Analyzing Specifications for Delay-Insensitive Circuits

Tom Verhoeff `<wstomv@win.tue.nl>`
Faculty of Mathematics and Computing Science
Eindhoven University of Technology, The Netherlands

Abstract

We present the XDI Model *for specifying delay-insensitive circuits, that is, reactive systems that correctly exchange signals with their environment in spite of unknown delays incurred by the interface. XDI specifications capture restrictions on the communication between circuit and environment, treating both parties equally. They can be visualized as state graphs where each arrow is labeled by a communication terminal and each state by a safety/progress label.*

We investigate various properties that can be extracted from XDI specifications: automorphisms, environment partitions, autocomparison matrix, and classifications of choice, order dependence, and nondeterminism. We introduce a distinction between static *and* dynamic *output nondeterminism, capturing the difference between design freedom and arbitration. Determining specification properties is useful for validation and design.*

Acknowledgment

This work has been supported by the European Commission under Working Group ACiD-WG (Esprit Nr. 21049) as part of the Fourth Framework.

Copyright Information

Analyzing Specifications for Delay-Insensitive Circuits

Tom Verhoeff `<wstomv@win.tue.nl>`
Faculty of Mathematics and Computing Science
Eindhoven University of Technology, The Netherlands

Abstract

We present the XDI Model *for specifying delay-insensitive circuits, that is, reactive systems that correctly exchange signals with their environment in spite of unknown delays incurred by the interface. XDI specifications capture restrictions on the communication between circuit and environment, treating both parties equally. They can be visualized as state graphs where each arrow is labeled by a communication terminal and each state by a safety/progress label.*

We investigate various properties that can be extracted from XDI specifications: automorphisms, environment partitions, autocomparison matrix, and classifications of choice, order dependence, and nondeterminism. We introduce a distinction between static *and* dynamic *output nondeterminism, capturing the difference between design freedom and arbitration. Determining specification properties is useful for validation and design.*

1. Introduction

We assume the reader is familiar with the basic ideas behind delay-insensitive (DI) circuits, a special class of asynchronous circuits (see e.g. [3]). DI circuits can be viewed as reactive systems that correctly exchange signals with their environment in spite of unknown delays incurred by the wire interface (see Fig. 1).

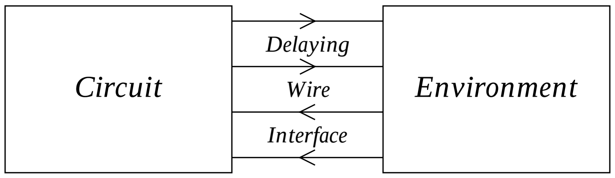

Figure 1: **Circuit-environment interface**

The Extended DI Model—XDI Model for short—was first introduced in [20]. It extends (with progress concerns) the models introduced by Udding [17, 18], Ebergen [5], and Dill [4, Ch. 3] for specifying DI circuits. The DI Algebra by Josephs and Udding [10] and its trace-theoretic semantic model [7, 12] allow the expression of progress requirements for the circuit, but the XDI Model also allows the expression of such requirements for the environment. This symmetry makes it possible to definite reflection and factorization for XDI specifications. Dill's model based on complete traces [4, Ch. 7] also treats progress symmetrically, but it uses infinite traces, whereas the XDI model uses only finite traces. The XDI Model is applied in [14] and [21].

We present the XDI Model in Section 2. The basic definitions appear in Section 2.1. In Section 2.2 we define XDI specifications and related concepts. We have chosen to present the XDI Model in some detail, because most technical points are not accessible outside [20]. We give mathematical definitions and mention important properties, but do not give proofs here. The presentation differs from [20], though the essence is the same. In Section 2.3 we show how XDI specifications can be rendered as labeled state graphs. At the end of that section we explain our drawing conventions for such graphs. In Section 2.4, various examples are given to illustrate the features of the XDI Model. Section 3 deals with the analysis of XDI specifications. Finally, Section 4 concludes the paper with some ideas for further work.

2. The XDI Model for Specifying DI Circuits

We start off with some basic definitions and then introduce the XDI model by defining the space $\mathcal{XDI}$ of XDI specifications. Some useful operators on $\mathcal{XDI}$ are introduced as well. Next we explain how an XDI specification can be visualized by a labeled state graph. Finally, we give some examples of XDI specifications, highlighting the strong points of the XDI Model.

2.1. Basic Definitions

Given set A of symbols, we denote by A^* the set of all finite-length sequences, also called **traces**, over A. The **empty trace** is denoted by ε, and **catenation** of trace u after trace t is denoted by tu.

Circuit terminals are identified by symbols, typically a, b, etc. A signal transition at a terminal constitutes a communication event. The order in which such events occur is captured in a trace, typically named t, u, etc.

Define the set Λ of **(trace/state) labels** as $\{\top, \nabla, \Box, \Delta, \bot\}$, and subset $\Lambda_{safe} = \{\nabla, \Box, \Delta\}$.

The naming and interpretation of the labels is as follows:

Label	Name	Interpretation
$\top$	top	(safety) error due to circuit
∇	transient	progress obligation for circuit
$\Box$	indifferent	no progress obligations
Δ	demanding	progress obligation for environment
$\bot$	bottom	(safety) error due to environment

Safety errors are also known as **(computation) interference** [16]. Operationally speaking, a top state is unreachable. The **reflection** of state label λ is denoted by $\backsim\lambda$, and defined by (1):

$$\begin{array}{c|ccccc} \lambda & \top & \nabla & \Box & \Delta & \bot \\ \hline \backsim\lambda & \bot & \Delta & \Box & \nabla & \top \end{array} \tag{1}$$

Reflection interchanges the role of circuit and environment. It is its own inverse. The **refinement order** on Λ, denoted by $\sqsupseteq$, is defined by (2):

$$\top \sqsupseteq \nabla \sqsupseteq \Box \sqsupseteq \Delta \sqsupseteq \bot \tag{2}$$

which can be read as 'top' refines 'transient', and 'transient' refines 'indifferent', etc. This order is motivated by the testing paradigm for comparing specifications. Under this paradigm, a circuit is executed in a (test) environment with its own specification; λ is considered a refinement of μ when λ passes at least the tests that μ passes (and possibly more). For details, the reader is referred to [20].

Refinement is a total order, reversed by reflection:

$$\lambda \sqsupseteq \mu \quad \equiv \quad \backsim\mu \sqsupseteq \backsim\lambda \tag{3}$$

Function application will be denoted by a dot and it binds weaker than trace catenation. Thus, $f.tu$ means $f.(tu)$.

