High-Level Modeling and Design of Asynchronous Interface Logic

ALEXANDRE V. YAKOVLEV
ALBERT M. KOELMANS
University of Newcastle upon Tyne
LUCIANO LAVAGNO
Politecnico di Torino

The authors' new methodology uses formal models of concurrency to synthesize speed-independent circuit implementations from high-level, abstract behavioral specifications. This methodology offers the advantages of preserving behavioral semantics and better control flow abstraction. It also provides heuristics at a higher level of abstraction to help solve the complete state-coding problem.

THE CRUCIAL ADVANTAGE of our methodology is preservation of the behavioral semantics linking the high-level, abstract synthesis and low-level, logical synthesis of asynchronous control circuits. We accomplish this by using the language of Petri nets at both stages. Our methodology also offers other major advantages:

- We obtain better abstraction of the synthesized circuit's control flow. For example, we can model and design a control circuit for a *k*-place buffer suitable for any buffer type (FIFO, LIFO, or RAM).
- We can derive useful heuristics at a higher level of abstraction. For example, by using the synchronization slack concept,[1] we avoided or resolved the complete state-coding problem for a control circuit at its abstract specification level. Solving this problem is crucial to ensuring the implementability of the initial description.[2,3]

We demonstrate our methodology using the classical example of interface circuitry, a FIFO buffer controller.

Two-stage methodology

We usually define a self-timed system as a collection of self-timed modules or elements that communicate using asynchronous protocols.[4] Such a system does not require a global clock signal. The causal relationships among the module actions order all system-level events in time. The designer guarantees correct operation by establishing an order that must be preserved in the final circuit.

We base our methodology[1] on this structural point of view and address high-level behavioral composition using interconnected Petri nets. We call the basic object in our hierarchy a discriminator, referring to a control component's natural ability to discriminate between signal transitions on its terminals during an operation, according to some prescribed strategy. As viewed from the user or environment perspective, a characteristic predicate specifies each discriminator. This predicate describes sequences (traces) of valid input-output behavior.

We initially use labeled Petri nets to model a discriminator's internal dependencies between actions. In a labeled Petri net, some communication action between the circuit and environment labels each transition.

Straightforward implementation of

0740-7475/95/$04.00 © 1995 IEEE

this model would require translation of abstract actions into up and down signal transitions. This task, however, is too complex to be solved at a high level. Therefore, we use hierarchical decomposition to iteratively refine the labeled Petri net into an interconnection of simpler ones. After decomposition, these simpler Petri nets can be expanded at the signal transition level and synthesized separately. The circuit obtained by structural composition of the submodules will satisfy the specification.

Hence our design methodology has two major stages: abstract or symbolic synthesis, and logic synthesis at the gate level. Abstract synthesis of the control circuit consists of the following steps:

1. We first perform an abstract decomposition of functionally independent units using characteristic predicates on traces and labeled Petri nets. This step characterizes each unit as a certain type of discriminator.
2. If direct translation into a circuit is too difficult, we further subdivide the unit. Subdivision continues until we find either a standard self-timed circuit element for each subunit or it becomes possible to create an implementable signal transition graph (STG) specification for each new subunit discriminator.

This first stage, then, results in an interconnection of discriminators, that is, abstract modules with symbolic ports. The structural part of this description is a netlist. Parallel composition of the individual discriminators defines behavior using labeled Petri nets.

The second stage of our procedure, logic synthesis from a binary-encoded behavioral specification, consists of

1. using a signal expansion (implementing abstract events using signal transitions) to convert each discriminator's internal, abstract behavioral description into its binary equivalent, represented by an STG
2. verifying the signal expansion's correctness and completeness to provide the final behavioral model for subsequent logic synthesis
3. deriving Boolean functions characterizing the design process results

During this stage, the designer may use circuit synthesis software tools based on STGs and related models.[2,5] Existing tools are usually adequate, but limited in power. Certain classes of useful behavior defined by STGs (that is, most forms of fair mutual exclusion) require a substantial manual synthesis effort, demonstrating the need for more research. Another problem with most existing automated algorithms is that they are based on state graphs. Since state graphs can be exponentially larger than an STG, such algorithms become too complex to be practical for large examples. For more background, see the STG-based synthesis box (next page).

