AUTOMATED SYNTHESIS OF MIXED-MODE (ASYNCHRONOUS AND SYNCHRONOUS) SYSTEMS

Pasupathi A. Subrahmanyam

Pasupathi A. Subrahmanyam *is a member of technical staff at in the VLSI Systems Research Department of AT&T Bell Laboratories, Holmdel, New Jersey where he works on formal methods for system design, VLSI CAD tools—including synthesis and verification tools—programming languages, and system design paradigms. He joined AT&T in 1984 with an M.S. in computer science from the Indian Institute of Technology, Madras, an M.S. in physics from the Indian Institute of Technology, Kanpur, and a Ph.D. in computer science from the State University of New York at Stony Brook.*

Many large scale integrated circuits and systems contain both synchronous and asynchronous subsystems (including self-timed subsystems). Examples include systems having asynchronous interfaces to busses or memories, and systems containing modules clocked by independent, locally generated clocks. This paper discusses specification and automated synthesis techniques for designing such systems. A graphical perspective of the temporal and interface constraints is provided via a timing diagram editor. The specification and synthesis techniques presented allow individual process implementations to be either synchronous, asynchronous, or combinational. We discuss factors influencing the decomposition of the overall system into subprocesses and the choice of implementation styles. Fragments of the design of a Processor Interface Board (PIB) are used to illustrate various concepts. The goal is to enable a designer to improve design quality by synergistically exploiting the advantages of both the synchronous and asynchronous design styles in a system, and to support experimentation with trade-offs in granularity and implementation strategies.

Introduction

Many large scale integrated circuits and systems contain a mixture of synchronous and asynchronous (i.e., self-timed) subsystems. Examples include systems having asynchronous interfaces to busses or memories, and systems that contain modules clocked by independent, locally-generated clocks. Most previous work on automated synthesis techniques has focused on synchronous systems;[12,3] and while there have been some attempts to automate self-timed designs,[1,8,2] almost no effort has considered the issues involved in automatically

Panel 1. Terms and Acronyms in This Paper

BIB	bus interface board
CAD	computer-aided design
CCS	Calculus for Communicating Systems
CHDL	C++-based Hardware Description Language
CMOS	complementary metal-oxide semiconductor
PIB	processor interface board
SG	state graph
STG	signal transition graph
VHDL	VHSIC Hardware Description Language (IEEE Standard)
VHSIC	Very High Speed Integrated Circuits
VLSI	very-large-scale integration

synthesizing systems that mix synchronous and asynchronous or self-timed systems, and in coupling them with pre-defined interfaces such as standardized bus interfaces. The lack of such efforts results mainly from a combination of two factors: an ideological split between the advocates of the two methodologies, and, until recently, insufficiently developed techniques for automating the synthesis of asynchronous and self-timed systems.

Motivated by these considerations, this paper discusses techniques for the specification and automated synthesis of such systems. The goal is to enable a designer to exploit synergistically the advantages of both the synchronous and asynchronous design styles in a system, and to support experimentation with trade-offs in granularity and implementation strategies. The added flexibility enables potentially improved design quality by allowing a designer to adopt the timing disciplines and architecture best suited to an application. Productivity is improved by providing automated support for various aspects of the design task.

The Context. Figure 1 depicts the general scenario presented in this paper. A designer starts with a conception of a system's desired behavior, interface, and performance. This conception is translated into a high-level initial specification.

Ideally, a specification at this level should avoid constraining the implementation by allowing for flexibility in the timing disciplines and the architecture used in it. The initial specification is then elaborated (either automatically or interactively) to yield descriptions for an ensemble of interacting subsystems that provide an implementation consistent with the specification. Each subsystem may in turn be implemented synchronously or asynchronously using appropriate synthesis techniques.

Many synthesis techniques generate, at some stage, a set of boolean logic equations to describe the system structure. Such descriptions of the boolean logic needed to implement the various subsystems may then be amalgamated, appropriately optimized, and eventually translated into mask layouts for circuit fabrication.

It is important to note that design iteration is not precluded from this process, since some characteristics of an intermediate design description may lead lead to a revision of earlier design decisions, and consequently revised designs.

Some of the important technical questions that arise in this context, relating to specification, decomposition, and synthesis, are the following:

- How can one specify a system to not imply (or unduly constrain) a timing discipline? What semantics underly such a specification? And, how can one gradually and smoothly incorporate the implications of adopting specific timing disciplines in this context?
- What criteria affect the choice of the implementation style for a subsystem? How do these criteria influence the architectural decomposition of a design?
- What techniques can assist in system decomposition, and in the synthesis that follows of the synchronous and asynchronous subsystems resulting from such a decomposition?

We have been experimenting with various techniques for addressing these issues. This paper provides an overview of a high-level specification technique, and the succeeding steps in system decomposition and subsystem synthesis.