2.2. XDI Specifications

An **XDI specification** X is a triple $\langle I, O, f\rangle$, where

- I is a finite set of input terminal identifiers, **inputs** for short,
- O is a finite set of output terminal identifiers, **outputs** for short, and
- f is a **trace function** from $(I \cup O)^*$ to Λ, classifying all traces,

satisfying nine conditions, for all symbols $a, b \in I \cup O$ and traces $t, u \in (I \cup O)^*$:

1. $I \cap O = \varnothing$: inputs and outputs are disjoint
2. $f.t = \top \;\Rightarrow\; f.tu = \top$: circuit errors are unrecoverable
3. $f.t = \bot \;\Rightarrow\; f.tu = \bot$: environment errors are unrecoverable
4. $f.t \neq \top \wedge f.ta = \top \;\Rightarrow\; a \in O$: circuit errors are caused only by outputs; this is equivalent to: $f.tu = \top \wedge u \in I^* \;\Rightarrow\; f.t = \top$
5. $f.t \neq \bot \wedge f.ta = \bot \;\Rightarrow\; a \in I$: environment errors are caused only by inputs; this is equivalent to: $f.tu = \bot \wedge u \in O^* \;\Rightarrow\; f.t = \bot$
6. $f.t = \nabla \;\Rightarrow\; (\exists a \in O : f.ta \in \Lambda_{safe})$: the circuit can make error-free progress when required
7. $f.t = \Delta \;\Rightarrow\; (\exists a \in I : f.ta \in \Lambda_{safe})$: environment can make error-free progress when required
8. $f.taa \notin \Lambda_{safe}$: two successive signals on the same terminal can cause **transmission interference** [16]
9. $a \in O \vee b \in I \;\Rightarrow\; f.tabu \sqsupseteq f.tbau$: captures limitations of the wire interface between circuit and environment; the situation where the circuit has done $tabu$ and the environment $tbau$ with $a \in I \wedge b \in O$ is impossible

Conditions 1–5, 8 and 9 cast into the XDI Model the rules characterizing delay-insensitive communication as first introduced by Udding in [17] and later slightly rephrased in [18]. In fact, condition 8 corresponds to R_0, and 9 generalizes R_1, R_2', and R_3''' of [18]. Condition 6, concerning progress obligations for the circuit, is implicit in Josephs's model [7], whereas condition 7, concerning progress obligations for the environment, is new. We come back to the more complicated condition 9, also called the **Neighbor-Swap Rule**, when discussing state graphs in Section 2.3.

The **fundamental property** of XDI specifications is that if circuit and environment adhere to their mutual specification X, then no errors and no deadlock will occur, even when they communicate over an interface of wires that incur unknown communication delays. Deadlock can arise when one party is demanding and the other is safe but not transient. The safety part of the fundamental property was proven for specifications without progress concerns (i.e., $f.t \in \{\top, \Box, \bot\}$) in [17]. The generalization with progress is covered in [20].

Given specification $X = \langle I, O, f\rangle$ we define its **alphabet** $\mathbf{a}X$ and its **trace set** $\mathbf{t}X$ by

$$\mathbf{a}X = I \cup O \tag{4}$$
$$\mathbf{t}X = \{ t \in (\mathbf{a}X)^* \mid f.t \in \Lambda_{safe} \} \tag{5}$$

The trace set consists of all traces where operation is error-free. It is prefix-closed (cf. conditions 2, 3). The **direction** of symbols in I is called input, and of symbols in O output. Input and output are called **opposite** directions.

The set of all XDI specifications is denoted by $\mathcal{XDI}$. The subset of all XDI specifications with inputs I and outputs O is denoted by $\mathcal{XDI}(I, O)$.

Reflection operator $\backsim$ and refinement order $\sqsupseteq$ on Λ can be lifted to trace functions by

$$(\backsim f).t \quad = \quad \backsim(f.t) \tag{6}$$

$$f \sqsupseteq f' \quad \equiv \quad (\forall t : f.t \sqsupseteq f'.t) \tag{7}$$

and to $\mathcal{XDI}$ by

$$\backsim\langle I, O, f\rangle \quad = \quad \langle O, I, \backsim f\rangle \tag{8}$$

$$\langle I, O, f\rangle \sqsupseteq \langle I', O', f'\rangle \quad \equiv \quad \begin{cases} I = I' & \wedge \\ O = O' & \wedge \\ f \sqsupseteq f' \end{cases} \tag{9}$$

These lifted versions inherit many properties of their original, e.g., the lifted reflections are also their own inverse, and they also reverse the (lifted) refinement order. Note, however, that the refinement orders on trace functions and $\mathcal{XDI}$ are partial—not total—orders.

Reflection on $\mathcal{XDI}$ interchanges input and output. $\mathcal{XDI}$ is closed under reflection, because the XDI conditions taken together are invariant under an input-output swap. The refinement order $X \sqsupseteq Y$ on $\mathcal{XDI}$ expresses that X **refines** Y, that is, any implementation of X also implements Y. Alternatively, we say that X leaves less freedom, or is more determined than Y. The examples given in Section 2.4 further clarify this interpretation.

There are three trivial processes in $\mathcal{XDI}(I, O)$ worth mentioning. The **quiet specification** $\Box_{I,O}$ is defined by $f.\varepsilon = \Box$, and for all a, t: $f.at = \top$ if $a \in O$ and $f.at = \bot$ if $a \in I$. The quiet specifications are the only specifications with trace set $\{\varepsilon\}$. The **miracle specification** $\top_{I,O}$ is defined by $f.t = \top$ for all t. The **chaos specification** $\bot_{I,O}$ is defined by $f.t = \bot$ for all t. The miracle and chaos specifications are the only ones with an empty trace set. Note that in [20], all miracle specifications are considered equivalent, and similarly for chaos. $\mathcal{XDI}(\varnothing, \varnothing)$ contains only these three specifications. Miracle $\top_{I,O}$ is the greatest specification in $\mathcal{XDI}(I, O)$ with respect to refinement: it refines everything, hence its name (unfortunately it cannot be physically implemented). Chaos $\bot_{I,O}$ is the least specification in $\mathcal{XDI}(I, O)$ with respect to refinement: it is refined by everything, hence its name (if you ask for chaos, anything will do).

2.3. XDI State Graphs

In this subsection, we consider a fixed XDI specification $X = \langle I, O, f\rangle$. Furthermore, let $A = \mathbf{a}X = I \cup O$. All traces are in A^* unless stated otherwise.

In order to define an abstract notion of state for X, we use the **'after' operator** on the trace function. For trace t, trace function $f/t\colon A^* \to \Lambda$, pronounced as '$f$ after t', is defined by

$$(f/t).u \quad = \quad f.tu \qquad \text{(for all } u) \tag{10}$$

The 'after' operator generalizes the derivative of a trace set with respect to a trace defined in [2] and the 'after' operator on CSP processes in [6]. Note that $f/\varepsilon = f$ and $f/t/u = f/tu$. Two traces t, u are considered equivalent under X when $f/t = f/u$, that is, when $f.tv = f.uv$ for all traces v (t, u have exactly the same 'future').

The equivalence class containing trace t is completely characterized by its common trace function f/t. Each trace function, characterizing an equivalence class, is considered an **(abstract) state** of X. Note that, in general, the number of states need not be finite.

The **state graph** of X is a directed, rooted graph with Λ-labeled nodes and A-labeled arrows, defined as follows. The nodes (also called **states**) of the graph are the abstract states of X: $\{ f/t \mid t \in A^* \}$. The label of node f/t is $(f/t).\varepsilon = f.t$ (note that this is independent of the choice of representative t from the equivalence class). The labeled arrows (also called **transitions**) of the graph are given by $\{ f/t \xrightarrow{a} f/ta \mid t \in A^* \wedge a \in A \}$, that is, there is a transition from state f/t to state f/ta with label a. The root (also called **initial state**) of the graph is state $f/\varepsilon = f$.

