1100, but are not covered by their least upper bound 2222 (recall that covering means being immediately above in the partial order).

A last class of ALTSs, significant because of some interesting results on sufficient conditions for its synthesis with realistic logic gates ([8]), is connected with the definition of *strict causality* (or, informally, of a "unique set of actions causing an action") described in Section 2.2.3.

- An ALTS (STD) is called *Distributive-1* if it is *strictly causal and locally confluent*.
- A finite ALTS (STD) is called *Distributive-2* with respect to a given state if the CD generated for this state is a *distributive lattice with a zero element*, according to the partial order defined in Section 2.4. The ALTS (STD) is Distributive-2 if it is Distributive-2 for every state in S.

A Proposition analogous to Proposition 2.1 can also be shown to hold about Distributive-1 and Distributive-2.

It should be obvious that the following inclusion holds for the classes of ALTSs (STDs): Distributive $\subset$ Semi-modular $\subset$ Speed-independent.

3 Modeling Asynchronous Logic Circuits

In this section we will show how "real" asynchronous circuits, built out of gates and wires, fit as a special case of our Binary Asynchronous Control Structures and State Transition Diagrams.

We will use two different delay models, pure and inertial, to describe the behavior of the circuit, and characterize circuit properties such as hazards in terms of ALTS properties such as local confluence.

3.1 A Low-level Model for Asynchronous Logic Circuits

Here, as in Section 2.1, we describe a circuit as the conjunction of a *structure* (a graph) and a *behavior* (a set of Boolean functions and delays).

An Asynchronous Logic Circuit (ALC, [19], [17]) is a triple $\langle X, Z, F \rangle$, where X is a set of *input signals* ($|X| = m$), Z is a set of *output signals* ($|Z| = n$), $F = \{f_1, f_2, ..., f_n\}$ is a set of Boolean functions, the *circuit element functions*, such that for each $i \in \{1 \dots n\}$, $f_i : \{0,1\}^{d_i} \rightarrow \{0,1\}$, where d_i is the number of inputs of element z_i. We denote by $Y = X \cup Z$, the set of signals of the ALC. The structure of an ALC can be represented by a directed graph with one node for each variable, and an arc connecting the node corresponding to each input of f_i with the node corresponding to y_i[6].

Structurally, an ALC is a special case of a BACS. The difference is that every structural component of the ALC is uniquely associated with a single variable, thus implying that each component, v_i, has *only one output*, $\{y_i\} = O^Y(v_i)$. The value of this output can be characterized either by the value of the corresponding Boolean function f_i (if y_i is an output signal) or by the value of the signal itself (if y_i is an input signal).

An ALC is *initialized* if its initial state is defined, as a binary vector $s0 \in \{0,1\}^{n+m}$. An ALC is *autonomous* if $X = \emptyset$.

Figure 3.(a) describes a very simple autonomous ALC, where $Z = \{z_1, z_2, z_3\}$ and $f_1 = \overline{z_3}$, $f_2 = \overline{z_3}$, $f_3 = z_1 + z_2$.

3.2 Taxonomy of Models for Asynchronous Logic Circuits

An initialized ALC produces a dynamic behavior, resulting from the transitions of both input and output signals. The input signals are changed by the environment and the output signals are changed by the ALC. The output values are determined by two factors. The first factor is the *evaluation* of the Boolean function associated with the element. The second factor is the inherent *switching delay* of the physical logic gate, which must be taken into account by the model.

[6] We associate a node with each input of an ALC to provide the way of modeling the input wires of the circuit as components with potential delay properties.

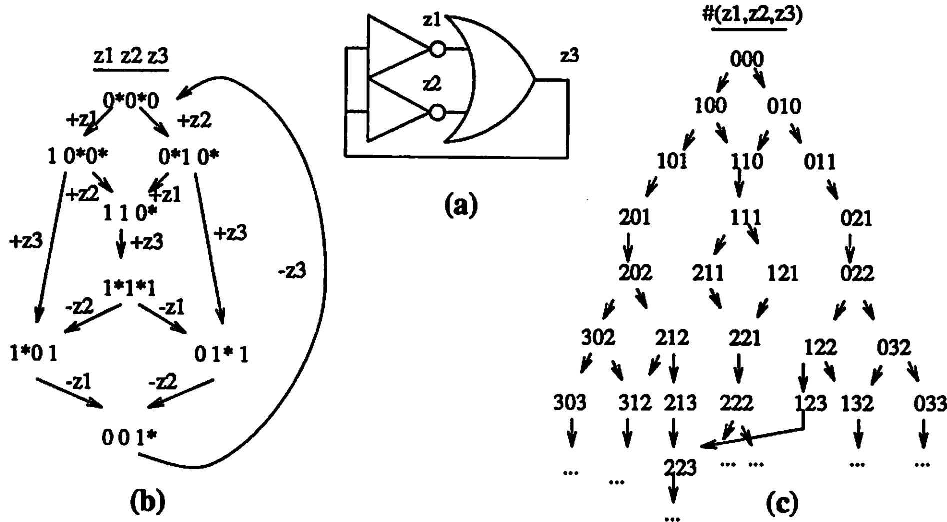

Figure 3: A circuit, its State Transition Diagram and its Cumulative Diagram with Inertial Delays

Therefore the dynamic behavior of the ALC is modeled through a number of abstractions, which can be classified as follows:

1. Delay model of an element (delay model "in small", see Section 3.3):
   - *pure delay* model,
   - *inertial delay* model.
2. Delay model of the circuit (delay model "in large"):
   - *feedback delay* model ([9]), assuming that there are delays only in the feedback wires.
   - *gate delay* model, ([19]), assuming that only the logic elements have finite delays.
   - *gate and wire delay* model, ([3] and [4]) assuming that both gates and wires have finite delays.
3. Environment behavior model (input change constraints):
   - *fundamental mode*, assuming that inputs can change their values only after the circuit has reached a stable state, where none of its variables is excited.
   - *input-output mode*, allowing the environment to change the input values in some states, not necessarily stable, in accordance with some protocol of interaction between the ALC and the environment. The fact that input values *may* change in some states, autonomously or in consequence of some output change, is explicitly indicated in the model (*reactive behavior*).
4. Circuit switching semantics ("race" model):
   - *general multiple winner*, assuming that any subset of the set of unstable signals may win the race due to a concurrent switching process.
   - *extended multiple winner*, where all concurrently changing signals go through a third, *undefined*, state before reaching their final value.

Brzozowski and Seger ([3]) showed that General Multiple Winner and Extended Multiple Winner yield *equivalent results*, but the latter allows *more efficient analysis* of the circuit hazards than the latter. *Ternary simulation*, can be used to analyze a circuit with the Extended Multiple Winner model, but *only in a rather limited case*, the Fundamental Mode operation. The ternary simulation of the dynamic behavior of an ALC in I/O mode (or of an autonomous ALC)

cannot give meaningful results for such effects like hazards or speed-independence[7]. Due to these reasons *our analysis assumes the General Multiple Winner race model and I/O mode.*

3.3 Delay Models of an Asynchronous Logic Circuit element

Let an ALC $\langle X, Z, F\rangle$ be initialized in some state s. Every element z_i of the circuit is modeled as a sequential composition of a delay-free logic function evaluator and a delay block. For every element z_i we call it (and the corresponding output variable) *stable* in s if its current value s_i is equal to the value of its function f_i. Otherwise we call it *excited.* We assume that if a variable is excited, then this variable *may* change its state after some finite time interval, which we call the *element delay.* For example, in the circuit in Figure 3.(a), initialized in state 000, z_3 is stable while z_1 and z_2 are excited. What happens next, whenever z_1 or z_2 changes value, depends on whether we use the *pure* or *inertial* delay model.

3.3.1 Inertial Delay Model

In the Inertial Delay model (ID), an excited variable *may* change its state after a finite delay. This means that for any excited variable z_i there are two possibilities. One is that its value, 0 or 1, changes to the opposite, i.e. to 1 or 0, after a finite but unbounded amount of time. The other possibility is that the value of its function f_i is changed before z_i manages to change, so that the previous value appears at the input of the delay block. In this case the output z_i of the element ceases to be excited and retains its previous value, which becomes stable (hence the term "inertial"). Speaking in more quantitative terms, the ID model means that if an element has a switching delay of d time units, pulses generated by the logic evaluator with duration less than d are filtered out, while pulses longer than d units appear at the output z_i shifted in time by d units.