Abstract synthesis

Control circuit design begins with a definition of the circuit's external behavior in abstract terms. Then we decompose this behavior into an interconnection of simpler submodules that collectively implement the same function. We must define the behavior of such a structural composition much more formally than existing hardware description languages (such as VHDL or Verilog) allow. This is because we do not want to rely on simulation to assess the circuit's correctness.

Thus, we define the circuit as discriminator D, a black box with a set of "pins" labeled by symbolic names of ports or events (like read, write, strobe, acknowledge) that can occur on the border between the circuit and its environment. At this level, we neglect issues such as operation encoding using signal levels or transitions on wires. Using such pins, we can structurally interconnect D with other discriminators, and thus form discriminators of a higher abstraction level. An interconnection of such discriminators communicates by performing shared actions on a hypothetical underlying medium.

We use a k-place buffer, denoted as $BUF_k(a,b)$, to model finite capacity FIFO storage. Here, event a means that a data item enters the buffer through the input port. Event b means that a data item leaves the buffer through the output port.

The following characteristic predicate defines the buffer's behavior. For every trace t defined on events a and b (that is, for every arbitrary, finite sequence of symbols a and b), $0 \le (t\#a - t\#b) \le k$, in which $t\#a$ (or simply $\#a$ when t is clear from the context) denotes the number of occurrences of symbol a in trace t. Note that since $BUF_k(a,b)$ does not define the item retrieval order, it is more appropriate to use the general term buffer here. In fact, this first part of the specification just ensures that we don't require too many resources from an underlying data path.

We describe the data path as follows. Let $d(p_i)$ denote the ith data value in the ordered sequence passing through port p. The buffer is FIFO if $d(a_0) = d(b_0)$, and $d(a_i) = d(b_i)$ implies $d(a_{i+1}) = d(b_{i+1})$ for all $i > 0$.

The predicate form of specification is convenient for logical reasoning but is not always intuitive to the designer. Using graphical capture often helps to define the internal causal relationship between events on the discriminator boundary. This is equivalent to replacing an abstract specification that states every p is followed by a q with an implementation where every occurrence of p causes, through some physical connection, an occurrence of q. Hence we can view the set of traces for a given discriminator as generated by an underlying formal dynamic model, called a

STG-based synthesis

During the last few years, researchers developed a large number of formal techniques and software tools to synthesize asynchronous control and interface circuits.[2,3,5-8] These techniques all differ in the way they model, specify, verify, and synthesize a circuit. The question of modeling, both high- and low-level, in both the behavioral and the structural domain, appears to be the key issue.

The Petri net, a well-known model for the dynamic behavior of concurrent and asynchronous systems,[9] is a useful tool for specifying asynchronous control circuits. The explicit notion of conditions and events in Petri nets creates a good framework for defining a circuit's behavior paradigms: causality, parallelism, choice, and conflict. The events of the net can be annotated with a specific interpretation, such as signal transitions. This type of net is a signal transition graph (STG). Such graphs, introduced by Rosenblum and Yakovlev,[8] and Chu,[6] as well as other closely related models like change diagrams,[5] have recently become very popular as a formalism for automated synthesis of asynchronous circuits.[2,5] This is because of their descriptive simplicity and similarity to timing diagrams.

The key step in developing signal transition graphs is the interpretation of Petri net transitions as rising or falling edges of input and output signals in the specified circuit. A reachability or state graph has a node for each Petri net marking and an edge for each transition from one marking to another. The specification is consistent if we can label the corresponding state graph with a string of signal values matching the transitions. That is, a rising signal transition requires the signal to have value 0 in the predecessor and value 1 in the successor marking label.

A consistently labeled state graph somewhat resembles the standard flow table specification used for asynchronous circuit design. A state graph, however, carries more information than a flow table because it explicitly tells which signal changes are possible in every state. As such, it allows formal proof of a circuit implementation's existence and correctness.[2,5]

We can directly implement the state graph generated by an STG as an asynchronous circuit if and only if the signals specified by the STG completely describe the circuit's state.

