# Some Recent Asynchronous System Design Methodologies

Ganesh Gopalakrishnan and Prabhat Jain

UUCS-90-016

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

September 26, 1990

## Abstract

We present an in-depth study of some techniques for asynchronous system design, analysis, and verification. After defining basic terminology, we take one simple example - a four-phase to two-phase converter - and present its design using (a) classical flow-tables; (b) Signal Transition Graphs of [8]; and (c) Trace Theory of [15]. We then present necessary and sufficient conditions for Delay Insensitivity, proposed by [38], and illustrate it on our example. Finally, we present the work of [13] on the verification of asynchronous circuits, and illustrate it on the circuits derived in the paper. The following points are emphasized: (i) presentation of techniques at more depth than in a general survey; (ii) illustration of all the aspects discussed on a common example; (iii) comparative study of the works presented. Many interesting works had to be left out, solely because of our lack of space and time.

## Contents

## List of Figures

## List of Tables

# 1 Introduction

Recently there has been renewed interest in asynchronous digital circuit design. This revival of interest seems to be largely due to the following reasons: (i) some recent publications (e.g. [35]) have provided the much needed inspiration; (ii) some large designs have been automatically synthesized (e.g. [27]); (iii) mathematical techniques for reasoning about asynchronous behavior have matured (e.g. [38,15,13]). Among the claimed advantages of asynchronous design are freedom from the constraint of lockstep synchronous execution where the clock cycle has to accommodate the slowest combinational path, ease of translation of a problem in a distributed computation, and even self diagnosis[11]. The reader is referred to [35] [28, Chapter 7] for excellent introductions to asynchronous design.

In this paper, we provide a glimpse of this research area by taking one simple example and lead the reader through various synthesis and verification techniques. The paper is organized as follows. We first present some basics of asynchronous systems. We then introduce our main example: a four-phase to two-phase converter with "quick return" (QR42[1]). We present the design of this example using:

- "Classical approaches" [16];
- Signal Transition Graphs [8];
- Trace Theory [15];

We then present some techniques for reasoning about and verifying asynchronous designs, based on the works of Ebergen[15] and Dill[13].

The following points are emphasized: (i) presentation of techniques at more depth than in a general survey; (ii) illustrating all the aspects discussed on a common example; (iii) comparative study of the works presented. Many interesting works had to be left out, solely because of our lack of space and time. (Some of these works are mentioned among our concluding remarks.) We have nothing to say (for or against) works that have not been surveyed here, and hope that they will be similarly studied elsewhere.

Finally, many of the terms that we have used here, and whose definitions we have paraphrased, have been used by past researchers. Wherever possible, we have cited researchers to whom the term is attributed, or in whose work we saw the term cited. In a few instances, we have to refer the reader to the "prevelant folklore" of this subject area.

# 2 Possible Meanings of "Synchronous" and "Asynchronous"

The words "asynchronous" and "synchronous" have acquired different meanings in hardware and software design. These different usages are beginning to be of concern during

[1] We were introduced to this example by Jo Ebergen.

modern times, because software specification and compilation techniques are now being used for hardware design also. Hence, researchers with a hardware background can end up confusing their counterparts with a software background. We do not attempt to provide the final answer here, but merely provide *working definitions* so that this paper will hopefully avoid certain confusions amongst readers who are primarily software oriented or primarily hardware oriented.

There are *two* aspects of a system that can be either asynchronous or synchronous: *computational steps* and *communication* [34].

**Definition 1** *A collection of subsystems within a system are* synchronous with respect to their computation *if step $M+1$ of computation in one system cannot occur unless step $M$ has occurred within all other systems. Example 1: Synchronous hardware. Example 2: Systolic arrays [40].*

**Definition 2** *A collection of subsystems within a system are* asynchronous with respect to their computation *if the computational rates of the systems are not* directly constrained *in any way. (However, they may be indirectly constrained by the communications.) Example 1: Asynchronous hardware. Example 2: Distributed implementations of CSP[19] or Ada[29] programs.*

**Definition 3** *A collection of systems are* synchronous with respect to their communication *if a set of $N$ processes wishing to communicate with each other are required to* rendezvous*; in other words, they are forced to communicate through a* zero sized buffer*; thus, all the senders and receivers must wait for each other to reach a common control state, and then "physically hand over data" through a memoryless medium (such as a wire). Example 1:* Rendezvous *of the language Ada, or CSP. For both Ada and CSP, $N = 2$. (Languages with $N > 2$ are said to support multi-way rendezvous.) Example 2: Synchronous hardware systems. Synchronous hardware systems actually engage in a rendezvous communication "implicitly"—by being in the "right pair of states at the right time".*

**Definition 4** *A collection of systems are* asynchronous with respect to their communication *if all the members of a set of processes wishing to communicate with each other need not wait for each other. Example: "wait free" communication supported by the use of buffers,* the presence of which is not reflected in the semantics of the language. *Example: The Kahn model[23] of functional multiprocessing, where processes communicate through infinite* streams.

**Definition 5** *For the purposes of this paper, we define a synchronous (hardware) system to one that is synchronous with respect to computation and communication, and an asynchronous (hardware) system to be one that is synchronous with respect to communication and asynchronous with respect to computation.*

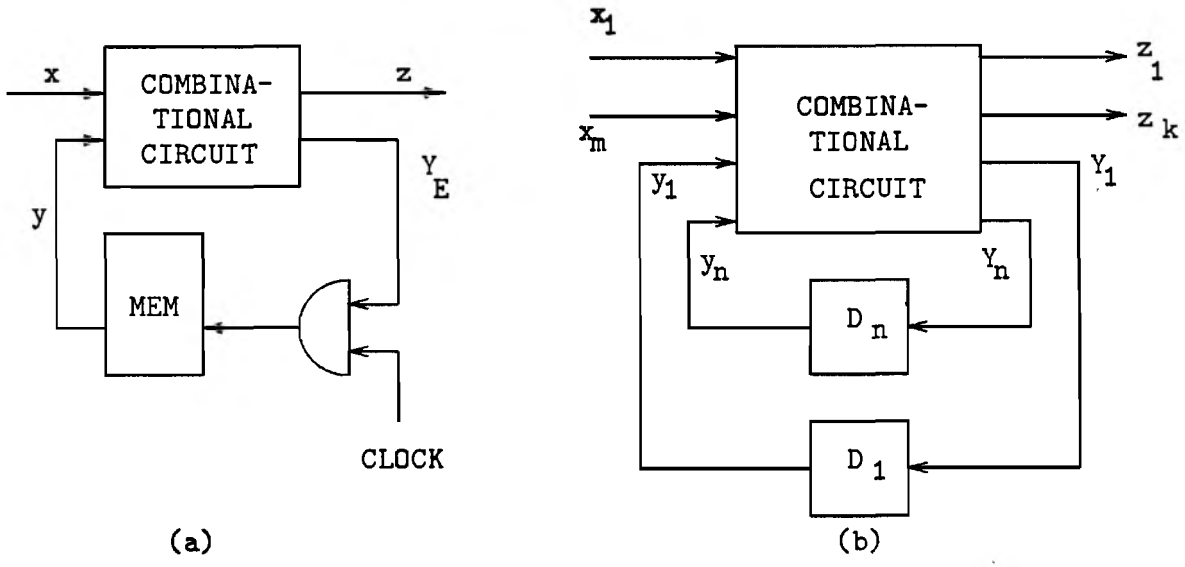


Figure 1: Models of Synchronous (a) and Asynchronous (b) Systems

## 3 Basics of Asynchronous Hardware Systems

By examining the general block diagram of sequential systems given in figure 1 for both synchronous and asynchronous systems, important differences between them can be pointed out. In synchronous systems operating on one central fixed-period clock, the clock period must be greater than the delay of the slowest combinational path that would materialize at least once during the computation. By virtue of this feature, *hazards*[16] occurring on both the state-feedback path as well as the outputs can be ignored (assuming that the outputs of the *entire system* are synchronously sampled). In asynchronous systems, the computational rate can vary across the spatial and temporal dimensions of the system. In asynchronous digital systems that do not rely on any absolute delays (such as absolute delay bounds for gates or wires), the computational rate is governed only by (a) the *synchronization skeleton*[25]—i.e. a graph of action dependencies on (other) actions; and (b) the rate at which *exogenous[8]* (externally generated) data arrives into the system.

The last sentence points to the existence of many different *styles* of asynchronous design. We now present some widely studied classification schemes. These schemes themselves are related to each other in many (complex) ways.

### 3.1 Fundamental .vs. Input-Output Mode

An asynchronous system is operated in the *fundamental mode* if the next exogenous excitation is not applied until every node of the system has updated itself and reached a steady state following the current exogenous excitation[16]. (The method to be employed to detect the attainment of steady state is not prescribed.) An asynchronous system is operated in the *Input-Output* (I/O) mode if it is permitted to apply the next exogenous excitation immediately after any activity is observed on any one of the output wires in response to the currently applied exogenous excitation.

There are many suggestive examples that indicate the limits as well as ramifications of fundamental and I/O mode operation. There are also many specific results that tell that for certain well-known components, the I/O mode of operation is not safe; for example, the most widely used asynchronous system building block—a C-element—can fail when operated under the I/O mode[5]. In practice, such failures can be avoided by resorting to knowledge about absolute gate delays[5].

### 3.2 Speed Independent .vs. Delay-Insensitive

The structure of a digital system can be viewed as a tuple $(M, i, o, N)$ where $M$ is a set of *modules*; $i(m)$, $m \in M$, are the *input ports* of module $m$, and $o(m)$ its output ports, and $N \subseteq \mathcal{P}(\cup_{m \in M} (i(m) \cup o(m)))$ are *nodes*—subsets of ports that are electrically connected. Each module is a black-box about whose innards nothing is known.

Predicate *speed-independent* is true of a system if, under the assumption that wires have zero delay, the system implements a certain logical behavior $B$ independent of the actual delays of the modules themselves. Predicate *delay-insensitive* is true of systems that implement a certain logical behavior $B$ independent of the actual delays of the modules as well as wires. We propose two memory aids to make these two terms more mnemonic: Speed-Independent = Module-Speed-Independent, and Delay-Insensitive = Module-and-Wire-delay-insensitive.

Unfortunately, the above statements are far from being precise definitions! In order to provide precise definitions, it is necessary to have a semantics for the specification language in question, so that the concept of "a behavior $B$" can be defined. Much controversy exists in the field currently, because there is often a lack of either clear understanding or agreement as to what the term "behavior of an asynchronous system" or "delay" means *in general.*

The situation has improved considerably, of late. Many theoretical models of asynchronous system behavior have been very carefully defined. Precise definitions of speed independence and delay insensitivity have been provided in the context of these works. Two recent important results are now listed. Udding [38] has identified four *necessary and sufficient* conditions for delay-insensitive signaling. (These conditions will be illustrated in section 9.) Brzozowski and Ebergen[4] have shown the impossibility of realizing certain basic components in a purely delay-insensitive manner if only gates may be used.

# 4 Terminology and Notations

A *signal* refers to a wire that carries a high (logical 1) or low (logical 0) voltage. If $s$ is a signal, $s+$ is a *signal transition* where $s$ is set to high, and $s-$ is a signal transition where $s$ is set to low. $s+$ is said to be the complement signal transition of $s-$ (and vice versa). If $s$ is an output signal, then $s+$ is read "generate $s$ going high". If $s$ is an input signal, $\underline{s+}$ is read "await $s$ going high"[8]. (However, when talking about signals in general (without considering their directions), we shall omit the underlines.)

A sequence $a+$; $b-$ indicates that $b-$ should come after $a+$. The notation $c+ \parallel d-$ says that $c+$ and $d-$ can occur in parallel. The notation $[a]^*$ denotes zero or more repetitions of $a$. The notations ';' and '||' can be combined, using parentheses if necessary, to specify partial orderings among a set of signals. Given two partial orders $R_1$ and $R_2$ over a set, $R_1$ is an *augment* of $R_2$ if $R_1$ is equally constrained, or more constrained than $R_2$. In other words, the *graph* of $R_1$ has the same or more edges than the graph of $R_2$. When $R_1$ is an *augment* of $R_2$, $R_2$ is said to *subsume* $R_1$[32].

In asynchronous systems, modules obey various signaling protocols. *Four-phase signaling* is a protocol followed by two systems to synchronize their computations. In this protocol, two systems are connected by a *channel* consisting of two wires *req* (for request) and *ack* (for acknowledge). One end of the channel is called *active* (because it initiates the synchronization) and the other end *passive* [26]. *req* is an output for the active end, and *ack* is an input for the active end (and vice versa for the passive end). The protocol at the active end is

$$req+;\ \underline{ack+};\ req-;\ \underline{ack-}\ .$$

The protocol at the passive end is

$$\underline{req+};\ ack+;\ \underline{req-};\ ack-\ .$$

These are complete *cycles* of the four-phase protocol. Four-phase signaling is also called *return-to-zero* signaling in literature.

In the *two-phase signaling* protocol, the interfaces skip the phase of returning the wires to their former state; example: $req+$; $\underline{ack+}$ is an active two-phase cycle. The next cycle for this interface would be $req-$; $\underline{ack-}$. Two-phase signaling is also called *non-return-to-zero* and *transition* signaling in literature.

# 5 Introduction of the example QR42

Module QR42 can be used to connect two systems $S1$ and $S2$ as shown in Figure 2, where $S1$ follows four-cycle signaling and $S2$ follows two-cycle signaling. Such a situation can arise (for example) while connecting two buffers obeying these two different signaling protocols.

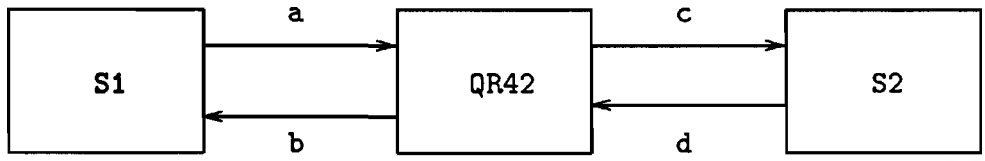


Figure 2: Block Diagram of the QR42 System

Assuming that all the signals start at low voltage, one cycle of QR42's activities that realizes its requirement is

$$\underline{a+};\ c+;\ \underline{d+};\ b+;\ \underline{a-};\ b-\ \ \underline{a+};\ c-;\ \underline{d-};\ b+;\ \underline{a-};\ b-$$

However, this cycle is not the most efficient one, because the acknowledgement to $S1$ is not generated until after an entire cycle has been performed on the $S2$ interface. To increase throughput, QR42 has to overlap the executions of its interfaces as much as possible, as in:

$$[\underline{a+};\ ((b+;\ \underline{a-})\ \|\ (c+;\ \underline{d+}));\ b-;\ \underline{a+};\ ((b+;\ \underline{a-})\ \|\ (c-;\ \underline{d-}));\ b-]^* \qquad (1)$$

In this cycle, notice that $b+$ and $c+$ are generated in parallel (as well as, later, $b+$ and $c-$). This allows $S1$ and $S2$ to respond to these *ack*s and do whatever internal processing that needs to be done *in parallel*, thus enhancing the system throughput. Also notice that the partial order implied by the second protocol subsumes that implied by the first protocol. One of the basic ideas in speeding up asynchronous computations is to find that partial order which is correct (with respect to a specification) and subsumes all other partial orders.