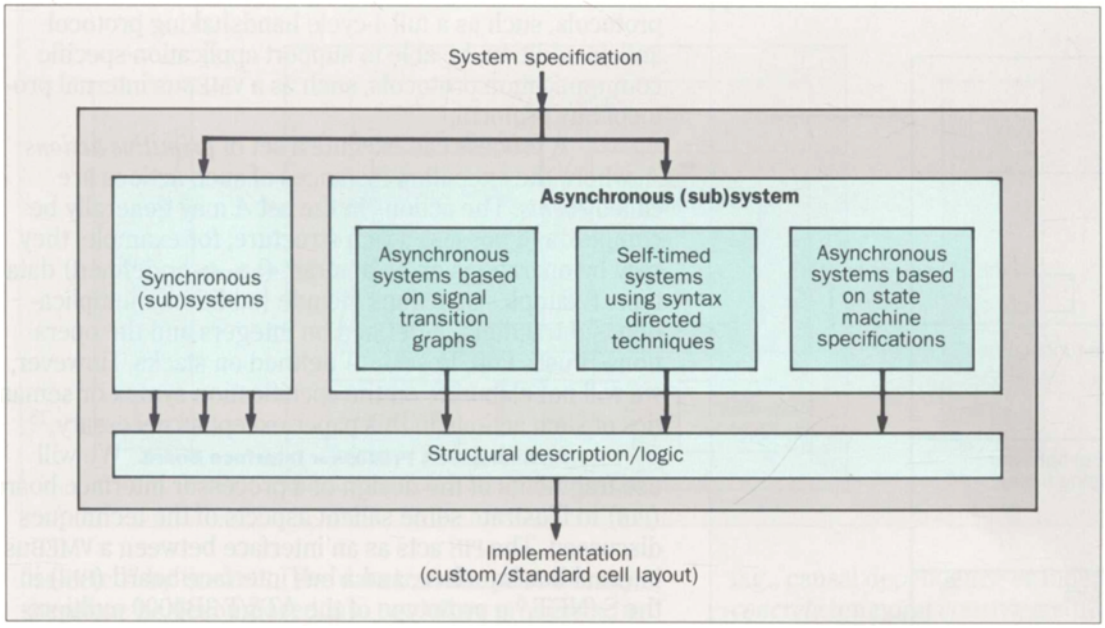

Figure 1. Synthesis of systems with synchronous and asynchronous components.

Process Specifications and Timing Constraints

We begin by discussing relevant aspects of a technique for specification of systems that exhibit a desired behavior, but do not necessarily have any commitment to a specific timing discipline or set of disciplines. In this discussion, a *specification technique* consists of an abstract *syntax* that defines the textual (or graphical) form of the specification, along with an associated *semantics*. The primitives underlying such an abstract syntax may be syntactically sugared to resemble the concrete syntax of languages such as VHDL[14] (Very High Speed Integrated Circuit Hardware Description Language, an IEEE standard) or CHDL (C++ Hardware Description Language), a hardware description developed by the author, and based on Concurrent C++.[4] ["Syntactic sugaring" refers to syntactically recasting (i.e., rephrasing) a statement to resemble a more familiar form, without changing the underlying semantics or meaning.] The input specification syntax we adopt here is a subset of a language based on a variant of a calculus for communicating systems (CCS)[11], syntactically sugared to resemble a Concurrent C++ program.

A program in the language describes a set of interacting processes. Each process (or hardware module) has several attributes. These include its external interface and behavior; the temporal aspects of its behavior, including interface protocols that must be obeyed; and its internal structure (see Figure 2).

An initial version of a system description can be either a single process that is eventually implemented using an ensemble of processes, or a set of interacting processes, where a process may have internal concurrency, i.e., it is not constrained to be sequential. Processes interact with each other by communicating over channels formed by connecting *typed ports*. Communication actions such as input and output are defined on the underlying port types. By generalizing the notion of port types from inputs, outputs, and bidirectional ports to

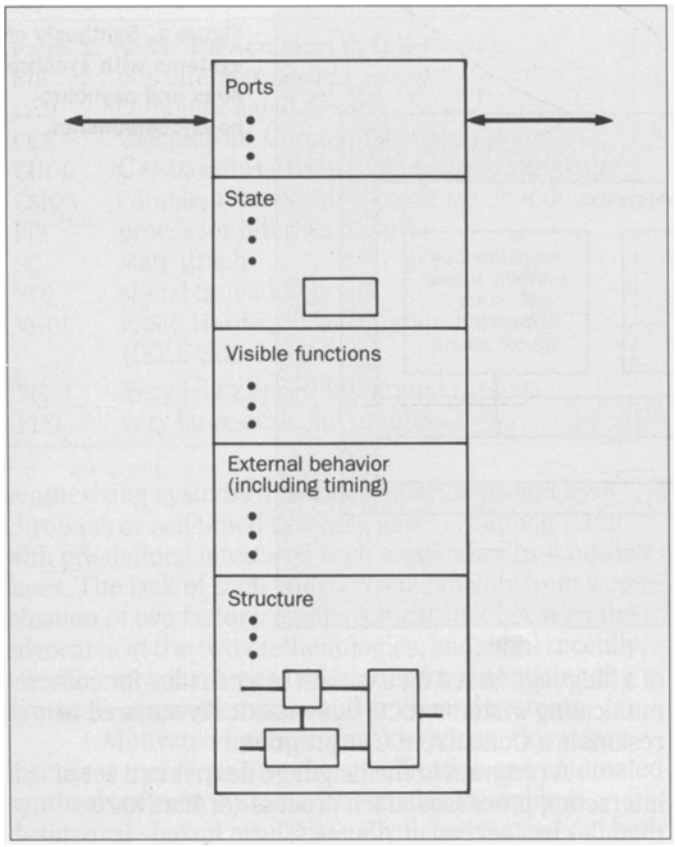

Figure 2. Some aspects of a module/process.

user-defined (i.e., abstract) port types at the specification level, we introduce the flexibility for *implementations* of such port types to have varying communication protocols, and to eventually use primitive input/output/bi-directional ports and signals. This enables a process to concurrently read and write values using such a typed port. Synchronization between processes is achieved by interprocess communication over a channel; some information may also be communicated during synchronization. Such a generalization makes it convenient to provide abstractions that support standard communication protocols, such as a full 4-cycle handshaking protocol and, in addition, be able to support application-specific communication protocols, such as a VMEBus internal protocol bus protocol.

A process can execute a set of *primitive actions* A, where the execution instances of such actions are called *events*. The actions in the set A may generally be complex and possess a rich structure; for example, they may be operations on an abstract (i.e., user-defined) data type. Examples of actions include {addition, multiplication, subtraction ... } defined on integers and the operations {Push, Pop, Reset, ... } defined on stacks. However, we will not elaborate on the specification syntax or semantics of such actions in this paper except as necessary.[15]

An Example: A Processor Interface Board. We will use fragments of the design of a processor interface board (PIB) to illustrate some salient aspects of the techniques discussed. The PIB acts as an interface between a VMEBus[7] internal bus interface, and a bus interface board (BIB) in the S/NET,[6] a prototype of the AT&T 3B4000 multiprocessor. It supports three categories of functions:

- Data transfer between the VMEBus and the BIB
- Parity and checksum logic to complement the data transfer between the VMEBus and the BIB
- Interrupt control functions to handle interrupts arising during data transfer, and asynchronous interrupts from the VMEBus and BIB.