Since we are dealing with completely asynchronous circuits, we cannot precisely say whether in the second situation above the element has or has not produced a short pulse at its output. We shall therefore regard this behavior (when f_i changes before z_i) as anomalous or *hazardous.*

3.3.2 Pure Delay Model

We can alternatively assume that the delay block of each element is not inertial when it becomes excited, i.e. it cannot filter out the pulses whose duration is less than a given value d (this behavior is close to reality for long wire delays). Therefore, even though the function value changes before the output z_i has changed, the element remains excited, and just shifts in time the complete sequence of its "expected" output transitions. With this Pure Delay model (PD), the value of the element in state s of the ALC, must be modeled by a pair, (r_i^s, τ_i^s), where the first component is the current binary value of z_i, i.e. $r_i^s \in \{0,1\}$, while the second component is the excitation number (recording how many excitations have been registered by the functional evaluator since the element was last stable), $\tau_i^s \in \{0,1,2,...\}$. In this model, the state of the ALC is a vector of length $|Y|$, with each component being of the above form.

We can now define an element z_i to be *stable*, according to the PD model, in state s if $\tau_i^s = 0$. Otherwise it is *excited.* The normal operation of the element is described by the following sequence of transitions: $(r_i^s, 0) \rightarrow (r_i^s, 1) \rightarrow (\overline{r_i^s}, 0)$. The hazardous operation, on the other hand, is described by the following sequences of transitions: either $(r_i^s, \tau_i^s) \rightarrow (r_i^s, \tau_i^s+1) \rightarrow (\overline{r_i^s}, \tau_i^s)$ (if $\tau_i^s > 0$) or $(r_i^s, \tau_i^s) \rightarrow (r_i^s, \tau_i^s+1) \rightarrow (r_i^s, \tau_i^s+2)$.

Now, with this definition of the behavior of each gate, we can describe the operation of the entire circuit for both the ID and the PD models.

3.4 Circuit Behavior Description with Inertial Delays

Let us consider, as an example, the circuit shown in Figure 3.(a). If we assign the all zero vector as the initial state of this ALC, variables z_1 and z_2 are excited in this state. As usual, we designate this fact by labelling the value of an excited variable with an asterisk (*). Therefore the initial state is marked as 0*0*0. Using the ID model of an element we can think about two possible states directly reachable from this state through the element normal switching behavior, 10*0* and 0*10*. Although variables z_1 and z_2 are excited concurrently and can switch independently, our interleaving semantics of concurrent actions

[7] Although some results on speed-independence can be obtained through such technique [3], they are meaningful only for a very restrictive modeling conditions. Namely, the circuit must operate in Fundamental Mode, and it is regarded as Speed-independent if its final equivalence class consists of a *unique stable* state. The existence of cyclic final equivalence classes cannot be detected by such ternary simulation, because each cyclically changing variable would have an *undefined* value.

requires that the first of the above two states is reached if variable z_1 changes before z_2 (and vice-versa). In both cases variable z_3 now becomes excited.

We can thus use a depth-first search procedure to generate the set of states reachable from the initial state. This set, together with the relation of direct reachability between states, can be represented by a graph, which satisfies our definition of an STD. Note that the transitions in this graph are labelled by the changes of variable values. Since the number of signals in the ALC is fixed ($|Y| = n + m$), it is obvious that the size of the STD, in terms of the number of its state labels, is bounded by 2^{n+m}.

Proposition 3.1 *The* STD *for any* ALC, *under the* ID *element model, is deterministic, commutative and non-contradictory*[8].

This Proposition follows directly from the ID model of an element and the uniqueness of the result of Boolean function evaluation for any given binary encoded state.

Therefore determinacy and commutativity are the intrinsic properties of the STD description for any ALC obtained using the ID model. This implies that *confluence and local confluence* are determined (up to sufficiency) by how the circuit satisfies the *persistency* condition.

We can now define speed-independence, semi-modularity, and so on for an ALC modeled with Inertial Delays. Let $\mathcal{C} = \langle X, Z, F \rangle$ be an ALC modeled with inertial delay and let $\mathcal{S} = \langle S, E, \lambda \rangle$ be its associated STD. According to the classification of Section 2.5 and Proposition 3.1:

1. $\mathcal{C}$ is *Speed-independent* if the STD $\mathcal{S}$ is *confluent.*
2. $\mathcal{C}$ is *Output-persistent* if it is Speed-independent and for each pair of edges $s_1 E(y_1^*) s_2$ and $s_1 E(y_2^*) s_3$, if $y_1 \in Z$, then y_1^* is enabled in s_3 (i.e. no output signal can ever be disabled).
3. $\mathcal{C}$ is *Semi-modular* if the STD of $\mathcal{S}$ is *locally confluent.*
4. $\mathcal{C}$ is *Distributive* if the STD of $\mathcal{S}$ is *strictly causal and locally confluent.*

Analogous definitions exist for a BACS (except for Output-persistent).

Output-persistency guarantees that no transition of an output signal will ever be disabled, thus guaranteeing a correct behavior for them (recall that disabling a transition means possibly causing a spurious pulse on the signal).

Obviously ID-Distributive$\subset$ID-Semi-modular$\subset$ID-Output-persistent$\subset$ID-Speed-independent.

Note that Semi-modular as defined above is equivalent to the "operational" definition due to Muller ([19]). An ALC is called Semi-modular with respect to a given state if the STD built from this this state has no transition from a state where some z_i is excited to another state where z_i is stable but has the same value. So an ALC is Semi-modular if its STD is persistent.

Also, note that Proposition 2.1 and 3.1 immediately imply that for an ALC with the ID model Speed-independent-1 (confluence) is implied by Speed-independent-2 (lattice), and similarly for Semi-modular-1 and Semi-modular-2 and Distributive-1 and Distributive-2.

The STD and an initial fragment of the CD for the ALC example in Figure 3.(a) is shown in Figure 3.(b) and (c). The STD is live, Speed-independent but not Semi-modular (persistency is violated in states 1*01 and 01*1, where z_2 and z_1 are disabled after the transition z_3^+ from states 10*0* and 0*10*, respectively). It is confluent but not locally confluent. Furthermore it is strongly connected, thus having single final equivalence class, and it contains only non-transient cycles of states. The CD is a lattice (one can easily prove that every pair of cumulative states has its least upper bound in the CD), but not semi-modular. It has a zero element, the empty multiset (or all zero vector).

3.5 Circuit Behavior Description with Pure Delays

Throughout this section we will refer to properties of the ALC under consideration *when analyzed with the Inertial Delay modes* by prefixing them with ID. Properties without the prefix, on the other hand, refer to the ALC analyzed with the Pure Delay model.

An Asynchronous Logic Circuit can be analyzed with the Pure Delay model in a similar way as in the Inertial Delay case, by building its CD with a depth-first search procedure, from the initial state. According to the notation introduced earlier, each state, s, is labelled by a vector of $|Y| = n + m$ pairs (r_i^s, τ_i^s), where the first component is the binary value on the output of the delay block, z_i, and the second component is the *number of potential changes* that the element will generate on its own

[8] Being non-contradictory, we can identify each state of the STD with its unique binary vector code.

before it will become stable. Let us call such a vector the PD-*vector*. This graph does not satisfy the definition of an STD, given in Section2.2, because the label is not binary. Nevertheless, it satisfies the definition of an ALTS, and again, due to the unique evaluation of the Boolean functions describing the ALC, we can make use of the fact that every state in set S is uniquely labelled by the PD-vector, and show that the following Proposition holds.

Proposition 3.2 *The* ALTS *for any* ALC, *under the* PD *element model, is deterministic, persistent and non-contradictory.*

Equality between PD-vectors requires also equality of the second component, so *commutativity does not hold* in this case. However, due to non-inertiality of elements behavior, we can claim *persistency*, because every element records its excitation, and cannot be disabled by changing the value at the output of its Boolean evaluator.

The latter detail drastically changes the role of the CD that can be built from the ALTS associated with the ALC. Such a CD is no longer a description that can be meaningfully used for characterizing the confluence properties of the ALC behavior.

The definition of a *bounded circuit* becomes crucial in such characterization. The PD model of an ALC (PD-ALC) is called *k-bounded* (or simply "bounded") if for every reachable state s in the associated ALTS, $\forall z_i : \tau_i^s \leq k$. An immediate consequence of boundedness is the *finiteness* of the ALTS of a PD-ALC.

The following Proposition is the implication of the fact that a PD-modeled ALC can accumulate unbounded "switching events" in its elements if its operation is cyclic.

Proposition 3.3 *The* PD *model of an* ALC, *which is* ID-*live*[9] *and non-*ID-*persistent, is unbounded.*

This is true because in a live, non-persistent ID model of a circuit there is no bound to the number of times the circuit can reach a state s where a variable y_i becomes disabled before it has a chance to fire. So every new arrival in this state will increment the corresponding τ_i^s.

- The PD model of an ALC is called *Speed-independent* if the associated ALTS is *finite and confluent.*
- The PD model of an ALC is called *Semi-modular* if the associated ALTS is *finite and locally confluent.*
- The PD model of an ALC is called *Distributive* if the associated ALTS is *finite, strictly causal and locally confluent.*

Analogous definitions exist for a BACS.