Before showing example state graphs, it is good to note some properties (which will simplify the drawing considerably). First, on account of condition 2, there is at most one state labeled $\top$ (any trace t with $f.t = \top$ has a future completely consisting of $\top$: $(f.t)u = \top$). Similarly, condition 3 implies that there is at most one state labeled $\bot$. Therefore, we may unambiguously speak of **the $\top$-state** and **the $\bot$-state**, meaning the state with that label. Specifications $\top_{I,O}$, $\bot_{I,O}$, and $\Box_{\varnothing,\varnothing}$ have only one state; all other specifications have at least two states.

Each state has, for each symbol of A, exactly one outgoing transition. The **reduced state graph** is obtained by omitting all transitions going to either $\top$ or $\bot$, and omitting $\top$ and/or $\bot$ if they thereby have become unreachable. The omitted transitions and states can be reconstructed as follows. Add a $\top$- or $\bot$-state as needed. Consider state p without outgoing a-labeled transition. In that case, a transition to $\top$ or $\bot$ was omitted. If p has label $\top$, then the missing transition is $p \xrightarrow{a} \top$ (cf. condition 2); if p has label $\bot$, then it is $p \xrightarrow{a} \bot$ (cf. condition 3); otherwise (p labeled neither $\top$ nor $\bot$), if $a \in O$, then it is $p \xrightarrow{a} \top$ (cf. condition 4), and if $a \in I$, then it is $p \xrightarrow{a} \bot$ (cf.

condition 5):

$$\top \xleftarrow{a} \top \xleftarrow{a\in O} \nabla/\Box/\Delta \xrightarrow{a\in I} \bot \xrightarrow{a} \bot \qquad (11)$$

Condition 6 for XDI specifications can be rephrased in terms of the reduced state graph as: each ∇-labeled state has an outgoing output transition; and condition 7 as: each Δ-labeled state has an outgoing input transition.

The reduced state graph of an XDI specification, other than $\top_{I,O}$ or $\bot_{I,O}$, has the following seven properties:

1. All node labels are in Λ_{safe}.
2. For each node, the outgoing arrows have distinct labels; hence, given node p, each trace t defines at most one node q in the graph by spelling out the symbols of t as the arrow labels (if that succeeds, we write $p \xrightarrow{t} q$; note that $p \xrightarrow{\varepsilon} p$ holds for all nodes p).
3. Each node is reachable from the root via the arrows.
4. It is minimal, that is, all nodes are distinguishable, in the sense that for each pair of distinct nodes p, p' there exists a trace t such that the labels of nodes q, q' in the complete graph with $p \xrightarrow{t} q$ and $p' \xrightarrow{t} q'$ differ.
5. Each ∇-labeled node has at least one outgoing output arrow, and each Δ-labeled node has at least one outgoing input arrow.
6. There are no self-loops and no 'successive' arrows with the same label.
7. It adheres to rules $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ defined below; these capture the Neighbor-Swap Rule.

Given a directed, rooted graph with Λ-labeled nodes and A-labeled transitions satisfying the seven properties above, there exists exactly one XDI specification whose reduced state graph it is.

The **Neighbor-Swap Rule** (condition 9 for XDI specifications) can be expressed in terms of the reduced state graph. For symbols a, b having the same direction, it implies $f.tabu = f.tbau$ (apply the rule twice, once with a, b interchanged, and use antisymmetry of $\sqsupseteq$), that is, $f/tab = f/tba$. Hence, the reduced state graph satisfies

Rule $\mathcal{X}$: For all nodes p, p', and p'', and symbols a and b in the same direction, if we have $p \xrightarrow{a} p' \xrightarrow{b} p''$, then there exists a node q with $p \xrightarrow{b} q \xrightarrow{a} p''$.

This is illustrated on the top-left in Fig. 2, where the presence of the thick elements implies the presence of the thin elements in the indicated relation to each other. Rule $\mathcal{X}$ can be paraphrased as: the state reached by successive symbols of the same direction, does not depend on the order of these symbols.

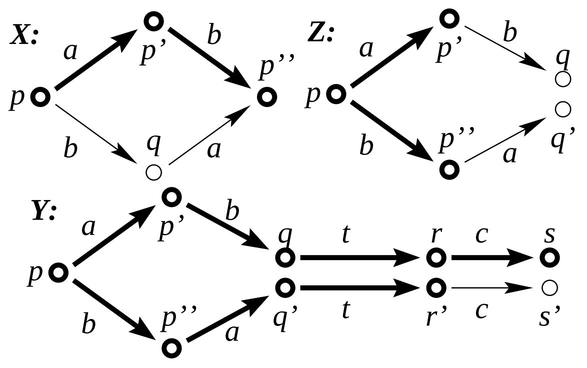

Figure 2: **Configurations in state graphs illustrating Rules $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$**

The case of symbols a, b in opposite direction is more complicated. We split it into two parts and leave out the argumentation. The easier of the two parts is

Rule $\mathcal{Z}$: For all nodes p, p', and p'', and symbols a and b in opposite direction, if we have $p \xrightarrow{a} p'$ and $p \xrightarrow{b} p''$, then there exist nodes q and q' with $p' \xrightarrow{b} q$ and $p'' \xrightarrow{a} q'$.

To paraphrase (see top-right in Fig. 2): symbols in opposite direction do not disable each other. Rule $\mathcal{Z}$ can be strengthened to **Rule** $\mathcal{Z}'$, by requiring that the if-then clause holds for *all distinct* a, b instead of just those having opposite direction. In practice, Rule $\mathcal{Z}'$ sometimes holds as well, that is, when there is no mutual exclusion between inputs, nor between outputs. Nodes q and q' may differ, but the difference is limited by (see bottom of Fig. 2)

Rule $\mathcal{Y}$: For all nodes p, p', p'', q, q', r, and r', symbols a, b, and c, and trace t with a, b in opposite direction, a, c in the same direction, and with $p \xrightarrow{a} p' \xrightarrow{b} q \xrightarrow{t} r$ and $p \xrightarrow{b} p'' \xrightarrow{a} q' \xrightarrow{t} r'$, we have

1. if $r \xrightarrow{c} s$ for some node s, then there exists a node s' with $r' \xrightarrow{c} s'$;
2. if $a \in I$, $b \in O$, and r is labeled ∇, then r' is labeled ∇ as well;
3. if $a \in O$, $b \in I$, and r is labeled Δ, then r' is labeled Δ as well.

Rule $\mathcal{Y}$ can be strengthened to **Rule** $\mathcal{Y}'$, by requiring that nodes q and q' are equal. For most—but not all—practical specifications, Rule $\mathcal{Y}'$ holds.

We adhere to the following **drawing conventions** for XDI state graphs. We draw the reduced state graph. The label of the initial state is drawn **bold-filled**. Input transitions are drawn **dashed**, and output transitions **solid**. For ease of reference, all states are numbered with a natural number, 0 for the initial state. We attempt to draw transitions with the same label in parallel. To avoid a cluttered picture, we may draw a transition not to its actual target but only show the identifying number of its target state at the arrow head. Again to avoid clutter, we may leave out some transition labels when they can be inferred from the remainder on account of the XDI conditions, in particular, the Neighbor-Swap Rule.