We now examine the specification and synthesis of QR42 using various approaches.

# 6 Design of QR42 Using Flow-tables

## 6.1 Primitive Flow-table for QR42

A flow-table has a row corresponding to each internal state, and a column corresponding to each input combination (also called *input state*). A combination of an input state and an internal state is called a *total state*. For every total state, a flow-table specifies the next internal state and the output.

Synthesis using flow-tables starts with system specification in the form of a *primitive flow table*. A primitive flow table has at-most one stable entry (see below) per row. The primitive flow-table for QR42 is shown in figure 3.

We denote the next state and the output for the total state $(q_i, I_j)$ by $N(q_i, I_j)$ and $Z(q_i, I_j)$ respectively. If $N(q_i, I_j) = q_i$, then $q_i$ is said to be a *stable state* under input $I_j$, and the

| state | ad | | | |
|---|---|---|---|---|
| | 00 | 01 | 11 | 10 |
| 1 | [1], 00 | – | – | 2, 11 |
| 2 | 3, 11 | – | 4, 11 | [2], 11 |
| 3 | [3], 11 | 5, 01 | – | – |
| 4 | – | 5, 01 | [4], 11 | – |
| 5 | – | [5], 01 | 6, 10 | – |
| 6 | – | 7, 10 | [6], 10 | 8, 10 |
| 7 | 1, 00 | [7], 10 | – | – |
| 8 | 1, 00 | – | – | [8], 10 |

Figure 3: Primitive Flow-table for QR42

entry corresponding to $q_i$ is boxed in the flow-table. All other states are unstable, and their entries are not boxed. State transitions are caused by input changes. If a transition from a stable configuration $(q_i, I_m)$ caused by changing $I_m$ to $I_n$ results in state $q_j$, and $(q_j, I_n)$ is not stable, the state changes again to $N(q_j, I_n)$, and continues to change within column $I_n$ until a stable configuration is reached.

For each row $q_i$ of the table, next state entries are specified for input states that may follow the input state containing the stable state in row $q_i$. As an example of how to read such a flow-table, assume that the system is in state 1 to begin with, with the current inputs being 00. From the flow-table we see that the system is in the stable state 1. Now suppose the input is changed to 10. From row 1 we see that the system performs a transition into state 2, and from row 2 of the flow-table, for input 10, we see that the system remains in state 2. The entries in row 1, and inputs 01 and 11 are unspecified. The remaining entries are specified similarly. Usually the number of states used to specify the problem via the primitive flow-table are much larger than what is necessary to realize the specified function. In order to represent the same behavior using a smaller number of states (for more economical realization), a *flow table reduction* is performed, by capitalizing on the unspecified entries. The reduced flow-table is shown in figure 4.

We now illustrate how to implement the compacted flow-table in terms of a gate/flip-flop network. The first step is to perform *state assignment.* This is done by representing the internal states by combinations of values of binary state variables. There are various state assignment schemes such as *connected row set assignment*, *shared row assignment*, *single transition time assignment*, etc. ([16, Chapter 6]). In our example, we find a simple *unicode assignment* (one code per state).

Once the state assignment is done, the *next state* and *output* functions can be defined.

| state | ad | | | |
|---|---|---|---|---|
| | 00 | 01 | 11 | 10 |
| A | **A**, 00 | **A**, 01 | C, 10 | B, 11 |
| B | **B**, 11 | A, 01 | **B**, 11 | **B**, 11 |
| C | A, 00 | **C**, 10 | **C**, 10 | **C**, 10 |

Figure 4: Reduced Flow-table for QR42

| | $y_1\ y_2$ | ad | | | |
|---|---|---|---|---|---|
| | | 00 | 01 | 11 | 10 |
| A | 00 | **00**, 00 | **00**, 01 | 01, 10 | 10, 11 |
| B | 10 | **10**, 11 | 00, 01 | **10**, 11 | **10**, 11 |
| C | 01 | 00, 00 | **01**, 10 | **01**, 10 | **01**, 10 |

Figure 5: Y, z-map for QR42

These functions can be presented in a tabular form shown in figure 5; such a table presents the so called $Y$ (next state) and $z$ (output) maps.

From these maps, we obtain the excitation function for the flip flops that implement the internal states. The equations are listed in figure 6. A circuit can be obtained from such equations using standard techniques.

## 7 Design of QR42 Using Signal Transition Graphs

In his dissertation[8], Chu has developed a methodology for synthesizing speed independent circuits starting from input specification in the form of graphs called *signal transition graphs*

$$
\begin{aligned}
Y_1 &= \bar{y_2} \cdot a \cdot \bar{d} + y_1 \cdot \bar{y_2} \cdot a + y_1 \cdot \bar{y_2} \cdot \bar{d} \\
Y_2 &= \bar{y_1} \cdot a \cdot d + \bar{y_1} \cdot y_2 \cdot d + \bar{y_1} \cdot y_2 \cdot a \\
b &= Y_1 + Y_2 \\
c &= \bar{y_2} \cdot a \cdot \bar{d} + \bar{y_2} \cdot \bar{a} \cdot d + y_1 \cdot \bar{y_2}
\end{aligned}
$$

Figure 6: Equations for next state ($Y_1$, $Y_2$) and output functions ($b$, $c$)

(STGs). Though Chu has provided a semantics for STGs based on *live-safe free-choice Petri nets*, we shall provide a direct (informal) semantics for STGs in this paper.

Basically, an STG is a finite directed graph in which the nodes are signal transitions, and the arcs are precedence constraints. An arc $st_1 \rightarrow st_2$ constraints signal transition $st_1$ to be an immediate predecessor of $st_2$. $st_2$ cannot occur unless all its immediate predecessors have. If $st_1 \rightarrow st_2$, then $st_2$ is an immediate successor of $st_1$. $st_1 \stackrel{*}{\rightarrow} st_2$ means $st_2$ is a successor of $st_1$. If a node label of an STG is underlined, the signal transition corresponding to this node is of an input signal. Non-underlined transitions are those of output signals. The distinction between inputs and outputs is maintained for many reasons, some of which will be illustrated in the following.

Notice that if there are two arcs incident on a node $n$, all the immediate predecessors of $n$ are constrained to occur before $n$. In this sense, $n$ serves the purpose of a *join* or *rendezvous* point. In [8], STGs are treated as interpreted *Free-choice Petri nets*[31] where STG nodes are modeled as *Petri net transitions* and STG arcs as *Petri net places.* Chu uses many results from Petri net theory, which, in this paper, we shall present more intuitively wherever possible.

A Petri net transition *fires* only when all its input places have at least one token. Firing reduces the token count of the input places by one, and increases the token count of the output places by one. A *marking* of a Petri net is an assignment of tokens to places. A marking also tells where "control" resides. When an STG is provided, an *initial marking* is also provided for it. It turns out that the class of Petri nets that model STGs always satisfy the *one token per loop* restriction—the total number of tokens in all the places on any loop must not exceed 1. Given this restriction, an initial marking need only specify a subset of STG edges (which then are endowed with one token each). The STG that captures the signaling requirements of QR42 is provided in figure 7(a). This STG specification captures equation 1 directly. The initial marking of this STG consists of the edge leading into the topmost $a+$ transition. Markings reachable from this initial marking are shown below: (Here, $a \rightarrow b$ is to be read: 'the edge from $a$ to $b$'.)

- Corresponding to the upper half of the graph, we have the edges:

$$\{a+ \rightarrow b+, a+ \rightarrow c+\},$$
$$\{a+ \rightarrow b+, c+ \rightarrow d+\},$$
$$\{a+ \rightarrow b+, d+ \rightarrow b-\},$$
$$\{b+ \rightarrow a-, a+ \rightarrow c+\},$$
$$\{b+ \rightarrow a-, c+ \rightarrow d+\},$$
$$\{b+ \rightarrow a-, d+ \rightarrow b-\},$$
$$\{a- \rightarrow b-, a+ \rightarrow c+\},$$

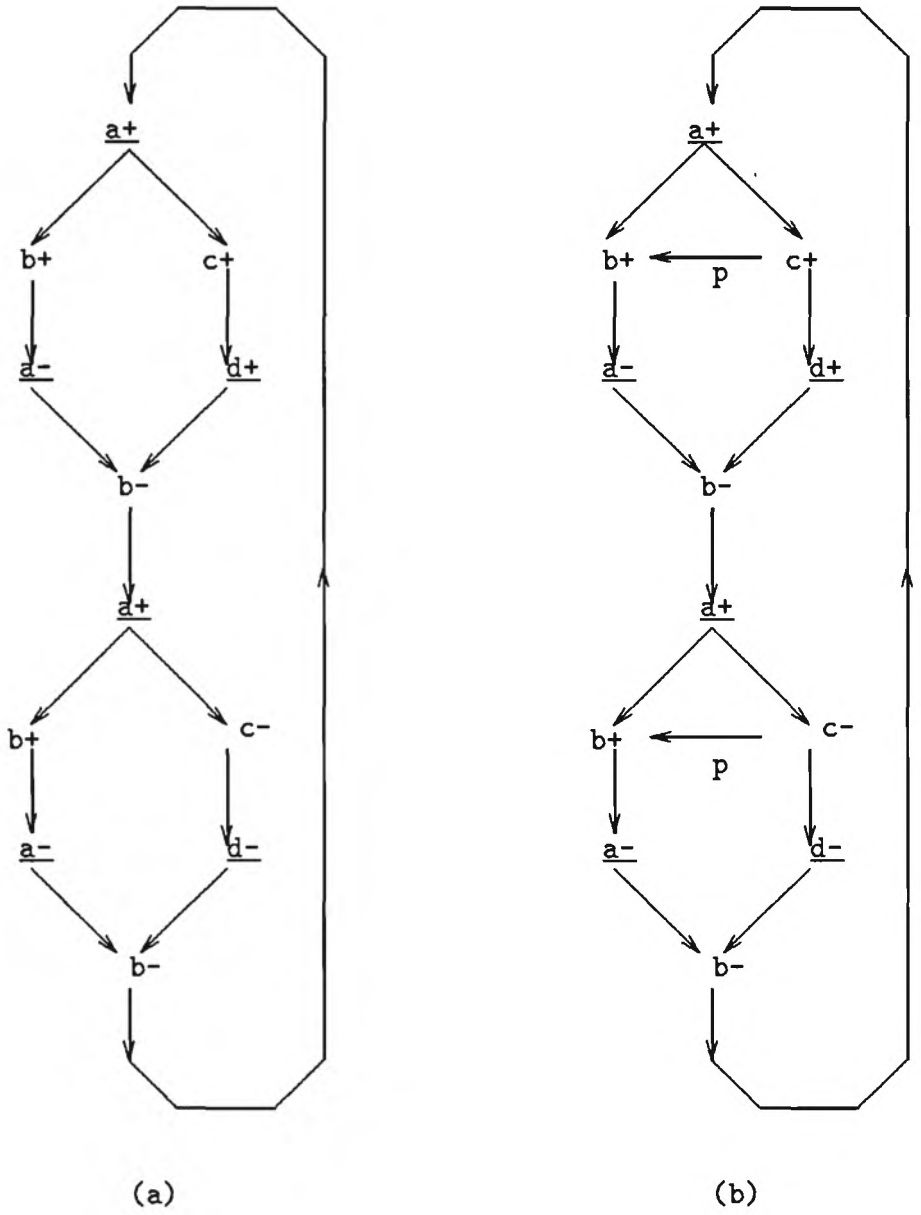


Figure 7: Original STG of QR42 (a), and the same with Persistency Edges (b)

$\{a- \rightarrow b-, c+ \rightarrow d+\}$,
$\{a- \rightarrow b-, d+ \rightarrow b-\}$

- the edge $\{b- \rightarrow a+\}$, and
- and a set of markings (similar to the upper-half of the STG) for the lower half of the STG also.

Notice that each such marking corresponds to a "cut" through the graph.

The synthesis procedure using the STG approach consists of the following steps:

1. Check the syntax of the given STG for liveness and persistency. If the STG does not pass any of these checks, modify it appropriately (usually by augmenting it with extra edges, section 7.1).
2. Derive the equivalent finite automaton of the STG and perform state assignment to produce a state graph. Then examine this state graph and check for state assignment problems, augmenting the specification with extra internal signals in case there are state assignment problems (section 7.2).
3. From the state graph obtained in step 2, it is possible to obtain a circuit that realizes the specification. However, the step of *contraction* applied to the STG of step 2 usually yields more efficient circuits. So the remaining steps are: perform contraction on the given STG to obtain many simple STGs, and map them to their respective state graphs. From these state graphs, obtain a K-map representation, and the final circuit corresponding to these K-maps. Assemble these circuits to obtain the overall circuit (section 7.3).

## 7.1 Syntactic Checks and Transformations on STGs

The checks performed on STGs are for liveness and persistency. An STG is defined to be *live* if it is *strongly connected*, and if in any of its simple cycles, for every signal $t$, transitions $t+$ and $t-$ alternate. The requirement of strong connection guarantees that after a signal transition, the 'next' signal transition is always defined. The requirement of alternation guarantees that a signal that has attained high voltage (representing, say, logic 1) will not be again required to go high without first having gone low. Likewise a signal that is low will first have to go high before being required to go low again. (Note that the definition of STGs does allow for non-alternating transitions.) If an STG doesn't meet these liveness requirements, it is rejected.

Next we address persistence.

**Definition 6** *In an STG, if* $s \xrightarrow{*} t_1$ *and* $s \rightarrow t_2$*, and if* $s$ *is the complement transition of* $t_1$*, then, in order to guarantee persistence,* $t_2 \xrightarrow{*} t_1$ *must hold.*

Persistence in this sense is necessary to guarantee *speed independence*, in Chu's approach. In the next section we discuss non-persistence due to state assignment (sometimes referred to as "the state assignment problem" in Chu's sense). This latter notion of persistence, taken together with the notion of persistence defined in definition 6 is necessary and sufficient to guarantee speed independence, in Chu's approach.