The PIB specifications can be decomposed into submodules that support the functions in each of these categories (see Figure 3). The data transfer functions constitute the major function supported by the PIB. The rest of this paper will consider fragments of an implementation of the data transfer functions.

Each instance of a PIB is hard-wired to respond to some subset of requests from the VMEBus, determined by some combination of the input signals `Address` [7..31] and `AddressModifier`[0..6] from the VMEBus. When a read or write request is made by the VMEBus, indicated by the signal `AddressStrobeBar` going low (i.e., as specified by the VMEBus protocol), a particular PIB instance responds to this request only if it has been

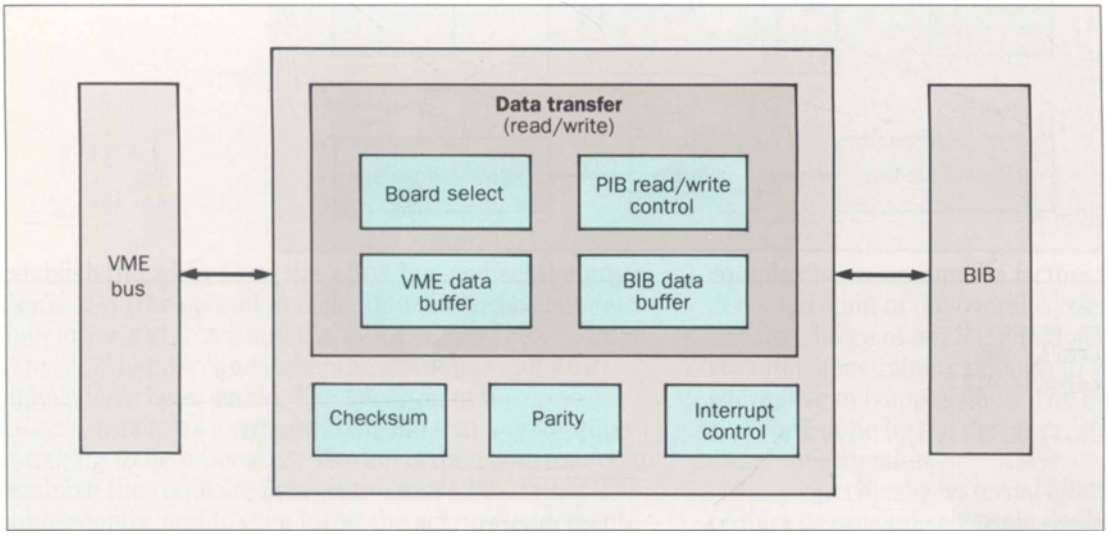

Figure 3. PIB decomposition.

"selected" by the VME. The data transfer process may therefore be decomposed into two other processes: a `BoardSelect` process that determines if a specific PIB instance should indeed respond to a read/write request, and a `Read/Write Control` process that handles the details of the transaction. This `Read/Write Control` process is thus triggered by the `BoardSelect` process, and in turn orchestrates the data transfers into and out of the local data buffers that interface to the VMEBus and BIB.

PIB System Description. The top level of description of the PIB system has the form shown in Figures 4 and 5. It consists of:

- The abstract port interfaces to the VME and BIB.
- The behaviors of the VME slave interface (`VMESlaveInterface`), the main PIB body, and the BIB master interface (`BIBMasterInterface`.

The architectural details are initially unspecified.

Specifying Timing Constraints. An important component of communication protocols (and many hardware interface specifications) is the set of timing constraints associated with the protocol. Such constraints may be broadly classified as either *abstract* temporal constraints—e.g., causal dependence or independence of events—or *concrete* temporal constraints that associate attributes with an interval of interest, e.g., < ***minimum***, ***maximum*** > (or even average) values of the duration between events. If a *signal* is viewed as a real-valued function of time, an *event* may be viewed as a boolean-valued predicate on signals. Often, only an abstracted (i.e., discrete) set of signal values is used in the context of interface protocols, e.g., `{0, 1, X (don't-care), Hi-Z (floating nodes), stable, unstable, rising, falling}` or subsets thereof. Similarly, typical events of interest tend to be signal transitions on selected signals. "Specification sheets" for systems frequently contain timing diagrams that show the *overall* communication action and timing constraints during an operation, augmented by informal natural language descriptions of the protocol for each process.

To assist in graphically displaying and editing many temporal constraints in a familiar format, we have designed an interactive menu-driven *window-oriented object-based editor* for specifying such constraints as 2-dimensional timing diagrams (Figure 7). Internally, the constraints are represented as expressions using an

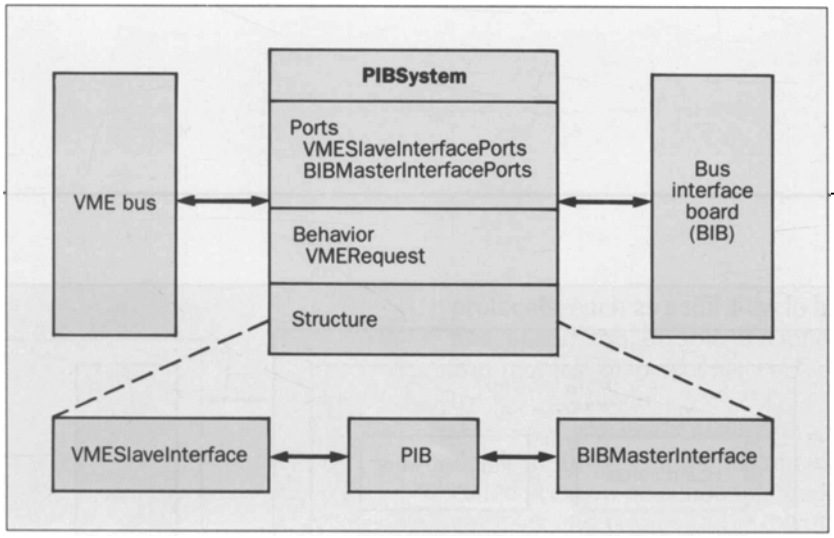

Figure 4. PIB system interface.