2.4. Examples of XDI Specifications

We consider circuits with inputs $I = \{ b, b0, b1 \}$ and outputs $O = \{ c, c0, c1 \}$ as depicted in Fig. 3.

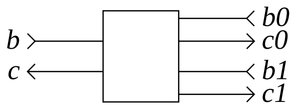

Figure 3: **Circuit interface diagram**

The first XDI specification E_Δ is given by the (reduced) state graph at the top in Fig. 4. It describes a handshake circuit (see [1]) with one passive handshake port $\{ b, c \}$ and two active handshake ports $\{ c0, b0 \}$ and $\{ c1, b1 \}$.

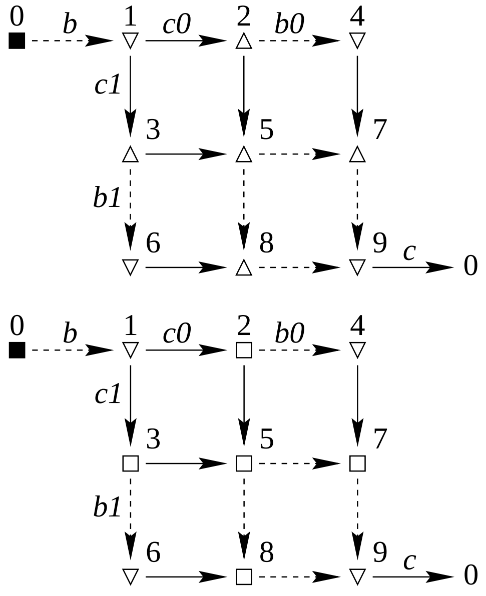

Figure 4: **State graphs for XDI specifications** E_Δ **(top) and** E **(bottom)**

Specification E_Δ can be interpreted as follows. The environment may (but need not, since the initial state is labeled indifferent) initiate a handshake on the passive port via (request) input b. The circuit must then send a request signal on at least one of the active ports via outputs *c0* or *c1*. It must do so because state 1 is labeled ∇. The circuit need not send both request signals, because states 2 and 3 are *not* labeled ∇. It may, however, send both requests, because state 2 has an outgoing *c1*-arrow and state 3 an outgoing *c0*-arrow. Because states 2, 3, 5, 7, and 8 are labeled Δ, the environment has the obligation to complete each requested handshake by acknowledging on *b0* and *b1* (which it is allowed to do, because of the appropriate outgoing arrows in those states). In fact, the environment may acknowledge each request output any time after its arrival. The circuit must issue the second request after the first was acknowledged, because states 4 and 6 are labeled ∇. The environment must postpone a new request on b until it has received acknowledge signal c from the circuit. The circuit must produce acknowledge c once it has received both acknowledges *b0* and *b1*, because state 9 is labeled ∇. After that the initial state is reached again.

XDI specification E (at the bottom in Fig. 4) is obtained from E_Δ by making the five demanding states indifferent. This relieves the environment from its obligation to complete the active handshakes, but does not change the circuit's obligations. We have $E \sqsupseteq E_\Delta$, that is, E refines E_Δ: every implementation of E also implements E_Δ. The reverse does not hold.

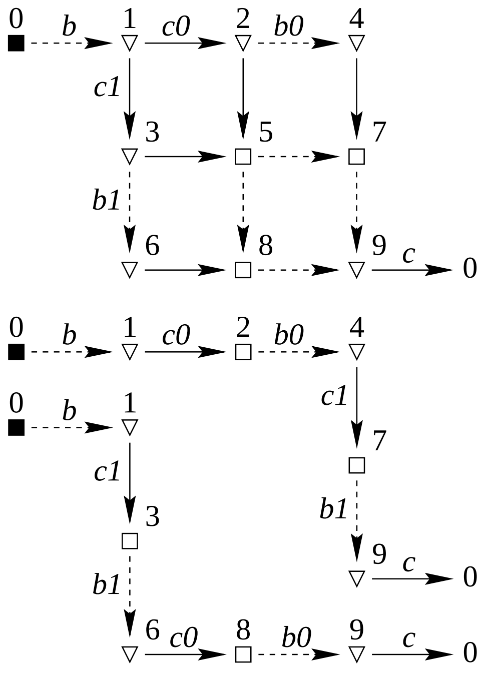

Figure 5: **State graphs for** P **(top),** S_0 **(middle) and** S_1 **(bottom)**

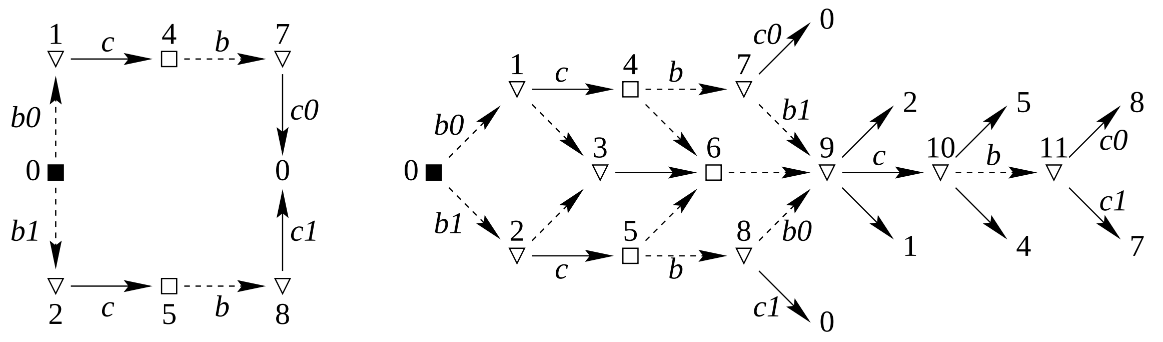

Figure 6: **State graphs for mixers M_{NR} (left) and M (right)**

Demanding states are to some extent irrelevant for circuit design; however, they play a crucial role in [20] relating refinement and parallel composition through reflection. Furthermore, they may help in verifying composite systems containing the circuit, similar to the role of preconditions in programming. Finally, recording that a state is demanding may encourage the designer to investigate the appropriateness of a dynamic rather than a static VLSI implementation.

Next we consider three related XDI specifications P, S_0, and S_1 with the same inputs and outputs as E_Δ and E (see Fig. 3). The state graphs are given in Fig. 5. P differs from E only in states 2 and 3, which are transient in P. After receiving the initial request, P must respond with requests on *both* active ports. P specifies the handshake PAR-element defined in [1].

Similarly, specifications S_0 and S_1 describe SEQ-elements of [1], which sequence the active handshakes within the passive handshake. The difference between S_0 and S_1 is the order of the passive handshakes. S_0 does $\{\,c0, b0\,\}$ before $\{\,c1, b1\,\}$, whereas S_1 does the reverse order.