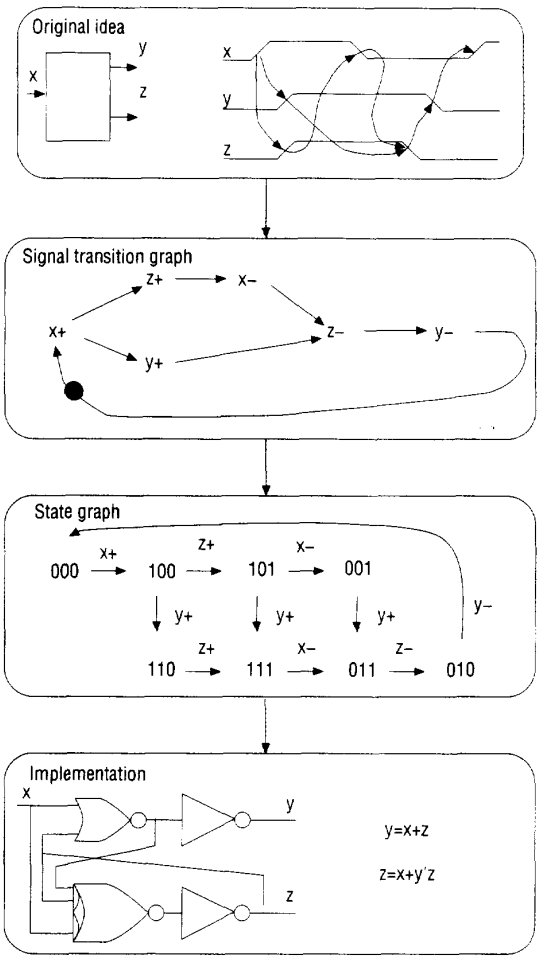


Figure A. *STG-based synthesis procedure.*

The STG is then said to have the complete state coding property. Otherwise, we must apply some adequate technique[10] to add signals that will complete the circuit description. Figure A summarizes the main steps in an STG-based synthesis procedure.

labeled Petri net (LPN),[11] rather than defined by a predicate. A LPN is a marked Petri net graph (we use the standard notation of ordinary Petri nets[9]) whose transition nodes are assigned a label from the alphabet of symbolic actions (or events).

We can create LPNs using parallel composition of elementary fragments that describe the basic paradigms of sequential behavior. Figure 1 shows a generic causality requirement represented as an LPN (Figure 1b) and a primitive discriminator (Figure 1a). The generic causality predicate assumes

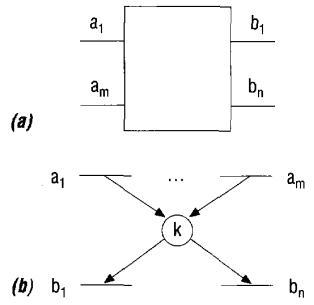


Figure 1. *Generic primitive discriminator* ***(a)*** *and corresponding general primitive LPN* ***(b)****.*

that if $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$ are actions of a system, then for every trace on $A \cup B$

$$\sum_{i=1}^{n} \# b_i - \sum_{i=1}^{m} \# a_i \leq k$$

Such paradigms are common to most high-level specification formalisms. These formalisms include guarded commands or CSP.[7] We claim that the LPN algebra[11] obtained by parallel composition of elementary one-place components (Figure 2) is a general way of defining behavior. The fragments describe sequencing aspects, while composition provides concurrency and synchronization.

To construct the LPN for our k-place buffer, we assemble its characteristic predicate (defined earlier) from two primitive causality requirements joined by a conjunction: $(\#a \geq \#b)$ and $(\#a \leq \#b+k)$. That is, if action a is "put a value in" and action b is "get a value out" then, for every execution trace, $\mathrm{BUF}_k(a,b) = \{(\#a \geq \#b) \text{ and } (\#a \leq \#b+k)\}$.

If we now apply parallel composition to the two one-place LPNs corresponding to these requirements, we have the LPN model for the k-place buffer, shown in Figure 3. The table in Figure 4 lists some typical discriminators along with their symbolic names and LPNs.