Figure 5. Skeletal specification of the PIB system.

```
module PIBSystem is
bidirectional port struct {
VMESlaveInterfacePorts VME;
BIBMasterInterfacePorts BIB;
};
//.....
public:
VMERequest;
//.....
behavior
{
PIBSystem = VMESlaveInterface | PIB | BIBMasterInterface
// Behavior defined by the composition (denoted by "|") of subsystem behaviors
}
//.....
architecture of PIBSystem
// .....
// The architecture consists of 3 submodules VMESlaveInterface | PIB | BIBMasterInterface
```


appropriate set of primitives derived from temporal logic and algebra. This approach combines the benefits of:

- A familiar graphical specification and presentation technique.
- The rigor associated with using description primitives having well-defined mathematical semantics.

`BIBMasterInterface`: **Ports and Behavior.** To illustrate the context in which such graphical perspectives are used, consider the description of the BIB master interface module, a subsystem of the PIB described above (Figure 6). The PIB acts in a "master" mode when requesting Read and Write services from the BIB. The overall description of the BIB master interface appears in Figure 8; its behavior supports the Read and Write operations invoked by the PIB.

Reading from the BIB (`BIBMasterInterface:Read.`) When reading data from the BIB, the BIB read protocol must be obeyed, and parity must be checked. Representative details of this protocol are given in the Read function (Figure 9).

For example, Figure 7 shows the timing diagram associated with a read operation on the BIB. This diagram show the *overall* transaction across the interface, in that it contains the actions performed by both the master (PIB) and slave (PIB) processes. When the PIB wants to read data from a specific BIB address, it must first

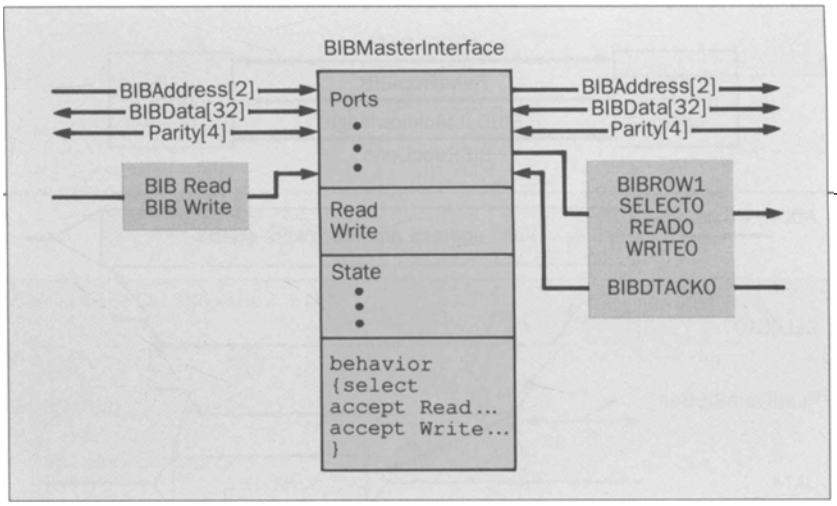

Figure 6. BIBMasterInterface.

establish the address on the ADDR bus and set the ROW1 signal to 0 (the operation code for a read). The PIB must then lower SELECT0 and the ReadFromBIBBar strobe, at least 50 nanoseconds after the ADDR bus and ROW1 signals have been stable. The lowering of the strobe ReadFromBIBBar signals to the BIB that a read request is waiting to be processed. The BIB is then constrained to establish the requisite data on the DATA bus within 200 nanoseconds, and to then lower the acknowledgement signal BIBDataAcknowledgeBar 70 nanoseconds after the data is stable. Lowering this acknowledgement signal indicates to the PIB master that the data it requested is available on the DATA bus. When the PIB is done with this data, it withdraws its read request by raising the ReadFromBIBBar signal. Then the read cycle is completed on both the master and slave side. The PIB master resets the signal SELECT0 to high, and releases the ADDR and ROW1 lines. In parallel, the slave releases the DATA bus and resets the BIBDataAcknowledgeBar signal to high within 290 nanoseconds.

For the purposes of synthesis, it is sufficient—and perhaps even desirable—to explicitly specify only the requirements imposed by a protocol on the individual processes (in this example, the PIB master and BIB slave). The combined behavior of the overall system can then be computed from this information and depicted graphically if desired.[16]

Decomposition Into Synchronous and Asynchronous Components

External interface constraints, system performance requirements, and system or subsystem design complexity are among the factors that influence the decomposition of the overall system into subprocesses, and the choice of implementation styles. These criteria are often constraining enough to suggest a small set of alternative decompositions. The decomposition is currently done by the designer; automated support is under investigation.

Specifically, external interface protocols may require or preclude a certain design style. For example, inherently asynchronous external inputs preclude a completely synchronous design; and externally specified signaling schemes such as that of the VMEBus interface[7] preclude a self-timed design that universally adopts a dual-rail encoding scheme. (A *dual-rail* scheme requires two wires to encode one bit of information, i.e., the set of values {0, 1}, in contrast to the more commonly used single-wire technique to encode one bit.) In principle, a completely unconstrained asynchronous design paradigm is always feasible, but is rarely adopted for complex systems because of the problems associated with such designs, including races and hazards.

Synchronous designs alleviate the design problems associated with unrestricted asynchronous designs, but preclude the designer from taking advantage of data-dependent performance improvements. Because of their speed-independence and composability, self-timed designs are more portable across technology enhancements. By contrast, synchronous systems with finely-tuned clock distribution networks tend to involve considerable simulation and redesign, and are therefore less robust and less portable.