This definition and Proposition 3.3 imply the following important result.

Theorem 3.4 *The* PD-*model of an* ALC, *which is* ID-*live, is* PD-*Speed-independent iff it is* ID-*Semi-modular*[10].

This theorem in practice claims that for the PD model of an asynchronous circuit, speed-independence amounts to semi-modularity, if one considers a cyclically operating circuit. This result provides a crucial justification for the restriction to semi-modularity, when we look for necessary and sufficient conditions for the hazard-free, speed-independent implementation of an ALC.

Note also that this equivalence result strongly favors the use of semi-modularity rather than speed-independence as the characterization of a "correct" circuit in the ID case. The ID model can be too optimistic in many practical cases, so a design made for semi-modularity (or output-persistency) will be more "robust" with respect to technology changes, different implementations of other components of the system, and so on.

Obviously PD-Distributive$\subset$PD-Semi-modular.

Our ALC example in Figure 3.(a) generates the ALTS whose initial fragment is shown in Figure 4. This ALTS is persistent but non-commutative. It is infinite and not locally confluent. The PD-ALC is unbounded, therefore it is not Speed-independent.

Our analysis of ID and PD models of ALCs has an important by-product. It gives concise and general characterization of hazards in the ALCs behavior.

[9] I.e. the STD of the same circuit, modeled with Inertial Delays, is live. That is, it forms a single equivalence class for any initial state (Section 2.2.3).
[10] I.e. the same circuit modeled with Inertial Delays is Semi-modular.

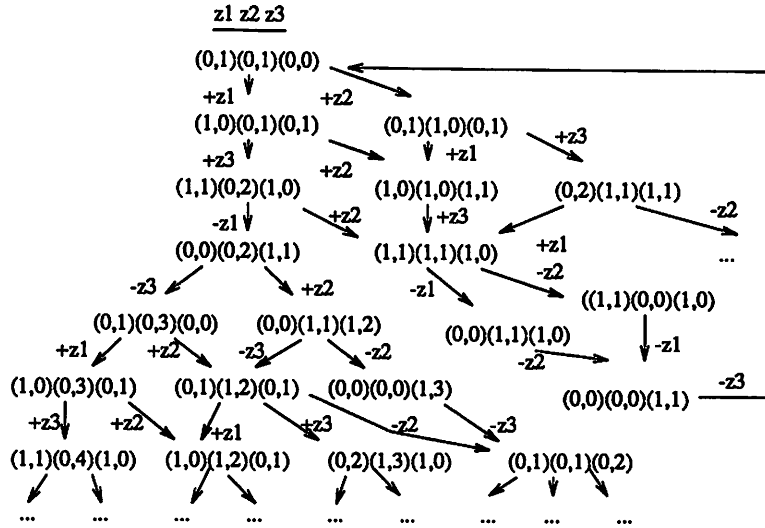

Figure 4: Cumulative Diagram for Pure Delay Model

3.6 On Static and Dynamic Hazards of Asynchronous Logic Circuits

The traditional definition of hazards ([22]) assumes two types of hazards, *static* and *dynamic*. Static hazards are again of two types, 0-1-0 and 1-0-1, and they model the behavior of an element which generates spurious pulse at its output. The dynamic hazards, 0-1-0-1 and 1-0-1-0, model the erroneous behavior of an element that should switch from 0 to 1 or 1 to 0, respectively, but during this process returns back to its previous state.

Leaving aside the question of how dangerous such hazards are in an asynchronous context, we can identify how this behavior can be analyzed at our ID and PD modeling level.

For a given ALC, the ID model, represented by the STD, can only depict *static hazards*. They are present if the ID-ALC is *non-persistent*. An element whose excitation is disabled, without switching, in state 0(1) is defined to have an 0-1-0 (1-0-1) static hazard[11]. Hence, if the ALC is ID-Semi-modular (or ID-Output-persistent), then it is free from hazards.

For a given ALC, the PD model, represented by its ALTS, can describe all hazards, characterized as follows. An element whose excitation number τ_i^s is greater than 1 has a hazard. The rank, k, of this hazard is equal to τ_i^s.

This characterization allows us to define not only "standard" static or dynamic hazards, but any dynamic hazardous behavior that can be generated by the circuit element. In fact, the "standard" static hazard corresponds to $\tau_i^s = 2$, while the "standard" dynamic hazard to $\tau_i^s = 3$.

4 A High-level Behavioral Model for Asynchronous Systems

The previous Sections discussed various inter-related models of Asynchronous Control Structures and Logic Circuits, and the relationship between model properties, such as confluence, and circuit properties, such as hazards.

In this section we develop a very general, *event-based*, model of BACS and ALC that, unlike STDs and Trace Models, has an explicit notion of *causality* and *concurrency*. So for example we will be able to distinguish the cases where events a and b are truly concurrent, independent of each other, and the case where either a can happen, and then cause b, or b can happen, and then cause a (an example of this distinction will be given in Figure 6).

The model, called Signal Transition Graph (STG), is based on interpreted Petri nets, and is a development of similar, but less general, models presented by [20] and [5].

We first recall some basic definitions from the theory of Petri nets then establish relationships between STGs and the models described in the previous Sections.

[11] Strictly speaking, this is not a hazard in the "ideal" model, because the output of the delay block does not change. Due to the physical considerations above, though, this kind of situation can actually generate a spurious pulse on the output.

4.1 Petri Nets

Petri nets are a widely used model for concurrent systems, because they have a very simple and intuitive semantics, that directly captures concepts like causality, concurrency and conflict between events.

A *Petri net* (PN) is a triple $\mathcal{P} = \langle T, P, F \rangle$. T is a non-empty finite set of transitions. P is a non-empty finite set of places $F \subseteq (T \times P) \cup (P \times T)$ is the flow relation between transitions and places[12].

A PN marking is a function $m : P \rightarrow \{0, 1, 2, \ldots\}$, where $m(p)$ is called the number of *tokens* in p under marking m. A marked PN is a quadruple $\mathcal{P} = \langle T, P, F, m_0 \rangle$, where m_0 denotes its initial marking. A transition $t \in T$ is *enabled* at a marking m if all its predecessor places are marked. An enabled transition t *may fire*, producing a new marking m' with one less token in each predecessor place and one more in each successor place (denoted by $m[t > m')$.

A sequence of transitions and intermediate markings $m[t_1 > m_1[t_2 > \ldots m'$ is called a *firing sequence from* m. The set of markings m' reachable from a marking m through a firing sequence is denoted by $[m >$. The set $[m_0 >$ is called the *reachability set* of a marked PN with initial marking m_0, and a marking $m \in [m_0 >$ is called a reachable marking. A PN marking m is *live* if for each $m' \in [m >$ for each transition t there exists a marking $m'' \in [m' >$ that enables t. A marked PN is live if its initial marking is live.

A marked PN is *k-bounded* (or simply "bounded") if there exists an integer k such that for each place p, for each reachable marking m we have $m(p) \leq k$. A marked PN is *safe* if it is 1-bounded. A transition t_1 *disables* another transition t_2 at a marking m if both t_1 and t_2 are enabled at m and t_2 is not enabled at m' where $m[t_1 > m'$. A marked PN is *persistent* if no transition can ever be disabled at any reachable marking.

A PN is a Marked Graph if every place has exactly one predecessor and one successor. A Marked Graph is persistent for every initial marking m_0, furthermore every strongly connected marked graph has at least one live and safe initial marking. A PN is *free-choice* if any two transition with a common predecessor place have only one predecessor.

A marked Petri net $\mathcal{P} = \langle T, P, F, m_0 \rangle$ generates an Arc-Labeled Transition System (ALTS) $([m_0 >, E, T, \delta)$ (Section 2.2) as follows. For each edge $(m_1, m_2) \in E$, where $m_1[t > m_2$, we have $\delta(m_1, m_2) = t$. Under this mapping, each Unique-Action Relation-class $[e]^t$, with $e = sE(t)s'$ corresponds to *a particular firing* of a transition t.

The following Proposition is an obvious consequence of the PN firing rule and of the results in [10]:

Proposition 4.1

1. *The* ALTS *corresponding to a marked* PN *is finite if and only if the* PN *is bounded.*
2. *The* ALTS *corresponding to a marked bounded live* PN *is live.*
3. *The* ALTS *corresponding to a marked* PN *is* deterministic *and* commutative.
4. *The* ALTS *corresponding to a marked persistent* PN *is* persistent, locally confluent *and* confluent.

4.1.1 Cumulative Diagram of a Petri Net

As in Section 2.4, we can define the Cumulative Diagram (CD) of a Petri net, and analyze its properties as a lattice. This will be useful in order to establish the desired correspondence between PN properties and circuit properties.