We have $P \sqsupseteq E$, $S_0 \sqsupseteq E$, and $S_1 \sqsupseteq E$. Thus, E can be implemented by either of P, S_0, or S_1. Note that P, S_0, and S_1 are mutually incomparable under $\sqsupseteq$. Specifications E_Δ, E, P, S_0, and S_1 all satisfy the stronger Rules $\mathcal{Y}'$ and $\mathcal{Z}'$.

Note that E does not suffice to specify a PAR-element, because E may deadlock in an environment that insists on receiving both requests before acknowledging either of them, whereas a PAR-element should not do so. For all one knows, E could have been implemented by an SEQ-element. Consider, for instance, an environment with communication between the components attached to the active ports, such as a C-element with inputs $\{\,c0, c1\,\}$ and forked outputs $\{\,b0, b1\,\}$. Specification E is, however, useful to postpone a design decision as illustrated by a decomposition for the Decision-Wait given in [19]. E only specifies that every passive handshake surrounds handshakes on *each* of the active ports, without imposing any order restrictions.

The following two specifications have yet again the same interface (see Fig. 3), but they use it differently. The state graphs of these specifications are given in Fig. 6. On the left is specification M_{NR} for the Non-Receptive Mixer of [1], having two passive ports $\{\,b0, c0\,\}$ and $\{\,b1, c1\,\}$, and one active port $\{\,c, b\,\}$. The environment *chooses* to request a handshake on *one* of the passive ports (after a request on either *b0* or *b1*, states 1 and 2 allow no further input). The circuit then must start a handshake on the active port, by outputting on *c*, and, after receiving acknowledge *b*, complete the passive handshake (states 1, 2, 7, and 8 are labeled ∇). Specification M_{NR} satisfies the stronger Rule $\mathcal{Y}'$ but not $\mathcal{Z}'$ (inputs *b0* and *b1* disable each other in the initial state).

Specification M (Fig. 6, right) describes the (Receptive) Mixer of [1], with the same port structure as M_{NR}. In this case, the environment may initiate handshakes on *both* passive ports (states 1 and 2 now allow further input, leading to the 'new' state 3, where both requests are pending). For each handshake on a passive port, the circuit must initiate a handshake on the active port. In case of contention, the circuit must order (by arbitration) the active handshakes and the completions of the passive handshakes. Specification M satisfies the stronger Rule $\mathcal{Y}'$ but not $\mathcal{Z}'$ (outputs *c0* and *c1* disable each other in states 9 and 10). Note that M refines M_{NR}. The state graph of M is harder to grasp than the others. We will get back to it in Section 3.4.

Our final example illustrates the relevance of Rule $\mathcal{Y}$. Consider specification Y, whose state graph appears on the left in Fig. 7. Depending on the order in which input a and output b occur, either state 4 (input before output) or state 5 (output before input) is reached. Hence, Y does not satisfy the stronger Rule $\mathcal{Y}'$. State 4 cannot be labeled ∇ (to 'enforce' output c; also see operator Φ in Section 3.4), because in that case Rule $\mathcal{Y}$ would be violated: by require-

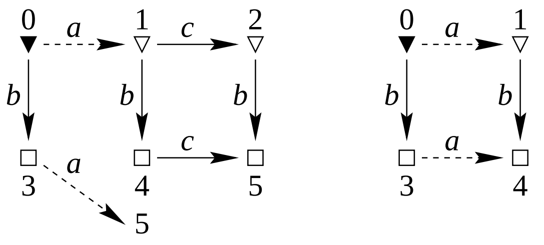

Figure 7: **State graphs for specifications** Y **(left) and** Y' **(right)**

ment 2 of $\mathcal{Y}$ state 5 should then also be transient, but it has no outgoing output arrows, thus violating XDI condition 6.

Specification Y may seem a bit contrived, but this same configuration 'naturally' appears in the state graph for the Nacking Arbiter considered below. Violation of Rule $\mathcal{Y}'$ is inherently tied to nondeterminism and it is hard to understand without taking progress into account. Note that specification Y' (Fig. 7, right) satisfies both Rule $\mathcal{Z}'$ and $\mathcal{Y}'$, and it refines Y: $Y' \sqsupseteq Y$.

3. Analyzing XDI Specifications

Two reasons to analyze specifications are validation and design guidance (see Fig. 8). When developing a new circuit, it is good practice to establish its specification first. For complicated circuits, it is not immediately obvious that a tentative specification is indeed as desired. During validation, key properties of the specification are examined to help eliminate errors and build confidence at an early stage.

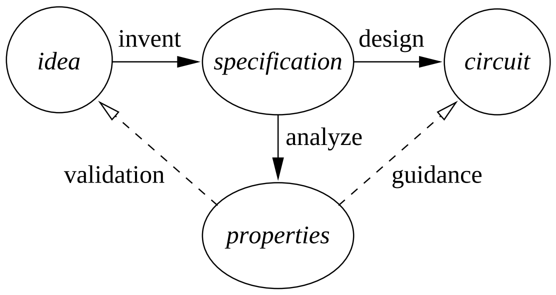

Figure 8: **The role of specification analysis**

When designing a circuit implementation from a specification, it may again be helpful to know the key properties of the specification, because they may provide guidance in making appropriate design decisions.

As an additional example, we consider the Nacking Arbiter, whose interface is given in Fig. 9. The idea is that there are two parties, one using $\{\,r0, a0, n0\,\}$ and the other $\{\,r1, a1, n1\,\}$, where each party either cycles through r_i, a_i, r_i, a_i (for 'request', 'grant', 'release', 'acknowledge') or r_i, n_i (for 'request', 'negative ack'), such that the (resource) grants are mutually exclusive. The negative ack should be given to a request when the resource has already been granted. It is surprisingly hard to give an XDI specification *Narb* that captures all requirements. When drawing a state graph of more than a few states, one already feels the need for some help to make sure that it properly captures all intended restrictions.

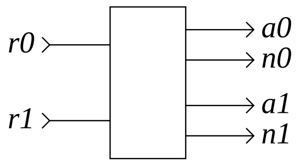

Figure 9: **Interface for Nacking Arbiter**

In this section, we look at four important classes of properties of XDI specifications that can be used for validation and design guidance. A tool has been built to extract these properties from finite XDI state graphs. It is explained and applied in [19].

3.1. Automorphisms

An **automorphism** of XDI specification $X = \langle I, O, f \rangle$ consists of an alphabet permutation φ and a state permutation ψ preserving symbol direction (i.e., $\varphi.I = I$ and $\varphi.O = O$) and state labeling (i.e., $(\psi.p).\varepsilon = p.\varepsilon$; recall that a state is a trace function p and that its label is $p.\varepsilon$). In terms of the (reduced) state graph of X, an automorphism is a graph isomorphism between the state graph and itself, preserving symbol direction and state labeling. Note that an automorphism need not map the initial state to itself.

State permutation ψ is completely determined by φ and $\psi.f$, the image of the initial state, because $\psi.(f/t) = (\psi.f)/(\varphi.t)$, where $\varphi.t$ is obtained from t by applying φ to each of the symbols in t.