The area overhead associated with using analog

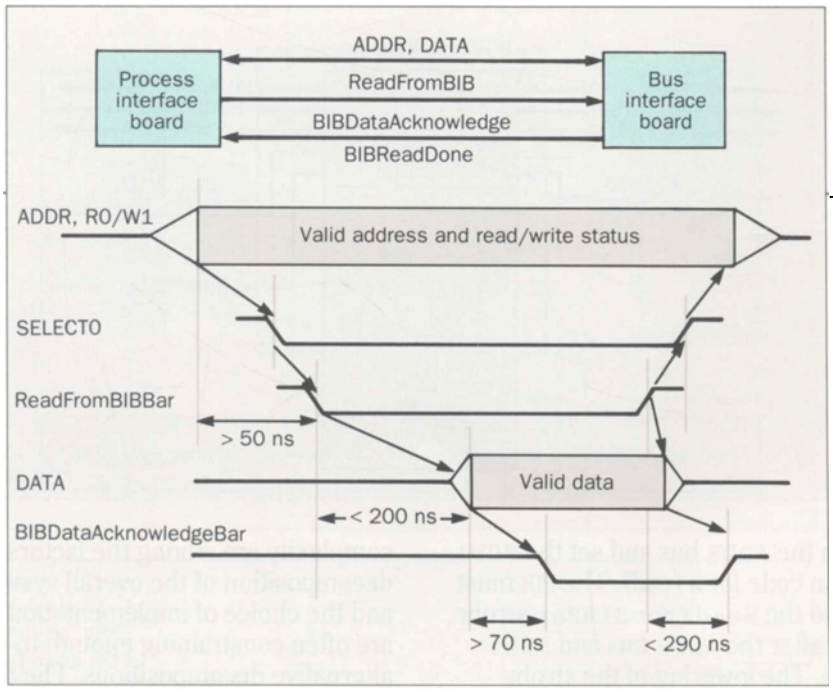

Figure 7. BIB read timing.

delays to generate completion signals is small. The drawback of this technique is that a detailed simulation and analysis of the circuit (somewhat analogous to synchronous designs) may be needed to compute the minimal values of delays that can be safely used. By comparison, the area overhead associated with dual-rail encoding can be significant, roughly doubling the size of a circuit if the data path is dominant; the extra logic may occasionally even degrade system performance.

Example: Data Transfer Module Decomposition. The PIB Read/Write control module handles most of the communication protocol, and is mandated to communicate via channels associated with explicit request-acknowledge signals. These characteristics make this module a good candidate for self-timed implementation. The `BoardSelect` process, on the other hand, involves a 32-bit wide data path with an asynchronous input and an asynchronous output. It is therefore well-suited for implementation as an asynchronous combinational module. The processor itself, though not part of the subsystem considered here, is implemented using a synchronous discipline.

Synthesis Techniques

Our synthesis method allows implementation of individual processes to be either synchronous, asynchronous, or combinational. Given a description of a system such as that illustrated in the Process Specifications and Timing Constraints section, augmented with directives for implementing various subsystems synchronously or asynchronously, each subsystem description is translated into a form that accepted by a program performing the appropriate kind of synthesis. In particular, we have interfaced to existing AT&T programs for synthesizing synchronous and combinational circuits. In addition, three techniques to synthesize asynchronous circuits have been explored.

- The first technique consists of using a set of self-timed circuit structures to implement the basic linguistic primitives in the language. This set performs a syntax-directed translation of the initial description to obtain an intermediate circuit description, and then improves the quality of the implementation by using local (peephole) and global (logic optimization) techniques. This technique uses combinational logic to generate the completion signals associated with the self-timed circuits, and a "one-hot" state-assignment paradigm[5]. One flip-flop is used for each state in the state diagram; this is set on entering the state, and reset upon exiting it.
- The second technique consists of generating completion signals using analog delays instead of combinational logic; the rest of the circuit generation process remains the same.
- In the third technique, the fragment of the initial

```
module BIBMasterInterface(BIBMaster, BIB)
{
//.....................................................................
inputs:
BIBDTACK0 boolean DTACK0; /* input */
outputs
port struct { // representation:
BIBAddress
{  ADDR[0..1]; /* output */ }
/* additional channel control signals: */
boolean BIBROW1, SELECT0, WRITE0, READ0, /* output */
bidirectional ports
BIBDataAndParity {
                Boolean Data[32]; /* bidirectional */
                Boolean Parity[4]; /* bidirectional */
}
}
//.....................................................................
public {
   BIBData Read(BIBAddress);
   BIBDataAcknowledge Write(BIBAddress, BIBData);
}
//.....................................................................
behavior of BIBMasterInterface is
{ do-forever
   select
           accept Read: Read(BIBOp, BIBAddress);
   or
           accept Write: Write(BIBAddress, BIBData);
}
```

Figure 8. BIB master interface.

description to be asynchronously implemented is first translated into an appropriate signal transition graph[2] (STG). As elaborated upon below, a logic description of the circuit is generated from this graph, and then optimized using existing logic optimization techniques. The state assignment is derived from consistent live-safe markings of the intermediate signal transition graph.

Asynchronous Circuit Synthesis From An STG. The STG resulting from the translation embodies the causal relations between signal transitions. Any concrete timing constraints between events are mapped into labels associated with the edge connecting these events.

The synthesis proceeds by deriving a *state graph* SG from the STG.[2] The input and output signals present in the STG are used to encode states in SG, i.e., each state s is labelled with a *unique* vector v of signal values. A state transition from s_1 to s_2 in the state graph is caused by the rising or falling transition of some signal s in STG. Consequently, the signal vectors labelling s_1 and s_2 must differ only in the value of the signal s. Further, if the state transition is caused by signal s rising, then the value of s must be 0 in s_1 and 1 in s_2, or vice versa if the state transition is caused by the signal s falling.