According to [26], we define the Cumulative Diagram of a marked PN as follows. Given a firing sequence $m_0[t_1 > m_1[t_2 > \ldots m$ of a marked PN $\mathcal{P} = \langle T, P, F, m_0 \rangle$, the corresponding *firing vector* is a mapping $V : T \rightarrow \{0, 1, 2, \ldots\}$ such that for each transition t, $V(t)$ is the number of occurrences of t in the sequence.

Let $\mathcal{V}$ be the set of all firing vectors of $\mathcal{P}$. We define a mapping $\mu : \mathcal{V} \rightarrow [m_0 >$ that associates each firing vector with the final marking of the corresponding firing sequence. Note that the mapping is well defined, since in any marked PN the marking reached after a sequence of transition firings from m_0 depends only on the number of occurrences of each transition in the sequence, not on the *order* of occurrence.

The set $\mathcal{V}$, called the *Cumulative Diagram* of $\mathcal{P}$, was shown in [26] to be a partial order when we define $V_1 \sqsubseteq V_2$ if:

- $V_1(t) \leq V_2(t)$ for all t and
- marking $\mu(V_2)$ is reachable from marking $\mu(V_1)$.

The following Theorem was proved in [2]:

[12] A PN can be represented as a directed bipartite graph, where the arcs represent elements of the flow relation.

Theorem 4.2 *The* ALTS *of a* PN *is confluent if the net is free-choice, bounded and live.*

The following Theorems were proved in [26]:

Theorem 4.3 *The* CD *of a marked* PN *is a semi-modular lattice with a zero element if the net is* persistent.

Theorem 4.4

1. *The* CD *of a marked* PN *is a distributive lattice with a zero element if the net is* safe and persistent.
2. *The* CD *of a Marked Graph is a distributive lattice with a zero element.*
3. *Let $\mathcal{P}$ be a* PN *whose* CD *$\mathcal{V}$ is distributive. There exists a safe and persistent* PN *$\mathcal{P}'$ whose transitions are labelled with the transitions of $\mathcal{P}$ and whose* CD *is isomorphic to $\mathcal{V}$.*
4. *Let $\mathcal{P}'$ be a safe persistent* PN, *let $\mathcal{V}$ be its* CD. *There exists a safe Marked Graph $\mathcal{P}''$ whose transitions are labelled with the transitions of $\mathcal{P}'$ and whose* CD *is isomorphic to $\mathcal{V}$.*
5. *Let $\mathcal{S}$ be a finite distributive* ALTS *with a set of labels T. There exists a safe persistent* PN *$\mathcal{P}$ and a safe Marked Graph $\mathcal{P}''$ whose transitions are labelled with those in T and whose* CDs *are isomorphic to the* CD *generated by $\mathcal{S}$.*

4.2 Signal Transition Graphs

Interpreted Petri nets, where transitions represent changes of values of circuit signals, were proposed independently as specification models for Asynchronous Logic Circuits by [20] (where they were called Signal Graphs) and [5] (where they were called Signal Transition Graphs, STGs). Both papers proposed to interpret a PN as the specification of an ALC $\mathcal{C} = \langle X, Z, F \rangle$ (where Y denotes, as usual, $(X \cup Z)$, by labelling each transition with an element of $Y \times \{+, -\}$. A label y_i^+ means that signal $y_i \in Y$ changes from 0 to 1, and y_i^- means that y_i changes from 1 to 0, while y_i^* denotes either y_i^+ or y_i^-.

An STG is a quadruple $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ where $\mathcal{P}$ is a *marked* PN, X and Z are (disjoint) sets of input and output signals respectively and $\Delta : T \rightarrow (X \cup Z) \times \{+, -\}$ labels each transition of $\mathcal{P}$ with a signal transition. An STG is *autonomous* if it has no input signals (i.e. $X = \emptyset$).

Both [20] and [5] gave also synthesis methods to translate the PN into an STD (called Transition Diagram in [20] and State Graph in [5]) and hence into an ALC implementation of the specified behavior.

Given an STG $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ and the ALTS $([m_0 >, E, T, \delta)$ corresponding to its PN $\mathcal{P}$, we define the associated STD $\mathcal{S} = \langle [m_0 >, E, \lambda \rangle$ as follows. For each $m \in [m_0 >$, we have $\lambda(m) = s^m$, where s^m is a vector of signal values. Let s_i^m denote the value of signal y_i in marking m.

Obviously the STD labelling must be *consistent* with the interpretation of the PN transitions, so we must have for all arcs $e = (m, m')$ in the STD:

- if $\Delta(\delta(e)) = y_i^+$, then $s_i^m = 0$ and $s_i^{m'} = 1$.
- if $\Delta(\delta(e)) = y_i^-$, then $s_i^m = 1$ and $s_i^{m'} = 0$.
- otherwise $s_i^m = s_i^{m'}$.

Figure 5 shows an example of an STG and the corresponding STD. Note that for historical reasons PN transitions are denoted by the corresponding labels and PN places are denoted by circles PN places with only one predecessor and one successor are generally omitted. So in Figure 5.(a) the initial marking of the PN (corresponding to the leftmost state in Figure 5.(b)) appears on the edge between y^- and x^+.

An STG is defined as *valid* if its STD is *finite* (i.e. the PN is bounded) and has a *consistent labeling*. In this paper we will only consider valid STGs (otherwise their interpretation as control circuit specifications would lose its meaning).

One consequence of this requirement is similar to Property 2.1:

Property 4.1 *In a valid* STG, *for all firing sequences of its* PN, *the signs of the transitions of each signal alternate.*

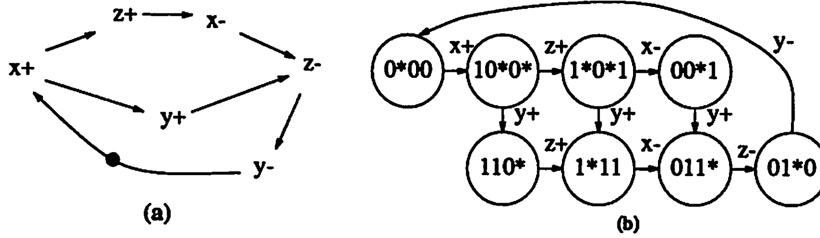

Figure 5: A Signal Transition Graph and its State Transition Diagram

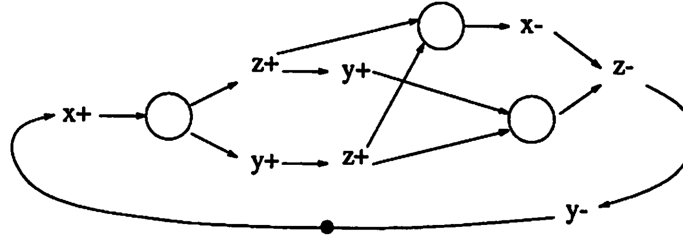

Figure 6: A Persistent Signal Transition Graph with non-persistent underlying Petri net

Two STG markings m_1 and m_1' are *equivalent* if for each finite firing sequence $m_1[t_1 > m_2[t_2 > \ldots$ there exists a firing sequence $m_1'[t_1' > m_2'[t_2' > \ldots$ such that $\Delta(t_i) = \Delta(t_i')$ for all i. This relation partitions the set of reachable markings into equivalence classes. The equality between STG markings (and hence STD states) in the following will always be modulo this equivalence[13].

An STG is *persistent* if for each reachable marking m_1, if t_1 is enabled in m_1 and $m_1[t_2 > m_2$, with $\Delta(t_1) \neq \Delta(t_2)$, then there exists a transition t_3 enabled in m_2 such that $\Delta(t_1) = \Delta(t_3)$. An STG is *output persistent* if the above definition holds for all t_1 such that $\Delta(t_1) \in Z$.

Note that this definition of STG persistence allows a case like that in Figure 6 to be treated as persistent, even though the underlying PN is *not persistent*. So PN persistency is a *stronger* condition than STG persistency. Only the transition labeling Δ that maps two different PN transitions into y^+ (and similarly for z^+) "erases" the distinction that was present between the PNs underlying Figures 6 and 5.(a), so that the two STGs generate isomorphic STDs. This Figure illustrates clearly the "semantic gap" arising from using a purely interleaving semantics (as in the STD) versus using a true concurrent semantics (as in the STG).

The following Theorem is a direct consequence of the results in [26]:

Theorem 4.5

1. *For every deterministic, commutative and persistent* STD S *there exists an* STG *whose* PN *is persistent, bounded and generates* S.
2. *For every distributive* STD S *there exists an* STG *whose* PN *is a safe Marked Graph and generates* S.
3. *For every distributive* STD S *there exists an* STG *whose* PN *is safe, persistent and generates* S.