The set of all automorphisms of X forms a (mathematical) group under permutation composition. It is called the **automorphism group**, and captures the **symmetries** in the specification. Example specifications E_Δ, E, P, M_{NR}, and M of Section 2.4, all have the same automorphism group consisting of two elements, viz. the identity automorphism (with $\varphi.a = a$ and $\psi.p = p$) and the automorphism that exchanges the roles of *b0*, *c0* with *b1*, *c1* and leaves the states unchanged. These automorphisms can be tabulated as follows:

0	b	c	*b0*	*c0*	*b1*	*c1*
0	b	c	*b1*	*c1*	*b0*	*c0*

(12)

Each row lists the image of the initial state and the images of all symbols. The identity automorphism is always present and listed in the first row as reference. Specifications S_0 and S_1 have an automorphism group of three elements. For S_0 it is:

$$\begin{array}{c|cccccc} 0 & b & c & b0 & c0 & b1 & c1 \\ \hline 2 & b0 & c0 & b1 & c1 & b & c \\ 7 & b1 & c1 & b & c & b0 & c0 \end{array} \qquad (13)$$

This reveals a three-fold symmetry that may guide the design process. Specifications Y and Y' (Fig. 7) have only one automorphism: the identity.

Certain desirable symmetries of a specification are often known beforehand. One way of validating a specification is to corroborate these symmetries. For example, the specification *Narb* for the Nacking Arbiter is expected to be symmetric in the two parties. It should therefore have two automorphisms: the identity, and the exchange of 0 and 1.

3.2. Independent Environment Partitions

Some circuits need to operate in an environment split into independent parts. This is, for instance, the case for handshake circuits, whose handshake ports are connected independently. Each part of the environment communicates with the circuit without directly seeing the communications of the other parts. To guarantee error-free operation, in spite of the more limited information that each part of the split environment has, the specification must be set up properly. Partitions into two independent parts were introduced in [17].

Given XDI specification $X = \langle I, O, f \rangle$ and partition $\mathcal{P}$ of $\mathbf{a}X$, the following properties are necessary, though not sufficient, for independence of $\mathcal{P}$:

1. Disabling inputs belong to the same part, that is, if for states p, p', p'' in the reduced state graph and symbols $a, b \in I$ we have $p \xrightarrow{a} p'$, $p \xrightarrow{b} p''$, and $p'.b \notin \Lambda_{safe}$, then a, b belong to the same part of $\mathcal{P}$.

2. An input and its enabling outputs belong to the same part, that is, if for states p, p', p'' in the reduced state graph and symbols $a \in O$, $b \in I$ we have $p \xrightarrow{a} p'$, $p' \xrightarrow{b} p''$, and $p.b \notin \Lambda_{safe}$, then a, b belong to the same part of $\mathcal{P}$.

These properties are easy to verify. In fact, a union-find algorithm can construct the *finest* partition satisfying the two properties above, called the **finest semi-independent partition (FSP)**. XDI specifications E_Δ, E, P, S_0, S_1, and M of Section 2.4 all have FSP $\{\{b, c\}, \{b0, c0\}, \{b1, c1\}\}$, as is to be expected for a handshake circuit. Note that specification M_{NR} (Fig. 6) has FSP $\{\{b, c\}, \{b0, c0, b1, c1\}\}$: since inputs $b0$ and $b1$ disable each other initially, the environment cannot operate the two passive handshake ports of M_{NR} completely independently.

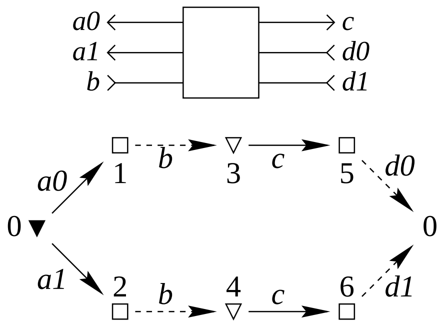

Figure 10: **XDI specification F: counterexample for independence of FSP**

The two properties above are not sufficient to guarantee complete independence of the parts, as is shown by the following counterexample. XDI specification F (see Fig. 10) has FSP $\{\{a0, a1, b\}, \{c, d0, d1\}\}$, but in order to send input $d0$ or $d1$ without causing interference, the environment must observe output $a0$ or $a1$, and output c.

Determining the FSP can provide insight into a specification, as in the case of the Nacking Arbiter. In [9], a DI Algebra specification is given:

$$\begin{aligned} NA &= [r0? \to a0!; G0 \,\Box\, r1? \to a1!; G1] \\ G0 &= [r0? \to a0!; NA \,\Box\, r1? \to n1!; G0] \\ G1 &= [r0? \to n0!; G1 \,\Box\, r1? \to a1!; NA] \end{aligned}$$

It happens to differ from the XDI specification given in EDIS [19] (also see Fig. 11), which has 28 states. The DI Algebra specification *NA*, when converted into XDI by '`DIGG`' [13], has 30 states. It turns out that *NA* has FSP $\{\{r0, a0, n0, r1, a1, n1\}\}$ whereas *Narb* has $\{\{r0, a0, n0\}, \{r1, a1, n1\}\}$. That is, in the DI Algebra version, the handshake ports are not independent. Indeed, Handshake Algebra [11, 8] was developed specifically to avoid these kinds of over-constrained specifications that arise in DI Algebra.

3.3. Autocomparison Matrix

It is relatively straightforward to decide the refinement relation between two specifications given by state graphs. For specifications $X = \langle I, O, f \rangle$ and $X' = \langle I, O, f' \rangle$, we are interested in determining $X \sqsupseteq X'$, that is $f \sqsupseteq f'$. It is convenient to generalize the comparison problem to determining

$$f/t \sqsupseteq f'/t' \qquad (14)$$

for *all* states f/t and f'/t', because of the recurrence relation

$$f/t \sqsupseteq f'/t' \quad \equiv \quad \begin{cases} f.t \sqsupseteq f'.t' \;\wedge \\ (\forall a : f/ta \sqsupseteq f'/t'a) \end{cases} \qquad (15)$$