Persistence means the following, as far as the behavior of a circuit is concerned. Suppose a circuit element $C$ is subject to the transition of a signal $s$ on one of its inputs. Then, signal $s$ must last at its present state long enough for $C$ to observe the value of $s$, and moreover, no other signal $t$ in the system must undergo a transition so as to cause another (the opposite) transition on $s$ meanwhile. (This notion is also related to what is called *Semi Modularity* in [36].) An STG may be inherently persistent; or it can be made persistent by adding extra edges in it, whose purpose is to delay the occurrence of the opposite transition of signal $s$ referred to above. Formally, in the given STG, if $s \xrightarrow{*} t_1$ and $s \rightarrow t_2$, and if $s$ is the complement transition of $t_1$, and if $t_2 \xrightarrow{*} t_1$ is *not present*, then add an edge (called the *persistency edge*) somewhere in the STG (usually directed from $t_2$ to $t_1$) such that $t_2 \xrightarrow{*} t_1$.

However, there is a caveat: if $t_1$ is an *input* transition, then adding such an edge would cause the in-degree of $t_1$ to go up by one. A constraint that Chu's approach imposes is that input transitions (such as $t_1$) have an in-degree of exactly one. This is explained as follows. If we were to allow an input transition to have an in-degree $> 1$, it is tantamount to imposing a constraint on the environment—that the environment generate $t_1$ only after its two immediate predecessor transitions have occurred. Constraining the environment in such complex ways (e.g. to force the environment to wait for *two responses from the system* before it is allowed to generate its next action) is not allowed in Chu's approach. Such situations are highly unusual in protocols such as *return-to-zero* for which Chu's approach is largely intended. Refer to [8] for further details.

Now let us proceed to make the STG of QR42 shown in figure 7(a) persistent. Notice that in the upper-half of this STG, $\underline{a+} \xrightarrow{*} \underline{a-}$ and $\underline{a+} \rightarrow c+$, but the constraint $c+ \xrightarrow{*} \underline{a-}$ does not exist. There is a similar situation in the lower-half, with $c-$ instead of $c+$. Going by our definition, it appears that one way to make this STG persistent would be to add two persistency arcs, $c+ \rightarrow \underline{a-}$ and $c- \rightarrow \underline{a-}$. However, this is not acceptable as pointed out above. Therefore, we add the persistency arcs $c+ \rightarrow b+$ and $c- \rightarrow b+$ as shown in figure 7(b), with the persistency edges labeled by the letter '**p**'.

### 7.2 From STGs to State Graphs

In this section, we obtain a state graph for QR42, and point out that it suffers from non-persistence due to *state assignment*. We then show how to modify the STG suitably.

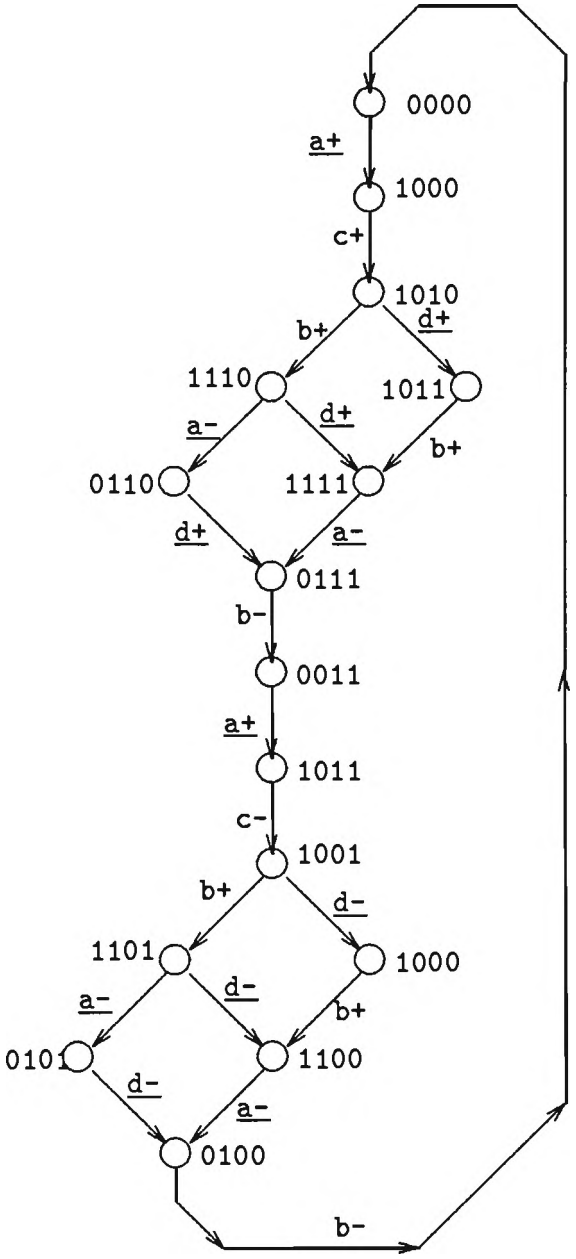


Figure 8: State Graph for QR42

The state graph for QR42 is shown in figure 8. In this graph the nodes are states, node labels are state assignments, and edges are signal transitions. If $node_1 \xrightarrow{s+} node_2$, then the state assignment of $node_1$ has bit corresponding to $s$ 0 and $node_2$ has this bit set to 1. Similarly if $s-$ is the transition label, then these bits are 1 and 0 respectively. A state graph can be arrived at in several ways. One way is to generate the set of reachable markings and determine the state corresponding to each marking.

The initial state graph obtained can be such that there exist two distinct states $s_1$ and $s_2$ carrying the same state assignment, and further, the set of transitions possible from $s_1$ and $s_2$ will allow for "behaviors not called for in the original specification". We refer to this as *non-persistence due to state assignment.* We now elaborate on this point.

An STG specifies a partial order of signal transition occurrences. Any sequence occurrences of signal transitions in the actual circuit must be consistent with this partial order. Now, if at least one of the transitions $t_{1i}$ possible from state $s_1$ is that of a *non-input* signal, and likewise there is a non-input signal transition $t_{2j}$ from $s_2$, and the occurrence of $t_{2j}$ can *disable* $t_{1i}$ from occurring, the following scenario is possible:

- The system reaches state $s_1$;
- It is in effect in state $s_2$ also (because the state assignments of $s_1$ and $s_2$ are the same);
- So it takes $t_{2j}$;
- Therefore it cannot take $t_{1i}$;
- But $t_{1i}$ was supposed to be possible in state $s_1$!

Thus, the system does not meet its specification.

The reason why we chose $t_{1i}$ and $t_{2j}$ to be non-input transitions is that it is precisely the *non-input transitions* that a system can make autonomously (or *endogenously*). If all transitions were to be input transitions, then, the external world would decided which transitions are taken, and so the problem alluded to above would not arise.

In the state graph for QR42 in figure 8, there are two pairs of states which have the same state assignment. Let us consider one of these pairs. For this pair of states, the transition of the output signal $c$ is non-persistent because the two signals $b$ and $c$ are enabled in the pair of states and if transition on $b$ is taken from one of them, the system goes to a state where transition on $c$ is disabled.

The obvious solution to non-persistence due to state assignment is to add an extra (internal) signal $x$, and disambiguate the state assignments of $s_1$ and $s_2$ using $x$. Transitions of $x$ can be introduced in the original STG as shown in figure 9 and the state graph can be re-obtained. Once the state graph is obtained, a Karnaugh map is constructed using the concept of *implied states of signals*, and the final circuit obtained.

The above steps will be illustrated in the next section, because, in practice, the approach of net contraction yields more efficient circuits. In net contraction, several STGs are obtained from the input STG, their state graphs are obtained, and these individual state graphs are mapped into individual circuits which are then wired together.

## 7.3 Synthesis Through Net Contraction

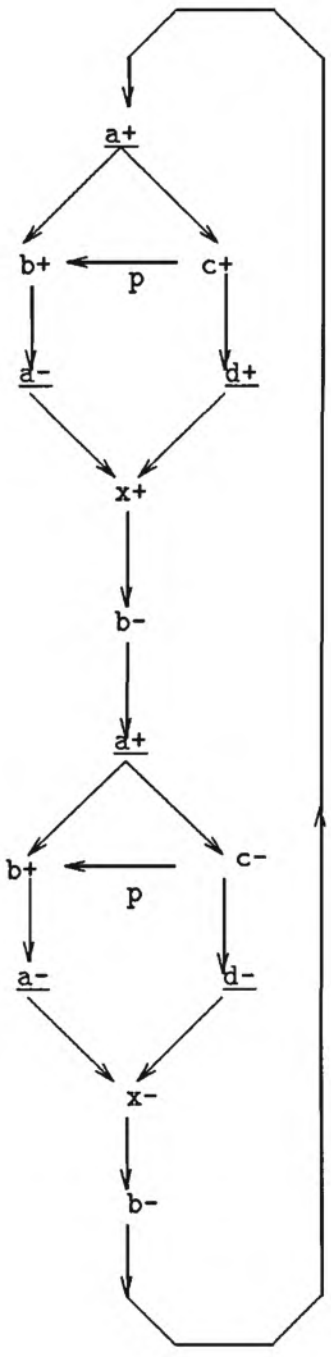


Figure 9: The Final STG

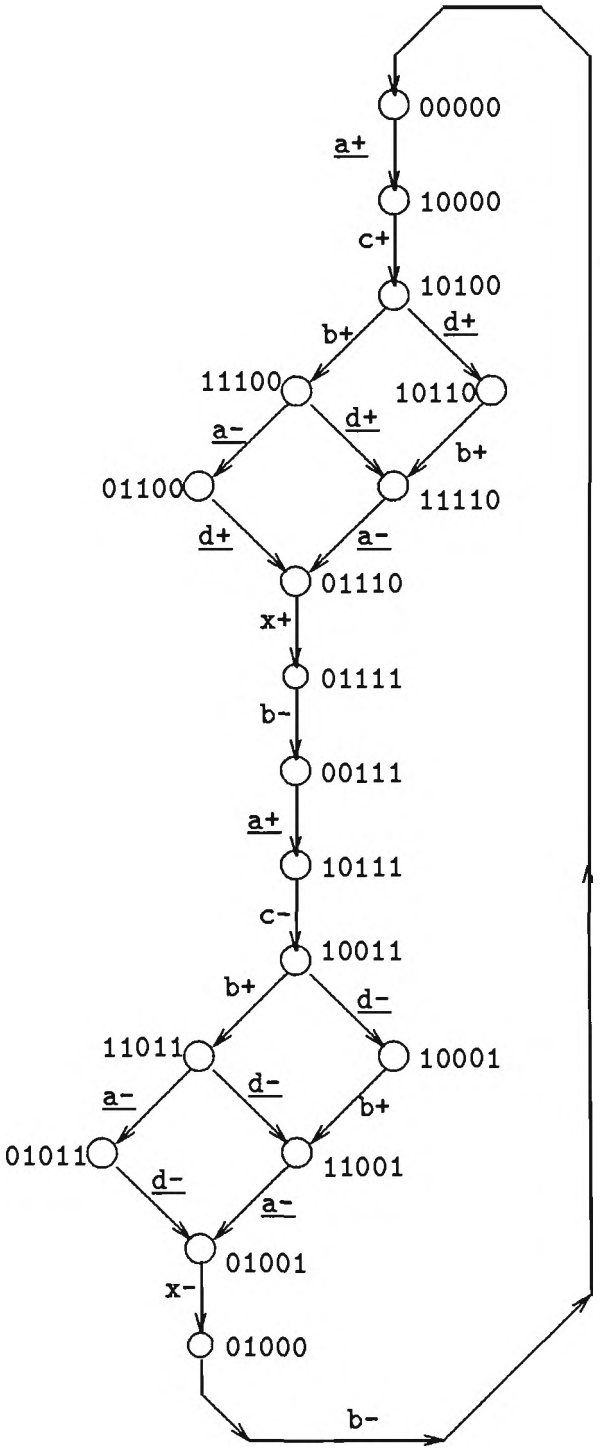


Figure 10: The Final State Graph

The final STG after the addition of two persistency arcs and the introduction of extra internal signal $x$ is shown in figure 9. The state graph corresponding to the final STG is shown in figure 10. Though the final state graph can be implemented directly, or more efficient circuit can be obtained by the decomposition technique. This decomposition technique is based on a graph-theoretic operation called *contraction*. We will illustrate this technique through our QR42 example. Only the output and the internal signals need be implemented. In our example, signals $b$, $c$, and $x$ need be implemented. For each output and internal signal to be implemented, the *input set* is found. Input set $I(i)$ of a signal $i$ is the set of signals whose transitions *cause* the transitions of signal $i$. Or, in other words, the input set of a signal $i$ is the set of signals whose transitions are the immediate predecessors of the transitions of the signal $i$ in the STG. Once the input set of all the output and the internal signals are found, the STG is decomposed in the smaller STGs, each of which contains the transitions of the signals in the set $i \cup I(i)$, for each $i$ which is a non-input signal.

In the implementation, signals in the input set $I(i)$ of a signal $i$ form the input to the logic element for $i$. The *contracted net* for a signal $i$ (output or internal) is obtained from the STG by eliminating the transitions of the signals which are not in the set $i \cup I(i)$, in such a way that the temporal relations among remaining transitions is preserved. The *contracted state graph* for the contracted net of a signal is obtained using exactly the same technique as discussed in the previous section. Contracted state graphs can also be obtained from the top-level state graph by using the *state graph contraction* in the following way. To obtain the contracted state graph for a signal $i$, the transitions of the signals which are not in the set $i \cup I(i)$ for a signal $i$ (output or internal) are replaced by the empty transition $\epsilon$, and then all states connected by $\epsilon$ are collapsed into a new "superstate". Thus, the state graphs obtained from the contracted nets are themselves contracted versions of the top-level state graph.

From a state graph, a logic implementation can be obtained directly. The logic function of a non-input (output or internal) signal $i$ can be obtained from the state graph as follows:

In state $s$, the *implied value* of signal $i$, denoted by $f(s,i)$, is given by

$$f(s,i) = \begin{cases} s'(i) & \text{if } \exists s' \in S : s \xrightarrow{t} s' \wedge t = i_* \\ s(i) & \text{otherwise,} \end{cases}$$

Where $i_*$ denotes a transition of signal $i$ (either $i_+$ or $i_-$). The set of implied values of $i$ in all the states of the state graph constitutes the logic function $f(i)$ of $i$. The K-map for the signal $i$ is obtained which contains the an entry corresponding to each state in the state graph of $i$. The entry corresponding to state $s$ of the state graph contains the implied value of signal $i$ in state $s$. The logic function $f(i)$ can be determined from such a K-map for signal $i$.