Abstract synthesis is essentially a process of decomposing more complex discriminators into simpler ones. The interconnection or netlist of elementary discriminators corresponds to the parallel composition of their LPNs.

The number of states an LPN model generates measures the complexity of a discriminator. We measure complexity this way because logic synthesis essentially draws upon the state graph representation. Therefore, although a decomposed discriminator's LPN can

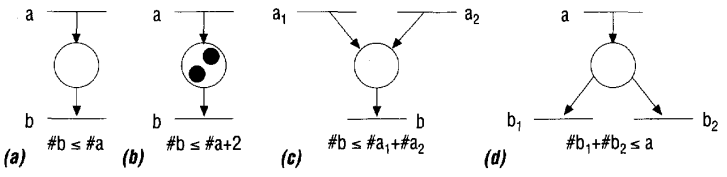


Figure 2. *Primitive LPN components representing specific causality cases: simple causality* ***(a)****, two-delayed causality* ***(b)****, simple OR-causality* ***(c)****, simple selection* ***(d)****.*

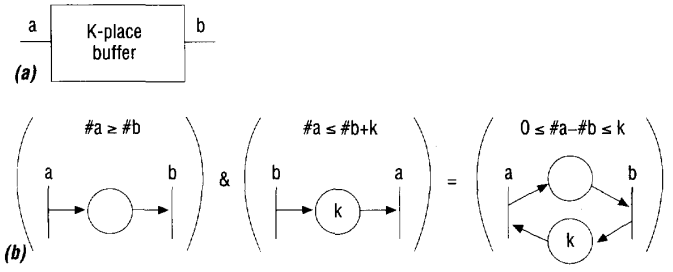


Figure 3. *Constructing an LPN for a k-place buffer: discriminator* ***(a)*** *and parallel composition of LPNs* ***(b)****.*

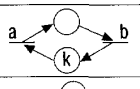
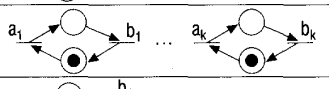
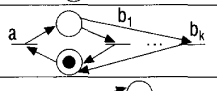
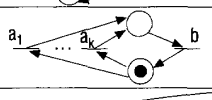
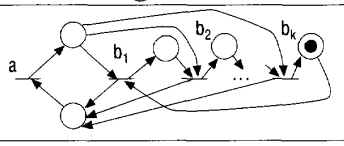
Discriminator	Symbolic name	Labeled Petri net
K-place buffer	BUF_k (a,b)	
Multichannel k-place buffer	MBUF_k (A,B)	
K-channel selector	SEL_k (a,B)	
K-channel multiplexer	MUX_k (A,b)	
Ordered k-channel selector	OSEL_k (a,B)	

Figure 4. *Typical discriminators and their LPNs.*

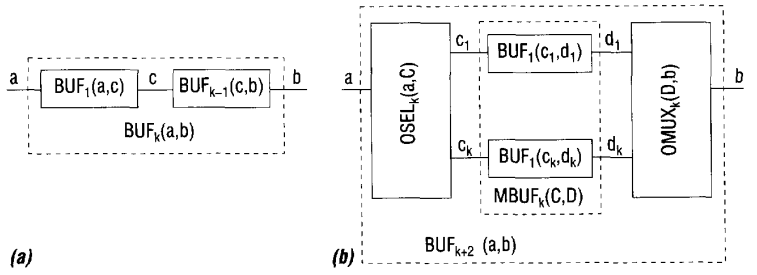


Figure 5. FIFO buffer decomposition: series or pipeline (a) and parallel (b).