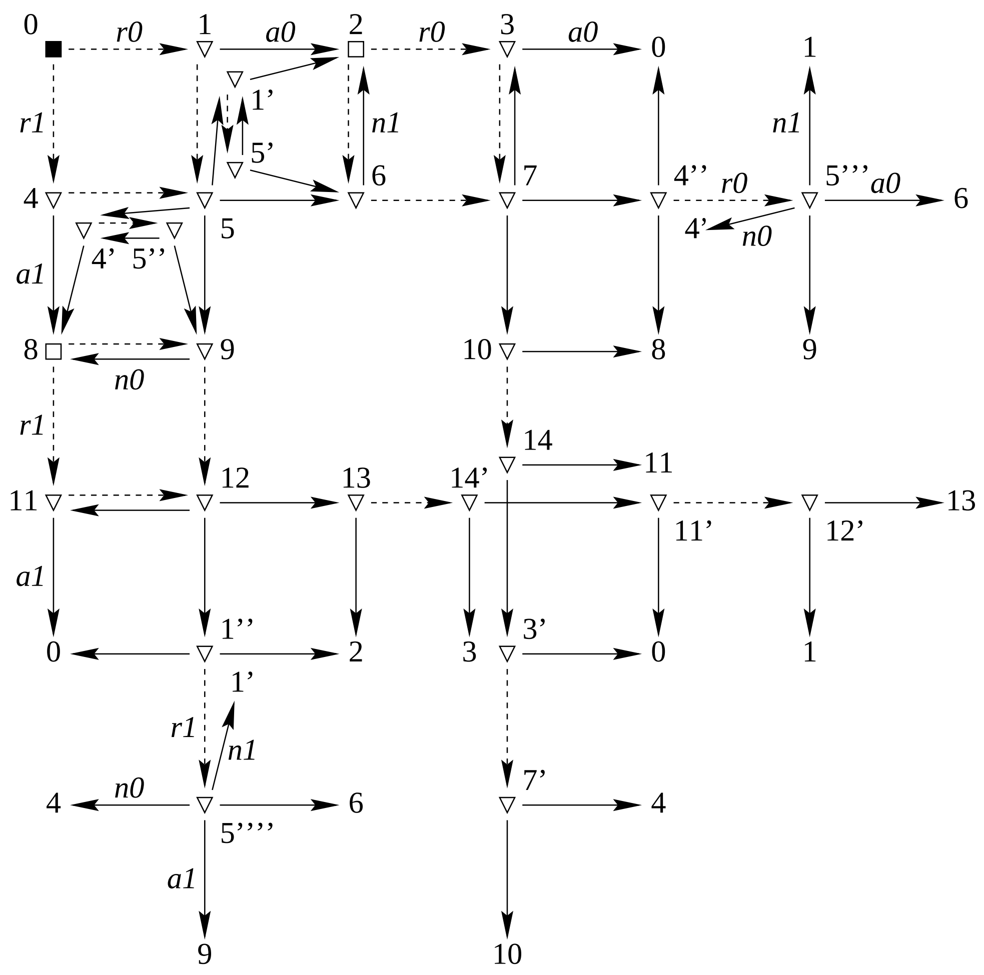

Figure 11: **State graph for Nacking Arbiter**

The answer to the original comparison problem is found by taking $t = t' = \varepsilon$ in (14). The result of (14) can be tabulated in a (comparison) matrix, with rows indexed by the states f/t of X and columns indexed by the states f'/t' of X'. When this is done for $X = X'$ one obtains, what we call, the **autocomparison matrix** of X, containing the value of $f/t \sqsupseteq f/t'$ for all state pairs. The diagonal is not interesting as it is always *true*. The rest, however, is useful to verify minimality:

> for distinct states p, q, we should have $p \not\sqsupseteq q \ \vee\ q \not\sqsupseteq p$,

and the Neighbor-Swap Rule

> for all states p and symbols a, b with $a \in O \ \vee\ b \in I$, we should have $p/ab \sqsupseteq p/ba$.

Furthermore, if we have

$$a \in O \ \wedge\ b \in I \ \wedge\ p/ba \not\sqsupseteq p/ab \qquad (16)$$

then Rule $\mathcal{Y}'$ does not hold, which says something about nondeterminism (see next section). As an example, we give the autocomparison matrix for specification Y in Fig. 7 ($+ =$ *true* and $- =$ *false*):

$p \sqsupseteq q$	0	1	2	3	4	5
0	+	+	+	−	−	−
1	−	+	−	−	−	−
2	−	+	+	−	−	−
3	−	−	−	+	+	+
4	−	−	−	−	+	−
5	−	−	−	−	+	+

In particular, the comparison of states 4 and 5 shows that Rule $\mathcal{Y}'$ is not satisfied. This situation also arises in the Nacking Arbiter, but this is not easy to verify manually.

3.4. Choice, Order Dependence, and Nondeterminism

We classify the various kinds of choice, order dependence, and nondeterminism that can be present in XDI specifications and mention some relevant theorems. For XDI specification $X = \langle I, O, f\rangle$, we define several predicates. X is **maximally transient** ([20]), if, whenever it *may* do output, it *must* do so; that is, when for all traces t and symbols a we have

$$ta \in \mathbf{t}X \ \wedge \ a \in O \quad \Rightarrow \quad f.t = \nabla \tag{17}$$

A specification that is *not* maximally transient leaves a choice between doing output or waiting (for input). All example specifications—*except* E_Δ, E (Fig. 4, states 2, 3), and Y (Fig. 7, state 4)—are maximally transient (see Table 1).

Spec	max^∇	$\mathcal{Z}^{out}$	$\mathcal{Z}^{inp}$	$\mathcal{Y}^{out}$	$\mathcal{Y}^{inp}$
E_Δ	**no**	*yes*	*yes*	*yes*	*yes*
E	**no**	*yes*	*yes*	*yes*	*yes*
P	*yes*	*yes*	*yes*	*yes*	*yes*
S_0	*yes*	*yes*	*yes*	*yes*	*yes*
S_1	*yes*	*yes*	*yes*	*yes*	*yes*
M_{NR}	*yes*	*yes*	**no**	*yes*	*yes*
M	*yes*	**no**	*yes*	*yes*	*yes*
Y	**no**	*yes*	*yes*	*yes*	*yes*
Y'	*yes*	*yes*	*yes*	**no**	*yes*
F	*yes*	**no**	*yes*	*yes*	*yes*
Narb	*yes*	**no**	*yes*	**no**	*yes*

Table 1: Classification of example specifications

X has **no disabling** ([17]) between symbols in $A \subseteq \mathbf{a}X$ when for all traces t and distinct symbols $a, b \in A$ we have

$$ta \in \mathbf{t}X \ \wedge \ tb \in \mathbf{t}X \quad \Rightarrow \quad tab \in \mathbf{t}X \tag{18}$$

If X has no disabling outputs, we say it satisfies **Rule** $\mathcal{Z}^{out}$. Dually, **Rule** $\mathcal{Z}^{inp}$ excludes disabling inputs. Disabling inputs require a choice from the environment. A circuit must somehow implement a choice between disabling outputs. Rule $\mathcal{Z}^{inp}$ is violated only by example specification M_{NR} (Fig. 6, state 0), and $\mathcal{Z}^{out}$ only by M (Fig. 6, states 9, 10), F (Fig. 10, state 0), and *Narb* (Fig. 11, e.g. state 5). We have

$$\mathcal{Z}' \quad \equiv \quad \mathcal{Z} \ \wedge \ \mathcal{Z}^{inp} \ \wedge \ \mathcal{Z}^{out} \tag{19}$$

X is called **(output) deterministic** ([20]) when it is maximally transient and has no disabling outputs. In that case, the output obligations leave no choice (other than order). Only example specifications P, S_0, S_1 (Fig. 5), M_{NR} (Fig. 6), and Y' (Fig. 7) are output deterministic.