Any function of form $x = S + x \cdot \bar{R}$ can be realized as a S/R flipflop with S and R being

functions of other signals. In CMOS, such a configuration can be combined into a complex DCVS gate, which will be no more complex than a C-element and is much simpler than a direct implementation using S/R flipflop.

Once the logic function for every non-input signal has been obtained, the complete circuit can be obtained by connecting these logic elements such that the input and output signals of the logic elements are connected properly to satisfy the input set requirement of all the constituent logic elements.

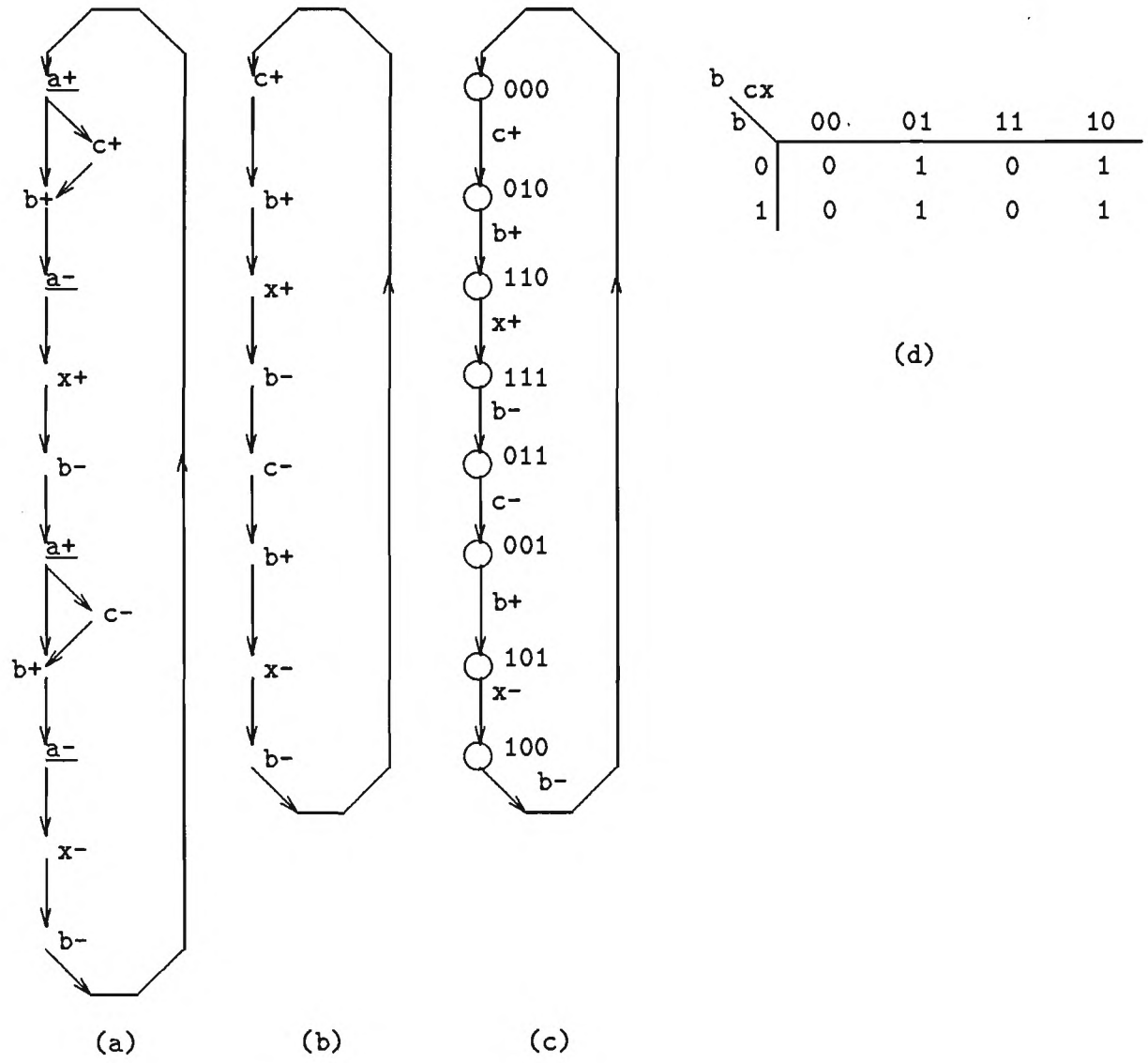


Figure 11: (a)Redundant STG for b; (b)'a' removed; (c)SG for (b); (d)K-map for 'b'

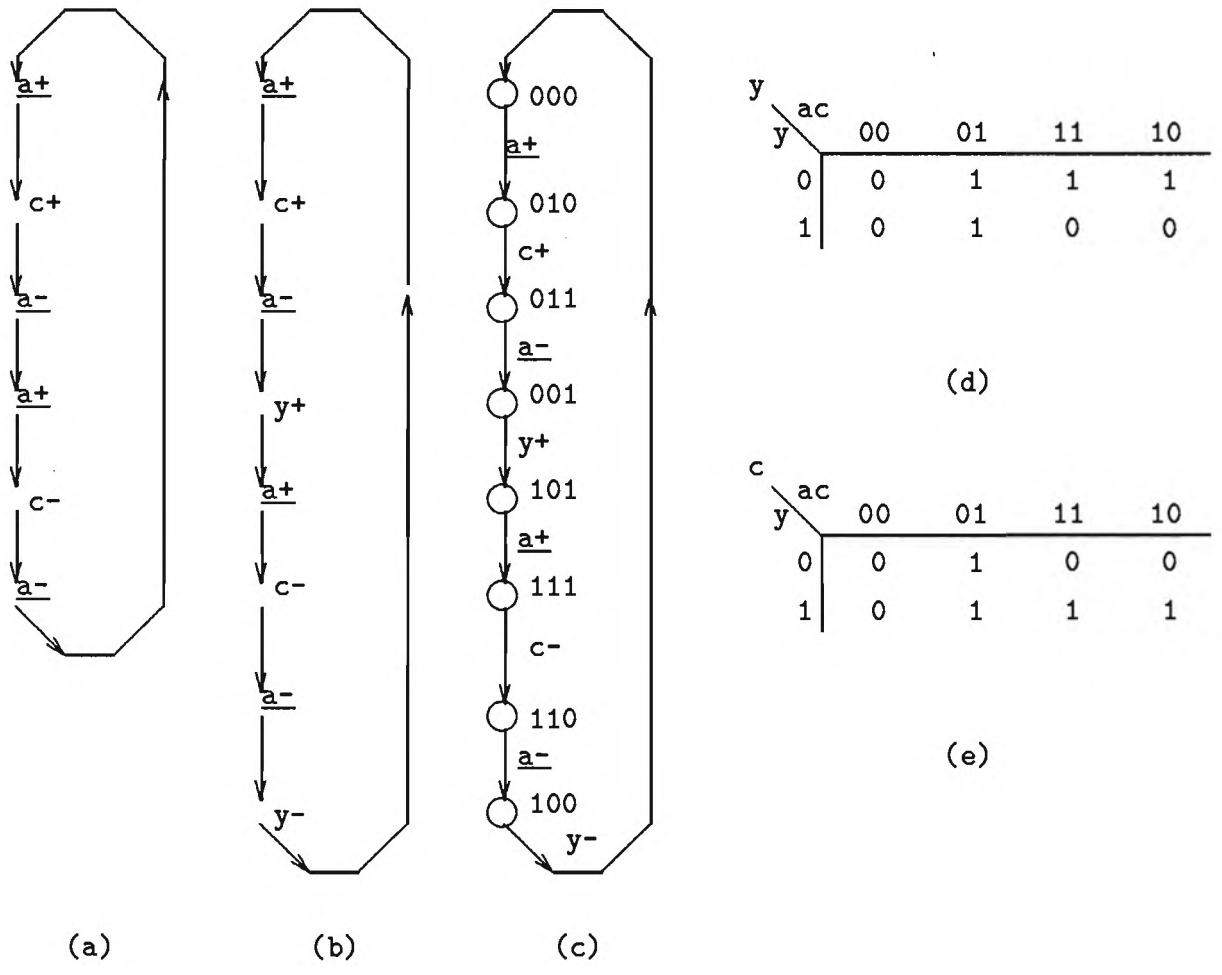


Figure 12: (a)Without 'y'; (b)With 'y'; (c)SG for (b); (d)K-map for 'y'; (e)K-map for 'c'

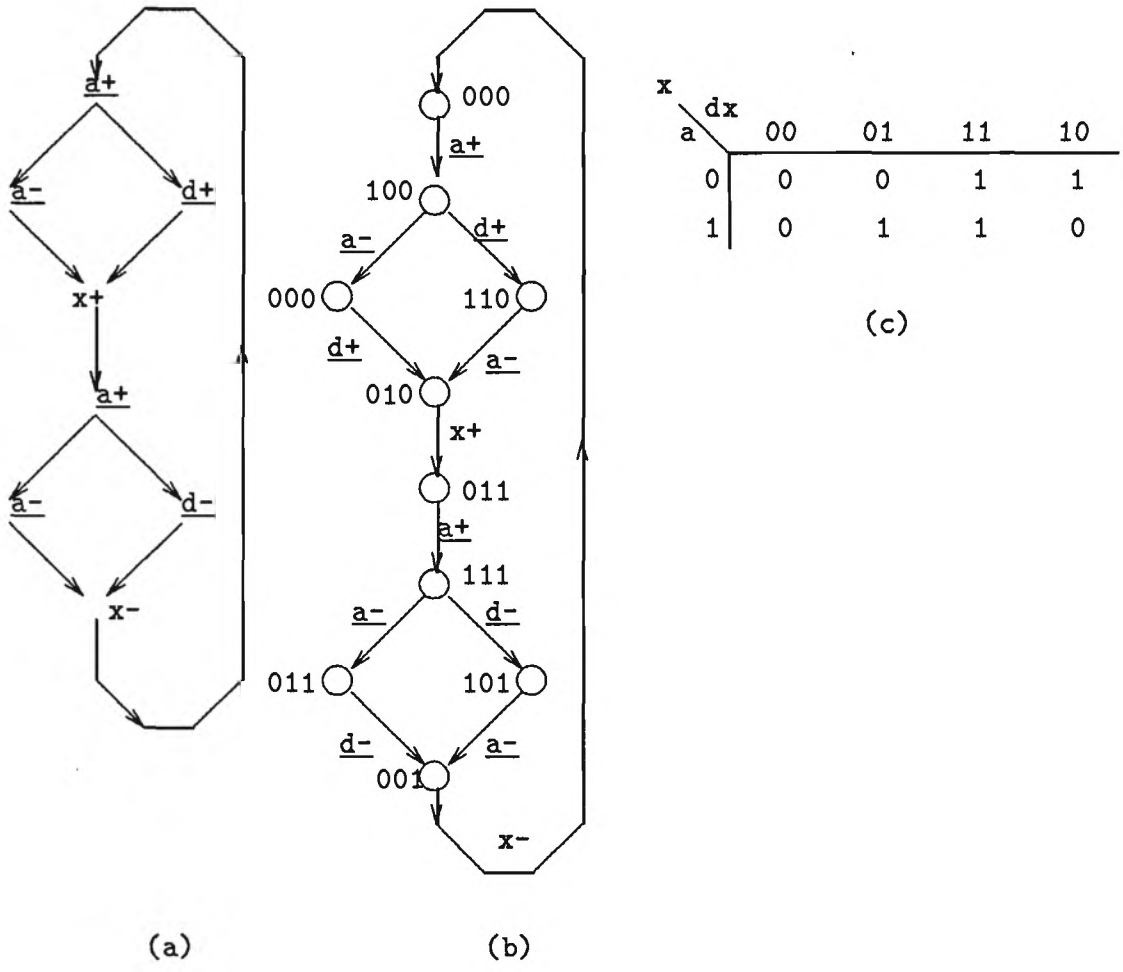


Figure 13: (a)STG for 'x'; (b)SG for 'x'; (c)K-map for 'x'

$$
\begin{aligned}
b &= \bar{c} \cdot x + c \cdot \bar{x} \\
y &= \bar{a} \cdot c + y \cdot c + y \cdot a \\
c &= \bar{a} \cdot c + \bar{y} \cdot c + \bar{y} \cdot a \\
x &= a \cdot x + d \cdot x + \bar{a} \cdot d
\end{aligned}
$$

Figure 14: Logic Equations for signals $b$, $y$, $c$, and $x$

Now, we illustrate the decomposition technique on QR42 example. The output and the internal signals in the STG of figure 9 to be implemented are $b$, $c$, and $x$. The input sets of these signals are $\{a, c, x\}$, $\{a\}$, and $\{a, d\}$ respectively. Contracted nets for these signals based on their input sets are shown in figure 11(a), 12(a), and 13(a) respectively. Now, let us consider signal $b$ first. As seen in the figure 11(a), there is a redundant arc from $\underline{a+}$ to $b+$ which can be removed and the contracted STG for signal $b$ can further be reduced to the one shown in figure 11(b). In this STG (figure 11(b)), $b$ has the input set $\{c, x\}$. The corresponding state graph is shown in figure 11(c). K-map for this state graph is shown in figure 11(d). Logic equations can be obtained from the K-map.

The state graph for signal $c$ has the state assignment problem due to $\underline{a-} \rightarrow \underline{a+}$ arc with no intervening transition of any other signal. To alleviate this problem, we introduce transitions of an extra internal signal $y$ to get the STG shown in figure 12(b). The state graph for this STG is shown in figure 12(c). K-maps for $y$ and $c$ are obtained from this state graph and are shown in figure 12(d) and figure 12(e) respectively.

Similarly, the state graph and K-map for signal $x$ are shown in figure 13(b) and figure 13(c).

The boolean equations for the signals $b, y, c$, and $x$ are obtained from their K-maps and are shown in figure 14.

The boolean equations for signals $y, c$, and $x$ can be realized as SR-flip flops. From the boolean equations of these signals, it is clear that the condition $S \cdot R = 0$ for the SR-flip flops is satisfied. Using the above implementation for the signals $b$, $y$, $c$, and $x$, we get the final circuit as shown in figure 15.