4.3 Signal Transition Graphs and Asynchronous Logic Circuits

Signal Transition Graphs were introduced to specify Asynchronous Logic Circuits, a special case of Binary Asynchronous Control Structures. We are now ready to establish a correspondence between the STD associated with a valid STG and the STD associated with a BACS. We shall also examine when a similar correspondence can be established with the STD associated with an ALC.

Intuitively, our target is to implement the STG as a circuit with one signal for each STG output signal, where the Boolean function computed by each gate maps each STD binary label into the corresponding *implied value* for that signal. The implied value for signal y_i in state s^m is defined as the complement of s_i^m if y_i is excited in s^m, s_i^m otherwise. So for example if

[13]Which amounts to observable behavior equivalence.

$s^m = 00^*1$ for signal ordering $y_0 y_1 y_2$, then the implied value of y_0 is 0, the implied value of y_1 is 1 and the implied value of y_2 is 1.

Both [20] and [5] recognized that an STG has an STD-isomorphic circuit implementation if (but not only if) output signal transitions are persistent, and the STD is non-contradictory. Later on Chu ([6]) formulated a *necessary and sufficient* condition for the existence of a circuit implementation of a valid STG, called Complete State Coding in [18] (who proved it to be necessary and sufficient).

An STG has the *Complete State Coding property* if *all markings with the same binary label have the same set of enabled output signal transitions.* So we can state the following Theorem.

Theorem 4.6 *Let S be the* STD *of a valid* STG. *Let $Y = X \cup Z$ be its set of signals. Let $C = \langle V, H, Y, \rho \rangle$ be a* BACS *whose* STD *is isomorphic to S.*

The output signals Z of S can be characterized as Boolean functions of signals in Y if and only if the STG *has the Complete State Coding property.*

This can be proved observing that if the STG has the Complete State Coding property, then the implied value rule defines a unique Boolean mapping between the set of STD labels and the value of each signal. On the other hand, suppose that the STG does not have the Complete State Coding property. Then two states with the same label have a different implied value for some output signal z_i, and there is no Boolean function of the set of STG signals that can characterize z_i.

Corollary 4.7 *Let S be the* STD *of a valid autonomous* STG *(i.e. $X = \emptyset$).*

There exists an autonomous ALC *such that its* STD *is isomorphic to S if and only if all the states of S have distinct labels.*

Theorem 4.6 means that each output signal of an STG can be implemented as a Boolean function if and only if its value and excitation in each STD state is uniquely determined by the STD state binary label itself.

The problem, given a valid STG $\mathcal{G}$ without the Complete State Coding property, to determine another STG $\mathcal{G}'$ with the Complete State Coding property and such that its set of traces (restricted to the signals of $\mathcal{G}$) is a subset of the traces of $\mathcal{G}$ was solved recently for various special classes of STGs (see, for example, [27] or [24] for Marked Graphs and [14] for free-choice STGs).

5 Classification of Models of Asynchronous Logic Circuits

We are now ready to proceed with the next contribution of this paper, a classification of the Signal Transition Graph models according to the type of Asynchronous Control Structure or Asynchronous Logic Circuit they give origin to.

Note that the major focus of this section is the analysis of the "protocol of interaction" between the nodes in a BACS or an ALC, rather than a synthesis methodology for the internal structure of each node. We are interested in checking when a given STG describes a speed independent or semi-modular interaction between nodes, and we assume that each node can be realistically modeled using an instantaneous Boolean evaluation and pure delay or inertial delay (see Sections 3.4 and 3.5). This point of view is interesting, for example, if the interconnections between nodes occur outside a single integrated circuit, so that the delays between them are large compared with the delays inside the circuit, and are outside of the direct designer's control. The reader is referred to [8], [13] or [1] for examples of techniques to realize the function of each node using a specific implementation technology.

The practical relevance of this section is that an STG that specifies a behavior that is not speed independent will almost certainly cause a malfunctioning implementation, since the behavior of the implementation will depend on the delays of the components. Similarly, an STG that specifies a behavior that is not output persistent will most likely be implemented with a circuit that suffers from hazard problems (due to the analysis in Section 3.5) unless a special care is devoted to its low-level design. Analogous considerations apply to the usefulness of having the STG describe a delay-insensitive behavior when the interconnection delays between the circuit components are widely variable and unpredictable.

5.1 Speed Independence with Inertial Delay

Using the definitions from Section 3.4, we can now check, given an STG, if the associated BACS or ALC belongs to any of the speed-independence classes. We will give sufficient conditions for each class (necessary and sufficient conditions can be derived using the definitions directly on the STD, but can be in general more expensive to compute).

The following Proposition is a result of Theorems 4.2, 4.3 and 4.4.

Proposition 5.1 *Let $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ be a valid* STG, *and let S be its associated* STD. *Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a* BACS, *modeled with inertial delay, such that its* STD *is isomorphic to S and node v_0 has $I^Y(v_0) = Z$ and $O^Y(v_0) = X$ (i.e. inputs Z and outputs X: this special node represents the* environment *where the circuit described by the* STG *will operate).*

- *$\mathcal{A}$ is* ID-*Speed-independent if $\mathcal{P}$ is free-choice, bounded and live.*
- *$\mathcal{A}$ is* ID-*Semi-modular if $\mathcal{P}$ is persistent.*
- *$\mathcal{A}$ is* ID-*Distributive if $\mathcal{P}$ safe and persistent.*
- *$\mathcal{A}$ is* ID-*Distributive if $\mathcal{P}$ is a Marked Graph.*

Obviously if the STG has the Complete State Coding property, and hence has an ALC implementation (Section 4.3), then the above results hold for the ALC as well, because it is just a special case of BACS.

The following Proposition is a result of Theorem 4.2.

Proposition 5.2 *Let $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ be a valid* STG *with the Complete State Coding property, and let S be its associated* STD. *Let $\mathcal{C} = \langle X, Z, F \rangle$ be an* ALC, *modeled with inertial delay, such that its* STD *is isomorphic to S.*

- *$\mathcal{C}$ is* ID-*Output-persistent if $\mathcal{P}$ is live and $\mathcal{G}$ is output persistent.*

The following Proposition is a result of Theorems 4.4 and 4.5.

Proposition 5.3 *Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a* BACS, *modeled with inertial delay, and let S be its associated* STD.

- *if $\mathcal{A}$ is* ID-*Semi-modular, then there exists an* STG *whose* PN *is bounded and persistent, and whose* STD *is isomorphic to S.*
- *if $\mathcal{A}$ is* ID-*Distributive, then there exists an* STG *whose* PN *is safe and persistent, and whose* STD *is isomorphic to S.*
- *if $\mathcal{A}$ is* ID-*Distributive, then there exists an* STG *whose* PN *is a Marked Graph, and whose* STD *is isomorphic to S.*

5.2 Speed Independence with Pure Delay

We will now give conditions, similar to the previous Section, in order to characterize an ALC implementation of an STG using the *pure* delay model (Section 3.5). We will only consider circuits that exhibit a *cyclic* behavior, that is whose operation does not "stop" after a finite number of transitions, because they have the most practical interest.

The following Proposition is a result of Theorems 4.2, 4.3 and 4.4.

Proposition 5.4 *Let $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ be a valid* cyclic STG, *and let S be its associated* STD. *Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a* BACS, *modeled with pure delay, such that its* ALTS *is bounded and isomorphic to S, and node v_0 has $I^Y(v_0) = Z$ and $O^Y(v_0) = X$ (i.e. inputs Z and outputs X: this special node represents the* environment *where the circuit described by the* STG *will operate).*

- *$\mathcal{A}$ is* PD-*Semi-modular if $\mathcal{P}$ is free-choice, live and safe.*
- *$\mathcal{A}$ is* PD-*Semi-modular if $\mathcal{P}$ is persistent.*
- *$\mathcal{A}$ is* PD-*Distributive if $\mathcal{P}$ safe and persistent.*
- *$\mathcal{A}$ is* PD-*Distributive if $\mathcal{P}$ is a Marked Graph.*

Obviously if the STG has the Complete State Coding property, and hence has an ALC implementation (Section 4.3), then the above results hold for the ALC as well, because it is just a special case of BACS. Proposition 5.3 holds also in the pure delay case.

5.3 Delay Insensitivity

Informally, a Delay-Insensitive circuit operates correctly under the inertial gate and wire delay model in input-output mode with general multiple-winner races (Section 3.2). A more formal definition was given by Udding in [21], using Trace theory.

Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a BACS, let $\mathcal{S} = \langle S, E, \lambda \rangle$ be its associated STD, and let $\mathcal{T} = \langle A, \Sigma \rangle$ be its associated Trace Model. Let $\nu(y)$ for signal $y \in Y$, denote the node $v \in V$ such that $y \in O^Y(v)$ (i.e. y is an output of v). Let v_0 be a special node qualified as "the environment" (this distinction is necessary to speak about "output persistency"). The delay insensitivity of $\mathcal{A}$ is captured by the following four rules ([21]):

R_0 For all $s \in A^*$ and $y_1^* \in A$, we must have $sy_1^* y_1^* \notin \Sigma$, i.e. Property 2.2 must hold for $\mathcal{T}$.