The following rules classify input-output order dependence ([20]). X satisfies **Rule** $\mathcal{Y}^{out}$ when for all traces t, u and symbols $a \in I$, $b, c \in O$ we have

$$tabuc \in \mathbf{t}X \ \wedge \ tbau \in \mathbf{t}X \quad \Rightarrow \quad tbauc \in \mathbf{t}X \tag{20}$$
$$tabu \in \mathbf{t}X \ \wedge \ f.tbau = \nabla \quad \Rightarrow \quad f.tabu = \nabla \tag{21}$$

Dually, X satisfies **Rule** $\mathcal{Y}^{inp}$ if $\backsim X$ satisfies $\mathcal{Y}^{out}$. Similar to (19), we have

$$\mathcal{Y}' \quad \equiv \quad \mathcal{Y} \ \wedge \ \mathcal{Y}^{inp} \ \wedge \ \mathcal{Y}^{out} \tag{22}$$

All example specifications satisfy Rule $\mathcal{Y}^{inp}$, and $\mathcal{Y}^{out}$ is violated by Y (Fig. 7, output c is enabled in state 4 but not in 5) and *Narb* (Fig. 11, states 3 and 11). Every output deterministic specification satisfies $\mathcal{Y}^{out}$ (theorem in [20]). In fact, an even stronger statement holds. Define operator Φ on specifications by trace functions by

$$\Phi.\langle I, O, f\rangle \quad = \quad \langle I, O, f'\rangle \tag{23}$$

where

$$f'.t \quad = \quad \begin{cases} \nabla \ \text{ if } (\exists\, a \in O : f.ta \in \Lambda_{safe}) \\ f.t \text{ if } (\forall\, a \in O : f.ta \notin \Lambda_{safe}) \end{cases} \tag{24}$$

that is, $f'.t$ yields ∇ whenever that does not violate condition 6 and otherwise $f.t$. We have

$$\text{‘}X \text{ satisfies } \mathcal{Y}^{out}\text{’} \quad \equiv \quad \Phi.\langle I, O, f\rangle \in \mathcal{XDI} \tag{25}$$

Example Y (Fig. 7) indeed satisfies the right-hand side of (25).

X has **static (output) nondeterminism** ([20]). when it is not deterministic but has an output deterministic refinement; the latter means a deterministic specification X' exists with $X' \sqsupseteq X$. X has **dynamic (output) nondeterminism** ([20]), when it has no deterministic refinement. Since refinement is reflexive, this also implies that X is not deterministic. Nondeterministic example specifications E_Δ, E, Y, and F have static nondeterminism, since they have deterministic refinements.

In CSP [6], *every* specification has at least one deterministic refinement. This is not the case in $\mathcal{XDI}$. Example specification M (Fig. 6) has dynamic nondeterminism, which can be verified as follows. A nondeterministic specification can be refined into something more deterministic in only a few ways:

1. In a non-transient state allowing output, one may either (a) omit some outputs, or (b) guarantee output by making the state transient.

2. In a transient state with disabling outputs, one may omit all but one of the mutually exclusive outputs.

When making such changes one must also see to it that the resulting state graph is valid.

M fails to be deterministic because of disabling outputs *c0* and *c1* in states 9 and 10. If we omit output *c*0 in state 9, then it must also be omitted from state 7 (cf.

Rule $\mathcal{Z}$), which then must be relabeled $\Box$ or less (cf. condition 6), which contradicts the original specification. Therefore, *M* has no deterministic refinement. The same reasoning holds for *Narb*, which has dynamic nondeterminism as well.

Static output nondeterminism can be eliminated by the designer. The nondeterminism is then resolved at **design-time**. Dynamic output nondeterminism can only be resolved at **run-time**. It is intertwined with the *behavior* of the environment, hence the name. Whether or not such nondeterminism is actually encountered during operation depends on the interaction between the circuit and its environment. For instance, the operation of *M* is deterministic as long as no more than one request on *b0*, *b1* is made at a time. Nondeterminism only plays a role when two requests are made "simultaneously". It cannot be resolved at design-time precisely because simultaneity is not a well-definable concept under DI operation. Dynamic nondeterminism in a specification indicates that some form of **arbitration** plays a role. A relation to 'confusion' in Petri nets [15] is suspected.

Giving a simple efficient algorithm for recognizing dynamic nondeterminism is still an open problem. We do have the following theorem ([20]): A specification satisfying Rules $\mathcal{Y}^{out}$ and $\mathcal{Z}^{out}$ has a deterministic refinement and, consequently, no dynamic nondeterminism.

4. Concluding Remarks

We have presented the XDI Model for specifying delay-insensitive circuits. The major improvement over Udding's model is that it allows the expression of progress requirements, both for the circuit and for its environment. A set of conditions has been formulated that an XDI specification must satisfy in order to be acceptable. XDI specifications can be visualized as labeled state graphs. The state labels distinguish between: no progress obligation (indifferent: $\Box$), a progress obligation for the environment (demanding: Δ), and a progress obligation for the circuit (transient: ∇). Since the model uses finite traces, it is not suitable for specifying fairness concerns.

Furthermore, we presented several classes of specification properties that can be interesting for the purpose of validation (when inventing a specification) or design (when implementing a specification). These properties can be automatically extracted from finite state graphs.

A tool has been developed to validate, analyze, and compare XDI state graphs expressed in AND/IF (Andover Interchange Format for the description of finite automata; see [19].) This tool is being applied in the development of the Encyclopedia of Delay-Insensitive Systems [19]. It is available through an e-mail interface described in [19]. All specifications and refinements in this paper have been verified by the tool.

Interesting topics for further work are the partitioning of environments into independent (not just semi-independent) parts and the distinction between static and dynamic nondeterminism. Concerning the latter, it is particularly interesting to study the relations with 'confusion' in Petri nets and to find an efficient algorithm to recognize dynamic nondeterminism in specifications.

Acknowledgments

I would like to thank Rudolf Mak for valuable comments on this paper. I also appreciated the many detailed suggestions provided by the anonymous reviewers.

References

[1] K. van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. International Series on Parallel Computing. Cambridge University Press, 1993.

[2] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[3] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[4] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[5] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[7] M. B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992.

[8] M. B. Josephs, P. G. Lucassen, J. T. Udding, and T. Verhoeff. Formal design of an asynchronous DSP counterflow pipeline: A case study in handshake algebra. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 206–215, Nov. 1994.

[9] M. B. Josephs and J. T. Udding. Delay-insensitive circuits: An algebraic approach to their design. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 342–366. Springer-Verlag, Aug. 1990.

[10] M. B. Josephs and J. T. Udding. An overview of DI algebra. In *Proc. Hawaii International Conf. System*

Sciences, volume I. IEEE Computer Society Press, Jan. 1993.

[11] M. B. Josephs, J. T. Udding, and Y. Yantchev. Handshake algebra. Technical Report SBU-CISM-93-1, School of Computing, Information Systems and Mathematics, South Bank University, London, Dec. 1993.

[12] P. G. Lucassen. *A Denotational Model and Composition Theorems for a Calculus of Delay-Insensitive Specifications*. PhD thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, May 1994.

[13] W. Mallon and J. T. Udding. Building finite automatons from DI specifications. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, 1998.

[14] J. O'Leary. *A Model and Proof Technique for Verifying Hardware Compilers for Communicating Processes*. PhD thesis, School of Electrical Engineering, Cornell University, Aug. 1995.

[15] W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

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

[17] J. T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1984.

[18] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[19] T. Verhoeff. Encyclopedia of delay-insensitive systems (EDIS). `http://edis.win.tue.nl/`

[20] T. Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994.

[21] R. van de Wiel. *Testing Handshake Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1998 (in preparation).