## 8 Design of QR42 Using Trace Theory

For his dissertation [15] Ebergen has developed a technique for synthesizing speed-independent as well as delay-insensitive circuits. The input to his procedure is a *command* specifying a *directed trace structure*. This command is subject to semantics preserving transformations, finally resulting in a set of commands that models a circuit. His technique is based on a formalism called trace theory. Trace theory is also used as the basic formalism in [42,41,38]. Dill [13] uses a variant of trace theory, but does not use this theory for circuit synthesis. Its

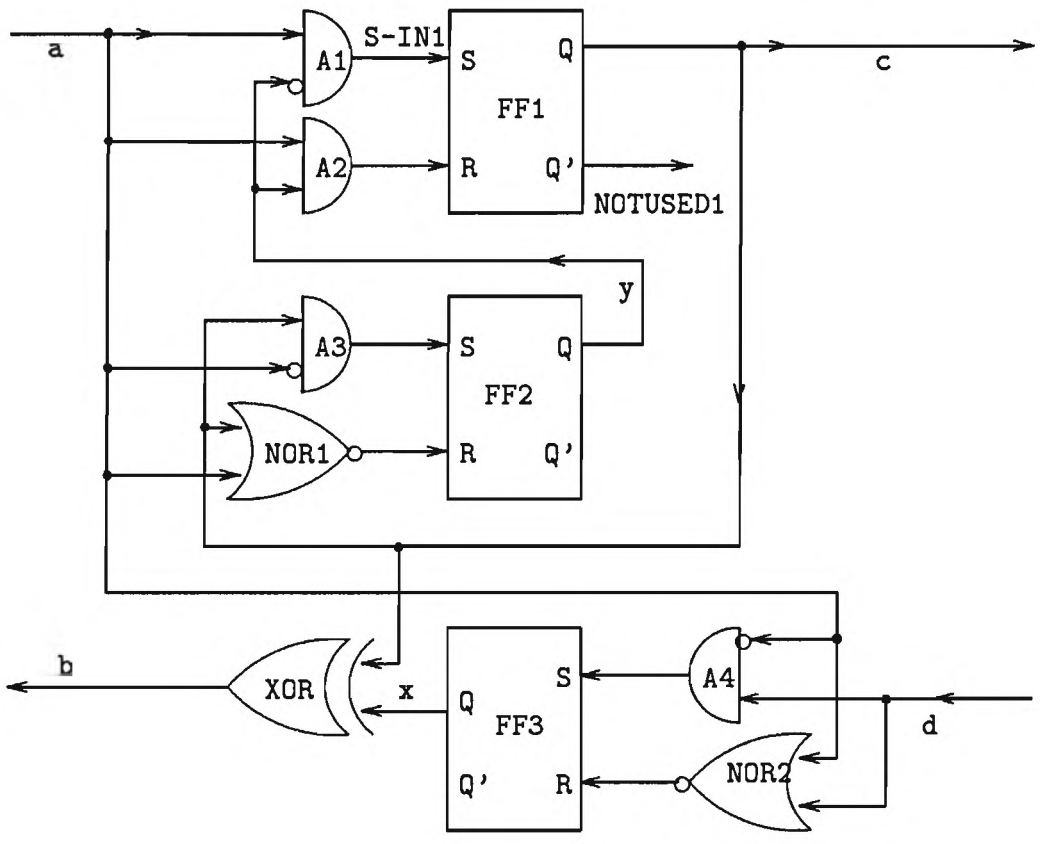


Figure 15: Final Circuit Resulting from Chu's Technique

use for circuit verification will be illustrated in the next section.

### 8.1 Directed Trace Structures

Taking an analogy with automata theory, trace structures are analogous to the pair $(alphabet, language)$ of a finite automaton. Commands, that denote trace structures, are analogous to regular expressions that denote the regular languages of automata. It may help readers to try this analogy on the definitions to follow, keeping in mind, however, that many of the operators used in trace theory have no counterparts in the traditional automata theory (e.g. [21]).

In trace theory, specifications are given by means of a directed trace structure, which is a triple $(A, B, X)$ where $A$ and $B$ are finite sets of symbols and $X \subseteq (A \cup B)^*$. The word "directed" captures the fact that *input* and *output* directions of symbols (synonymous with *signal transitions* as will be seen shortly) is retained. Here $(A \cup B)^*$ is the set of all finite sequences of symbols over $A \cup B$—in other words *traces*. The alphabet of a directed trace structure $R = (A, B, X)$ is $A \cup B$, and is denoted by $\mathbf{a}R$. $A$, the input alphabet, is denoted by $\mathbf{i}R$; $B$, the output alphabet, by $\mathbf{o}R$; and the traces of $R$ by $\mathbf{t}R$.

Given two trace structures $R$ and $S$ and a set of symbols $A$, new trace structures may be obtained by the following constructs:

*Concatenation:* $R;S = (iR \cup iS, oR \cup oS, tRtS)$. $R;S$ is a trace structure whose input alphabet is the union of the input alphabets of $R$ and $S$, whose output alphabet is the union of the output alphabets of $R$ and $S$, and whose traces are the concatenation of the traces of $R$ and $S$.

*Union:* $R \mid S = (iR \cup iS, oR \cup oS, tR \cup tS)$ where the alphabets are united, and the trace sets are also united.

*Repetition:* $[R] = (iR, oR, (tR)^*)$, where the alphabets are retained, and the Kleene-closure of the trace set is taken.

*Prefix-closure:* $\mathbf{pref}R = (iR, oR, \{t_0 \mid (\exists :: t_1 \;.\; t_0t_1 \in tR)\})$. Here, the alphabets are retained. If there is a trace $t$ in $R$, all the proper prefixes of $t$ are added to the trace set of $\mathbf{pref}R$. For example, if $tR = \{a, abc, da\}$, then $t(\mathbf{pref}R) = \{\epsilon, a, ab, abc, d, da\}$. The prefix-closure operator is quite handy for capturing the true behavior of real world systems. For example, if a real world system has the capability to produce a sequence of actions $abc$, it could not have done so without first having produced the sequences $\epsilon$, $a$, and $ab$. This set of sequences is precisely the one given by taking **pref**.

*Projection:* $R \downarrow A = (iR \cap A, oR \cap A, \{t \downarrow A \mid t \in tR\})$. Here, portions of the alphabet that overlap with $A$ are retained, and only those symbols of $tR$ that are contained in $A$ are

retained. This construct is handy in modeling a system whose "internal" signals are of no interest for the outside of the system, and hence can be projected away.

*Weave:* This is a powerful operator used usually for writing the specification of systems exhibiting a large amount of concurrency. Its definition is

$$R \parallel S = (iR \cup iS, oR \cup oS, \{t \in (aR \cup aS)^* \mid t \downarrow aR \in tR \wedge t \downarrow aS \in tS\})$$

The input and output alphabets are united. The set of traces of $R \parallel S$ is a subset of $(aR \cup aS)^*$ such that if $t$ is in this subset, then $t$ *is consistent with* $R$*'s behavior* ($t \downarrow aR \in tR$) and $t$ *is consistent with* $S$*'s behavior* ($t \downarrow aS \in tS$). As we shall see soon, $\parallel$ is used usually as follows: specify various *facets* of the system behavior (in the form of $R$ and $S$); then *take the conjunction* of the constraints implied by these facets (obtain $R \parallel S$).

As an example to further illustrate $\parallel$, the behavior of QR42 can be specified by specifying the behaviors of its two interfaces separately (the "facets"), and then specifying the constraints that exist between the actions of these facets by using $\parallel$.

### 8.2 Commands

Commands denote directed trace structures. The BNF for commands that we consider, and the directed trace structures that they denote, are now presented. We use $dts(cmd)$ to denote the directed trace structure of command *cmd*.

| | | |
|---|---|---|
| $cmd ::=$ | $a?$, | *denoting* $(\{a\}, \emptyset, \{a\})$ |
| | $\mid a!$, | *denoting* $(\emptyset, \{a\}, \{a\})$ |
| | $\mid !a?$, | *denoting* $(\{a\}, \{a\}, \{a\})$ |
| | $\mid cmd_1; cmd_2$ | *denoting* $dts(cmd_1); dts(cmd_2)$ |
| | $\mid cmd_1 \parallel cmd_2$ | *denoting* $dts(cmd_1) \parallel dts(cmd_2)$ |
| | $\mid cmd_1 \mid cmd_2$ | *denoting* $dts(cmd_1) \mid dts(cmd_2)$ |
| | $\mid [cmd]$ | *denoting* $[dts(cmd)]$ |
| | $\mid \mathbf{pref} cmd$ | *denoting* $\mathbf{pref} dts(cmd)$ |

In the trace model, a commonly used interpretation for an occurence of a symbol such as $a$ is that of a signal transition on a terminal of a component—high going, or low going. The signaling discipline employed is what is commonly known as *transition signaling* or *non-return-to-zero* (NRZ) signaling [35]. It is possible to realize circuits employing four-cycle signaling by making sure that there are an even number of alternating signal transitions in any cyclic activity; however, there are many inherent advantages to employing transition signaling and most of Ebergen's circuits are synthesized in this style. By way of examples, we specify a *unidirectional* WIRE, a MERGE, a C-ELEMENT, and a TOGGLE using the command notation (figure 16). These will be the primitives used in QR42.

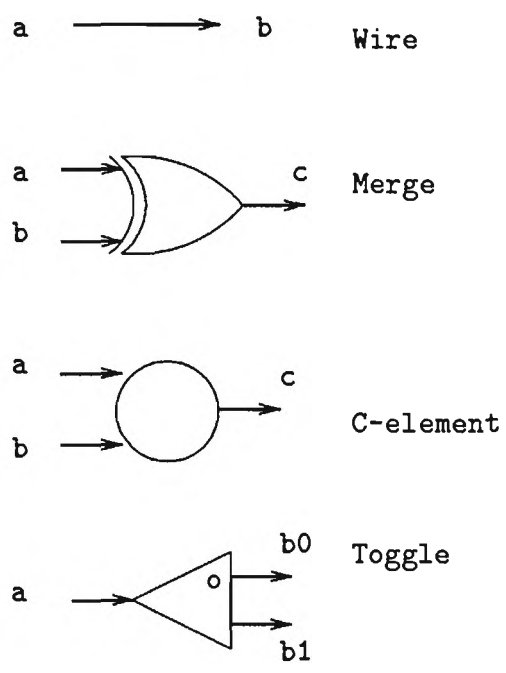


Figure 16: A Wire, Merge, C-element, and Toggle

### 8.2.1 A Unidirectional WIRE

A WIRE conducts transitions from one of its terminals to the other terminal. The command describing a WIRE is

$$\mathbf{pref}[a?; b!].$$

This denotes the trace structure

$$(\{a\}, \{b\}, \{\epsilon, a, ab, aba, \ldots\}).$$

By examining the possible traces, it becomes clear that each trace captures a 'moment in the lifetime' of a WIRE—a complete record of the role it has played in the system thus far. For example, the trace $a$ specifies that moment in a WIRE's lifetime, where it has been subject to an $a$ input, but has not responded yet with a $b$ output. Notice that it is wrong to write the WIRE's behavior as

$$[a?; b!],$$

because it allows for the traces $\epsilon, ab, abab, \ldots$ only; this does not capture those moments where the WIRE has been subject to an excitation but has not responded.

Notice also that we exclude $aa$ from the trace set, for this connotes "data overrun" or "choking"—a situation in which the WIRE has been subject to the next excitation before it has responded to the current one. As we shall see in section 9, the prohibition of such traces is *necessary* for guaranteeing delay-insensitive behavior. In the case of the WIRE, the environment of the WIRE has to 'cooperate' with it by first applying an $a$ and *waiting* for a $b$ before another $a$ is issued.

Thus commands that describe circuit elements are chosen to be *prescriptions* for safe usage so as to get a certain I/O functionality. Commands are not *descriptions* of all possible behaviors.

### 8.2.2 A MERGE Element

A MERGE, usually realized by an XOR gate, merges signal transitions on its inputs. The command that describes its behavior is

$$\mathbf{pref}[(a?|b?); c!].$$

This allows for either $a$ or $b$ to be subject to a transition; the environment then waits for $c$ to respond before $a$ or $b$ is excited again.

### 8.2.3 A C-ELEMENT

A C-ELEMENT, also known as a *rendezvous* element, awaits transitions on $a$ and $b$ in some order before it produces a transition on $c$. Its behavior is captured by the command

$$\mathbf{pref}[(a? \parallel b?); c!].$$

There are many different commands that describe the same behavior. Command

$$\mathbf{pref}[(a?;c!) \parallel (b?;c!)]$$

also captures a C-ELEMENT's behavior. The *weave* of $a?;c!$ and $b?;c!$ allows an $a?$ and a $b?$ to occur in either order, followed by the occurrence of a $c!$, before $a?$ or $b?$ can occur again. Yet another way of writing the behavior of a C-ELEMENT is

$$\mathbf{pref}[a?;c!] \parallel \mathbf{pref}[b?;c!].$$

In Ebergen's technique, it is important to have many such alternative descriptions available for a circuit element, so that the semantics preserving transformations may be directed with greater flexibility towards any of the many equivalent "patterns" (commands).

### 8.2.4 A TOGGLE

A TOGGLE steers every even numbered (starting to count from 0) transition on $a$ into $b0$ and every odd numbered transition on $a$ into $b1$. Its behavioral description is

$$\mathbf{pref}[a?;b0!;a?;b1!]$$

## 8.3 Specification of QR42

The QR42 system can be specified in Ebergen's notation by command $E$:

$$E = \mathbf{pref}[a?;((b!;a?) \parallel (c!;d?));b!].$$

The sub-command $a?;((b!;a?) \parallel (c!;d?));b!$ is repeated. All its prefixes are also allowed, because of the **pref** operator applied to the repetition. The sub-command allows an $a?$ to occur first. Then, the sequences $b!;a?$ and $c!;d?$ are *woven*. Since these sequences do not share any symbol, (by studying the definition of *weave*) it may be concluded that these two sequences are completely concurrent. Once they both have occurred, action $b!$ can occur. Thus, notice that an $a?$ is allowed, followed by a $b!$, followed by another $a?$. However, at this point, a $b!$ is delayed until a $c!$ has been issued and a $d?$ has been received in response to it. Thus, a four cycle handshake is completed on $a, b$ only after a two cycle handshake is completed on $c, d$, but the systems connected to the interfaces are allowed to work concurrently.

After writing such a specification, one can gain confidence in its correctness through several means, but mainly by proving a "catalog" of desired properties based on it. Ebergen's approach is elegant in the sense that the proof of such properties is well supported through a detailed body of mathematical results developed in [15].

There is an algorithm described in [15], but this is a very general algorithm and does not yield the optimal result in most cases. To obtain an optimal result, we have to resort to heuristics and experience. The situation is similar to algorithm design.

We illustrate one such sequence of rule selections on the QR42 system. A general treatment of the various rules, and when they are appropriate, is beyond the scope of this survey. All we hope to convey through the following section is the overall nature of the semantics-preserving transformations.