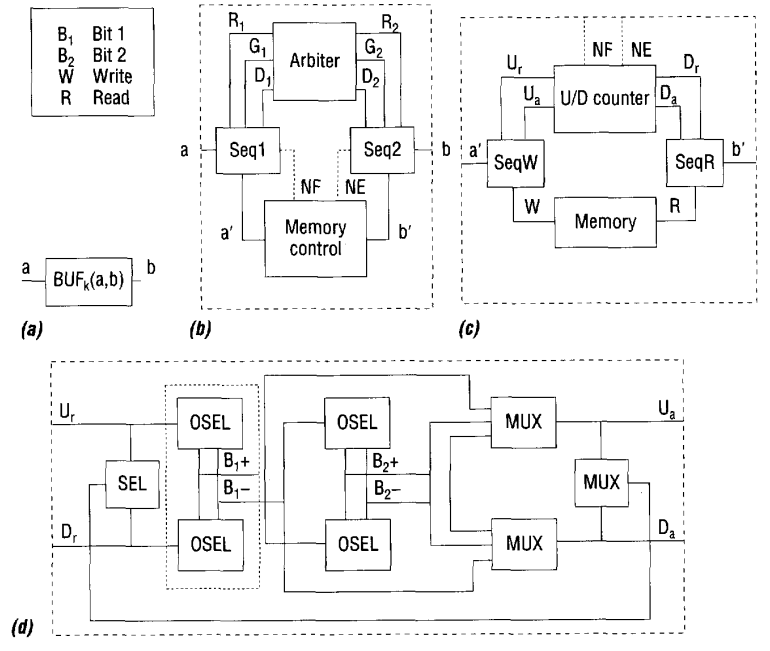


Figure 6. Structural buffer decomposition from top-level model (a) to successive refinements: $BUF_k(a,b)$ (b), memory control (c), and U/D counter (d).

be descriptively simple, its number of states is a rather crude estimate of the implementation's final area and delay cost, and also indicates the complexity of the logic synthesis process.

FIFO buffer decomposition. We can decompose the behavior of a FIFO buffer, as shown Figures 5a and 5b, in (at least) two possible ways. The first decomposition is a pipeline of lower capacity buffers connected in series. Each subbuffer must have its own storage, and therefore every data item travels across all subbuffers before leaving the module.

The second decomposition is a parallel interconnection of k buffers of capacity 1, which together correspond to a multichannel buffer of capacity k. Two additional submodules, ordered selector and multiplexer, order the input and output flow to and from these elementary buffers. Both organizations are FIFO.

Both these solutions have some good properties because of their regularity and the generic structure of the buffer control circuit. The second solution, however, is faster. Although both solutions have the same throughput, the second solution has shorter propagation delays. This is because the propagation delay for one data item is proportional to buffer length, and the second solution has shorter buffers.

Although superior in speed, the second solution requires more silicon area, a crucial factor in large-capacity buffers. The area of a first-cut implementation of the control circuitry for both solutions is linear with k, but the second solution has a larger multiplicative factor.

We can improve the size of the control module, making its largest component logarithmic in k, by adding a counter to the buffer control circuit. The data path in such a buffer will also be different from that of the previous two solutions. It will be based on ordinary memory with built-in mod-k counters to generate write and read addresses. This solution is similar to the buffers described in Sutherland[12] and Dill et al.[13] However, in contrast to Dill's approach, which deals with subsequent design verification, we formally synthesize the circuit at the discriminator level.

Ignoring data path synthesis for now, our main problem is finding an adequate LPN description for a new discriminator, the mod-k counter, that is now part of the control circuit. Furthermore, this counter must be reversible, that is, able to count up and down. Using a counter to implement the control flow of the k-place buffer is an example of an often used technique: adding more data-path layers inside the control circuit, to prevent it from growing faster in size than the data circuit. (We ignore, for now, the alternative of using information from the data path to implement control; that is, by compar-