An STG can be mapped into a state graph with unique state codes only if both the rising and falling

Figure 9. Read operation on the BIB.

```
BIBData BIBMasterInterface::Read(BIBOp, BIBAddress)
{
/* Local Variables */
/* Read Protocol: */
/*--| assume initial conditions: SELECT0, READ0 (strobe), DTACK0 high. */
/*--| assume initial conditions: ADDR, R0W1, DATA not established. */
[ ADDR = BIBAddress || BIBR0W1 = BIBOp ] // establish op and address
SELECT0-; /* lower SELECT0, thus selecting BIB */
wait-for (50 ns);
READ0-;  /* strobe READ operation */
wait-for BIBDTACK0-;
/*--| guaranteed within 270 ns for normal read, and within 570 ns on status read */
/*--| 70 ns. setup time guaranteed by BIB (slave) */
BIBMaster.ProcessData(BIBDataAndParity); /* read in BIBData and use it in PIB */
READ0+; /* terminate READ operation */
/* Raise the SELECT0 signal; and release control of the Address bus and BIBR1W0 */
[ SELECT0+; || ADDR = FLOAT; || BIBR0W1 = FLOAT; ]
/* end of Read Protocol cycle */
}
```


transitions of the same signal never are simultaneously enabled. This fact enables the desired property to be checked directly for a given STG, rather than after the derivation of SG. If a state graph with unique state codes cannot be derived from the initial STG, then additional internal signals are introduced so as to guarantee unique codes. Alternatively, additional arcs may be added to the STG, although this has the deleterious effect of reducing the concurrency allowed in the circuit.

Once a state graph SG is obtained from the STG, a logic function describing each signal is obtained from SG and then mapped into a circuit. It is possible for an arbitrary circuit implementation of a state graph to exhibit hazards. Informally, a *hazard* is a transient (spike or glitch) on the output of a circuit that follows a legitimate change in the inputs; it typically results from unequal path delays through a combinational network, and is a circuit malfunction because the transient is inconsistent with the specification. A *static hazard* is a $1\rightarrow0\rightarrow1$ transition or a $0\rightarrow1\rightarrow0$ transition in a signal when no change should have occurred. A *dynamic hazard* is a $1\rightarrow0\rightarrow1\rightarrow0$ transition or a $0\rightarrow1\rightarrow0\rightarrow1$ in a signal when only the transition $1\rightarrow0$ or $0\rightarrow1$ should have occurred.

The analysis of hazards in a circuit depends on the delay model that properly mirrors the physical implementation. For example, wires connecting circuit elements may be modeled as having zero delay, a finite (but bounded) delay, or an unbounded delay. If the wires are assumed to have zero delay, the delay associated with a gate may be modeled as being "lumped" at its output. In addition, the occurrence of hazards depends upon whether inputs are allowed to change before the circuit has stabilized, and on whether multiple inputs can simultaneously change.

There are two categories of approaches for avoiding hazards in a circuit: *a posteriori* and *a priori*. An *a posteriori* method consists of first constructing a circuit, and then analyzing it for the presence (or absence) of hazards using the most appropriate delay model. Existing analysis techniques permit such analysis using both bounded and unbounded delay models, although they become intractable for large circuits.[16] If a hazard is feasible, then a redesign needs to remedy this defect. An *a priori* method ensures the absence of hazards by using

```
process PIB-ReadWrite-Control
/* Interface */
Inputs: PIBRequestForMe (Receive<PIBRequestForMe>, Transmit<VMEDataAcknowledge>),
RWOp (Req<ReadRequest, WriteRequest>, Ack<VMEDataAcknowledge>);
Outputs: ReadFromBIB (Req<ReadFromBIB>, Ack<BIBDataAcknowledge>),
         WriteToBIB (Req<WriteToBIB>, Ack<BIBDataAcknowledge>),
         TransmitDataToVME (Req<TransmitDataToVME>, Ack<AddressStrobe>),
         DataAcknowledge (Req<VMEDataAcknowledge>, Ack<AddressStrobe>),
         LoadPIBBuffer;
Internal State: boolean Busy initially false;
/* Behavior */
do forever {
wait-until request-on(PIBRequestForMe) and (not Busy);
switch (RWOp) {
        case 1:  ReadFromBIB; TransmitDataToVME; DataAcknowledge
    |   case 0: raise Busy; LoadPIBBuffer; DataAcknowledge; WriteToBIB; lower Busy
}
}.
```

Figure 10. Behavior of the PIB read-write control module.

a set of transformations and circuit implementation techniques which guarantee (i.e., enforce) the assumptions made about the circuit delay model.

Static hazards can be prevented in a two-level circuit implementation derived from the state graph by the inclusion of redundant prime implicants in the cover. (A two-level circuit consists of a bank of AND-gates followed by a bank of OR-gates.) If a multi-level logic circuit is preferable for reasons of area or delay efficiency, it may be obtained by applying a restricted subset of logic optimization techniques, e.g., algebraic factorization. This is because the static hazards in a two-level logic circuit are invariant under the application of associative, distributive and DeMorgan laws.[17] Dynamic hazards can be masked by the introduction of appropriate delay elements and state holding elements.[17]

In the case of a circuit derived from a state graph, the physical circuit implementation must preserve the transition ordering in the STG. A hazard can occur if the time between two transitions is less that the difference in delay between two different paths in a subcircuit computing the value of a signal. If wire delays are assumed to be bounded, then such hazards can be avoided by introducing additional delays along paths so as to avoid this condition.

A different example of an *a priori* method for unbounded wire delay model is the use of dual rail encoding and a circuit implementation technique which ensures that the delay along the two paths of a fork are equal; this is referred to as the isochronic fork assumption[8] and is relevant to the first technique described above.

The paths to the physical circuit layout are similar. This phase uses existing tools, and can produce either standard cell or custom CMOS (complementary metal-oxide semiconductor) layouts. The rationale for exploring more than one strategy is that, depending on the context of use, one of these strategies may yield a design superior to that generated by others. Such an option is useful, because a primary intent of this experiment was to provide architectural alternatives to the designer.