R_1 For all $s, t \in A^*$ and $y_1^*, y_2^* \in A$, such that $\nu(y_1) = \nu(y_2)$ (i.e. are both outputs of the same node), we must have $(sy_1^* y_2^* t \in \Sigma) = (sy_2^* y_1^* t \in \Sigma)$. So signals that are output of one node cannot be ordered.

R_2 This rule, dealing with commutativity, takes two forms, in decreasing order of strictness:

- R_2' For all $s, t \in A^*$ and $y_1^*, y_2^* \in A$, such that $\nu(y_1) \neq \nu(y_2)$ (i.e. outputs of different nodes), if $sy_1^* y_2^* \in \Sigma \wedge sy_2^* y_1^* \in \Sigma$ then $(sy_1^* y_2^* t \in \Sigma) = (sy_2^* y_1^* t \in \Sigma)$. So no change in the ordering between two transitions produced by two different nodes can change the subsequent behavior of *any* node. This rule is satisfied if $\mathcal{S}$ is *commutative*.
- R_2'' For all $s, t \in A^*$ and $y_1^*, y_2^*, y_3^* \in A$ such that $\nu(y_1) \neq \nu(y_2)$ and $\nu(y_3) \neq \nu(y_2)$ (i.e. y_2 is the output of a node different from y_1 and y_3) if $sy_1^* y_2^* t y_3^* \in \Sigma \wedge sy_2^* y_1^* t \in \Sigma$ then $sy_2^* y_1^* t y_3^* \in \Sigma$. So no change in the ordering between two transitions produced by two different nodes can change the subsequent behavior of *another* node (i.e. y_1 and y_2 can "have memory" of this change in the ordering, but no-one else can).

R_3 This rule, dealing with persistence, takes three forms, in decreasing order of strictness:

- R_3' For all $s \in A^*$ and $y_1^*, y_2^* \in A$, $y_1^* \neq y_2^*$, if $sy_1^* \in \Sigma \wedge sy_2^* \in \Sigma$ then $sy_1^* y_2^* \in \Sigma$. This rule is equivalent to $\mathcal{S}$ being commutative.
- R_3'' For all $s \in A^*$ and $y_1^*, y_2^* \in A$, $y_1^* \neq y_2^*$, such that either $\nu(y_1) \neq v_0$ or $\nu(y_2) \neq v_0$, if $sy_1^* \in \Sigma \wedge sy_2^* \in \Sigma$ then $sy_1^* y_2^* \in \Sigma$, i.e. only transitions produced by the environment can disable each other. This rule is similar to the idea of output persistency.
- R_3''' For all $s \in A^*$ and $y_1^*, y_2^* \in A$, $y_1^* \neq y_2^*$, such that $\nu(y_1) \neq \nu(y_2)$, if $sy_1^* \in \Sigma \wedge sy_2^* \in \Sigma$ then $sy_1^* y_2^* \in \Sigma$, i.e. no transitions produced by different nodes can disable each other.

In cases R_3'' and R_3''' it is assumed that the implementation of nodes with mutually disabling transitions can do so without hazards, using appropriates circuit design techniques.

All the circuits that Udding considers must satisfy R_0 and R_1. In addition:

1. the circuits that satisfy R_2' and R_3' are called the *synchronization* class, C_1.
2. the circuits that satisfy R_2' and R_3'' are called the *data communication* class, C_2.
3. the circuits that satisfy R_2' and R_3''' are called the *arbitration* class, C_3.
4. the circuits that satisfy R_2'' and R_3''' are called the *delay insensitive* class, C_4.

Obviously $C_1 \subset C_2 \subset C_3 \subset C_4$.

Now we can verify, given an STG, whether the BACS (or ALC, if it exists) implementing its output signals belongs to any of the delay insensitivity classes. An STG generates a Trace Model as the set of its firing sequences. So in principle a check whether an STG specification describes a system in C_1, C_2, C_3 or C_4 would require a check whether a potentially infinite set of traces satisfies the above rules.

Fortunately we can do better than that, and examine the STD generated by the STG, which is finite for valid STGs[14].

The following Proposition is a trivial consequence of the PN firing rule:

Proposition 5.5 *Let $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ be a valid* STG, *and let $\mathcal{S}$ be its associated* STD. *Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a* BACS *such that its* STD *is isomorphic to $\mathcal{S}$ and node v_0 has $I^Y(v_0) = Z$ and $O^Y(v_0) = X$ (i.e. inputs Z and outputs X).*

[14]This can still be very costly, in practice, since the size of the STD can be exponential in the size of the STG.

R_0 *is automatically satisfied, since $\mathcal{G}$ satisfies Property 4.1.*

R_1 *is satisfied if and only if for all states m_1, m_2 and m_3 of S such that $m_1 E(y_1^*) m_2 E(y_2^*) m_3$ and $\nu(y_1) = \nu(y_2)$ there exist m_4 and m_5 such that $m_1 E(y_2^*) m_4 E(y_1^*) m_5$.*

R_2 :

- R_2' *is satisfied if and only if for all states m_1, m_2, m_3, m_2' and m_3' of S such that $m_1 E(y_1^*) m_2 E(y_2^*) m_3$ and $m_1 E(y_2^*) m_2' E(y_1^*) m_3'$, we have $m_3 = m_3'$ (modulo the equivalence described in Section 4.2).*
- R_2'' *is satisfied if and only if for all pairs of simple paths $m_1 E(y_1^*)\ m_2 E(y_2^*)\ m_3 E(y_3^*) \ldots m_{i-1} E(y_{i-1}^*)\ m_i E(y_i^*)\ m_{i+1}$ and $m_1 E(y_2^*)\ m_2' E(y_1^*)\ m_3' E(y_3^*) \ldots m_{i-1}' E(y_{i-1}^*)\ m_i'$ we have $E(y_i^*)$ enabled in m_i'.*

R_3 :

- R_3' *is satisfied if and only if $\mathcal{G}$ is* persistent.
- R_3'' *is satisfied if and only if $\mathcal{G}$ is* output persistent
- R_3''' *is satisfied if and only if for all states m_1, m_2, m_2' of S such that $m_1 E(y_1^*) m_2$ and $m_1 E(y_2^*) m_2'$ and $\nu(y_1) = \nu(y_2)$ there exist m_3 and m_3' such that $m_2 E(y_2^*) m_3$ and $m_2' E(y_1^*) m_3'$.*

6 On the relationship between Signal Transition Graphs and Change Diagrams

Change Diagrams, described more in detail in [25, 11, 12], are an event-based model for Asynchronous Logic Circuits that bears some resemblance to Signal Transition Graphs, but has some interesting properties of its own. In this Section we compare the two models, and show how, when we limit ourselves to Semi-modular circuits, they have similar modeling power. So the choice between the two is just dependent on the need for choice modeling (as in Output-persistent circuits) and on the availability of analysis and synthesis algorithms.

The definition of Change Diagrams is based on two types of precedence relations between transitions in Asynchronous Logic Circuits.

1. the *strong precedence* relation between transitions a^* and b^*, usually depicted by a solid arc in the graphical representation of Change Diagrams, means that that b^* *cannot occur without the occurrence of* a^*.
2. the *weak precedence* relation between transitions a^* and b^*, usually depicted by a dashed arc in the graphical representation, means that b^* *may occur after an occurrence of* a^*. But b^* may also occur *after some other transition* c^*, which is also weakly preceding b^*, without the need for a^* to occur.

A Change Diagram is therefore formally defined as a tuple $(A, \rightarrow, \vdash, M, O)$, where:

- A is a set of *transitions* or *events*, labeled with transitions of a set of signals Y (as in Section 4.2).
- $\rightarrow \subseteq (A \times A)$ is the *strong* precedence relation between transitions.
- $\vdash \subseteq (A \times A)$ is the *weak* precedence relation.
- M is a set of initially *active* arcs.
- O is a set of so-called *disengageable* arcs.

The relations $\rightarrow$ and $\vdash$ are mutually exclusive (i.e. $(a^*, b^*) \in \rightarrow$ implies that $(a^*, b^*) \notin \vdash$ and vice-versa), and all the predecessors of a transition a^* must be either of the *strong* type or of the *weak* type. Hence the set of transitions A is partitioned into *AND-type* transitions (with strong predecessors) and *OR-type* transitions (with weak predecessors).

The firing rule of Change Diagrams is similar to that of PNs, with arcs playing the role of places and flow relation elements at the same time. Each arc is assigned an integer *activity* which, unlike PN marking, can be *negative*. Initially each arc in M has activity 1, and each arc not in M has activity 0.