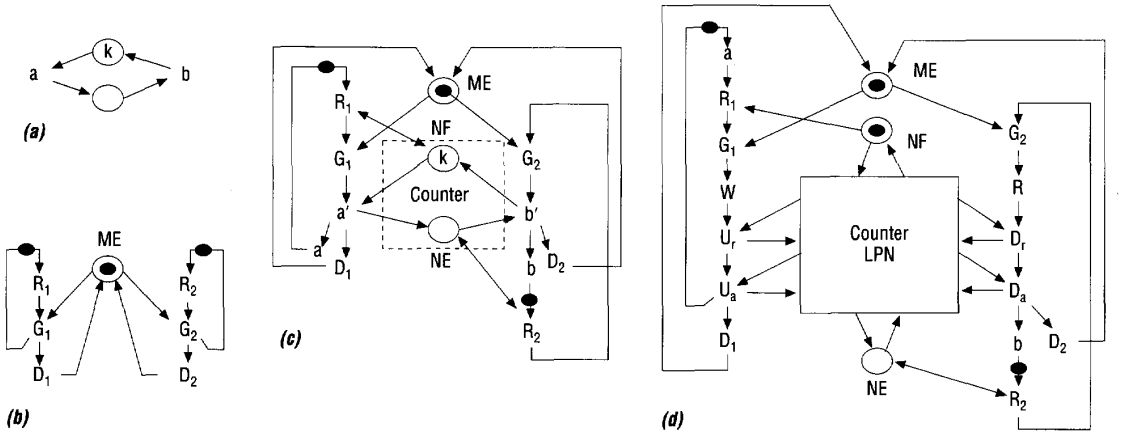


Figure 7. *Labeled Petri nets for counter-based buffer design: original $BUF_k(a,b)$ model* ***(a)****; request-grant-done arbiter control* ***(b)****; arbiter and counter* ***(c)****; and arbiter, counter, and memory control* ***(d)****.*

ing read and write counters to detect full or empty conditions.)

The memory and up/down (U/D) counter, together with the write, read, count-up, and count-down operations, form a critical region that must be protected from concurrent access. Primary buffer actions Put Data and Get Data, denoted by a and b, provide this protection. In the solutions represented by Figures 5a and 5b, the order of write and read operations to a buffer was enforced locally at the level of each primitive 1-place buffer. This third solution cannot rely on such an implicit ordering and hence, our circuit must include a mutual-exclusion element.

It is important to realize that adding mutual exclusion between a pair of atomic actions does not change their trace semantics. It is these semantics that allow the previous FIFO buffers[12,13] to formally satisfy the specification of a k-place buffer, in which both ports are considered independent and can thus be activated simultaneously.

Refining buffer control structure. The structure shown in Figure 6b refines the top-level model in Figure 6a. To allow a and b to occur concurrently, we introduce mutual exclusion (the ME element) inside the buffer component by including the arbiter whose LPN is shown in Figure 7b. It operates in parallel with two sequencers, Seq1 and Seq2. This composition provides control for the memory control unit by ensuring that the actions on ports a' and b' occuring in critical sections (like memory and counter access) are mutually exclusive.

Interaction with the arbiter requires the following three actions from each of two sides (denoted by i=1,2): request (R_i) to enter the critical section; grant (G_i), permitting entry; and done (D_i), acknowledging exit. Figure 7c shows this system's LPN model, where the fragment inside the dashed counter box models memory control. This fragment also provides conditions Not Full (NF) and Not Empty (NE) to the control flow in the sequencers. Marking in the places labeled NF and NE can easily provide these conditions. It is easy to prove that the LPN in Figure 7c correctly models the k-place buffer: We can compress the sequence of R_1, G_1, a', and D_1 actions into action a giving us the LPN that is trace equivalent to the original k-place buffer model in Figure 7a.

Having protected the critical sections of a' and b', we can now refine them to separate the actions on the data path, memory write (W) and read (R) operations, from those on the remaining control path, counter increment and decrement. For counter operations it is convenient to separately consider request and acknowledgment actions of the up and down operations. Thus we have the (U_r, U_a) and (D_r, D_a) action pairs.

Figure 6c shows the structural refinement of the memory control, including separate sequencers SeqW and SeqR for buffer write and read operations. The U/D counter module, synchronized with the sequencers, is also responsible for producing NF and NE flags. Assuming that it works in synchronization with the control path mechanisms, we abstract the memory unit's internal structure and behavior by having separate W and R ports. Figure 7d shows the corresponding LPN buffer; only the counter LPN is not explicitly defined in that figure.