Example: PIB Read-Write Control Behavior. Figure 10 shows a description of the behavior of the PIB read-write control module that is suitable for generating a self-timed module. The active port `ReadFromBIB` interfaces to a

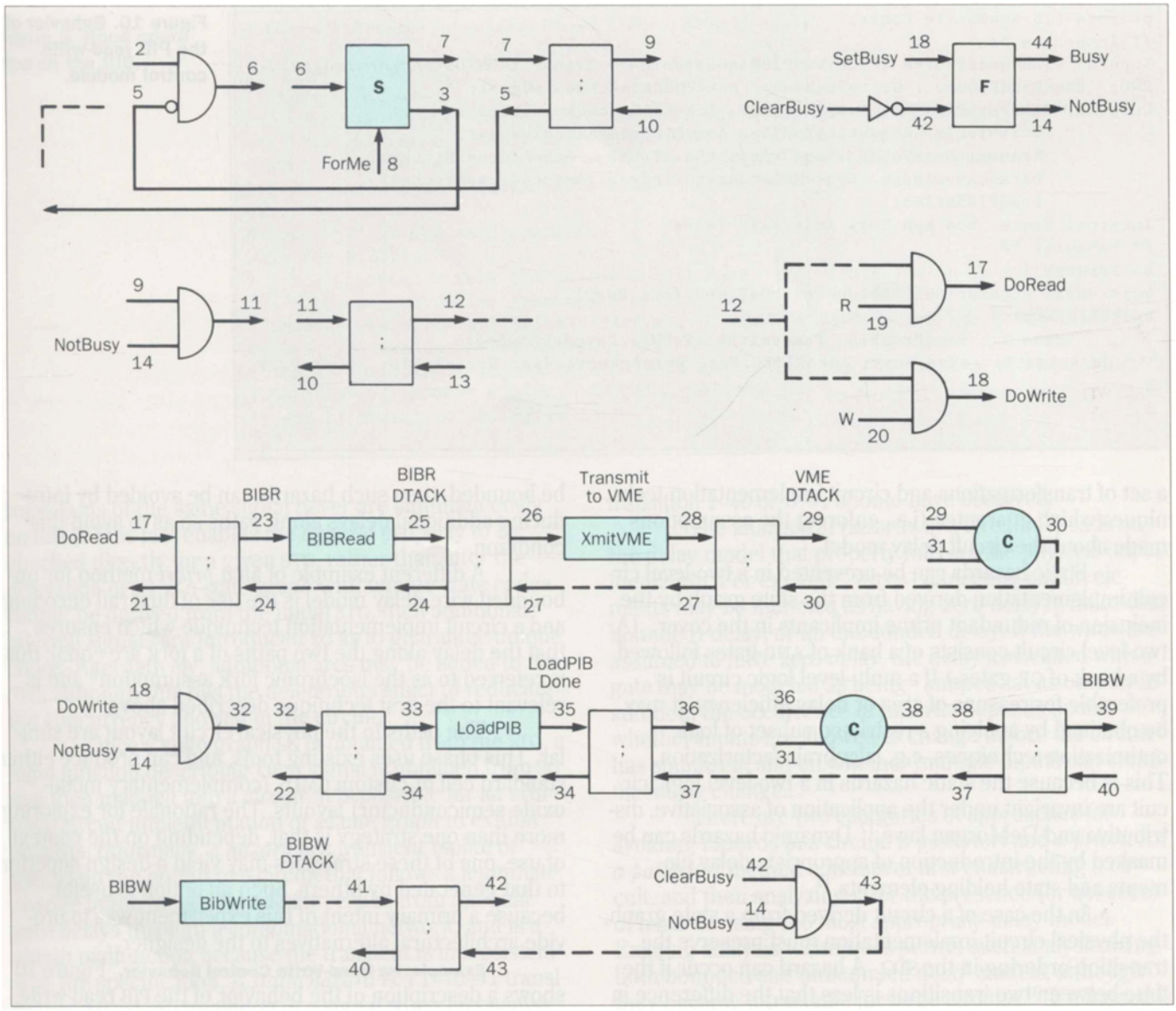
2
5
6
6
S
7
3
ForMe
8
7
5
;
9
10
SetBusy
18
ClearBusy
42
F
44
Busy
NotBusy
14
9
NotBusy
14
11
11
;
12
10
13
12
R
19
17
DoRead
W
20
18
DoWrite
DoRead
17
;
21
BIBR
23
BIBRead
24
BIBR
DTACK
25
;
24
26
Transmit
to VME
XmitVME
27
28
;
27
VME
DTACK
29
30
29
31
C
30
DoWrite
18
NotBusy
14
32
32
;
22
33
LoadPIB
34
LoadPIB
Done
35
;
34
36
36
C
31
38
38
;
37
BIBW
39
40
BIBW
DTACK
BIBW
BibWrite
41
;
40
42
43
ClearBusy
42
NotBusy
14
43

Figure 11. Fragmented view of a circuit obtained when the PIB read-write control specification is translated. The state-holding flip-flop (F) is created for the internal state variable `Busy`**. Syncronizer element (S) detects the asynchronous** `ForMe` **signal.** `DoRead` **is raised when read is requested, and** `DoWrite` **when a write operation is requested.**

channel used to communicate with a module that handles the details of the read protocol between the PIB and BIB. Analogously, `WriteToBIB` is used to communicate with a module that handles the details of the write protocol between the PIB and BIB. `TransmitDataToVME` transfers data from the internal PIB buffer to the VMEBus. `DataAcknowledge` signals the end of a read/write cycle to the VMEBus; `raise Busy` sets the internal state to be true. `LoadPIBBuffer` transfers data from the VMEBus into the internal PIB buffer.

Example: PIB Read-Write Control Synthesis. Figure 11 shows fragments of a circuit obtained when the PIB read-write control specification (Figure 10) is translated using a program based on the techniques presented here. A state-holding flip-flop (labeled F) is created for the explicitly declared internal state variable `Busy`; the two outputs of this flip-flop denote `Busy` and `NotBusy` signals. A synchronizer element (labeled S) detects the asynchronous arrival of a value 1 on the signal `ForMe` from the BoardSelect module. When this signal is detected, and the `NotBusy` signal is high, the appropriate sequence of actions begins; a semicolon is used to label the sequencing modules. The signal labeled `DoRead` is raised when a read operation is requested, while the signal labeled `DoWrite` is raised when a write operation is requested.