### 8.4 Synthesis Step 1: Factoring

The first step in the QR42 example is (what we call) the *factoring step*, where a series of semantics-preserving transformations are applied to the original specification (a command) to obtain a command in a form amenable to applying the *decomposition* step.

We take command $E$ and identify distinct occurrences of $a$ and $b$ in it, calling the first occurrence of $a$ $a_0$ and the second $a_1$. As we shall see momentarily, internal signals $a_0$ and $a_1$ can be generated within the system by using a TOGGLE. The main advantage gained is that whenever any one of the signals $a_0$, $a_1$, $b_0$, $b_1$, $c$, or $d$ undergo a transition, the state of the system is determined uniquely, because these signal transitions occur exactly once in a cycle of activity. Using this idea, we first study the command $E'$ as

$$E' = \mathbf{pref}[a_0?; ((b_0!; a_1?) \parallel (c!; d?)); b_1!].$$

This can be rewritten as

$$E' = \mathbf{pref}[a_0?; b_0!; a_1?; b_1!] \parallel \mathbf{pref}[a_0?; c!; d?; b_1!].$$

The idea behind this transformation is to move $\parallel$ outermost. This form is preferred, because it lends itself to the step of decomposition. The correctness of this command can be verified by noting that the '*weavands*' (arguments of the *weave* operation) both begin with $a_0?$ and end with $b_1!$, and have different characters in the middle. In such commands, the occurrence of the $a_0?$ and $b_1!$ of the weavands correspond (or *synchronize*), and the actions in-between can occur in any order.

Now we notice that the occurrence of $a_0$ causes $b_0!$ in the first weavand, and $c!$ in the second. When this situation occurs, a fruitful direction in which to guide the synthesis approach is to introduce an internal signal $!X_1?$ after $a_0$ in both the weavands. Likewise, we notice that after $a_1?$ in the first weavand and $d?$ in the second weavand, $b_1$ occurs in both weavands, and so we introduce $!X_2?$ before $b_1!$ in the weavands. Internal actions of the form $!X?$ are both inputs and outputs; they will end up being inputs for certain internal elements, and outputs for certain others. Now, $E$ has been rewritten to

$$E' = (\mathbf{pref}[a_0?; !X_1?; b_0!; a_1?; !X_2?; b_1!] \parallel \mathbf{pref}[a_0?; !X_1?; c!; d?; !X_2?; b_1!]) \downarrow \{a_0, a_1, b_0, b_1, c, d\}$$

Let us call the weavands $E_1$ and $E_2$.

## 8.5 Synthesis Step 2: Decomposition

One of the most important concepts in Ebergen's work is that of *decomposition.* In this section we digress a bit from the development so far and present the theory and intuition behind decomposition. Then we shall apply decomposition to $E_1$ and $E_2$ obtained above.

Decomposition is a relation between a command and a list of commands, and is written thus:

$$E \rightarrow (E_1, \ldots, E_n).$$

This is to be read: $E$ is decomposed into $(E_1, \ldots, E_n)$. The intuition (and purpose) behind decomposition is to decompose a complex command $E$ into a list of simpler commands $E_1, \ldots, E_n$ such that if these simpler commands are, in turn, realized as circuits $C_1, \ldots, C_n$, then interconnecting these circuits to form a circuit $C$ guarantees that:

- $C$ is well-formed. In other words, $C$ when connected to its environment (call it $\overline{C}$) is such that: (i) it has no unconnected ("dangling") inputs or outputs; (ii) no two subcomponents of $C$ have their outputs connected together;
- *The closed system consisting of $C$ together with its environment $\overline{C}$ is speed independent.* In other words, neither the environment $\overline{C}$ nor any of the subcomponents $C_1, \ldots, C_n$ of $C$ will ever be in a state such that: (i) the subcomponent is in a position to produce a certain output signal $a$; (ii) none of the connected subcomponents are in a position to accept $a$. In other words, no component will be subject to computation interference.
- The interconnection of $C$ with its environment $\overline{C}$ becomes a closed system that can "run all by itself". (In fact it has no inputs or outputs available any more.) This closed system exhibits all the traces that were called for in the original specification.

More precisely, to define when $E$ and $E_1, \ldots, E_n$ are in the decomposition relation, we first define the *reflection* of $E$, that models the environment. This process is denoted by $\overline{E}$. The directed trace structure of $\overline{E}$ is obtained from that of $E$ by swapping the input and output alphabets of $E$, and retaining the same traces of $E$, except that every output action of $E$ is changed to an input action of $\overline{E}$, and vice versa. $\overline{E}$ models the environment of $E$ because:

- It is capable of supplying any event that is awaited by $E$;
- It is capable of accepting any of the events generated by $E$;
- Its state transitions match those of $E$, so that the above correspondence between $E$ and $\overline{E}$ is maintained.

In fact, it is a notable feature of Ebergen's work that when a command $E$ is synthesized into a circuit, that circuit is guaranteed to work only for environments that *conform to* $\overline{E}$. This notion of conformance is also used by Dill (discussed in section 10).

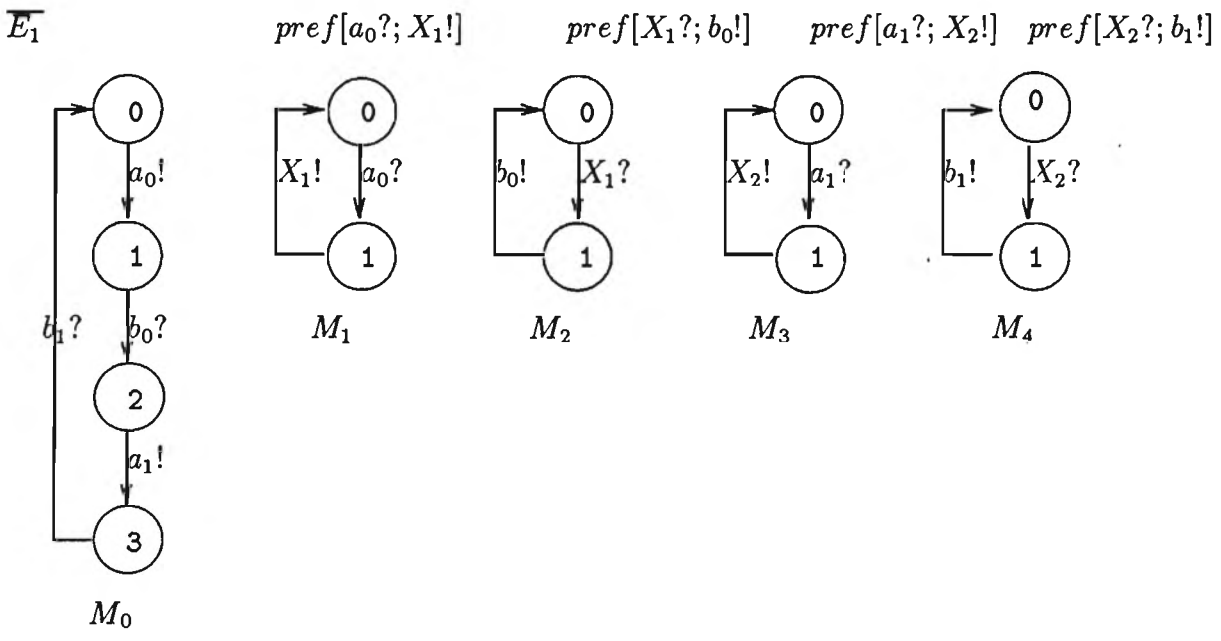


Figure 17: An Example Showing Decomposition

**Definition 7** *Process A conforms to process (written $A \preceq B$) B if (i) A accepts all the inputs that B does, and (ii) A does not generate an output unless B does so.*

An environment $\overline{EE}$ that conforms to $\overline{E}$ is a "lesser" environment, in that it demands less of $E$ than $\overline{E}$ does, but is willing to accept anything that $\overline{E}$ can.

Now we define the decomposition relation more formally. Let $E_0 = \overline{E}$. Then $E$ and $E_1, \ldots, E_n$ are in the decomposition relation if:

*No unconnected I/O*: The system $E_0, \ldots, E_n$ is *closed*;

*No clashing outputs*: The system $E_0, \ldots, E_n$ has no two outputs tied to each other;

*Simulation is Correct*: Conduct the 'simulation' of $E_0, \ldots, E_n$ as follows:

- Represent the commands $E_0, \ldots, E_n$ by their equivalent finite-state diagrams. We are assuming that $E_0, \ldots, E_n$ are $\|$ free commands, and for such commands, the finite-state diagram is easy to obtain. Some examples are provided in figure 17. This figure illustrates the decomposition of command $E_1$.
- Start $E_0, \ldots, E_n$ in its start state;

- If none of $E_i$ is in a position to produce an endogenous action, then the simulation stops;
- Otherwise, for every member of the set $E_i, \ldots, E_j$ of systems capable of performing an endogenous action, (with $E_k$ used as a representative in the following) do:
  - Perform one of the endogenous actions $a$ possible for $E_k$. (Repeat this for every such action $a$ possible.) This action is necessarily an output action.
  - For all remaining $E_l$ whose input alphabet contains $a$, check that these commands can accept $a$. *If not, then there is computation interference and the decomposition relation does not hold between $E$ and $E_1, \ldots, E_n$.* Were such a decomposition to be attempted, "choking" would result.
  - Assuming that all the processes whose input sorts contain $a$ are receptive to $a$ (check for "no choking" passed in the previous step), advance the state of all these processes, as well as that of the process generating $a$.
  - Repeat the above steps until no more new states are visited.
- Every trace specified in $E$ may occur in the simulation.

### 8.5.1 Example of Decomposition

Let us check if the following decomposition relation is true:

$$E_1 \rightarrow (\mathbf{pref}[a_0?; X_1!], \mathbf{pref}[X_1?; b_0!], \mathbf{pref}[a_1?; X_2!], \mathbf{pref}[X_2?; b_1!]).$$

Recall that

$$\begin{aligned} E_1 &= (\mathbf{pref}[a_0?; !X_1?; b_0!; a_1?; !X_2?; b_1!]) \downarrow \{a_0, b_0, a_1, b_1\}. \\ &= \mathbf{pref}[a_0?; b_0!; a_1?; b_1!]. \end{aligned}$$

The finite-state diagrams of $\overline{E_1}$, and the commands it (supposedly) decomposes into, are shown in figure 17. All these "machines" are initially in their state 0. The state of the simulation is the tuple $(0,0,0,0,0)$.

1. The only machine capable of an endogenous action, $a_0!$, is $M_0$. The only machine which has $a_0$ in its input sort is $M_1$. It is capable of accepting $a_0$. So advance $M_0$ and $M_1$. The state of the simulation is $(1,1,0,0,0)$.

2. Now $M_1$ alone is capable of an endogenous action, namely $X_1!$. $X_1$ is in the input sort of $M_2$ which is in a position to accept it. The simulation state is advanced to $(1,0,1,0,0)$.

3. Now $M_2$ alone is capable of an endogenous action, namely $b_0!$ which is accepted by $M_0$. Following this action, the simulation state is advanced to $(2,0,0,0,0)$. By similar reasoning, $(3,0,0,1,0)$, $(3,0,0,0,1)$, and finally $(0,0,0,0,0)$ are attained.

4. Since $(0,0,0,0,0)$ is re-visited, the simulation can be stopped.

In this example, at every simulation state, only one endogenous action was possible. If more than one endogenous action is possible, the simulation is, in effect, *forked* into as many simulations, and these possibilities are pursued separately.

From the above example, we can see that the claimed decomposition is indeed correct. Likewise, we decompose $E_2$ into

$$E_2 \rightarrow (\mathbf{pref}[a_0?; X_1!], \mathbf{pref}[X_1?; c!], \mathbf{pref}[d?; X_2!], \mathbf{pref}[X_2?; b_1!]).$$

### 8.6 Synthesis Step 3: DI Decomposition Check

Decomposition guarantees speed independence as explained in the previous section. However, for delay-insensitive behaviors, what is known as *delay-insensitive decomposition* (*DI decomposition* for short) is to be performed. One easy way to do this is to perform ordinary decomposition and then verify that the components obtained are all *DI components.*

DI components are those that exhibit the *DI signaling protocol.* This concept has been extensively studied by both Ebergen and Udding (we shall discuss the latter's work in section 9). Udding was the first to characterize a DI component formally by means of trace theory. Ebergen gave an alternative characterization by means of the decomposition relation, which turns out to be equivalent to to Udding's [15]. In [13] yet another, also equivalent to udding's, characterization can be found. Dill's verifier can verify if a component satisfies the DI signalling protocol. In [15] a syntactic check for commands is described. (This is one of the significant recent results in the area of asynchronous system design.)

Any component whose behavior is described by commands of the form

$$\mathbf{pref}[a?; b!]$$

is a DI component. Since our decompositions yield only such components, we have, in effect, achieved a DI decomposition without further ado.

### 8.7 Synthesis Step 4: Application of Separation Theorem

The Separation Theorem is one of the important theorems applied during design refinement. Due to lack of space, we resort to a correct, but perhaps less formal statement of the theorem.

**Definition 8** *Suppose* $E = E_1 \parallel E_2$. *Suppose* $E_1$ *and* $E_2$ *have been decomposed into* $E_{11}, \ldots, E_{1i_1}$ *and* $E_{21}, \ldots, E_{2i_2}$ *respectively. Divide*

$$Es = \{E_{11}, \ldots, E_{1i_1}\} \cup \{E_{21}, \ldots, E_{2i_2}\}$$

*into a set of* clusters $cl_1, \ldots, cl_m$ *such that:*

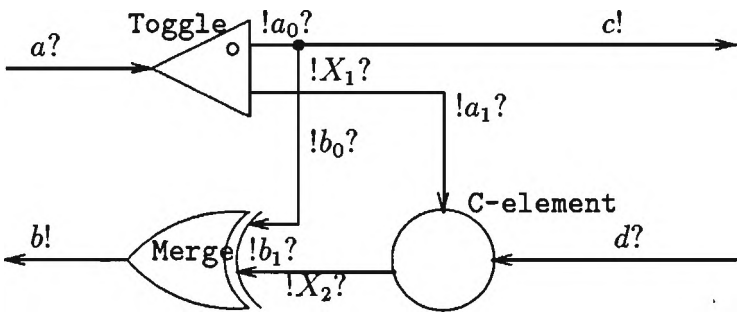


Figure 18: Circuit Synthesized Using Ebergen's Approach

- *The union of the clusters = Es;*
- *For every cluster that has more than one element, the elements of that cluster taken pairwise are found to share at least one output symbol;*
- *An element from one cluster does not share an output port with any element from any other cluster.*

*Apply the weave operator to each cluster—written as* $\| \, cl_i$. *(Since* $\|$ *is commutative and associative, this notation means the repeated application of* $\|$ *to the elements of* $cl_i$.*) Call these commands* $parcl_1, \ldots, parcl_m$. *Then,*

$$E \rightarrow (parcl_1, \ldots, parcl_m).$$

In our example, we can identify only one cluster containing two elements:

$$\{\mathbf{pref}[a_1?; X_2!], \mathbf{pref}[d?; X_2!]\}.$$

All other clusters have only a single element.