- An *AND-type* transition is enabled if *all* its predecessor arcs have activity greater than 0.
- An *OR-type* transition is enabled if *at least one* predecessor arc has activity greater than 0.

When an enabled transition fires, the activity of each predecessor arc is decremented, and the activity of each successor arc is incremented[15]. A Change Diagram is *bounded* if the activity on each arc is bounded (both above and below) in all possible firing sequences.

Disengageable arcs are "removed" from the Change Diagram after the *first firing* of their successor transition. They are used to represent the *initialization sequence* of a circuit, and we will not enter into details concerning their usage.

Following [25], we can associate a State Transition Diagram $\mathcal{S} = \langle S, E, \lambda \rangle$ with a Change Diagram, as we did in Section 4.2 for STGs. Let S be the set of reachable activity vectors (similar to PN markings). An arc $(s, s') \in E$ joins two activity vectors $s, s' \in S$ if there exists a transition $y_i^* \in A$ that is enabled in s and whose firing produces s'. The labeling must be consistent, so for each arc $(s, s') \in E$ corresponding to transition y_i^* we must have:

- $\lambda(s)_i = 0$ and $\lambda(s')_i = 1$ for an arc associated with y_i^+.
- $\lambda(s)_i = 1$ and $\lambda(s')_i = 0$ for an arc associated with y_i^-.
- otherwise $\lambda(s)_i = \lambda(s')_i$.

A Change Diagram is *correct* if it satisfies the following conditions, ensuring that the above labeling is consistent:

- for all firing sequences, the signs of the transitions of each signal alternate (similar to Property 4.1).
- no two transitions of the same signal can be concurrently enabled in any reachable activity vector.
- the Change Diagram is connected and *bounded* (i.e. the set S is *finite*).

The main theoretical result concerning Change Diagrams is stated (without proof) in [25]. A *transient cycle* in an STD is defined as a cycle where at least one variable is continuously excited with the same value (see Figure 9 for an example).

Proposition 6.1 *Each Semi-modular* STD *without transient cycles has a corresponding correct Change Diagram.*
Each correct Change Diagram has a corresponding Semi-modular STD.

Thus Change Diagrams are *equivalent in modeling power* to Semi-modular STDs, and hence to Semi-modular ALCs.

Change Diagrams are useful in practice because of the availability of low-complexity polynomial time *analysis* algorithms to decide, e.g.:

- whether a given Change Diagram is correct, and hence it can be used as a valid specification of a Semi-modular ALC.
- whether a given ALC has a distributive Change Diagram, and hence a distributive STD[16].

Furthermore synthesis algorithms from Change Diagrams to ALCs in various technologies were outlined in [11].

The main limitation of Change Diagrams is their inability to describe *choice* among alternative behaviors, as modeled by places with more than one successor in PNs. So a designer faced with the description, for example, of a self-timed memory element, must describe the various possible read/write cycles of a 0/1 datum as an *alternation* rather than a *choice* between them.

Given Theorems 5.3 and 6.1, we can see that there is a strong modeling similarity between Change Diagrams and STGs. Both can model all Semi-modular ALCs, that is a broad class of interest for asynchronous design. At the same time this similarity is only limited because STGs can describe choice, and there is a difference in the modeling power when *unbounded* PNs and Change Diagrams are considered.

Figure 7.(a) shows a cyclic finite Change Diagram with unbounded arc activity[17]. Such unbounded behavior, in which the i-th ($i = 1, 2, \ldots$) occurrence of event c is caused either by the i-th occurrence of a or by the i-th occurrence of b, is represented in Figure 7.(b) as a Change Diagram unfolding ([11]). In the unfolding each event a^i represents a *unique occurrence* of the corresponding event a in a firing sequence of the Change Diagram (similarly for b^i with respect to b and c^i with respect to c).

The same behavior can be represented by the *infinite* PN in Figure 7.(c). We conjecture that there exists no finite PN representing it. The seemingly equivalent PN shown in Figure 8.(a) describes in effect a different behavior. Its "unfolding"

[15] The activity of a predecessor arc $a^* \vdash b^*$ of an *OR-type* transition b^* can become negative as a consequence of a firing of b^* due to positive activity on some other arc $c^* \vdash b^*$. It can then become positive again when b^* fires in turn.

[16] Note that this analysis can be performed by direct construction of the Change Diagram, without going through the exponential size STD.

[17] Here events are not labeled with signal transitions for the sake of simplicity. It is possible to construct a correct interpreted Change Diagram showing the same type of behavior.

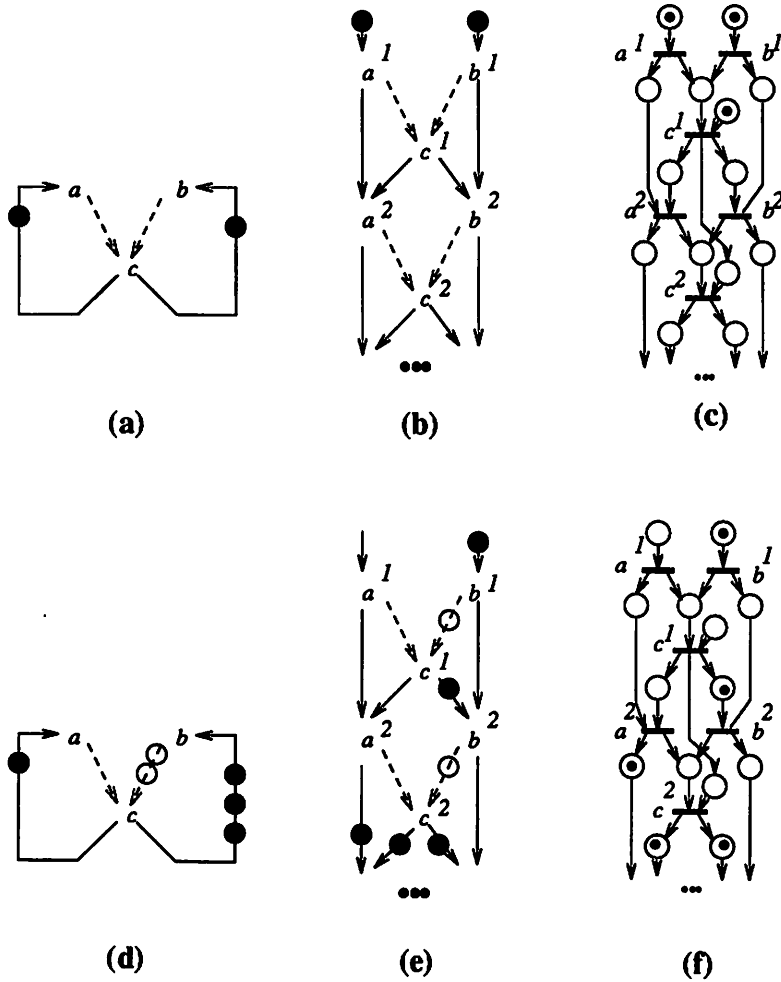

Figure 7: A Change Diagram without an equivalent finite Petri net

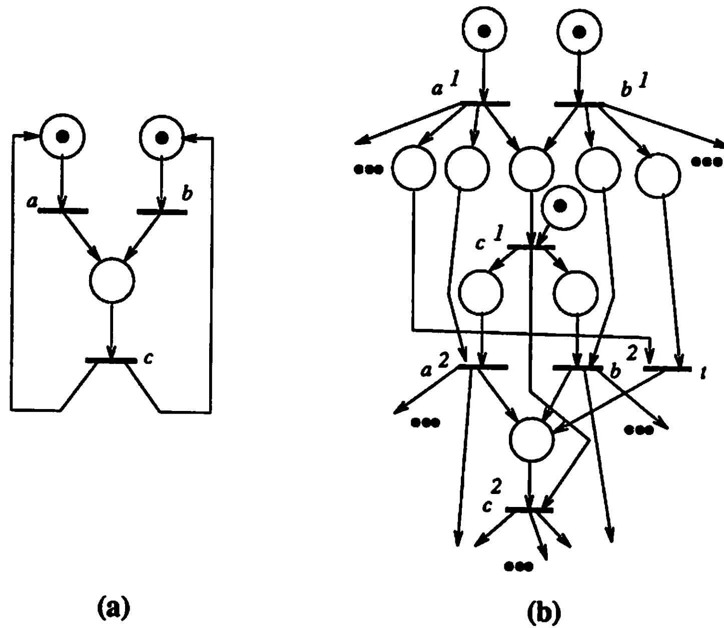

Figure 8: A Petri net only similarly equivalent to Figure 7

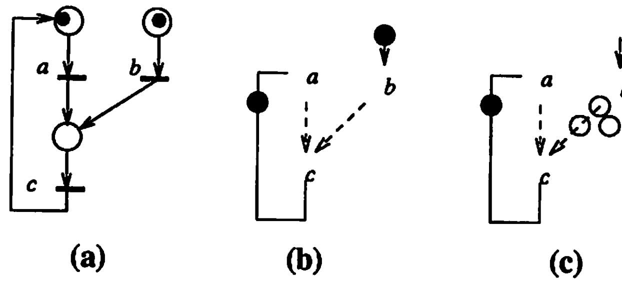

Figure 9: An unbounded Change Diagram with an equivalent bounded Petri net

into an execution net (see [26]) in Figure 8.(b) shows that here the i-th occurrence of event c can be caused by any combination of pairs of the form a^k and b^{i-k} where k can be of any value between 0 and i. The difference between these behaviors is obvious.