For example, if a read is requested, the signal `DoRead` is eventually raised high, triggering the BIB read protocol (not shown in the figure). When the asynchronous read operation from the BIB completes, this is signalled by the `BIBDataAcknowledge` going high (actually, `BIBDataAcknowledgeBar` going low, which is logically equivalent). The data on the BIB data bus is then transmitted to the VMEBus ; the `VMEData-Acknowledge` signal is then raised, signifying completion of the cycle. Analogously, the sequence of actions corresponding to a write request mirrors the specified sequence of actions; the one distinguishing characteristic is the explicit setting (and subsequent resetting) of the flip-flop representing the busy state.

The implementation just discussed makes explicit the separation between the PIB read-write control module and the module used to implement the BIB read and write protocols. Such a separation is by no means necessary, and therefore several variations of the above implementation are possible.

Example: `BoardSelect` **Implementation.** As mentioned earlier, each PIB instance responds to a subset of requests from the VMEBus, determined by some combination of the input signals `Address[7..31]` and `AddressModifier[0..6]` from the VMEBus; we will refer to this as the `BoardSelect` function. The `BoardSelect` logic involves an asynchronous input `AddressStrobe` that affects the computation of the `BoardSelect` function. The output `ForMe` of the `BoardSelect` module triggers much of the computation in the PIB. The logic for such a module can be generated by using standard synthesis tools for combinational and synchronous system synthesis.

Summary

The primary goal of the specification and synthesis techniques discussed in this paper is to enable a designer to exploit asynchronous, self-timed, and synchronous disciplines when and if appropriate. Other goals were to increase the practicality of such techniques and tools: this was done by introducing three features. The first was to support the design of self-timed systems where completion signals are generated by using delay elements, instead of being generated combinationally. The second was to provide a facility to explicitly identify both request-acknowledge signals lines to support externally fixed interfaces (such as the VMEBus2 interface).

The third was to support the specification of communication protocols in terms of signal transitions.

We believe the ability to merge paradigms that hitherto have been isolated can move both automated and interactive synthesis systems a step closer to assisting the design of "real" systems. The use of high-level specifications incorporating the notion of processes, typed ports and system timing is important in such a context, although the best linguistic incarnation for these notions is open to further exploration. In common with a major goal of high-level design tools—to support rapid prototyping and the exploring alternative architectures—the overall granularity of the various types of subsystems can be controlled by the user. There are some limitations in the current implementation of the system, owing to the fact that the tools are not completely integrated with the other parts of the existing design environment. Our preliminary results are encouraging, and ongoing work is directed towards explicating the theoretical underpinnings,[13] exploring strategies for decomposing systems into subprocesses, further experiments, and improving the optimizations performed.

Acknowledgments

Thanks to Bryan Ackland for extremely valuable feedback on earlier drafts of this paper, David Horn for the many hours he devoted explaining to me the intricacies of PIBs and BIBs, Mary Keefe for implementing a new version of the timing diagram editor, and Mark Yu for enlightening discussions.

References

1. T. S. Balraj and M. J. Foster, "Miss Manners: A Specialized Silicon Compiler for Synchronizers," *Proceedings of the 1986 MIT Conference on Advanced Research in VLSI*, MIT Press, Cambridge, Massachusetts, 1986, pp. 3-20.
2. T. A. Chu, "Synthesis of Self-Timed Circuits From Graph-Theoretic Specifications," Ph.D. Thesis, Massachusetts Institute of Technology, June 1987.
3. R. K. Brayton et al., "The Yorktown Silicon Computer," *Silicon Compilation*, ed. D. Gajski, Addison-Wesley, Reading, Massachusetts, 1988, pp. 204-311.
4. N. Gehani and W. Roome, *The Concurrent C Programming Language*, Silicon Press, Summit, New Jersey, 1989.
5. L. A. Hollaar, "Direct Implementation of Asynchronous Control Units," *IEEE Transactions on Computers*, Vol. C-31, No. 12, December 1982, pp. 1133-1141.
6. S. R. Ahuja, "S/NET: A High Speed Interconnect for Multiple Computers," *Proceedings of the National Communications Forum*, Chicago, Illinois, September 24-26, 1984, pp. 245-249.
7. *IEEE Standard for a Versatile Backplane Bus: VMEBus*, IEEE Press, New York, 1988.
8. A. Martin, "The Design of a Self-Timed Circuit for Distributed Mutual Exclusion," *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Computer Science Press, 1985.
9. T. H.-Y. Meng et al., "Asynchronous Processor Design for Digital Signal Processing," *IEEE ICASSP*, April 1988.
10. B. Moszkowski, "A Temporal Logic for Multilevel Reasoning About Hardware, *IEEE Computer*, Vol. 18, No. 2, February 1985, pp. 10-21.
11. R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag LNCS 92, 1980.
12. R. A. Newton and B. Preas, *25 Years of Electonic Design Automation*, Association for Computing Machinery, 1988.
13. P. A. Subrahmanyam, "Specification and Synthesis of Mixed-Mode Systems: Mathematical Aspects," *Proceedings of MSI Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, ed. M. Lesser and G. Brown, Springer-Verlag LNCS 408, 1989, pp. 202-223.
14. *IEEE Standard VHDL Language Reference Manual*, IEEE Std-1076-1987, IEEE Press, New York, 1986.
15. J. A. Goguen, "Parameterized Programming," *IEEE Transactions on Software Engineering*, Vol SE-10, September 1984, pp. 582-552.
16. P. A. Subrahmanyam, "Applications of Finite State Modelling and Analysis in Asynchronous/Synchronous Circuit Design," *Formal VLSI Specification and Synthesis*, VLSI Design Methods, Vol. 1, ed. L. J. M. Claesen, North-Holland Elsevier, 1990, pp. 247-260.
17. S. H. Unger, *Asynchronous Sequential Switching Circuits*, John Wiley, New York, 1969.

(Manuscript received November 9, 1990)