Thus,

$$\begin{aligned} E \rightarrow\ & (\mathbf{pref}[a_0?; X_1!], \mathbf{pref}[X_1?; b_0!], \\ & \mathbf{pref}[a_1?; X_2!] \parallel \mathbf{pref}[d?; X_2!], \\ & \mathbf{pref}[X_2?; b_1!], \mathbf{pref}[X_1?; c!]). \end{aligned} \tag{2}$$

## 8.8 Synthesis Step 5: Identification of the Primitives

At this stage, we are in a position to match the result of the decomposition with standard asynchronous components. The overall approach is the following:

- Treat the right-hand side of equation 2 as a network of components.
- Generate $a_0$ and $a_1$ from $a$ using a TOGGLE.
- We notice that whenever there is a $a_0?$, an $X_1!$ is generated. A *unidirectional* WIRE will achieve this behavior. Likewise, $X_1?$ causes a $c!$, $X_1?$ causes a $b_0!$, and $X_2?$ causes a $b_1!$, all of which are achieved by WIREs.
- Whenever there is an $a_1?$ and a $d?$ in either order, an $X_2!$ is generated. A C-ELEMENT will achieve this behavior.
- A $b$ is generated whenever there is a $b_0$ or a $b_1$. This is achieved by a MERGE element.
- The final circuit using these components is shown in figure 18.

# 9 Characterizing Delay Insensitivity

In his dissertation [38], Udding has stated and proved four necessary and sufficient conditions for a circuit specification to be called *delay insensitive*. Circuit specifications are expressed in a language of *commands*. Commands denote directed trace structures. But for a few minor syntactic differences, Udding's notation is similar to the one used by Ebergen. In this paper we shall discuss Udding's work using Ebergen's notations. In the following, we shall refer to trace structures by $T$, and denote its traces (its trace set) by $tT$ and its alphabet by $aT$.

## 9.1 Absence of Transmission Interference

The first condition, referred to as $R_0$, helps prevent *transmission interference*. Suppose a system generates a signal transition $a+$ on a wire and very soon thereafter generates an $a-$ on the same wire. Since nothing can be assumed of absolute or relative delays of wires in a circuit, a "runt" pulse can form on wire 'a'. (A *runt pulse* is one which rises partly to a logical value (one or zero), but before it attains this logic value, proceeds to change in the other direction. The amplitude of voltage swing of a runt pulse can, in the worst case, be zero.) This is a well-known source of circuit malfunction. Transmission interference is one of the sources of non-persistence (or "choking") referred to, earlier.

Condition $R_0$ helps prevent transition interference. It prevents a system from sending or receiving two consecutive signal transitions on a wire. More precisely,

$$s \in tT \Rightarrow saa \notin tT, \ for\ a \in aT$$

As an example, in all hand-shake protocols, request and acknowledge alternate. Two requests or acknowledges that immediately follow one another are never issued.

### 9.2 Non-Reliance on Relative Delays

The second condition, referred to as $R_1$, helps prevent two signals arriving in "opposite order than expected". Consider a system which waits for a signal $a$ to arrive first, then a signal $b$, and then proceeds in its computation. Suppose the system considers it to be erroneous if $b$ were to arrive first. If signals $a$ and $b$ of the system were to be connected to the external world via actual wires that have unknown delays, then, regardless of the order in which the environment of the system transmits transitions on $a$ and $b$, it is possible for the system to receive a $b$ first.

Condition $R_1$ asserts that if a system accepts the order $ab$ of signal arrivals, it must also accept the order $ba$. More precisely, for traces $s, t \in tT$, and for symbols $a, b \in aT$ of the same type (i.e. $a$ and $b$ are both inputs or both outputs), $sabt \in tT \Leftrightarrow sbat \in tT$.

As an example, the C-ELEMENT accepts its inputs in either order before it generates its output.

### 9.3 Absence of Computation Interference

This condition, referred to as $R_2'$, helps avoid *computation interference* as well. (Condition $R_2$, a condition stronger than $R_2'$, is introduced by Udding in his paper.) Computation interference is said to occur when a signal arrives at a system which is not in a state where it is prepared to accept the signal. Computation interference is the other source (over and above transmission interference) of non-persistence.

Following Udding's terminology, let us refer to the system under study as *mechanism*. Let the mechanism have two input signals $a', c'$ and an output signal $b'$. Picture these signals being connected to the environment by very long wires with arbitrary delays. Let the other ends of the wires be referred to by $a, b$ and $c$. We write $x < y$ to mean $x$ occurs before $y$.

Though the wires have arbitrary delays, due to causality, we have $a < a'$. In other words, an $a$ is input at the environment's boundary before it is input at the mechanism's boundary. Similarly, $b' < b$. So, if $b < a$, then $b' < a'$. However, if $a < b$, then both $b' < a'$ and $a' < b'$ are possible. Thus, it is possible for the environment's boundary to be experiencing a trace $sabt$, $s, t \in tT$, whilst the mechanism's boundary is experiencing the trace $sb'a't$. However, if the mechanism *does not have* a trace $sb'a't$, then this possibility does not arise.

We are now going to leave out the primes associated with $a'$ etc., as they were introduced only for notational clarity. (In studying DI circuits, wires are not explicitly modeled in the above fashion by giving distinct names to their ends.)

Condition $R_2'$ is

$$sabtc \in tT \wedge sbat \in tT \Rightarrow sbatc \in tT$$

With respect to the explanation given above, this means that if $sabtc$ is a possible trace at the environment's boundary, and if $sbat$ is a trace possible at the mechanism's boundary,

then *sbatc* must be possible at the mechanism's boundary.

In other words, as we saw above, the environment could be doing the sequence *sabt* while the mechanism is engaging in *sbat*. Now, if the environment attempts an extension of the trace *sabt* to *sabtc*, then the mechanism must be prepared to allow the extension of *sbat* to *sbatc*. If not, the mechanism will find itself subject to a $c$ that it is not prepared to accept—or computational interference will result.

This condition is not that easy to fathom. Refer to [38] for more examples.

### 9.4 Proper Arbitration

This condition is referred to as $R_3''$. (Udding also presents two stronger conditions $R_3$ and $R_3'$.) An arbiter can make a choice between two inputs—in other words, the acceptance of one input by the arbiter is tantamount to the *refusal* of the other input. Similarly, many asynchronous systems make *autonomous* non-deterministic decisions, where it choses between two outputs; in this case, the choice of one output is tantamount to the rejection of the other. (E.g. a "coin tosser" circuit.)

Udding's last condition states that an *input may not cause an output to be refused, and vice versa.* Let us do a case analysis and see why this is desirable. Suppose the mechanism is in a state where it is faced with the choice between accepting an input and generating an output. Suppose it chooses to generate the output. The environment has no immediate way of knowing that this choice was taken by the mechanism (because the wire from the mechanism to the environment can be very slow), and so the environment can end up generating the input to the mechanism which the mechanism chose not to accept. This can result in computation interference. The complementary situation is similar. Condition $R_3''$ states

$$sa \in tT \wedge sb \in tT \Rightarrow sab \in tT$$

where $a, b \in aT$ and $a, b$ are of opposite type.

### 9.5 Checking Udding's Conditions on QR42

The traces of QR42 are more readily seen if the command describing its behavior is written as

$$\begin{aligned}\textbf{pref}[\quad & a?; b!; a?; c!; d?; b! \\ \mid\ & a?; b!; c!; a?; d?; b! \\ \mid\ & a?; b!; c!; d?; a?; b! \\ \mid\ & a?; c!; b!; a?; d?; b! \\ \mid\ & a?; c!; b!; d?; a?; b! \\ \mid\ & a?; c!; d?; b!; a?; b!]\end{aligned}$$

None of the alternatives contains two consecutive occurrences of the same symbol. Each alternative begins with an $a?$ and ends with a $b!$, and hence the repetition of the alternatives will not give rise to two consecutive occurrences of the same symbol. Hence, condition $R_0$ is satisfied, and so QR42 is free of transmission interference.

Coming to condition $R_1$, the only two pairs of symbols of the same type that can appear consecutively are $(b, c)$, and $(a, d)$. We can see that for any trace of the form *sbct* where $s$ and $t$ are traces, *scbt* is also a legal trace of QR42. Similarly for a trace of the form *sadt*, *sdat* is also a trace of QR42. Hence condition $R_1$, non-reliance on relative delays, is satisfied.

Checking $R_2'$ by studying the different traces requires extensive case analysis. A formal proof is to be preferred. However, here we take one example and show $R_2'$ for that example. Consider the input symbols $a, d$ and output symbol $b$. If $R_2'$ holds, $sactd \wedge scat \Rightarrow scatd$, for $s, t \in tT$. Trace *abacd* is in $tT$, and this matches with *sactd* for $s = ab$ and $t = \epsilon$. Trace *abca* is in $tT$, and this matches with *scat*, again for the same $s$ and $t$. However, we see that *abcad*, which matches with *scatd*, is also in $tT$.

Finally, let us check $R_3''$. Consider $b$ and $d$ to be the symbols of opposite types. It is to be checked that $sa \in tT$ and $sb \in tT$ implies $sab \in tT$. As an example, $acb \in tT$ and also $acd \in tT$. We see that $acbd \in tT$ and also $acdb \in tT$.

Since the command describing QR42 is repetitive, a formal proof of the above conditions through induction is possible but tedious.

## 10 Verification of Asynchronous Circuits

In this section, we first introduce the notion of conformance and present two uses of this notion for asynchronous circuit verification.

### 10.1 Conformance

Given two modules $M_1$ and $M_2$, if $M_2$ implements a subset of the traces of $M_1$, it is, in general, not "safe" to substitute $M_2$ for $M_1$ in an arbitrary context. For example, if the command of a C-ELEMENT is reduced to

$$\mathbf{pref}[(a?; b?); c!]$$

then clearly it is not safe to substitute this "C-ELEMENT" for a genuine C-ELEMENT. As an example, the environment can generate a $b$ action after it has received a $c$ action, thus choking this "C-ELEMENT". (Also, notice that Udding's condition $R_1$ is violated by this "C-ELEMENT".)

Now, consider the command describing the "quick return" unit QR42, and also commands describing two variants of QR42:

*QR42*:

$$E = \mathbf{pref}[a?; ((b!; a?) \parallel (c!; d?)); b!].$$

*MR42, a 'medium-speed return unit'*:

$$\mathbf{pref}[a?; ((b! \parallel c!); (a? \parallel d?)); b!].$$

*SR42, a 'slow-speed return unit'*:

$$\mathbf{pref}[a?; c!; d?; b!; a?; b!].$$

We notice that:

- Both $MR42$ and $SR42$ implement a subset of the traces of $QR42$.
- Yet, it is safe to substitute $SR42$ for $QR42$, but not $MR42$ in any context, and not lead to any choking. (The operation of the overall system would slow down as $SR42$ has no parallelism exhibited by $QR42$.)

What is it that makes the subset of traces provided by $MR42$ and $SR42$ "special"?

The answer to this question provided by Dill is through the notion of *conformance* given in definition 7 [14]. Basically, $SR42$ conforms to the behavior of $QR42$. Thus, $SR42$ accepts all the inputs that $QR42$ accepts and *does not generate* an output *unless* $QR42$ does so. (If not, it could cause failure of the surrounding circuitry when $QR42$ would not have.)

For a given input and output alphabet, conformance is a partial order, with a *least element* that conforms to all other modules. For example, the least element in the conformance ordering over the input alphabet $a, d$ and output alphabet $b, c$ is

$$\mathbf{pref}[a?|d?].$$

This module, called a *universal do-nothing*[13] can be realized using a block of wood with two nails labeled $a$ and $d$ driven into it, spaced apart! This fact graphically illustrates that when $M_2$ conforms to $M_1$, $M_2$ will be required to do no harm, but not required to do any good!

## 10.2 Using Conformance Checking for Verification

Having introduced conformance, we are now in a position to illustrate two of the verification techniques proposed by Dill. (These techniques are, in effect, also captured by Ebergen's definition of *decomposition*.)

### 10.2.1 Verification for Speed Independence

- Let $T_S$ be the trace structure specifying the desired behavior of a circuit.
- Let $T_I$ be a claimed implementation of the circuit.
- $T_I$ is a speed-independent circuit realization of $T_S$ if $T_I \preceq T_S$.

Relation $\preceq$ is established in Dill's verifier as follows:

- The environment "implied by" $T_S$, called the reflection of $T_S$ in section 8, is denoted by $\overline{T_S}$.
- $T_I \preceq T_S$ if the simulation (in the sense of section 8) of $T_I$ and $\overline{T_S}$ does not fail due to choking.
- Dill introduces the operator $|$, a function from two trace structures to a trace structure, that denotes the effect of "parallel composition", or "simulating the concurrent behavior" of two trace structures. In terms of $|$, $T_I \preceq T_S$ if $T_I \mid \overline{T_S}$ is free of failures due to choking.

### 10.2.2 Verification for Equivalence

As noted before, a universal do-nothing module over a certain alphabet conforms to any behavior over the same alphabet. Clearly, something more than conformance, which is merely a partial order, needs to be checked in order to certify purported implementations of given specifications.

In Dill's approach, in order to certify that $T_I$ is equivalent (with respect to traces accepted *and* generated), to $T_S$, it is necessary to establish *conformation equivalence*, $\overset{conf}{\equiv}$. $T_I \overset{conf}{\equiv} T_S$ if $T_I \preceq T_S$ and $T_S \preceq T_I$.

## 10.3 Illustration of the Verifier

In this section we illustrate the use of Dill's verifier for the following tasks: (a) certify the circuit obtained using Ebergen's technique to be speed-independent; (b) certify the circuit obtained using Chu's technique to be speed-independent.