The Change Diagram is able to remember the number of occurrences of events a and b, using the negative activity mechanism. So if a fires twice, as represented in Figures 7.(d), 7.(e) and 7.(f) (empty circles represent negative activity on the arc between b and c), and then it stops firing, c can fire again only after b has fired *three* times, in order to "re-absorb" the negative activity. On the other hand in Figure 8.(a), if a fires twice and then stops, c can begin firing again as soon as b fires, because there is no way to remember an *unbounded* "debt" of tokens.

As a final example, consider Figure 9.(a). It represents an initially one-place buffer that becomes two-place when event b occurs. The behavior is Semi-modular, because no event is disabled. Yet there is no bounded equivalent Change Diagram, because bounded Change Diagrams can represent only Semi-modular behaviors without *transient cycles*. In this example event b is continuously enabled during the cyclic firing of a and c. An equivalent Change Diagram, shown in Figure 9.(b), has an *unbounded* negative activity on the arc between b and c (Figure 9.(c) shows such negative activity after three occurrences of a and c).

In summary, both Signal Transition Graphs and Change Diagrams can represent Semi-modular circuits, and their modeling power differs only when it comes to describe unbounded behaviors (not interesting for speed-independent circuit design) and *choice*. The designer can choose between them depending on the need, respectively, for an explicit representation of choice in Signal Transition Graphs and polynomial time analysis and synthesis algorithms in Change Diagrams.

7 A Design Methodology based on Asynchronous Control Structures and Signal Transition Graphs

To summarize the major contributions of the paper, we outline a design methodology based on the proposed modeling formalism.

- The design process begins when the designer selects:
  - the circuit and element delay models that need to be used (Section 3.2), depending on, for example:
    * the chip implementation technology, where gate delays or interconnection delays may dominate.
    * the partitioning of the system into chips, boards and cabinets, that may dictate to consider some wire or component delay to be unbounded.
    * the trade-off between modularity (when little is assumed on the environment where the circuit will operate) and performance (when such knowledge of the environment can greatly help to improve the performance).
  - the class of behavior that best fits the synthesis algorithms and the system-level requirements. For example:
    * good synthesis algorithms exist for distributive specifications ([8]).
    * using a purely speed-independent specification, rather than a semi-modular one, increases the risk of malfunctioning due to hazards, as we argued in Section 3.5,.
    * on the other hand, there are inherently non-semi-modular behaviors, such as arbitration, which may be required by the type of application. In this case, one would like to use the output-persistency criterion to at least make sure that the parts of the system that need to be designed with standard logic gates will not be *inherently* (i.e. independent of the implementation style) prone to hazards.

- the class of Petri net underlying the Signal Transition Graph specification that offers the optimal trade-off between ease of analysis and descriptive power:
    * structural properties[18], such as being a Marked Graph or being free-choice, are very easy to compute, yet allow to use, for example, Proposition 5.1 to design distributive circuits.
    * behavioral properties[19], such as liveness, safeness, boundedness, persistency, are in general harder to compute, but the PN literature offers a wealth of efficient algorithms. So we can use again Proposition 5.1 to design semi-modular circuits.
    * on the other hand, results like Proposition 5.3 show that if the designer chooses to describe, say, a distributive behavior, then it can be specified using an STG whose underlying PN is a Marked Graph.

- Once the above choices have been made, the STG describing the desired behavior can be verified against the chosen properties, using the results from the literature summarized in this paper.

- Then an implementation is produced using a synthesis algorithms from the literature (e.g. [8], [13], [1]).

- Finally, the resulting circuit can be verified (using, for example, [7]) *against the very same properties selected in the first step*, since our framework provides a *uniform representation* for such properties both at the specification and at the implementation level.

8 Conclusions

This paper has achieved three major objectives:

- Define a low-level *structural* and *behavioral* model for asynchronous systems, the Asynchronous Control Structure and its companion Arc-Labeled Transition System. This model is more powerful than similar models known from the literature (e.g. [19] or [12]) because it allows to use non-determinism for behavior abstraction and for components that cannot be characterized as Boolean functions, e.g. arbiters.

- Characterize concepts such as hazards, delay-insensitivity and speed-independence in terms of formal properties of the Arc-Labeled Transition System and its Cumulative Diagram, both using pure delays and inertial delays. In this context, we proved that in the pure delay case speed-independence is equivalent to semi-modularity.

- Relate those properties of the Arc-Labeled Transition System (and hence of the circuit) with *high-level* properties of a general, Petri net based specification, the Signal Transition Graph. So we can constrain the Signal Transition Graph representation depending on the *class of circuits that we want to describe*, rather than artificially imposing constraints such as safeness and liveness.

In conclusion we provide a unified model, based on a Signal Transition Graph specification, where the desired properties *common to every implementation* (because they are part of the specification itself) can be formally analyzed and verified.

References

[1] P. A. Beerel and T. H-Y. Meng. Gate-level synthesis of speed-independent asynchronous control circuits. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, March 1992.

[2] E. Best and K. Voss. Free choice systems have home states. *Acta Informatica*, 21:89–100, 1984.

[3] J. A. Brzozowski and C-J. Seger. Advances in asynchronous circuit theory – part i: Gate and unbounded inertial delay models. *Bulletin of the European Association of Theoretical Computer Science*, October 1990.

[4] S. Burns and A. Martin. A synthesis method for self-timed VLSI circuits. In *Proceedings of the International Conference on Computer Design*, 1987.

[18] I.e. properties that depend on the graph underlying the PN, and not on its marking.
[19] I.e. properties that depend both on the graph and on the marking of the PN.

[5] T. A. Chu. Synthesis of self-timed control circuits from graphs: an example. In *Proceedings of the International Conference on Computer Design*, pages 565–571, 1986.

[6] T. A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[7] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. The MIT Press, Cambridge, Mass., 1988. An ACM Distinguished Dissertation 1988.

[8] V. I. Varshavsky et al. *Self-timed Control of Concurrent Processes*. Kluwer Academic Publisher, 1990. (Russian edition: 1986).

[9] D. A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Institute*, 257:161–190,275–303, March 1954.

[10] R.M. Keller. A fundamental theorem of asynchronous parallel computation. *LNCS*, 24:103–112, 1975.

[11] M. A. Kishinevsky, A. Y. Kondratyev, and A. R. Taubin. Formal method for self-timed design. In *Proceedings of the European Design Automation Conference*, 1991.

[12] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. On self-timed behavior verification. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1992.

[13] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the Design Automation Conference*, June 1991.

[14] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proceedings of the Design Automation Conference*, June 1992.

[15] A. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communications*, The UT Year of Programming Series. Addison-Wesley, 1990.

[16] A. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.

[17] R. E. Miller. *Switching theory*, volume 2. Wiley and Sons, 1965.

[18] C. W. Moon, P. R. Stephan, and R. K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[19] D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.

[20] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *International Workshop on Timed Petri Nets, Torino, Italy*, 1985.

[21] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

[22] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley Interscience, 1969.

[23] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1985.

[24] P. Vanbekbergen, G. Goossens, and H. De Man. A local optimization technique for asynchronous control circuits. In *Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[25] V. I. Varshavsky, M. A. Kishinevsky, A. Y. Kondratyev, L. Y. Rosenblyum, and A. R. Taubin. Models for specification and analysis of processes in asynchronous circuits. In *Izvestiia Akademii nauk SSSR, Tekhnicheskaya Kibernetika*, 2, pages 171–190, 1988. English translation: Soviet Journal of Computer and Systems Sciences.

[26] A. V. Yakovlev. Analysis of concurrent systems through lattices. *Theoretical Computer Science*, Submitted for publication.

[27] A. V. Yakovlev and A. Petrov. Petri nets and parallel bus controller design. In *International Conference on Application and Theory of Petri Nets, Paris, France*, June 1990.

[28] A.V. Yakovlev. On limitations and extensions of STG model for designing asynchronous control circuits. In *Accepted for publication in Proc. of International Conference on Computer Design*, October 1992.