Consider task (a). The steps in this verification are as follows:

1. Specify the desired behavior of $QR42$ by presenting a finite state machine that has the desired traces;
2. First specify the components used in the circuit in figure 18 using state machines; Then specify the interconnection among these components to form the realization;

3. Invoke the function `conforms-to-p` of the verifier.

Ebergen's circuit was successfully verified by the verifier, for absence of speed independence. However, the circuit in figure 15 exhibited a choking error, as shown by the following excerpt from a terminal session with the verifier:

```
(conforms-to-p (qr42-imp-chu) *qr42spec*)
Error: Choke in component 2, which is ... (essentially the gate NOR2)
Path: (A S-IN1 NOTUSED1 C B A)
```

The verifier indicates that the problem lies in gate NOR2. It also prints out the sequence of signal transitions leading to the error. The above printout shows, against `Path:`, that

- An `a` input came, and caused transitions on `s-in1`, and `notused1`. It also caused a transition on `c`, which caused a transition on `b`.
- This last `b` transition caused an `a` transition to be generated by the system.
- It can so happen that `NOR2` has not yet responded to the first `a` transition. If this is the case (and this *is* the choking error that the system reports), then when the second `a` comes, it can *cancel* the change in output that `NOR2` was about to schedule on its output; in other words, `NOR2` chokes!

Does this mean that Chu's technique is flawed? Or, is Dill's notion of conformance too strong?

### 10.4 A Deeper Analysis of Chu's Technique

To answer the above crucial question, we considered the simplest possible situation in which we could think of recreating the same error condition. It turns out that we can recreate the situation on the design of a wire!

Consider a unidirectional wire with input $a$ and output $b$ specified using Signal Transition Graphs (STGs). We name a node of the STG *wire*, and specify the graph through the following simple tail recursion:

$$wire = a+ \rightarrow b+ \rightarrow a- \rightarrow b- \rightarrow wire$$

The technique of section 7 can be applied to obtain a state graph, perform state assignment, obtain a Karnaugh map, and finally, *as is to be expected*, the technique yields the boolean equation $b = a$. Now how do we implement this equation? One way is to simply provide a wire. Another way is to use two inverters in cascade. Yet another way is to use the circuit shown in figure 19. This circuit does not conform to the behavior of a wire, according to

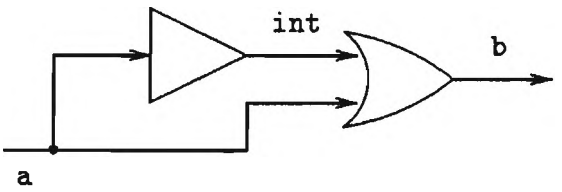


Figure 19: Wire Implemented as a Buffer and an Or gate

Dill's verifier—specifically, the sequence $a; b; a$ can cause the buffer to choke, assuming that it is very slow.

Here is the answer to all these apparent "problems" with Chu's technique. His synthesis technique assumes that for every circuit synthesized from the boolean equations obtained from the state graphs, all the internal nodes of the circuit stabilize before any output node of the circuit changes in value. Therefore, the choking errors reported by Dill's verifier can be ignored, and *it becomes the designer's responsibility* to make sure that this *fundamental mode assumption* is satisfied.

There is yet another place where caution is to exercised with the circuit of figure 15. (We had to observe this possible error manually, as Dill's verifier did not get past the choking error in gate `NOR2`.) When the signal `d` makes a transition, it is possible for `FF3`'s `s` and `r` inputs to be momentarily set to a `1`. This problem can be avoided by using a *reset dominant* SR flip-flop in place of `FF3`.

# 11 Concluding Remarks

## 11.1 On the Works Not Studied Here

In this section we attempt to briefly mention some of the works not studied here. (We are fully aware of the omissions we are bound to make in this attempt, and request the reader to kindly inform us of these.)

[1] presents a concurrent programming notation for describing computations slated for asynchronous and synchronous implementation. The notion of *event refinement and abstraction* is being studied.

[3] presents a technique for compiling OCCAM programs into VLSI circuits. The approach is to first generate direct implementations for the various OCCAM constructs, and then perform optimizations on the circuits generated.

[6] presents a tool for compiling asynchronous state machines into PPL[30,33]. [7] presents a verification technique for asynchronous processes based on a special form of finite automata.

[2] present techniques for verifying asynchronous sequential circuits using Temporal Logic model checking.

[9] present a technique for realizing speed-independent combinational logic in the asynchronous style. [10] present a technique for realizing speed-independent state machines in the asynchronous style. [12] present the design of a simple asynchronous RISC processor. [11] show that self-timed designs can also be self diagnostic.

[20] presents a technique for the direct implementation of asynchronous systems, using 'one-hot codes'. [39] presents a language for designing asynchronous systems. [24] has studied asynchronous computations in an abstract setting. [18] presents a system for verifying asynchronous designs, using a special form of asynchronous automata.

[26] has developed a technique for synthesizing asynchronous circuits from specifications in a language similar to Dijkstra's guarded command language. An asynchronous microprocessor has been compiled.

[36] presents a technique based on Signal Transition Graphs for realizing asynchronous systems. Efficient VLSI realizations of circuits using cascade logic, and signal-processing applications are emphasized.

Much of the early pioneering work in asynchronous design, including the detailed study of *metastability* was done by [37].

[41] has applied trace theory to asynchronous design. [42] investigated the application of trace theory for asynchronous design. [22] have developed a notation called *Synchronous Transition Systems*, and have designed asynchronous systems using it. They have also verified asynchronous systems using the Larch Theorem prover [17].

## 11.2 Closing Thoughts

We have merely scratched the surface of a very fast moving area of research. It is our observation that despite the attractions of asynchronous circuits in terms of modularity, performance, etc., their design calls for a totally new approach to circuit design, analysis, verification, synthesis, and testing, to name a few areas.

It seems that an asynchronous circuit designed without adequate mathematical analysis is bound to contain speed-dependent errors. "Mental simulation" of "typical" scenarios has proved to be too inadequate time and again, because of the vast number of combinations of scenarios that can actually arise in asynchronous designs. Dill[13] provides several interesting examples of what can go wrong (and what *has* gone wrong with published circuits!). On the other hand, techniques similar to [15] have (within their currently limited scope of application) have largely succeeded in reducing asynchronous design to a routine *calculational* activity. Such techniques do hold a great deal of promise. It is our opinion that in order to make progress in asynchronous design, in addition to widening the scope of existing formal design techniques, *our education system* calls for many changes. For a start, beginning

students in digital design must be taught rigorous mathematical techniques for asynchronous (and even synchronous!) design.

**Acknowledgements:** Sincere thanks to: (a) Venkatesh Akella, for initiating the study of asynchronous systems in the first author's research group; (b) Prof.Sanjay Rajopadhye for spending a week at Utah, discussing asynchronous system design and offering many insightful comments; (c) Prof.Ran Ginosar, for co-teaching an asynchronous system design course with the first author, and providing much valuable feedback; (d) Prof.Jo C. Ebergen, for spending five days at Utah, and teaching the authors many things—too many to mention; (e) Dr.Tam-Anh Chu, for spending five days at Utah and teaching the authors many things regarding his work, and other works; (f) Prof.David Dill, for making his verifier available to us, and for feedback; (g) Prof.Erik Brunvand, for many valuable suggestions; (h) Prof.Graham Birtwistle for his feedback and encouragement. This work was supported by NSF under award MIP-8902558.

## References

1. Venkatesh Akella. Action Refinement based Transformation of Concurrent Processes into Asynchronous Hardware. Ph.D. research in progress.

2. M. Browne, Edmund Clarke, D. Dill, and B. Mishra. Automatic Verification of Sequential Circuits using Temporal Logic. In *Proceedings of the Seventh International Conference on Computer Hardware Description Languages*, pages 98–113, North-Holland, 1985.

3. Erik Brunvand and Robert F. Sproull. Translating Concurrent Communicating Programs into Delay-Insensitive Circuits. In *International Conference on Computer-aided Design, ICCAD 89*, April 1989.

4. John Brzozowski and Jo C. Ebergen. *On the Delay-Sensitivity of Gate Networks.* Technical Report CSN90/X, Eindhoven University of Technology, 1990. *Submitted to IEEE Transactions on Computers.*

5. John Brzozowski and Jo C. Ebergen. *Recent Developments in the Design of Asynchronous Circuits.* Technical Report CS-89-18, Department of Computer Science, University of Waterloo, May 1989.

6. Tony M. Carter. *ASSASSIN: A CAD System for Self-Timed Control-Unit Design.* Technical Report UTEC-82-013, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, October 1982.

7. Eduard Cerny, P.Rioux, and C.Berthet. Comparison of Specification and Implementation for Asynchronous Circuits with Arbitrary Delays. In Luk J.M. Claesen, editor,

*Formal VLSI Specification and Synthesis (Proc. of the IFIP WG 10.2/WG 10.5 International Workshop on Applied Formal Methods for Correct VLSI Design, Houthalen, Belgium, November, 1989)*, pages 133–149, 1990.

8. Tam-Anh Chu. *Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specifications.* PhD thesis, Department of EECS, Massachusetts Institute of Technology, September 1987.

9. Ilana David, Ran Ginosar, and M.Yoeli. *An Efficient Implementation of Boolean Functions as Self-Timed Circuits.* Technical Report 678, Department of Electrical Engineering, Technion, Haifa, Israel, September 1988.

10. Ilana David, Ran Ginosar, and M.Yoeli. Implementing Sequential Machines as Self-Timed Circuits.

11. Ilana David, Ran Ginosar, and M.Yoeli. Self Timed is Self Diagnostic. 1990.

12. Ilana David, Ran Ginosar, and M.Yoeli. Self-timed Architecture of a Reduced Instruction Set Computer.

13. David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989. *An ACM Distinguished Dissertation.*

14. David L. Dill, Steven M. Nowick, and Robert F. Sproull. *Specification and Automatic Verification of Self-Timed Queues.* Technical Report CSL-TR-89-387, Computer Systems Laboratory, Stanford University, August 1989.

15. Jo C. Ebergen. *Translating Programs into Delay Insensitive Circuits.* Centre for Mathematics and Computer Science, Amsterdam, 1989. *CWI Tract 56.*

16. Arthur D. Friedman. *Fundamentals of Logic Design and Switching Theory.* Computer Science Press, 1986.

17. Stephen Garland, John Guttag, and Jorgen Staunstrup. Verification of VLSI circuits using LP. In George Milne, editor, *1988 Glasgow Workshop (IFIP WG 10.2) on Hardware Verification*, 1988.

18. Z. Har'El and Robert P. Kurshan. Software For Analytical Development of Communication Protocols. *AT&T Technical Journal*, January 1990. *To appear.*

19. C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978. Original article on CSP.

20. Lee Hollaar. Direct Implementation of Asynchronous Control Units. *IEEE Transactions on Computers*, c-31(12):1133–1141, December 1982.

21. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

22. J.Staunstrup and M.R.Greenstreet. Designing Delay Insensitive Circuits using "Synchronized Transitions". In Luk J.M. Claesen, editor, *Formal VLSI Specification and Synthesis (Proc. of the IFIP WG 10.2/WG 10.5 International Workshop on Applied Formal Methods for Correct VLSI Design, Houthalen, Belgium, November, 1989)*, pages 209–226, 1990.

23. Gilles Kahn and David MacQueen. Couroutines and Networks of Parallel Processes. In *IFIP-77*, pages 993–998, North-Holland, 1977.

24. Robert M. Keller. Towards a Theory of Universal Speed-Independent Modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974.

25. Zohar Manna and Pierre Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. In *Proc. of the Workshop on Logics of Programs, Yorktown-Heights, NY*, Springer-Verlag, 1981. LNCS.

26. Alain J. Martin. Formal Program Transformations for VLSI Circuit Synthesis. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, Addison-Wesley, 1990.

27. Alain J. Martin, Steven Burns, T.K.Lee, D.Borkovic, and P.J.Hazewindus. The Design of an Asynchronous Microprocessor. In C.L.Seitz, editor, *Proc. Decennial Caltech Conference on VLSI*, MIT Press, 1989.

28. C. A. Mead and L. Conway. *An Introduction to VLSI Systems.* Addison Wesley, 1980.

29. Dept. of Defense. *The ADA Language Specification, MIL-STD-1815-A.* American National Standards Institute, 1430 Broadway, NY (212) 642-4900, 1983.

30. Suhas S. Patil and T. A. Welch. A Programmable Logic Approach for VLSI. *IEEE Trans. on Computers*, C-28(9):594–601, 1979.

31. J. L. Peterson. Petri Nets. *Computing Surveys*, 9:223–252, September 1977.

32. Vaughan Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, (1):33–72, February 1986.

33. K.F. Smith, T.M. Carter, and C. E. Hunt. Structured Logic Design of Integrated Circuits Using the Stored Logic Array. *IEEE Transactions on Electron Devices*, ED-29(4):765–776, April 1982.

34. Eugene W. Stark. Graduate Seminar on "Concurrency Theory" taught at SUNY, Stony Brook, 1985.

35. Ivan Sutherland. Micropipelines. *Communications of the ACM*, June 1989. *The 1988 ACM Turing Award Lecture.*

36. David G. Messerchmitt Teresa H.-Y. Meng, Robert W. Brodersen. Automatic Synthesis of Asynchronous Circuits from High-level Specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

37. T.J.Chaney and C.E.Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

38. Jan Tijmen Udding. A Formal Model for Defining and Classifying Delay-insensitive Circuits and Systems. *Distributed Computing*, (1):197–204, 1986.

39. Jan Tijmen Udding and Mark B. Josephs. The design of a delay-insensitive stack.

40. Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

41. C. van Berkel, C. Niessen, M.Rem, and R.Saeijs. VLSI Programming and Silicon Compilation: a Novel Approach from Phillips Research. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, 1988.

42. Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design.* Springer Verlag, 1985. LNCS 200.

*Formal Aspects of VLSI Research Group*
*University of Utah, Department of Computer Science*

# Some Recent Asynchronous System Design Methodologies

GANESH GOPALAKRISHNAN (*ganesh@cs.utah.edu*)
PRABHAT JAIN (*jain@cs.utah.edu*)

*University of Utah*
*Dept. of Computer Science*
*Salt Lake City, Utah 84112*

**Keywords:** Asynchronous Design, Formal Specification, Verification, Delay Insensitivity, Speed Independence

**Abstract.** *We present an in-depth study of some techniques for asynchronous system design, analysis, and verification. After defining basic terminology, we take one simple example—a four-phase to two-phase converter—and present its design using (a) classical flow-tables; (b) Signal Transition Graphs of [8]; and (c) Trace Theory of [15]. We then present necessary and sufficient conditions for Delay Insensitivity, proposed by [38], and illustrate it on our example. Finally, we present the work of [13] on the verification of asynchronous circuits, and illustrate it on the circuits derived in the paper. The following points are emphasized: (i) presentation of techniques at more depth than in a general survey; (ii) illustration of all the aspects discussed on a common example; (iii) comparative study of the works presented. Many interesting works had to be left out, solely because of our lack of space and time.*