Figure 6d shows the structural implementation of the U/D counter, for the case of a modulo-4 counter. Figure 8 (next page) shows the corresponding

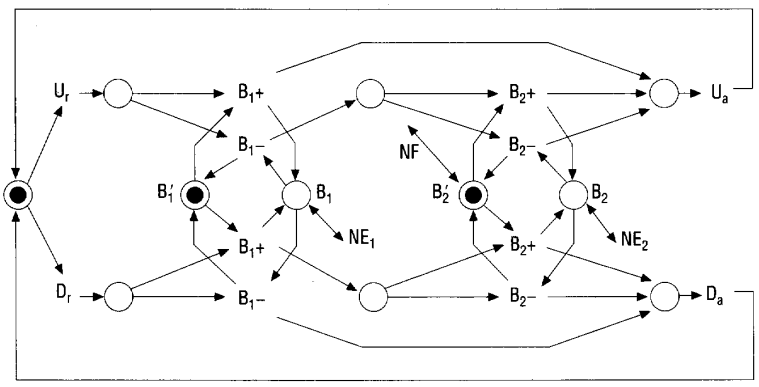


Figure 8. Labeled Petri net counter model.

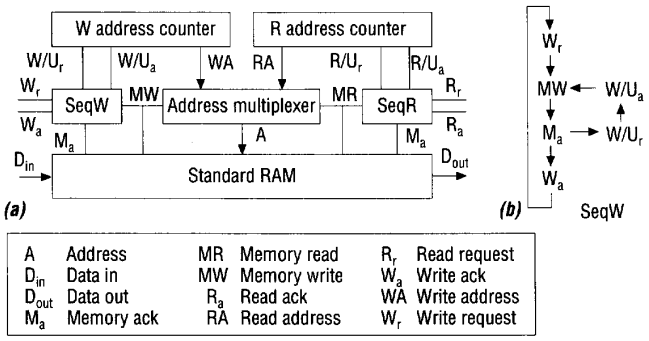


Figure 9. FIFO memory unit structural representation (a) and LPN (b).

LPN, which can be used to trace the interconnection's behavior.

The design of the buffer control circuit is generic in the sense that we have only implemented the $BUF_k(a,b)$ part of the specification. The implementation does not restrict access to data stored in the memory unit, apart from mutual exclusion between read and write actions. This approach distinguishes this decomposition in a major way from those in Figure 5. We now refine the FIFO memory unit (Figure 9a) using the access discipline requirement. Figure 9b shows the refined data path. We obtain a last-in/first-out structural organization (Figure 10a) by simply changing only the data path (Figure 10b).

Logic synthesis

Completion of the first stage of our procedure results in an interconnection of LPNs that represent discriminators. In the second stage, we must first convert those LPNs to STGs. Converting LPNs used for specification to STGs is essentially an expansion of events from symbolic labels to value changes on physical wires. This expansion has both structural and behavioral aspects. Structurally, we need to obtain a black box with a set of input and output signals. The expansion associates each port and event in the original specification with a subset of signals and signal transitions.

Several types of signal expansion exist. Handshake expansion[7] of port actions associates port *a* with a pair of signals, request signal a_{req} and acknowledgment signal a_{ack}.

Simple signal casting associates port *a* with single signal z_a that carries an event between subcircuits. This expansion does not provide direct notification of event reception; that is, there are no separate acknowledgment signals. Therefore, it is especially useful within a circuit because a circuit's global LPN behavior indicates acknowledgment.

Signaling types may also differ in the number of phases that are significant in the two-phase (up/down) cycle of each binary signal.[12] Signaling on port *a* is 4-phase or return-to-zero (RZ) protocol if only one of the transitions of z, either z_a+ or z_a-, is assumed to be significant to the action on *a*. In two-phase signaling or non-return-to-zero (NRZ) protocol, both transitions are significant and semantically equivalent in representing the occurrence of the same action on *a*.

The behavioral relationship between original events and signal transitions must be formally and semantically justified. Thus, for the critical signal transitions, the original specification semantics must conform to those of the refined one.

We will use a handshake expansion to explain critical signal transitions. For example, if abstract action *a* denotes writing to a buffer, the corresponding signal transition can be an assertion of request a_{req}, denoted as $a_{req}+$, in the handshake pair (a_{req}, a_{ack}) associated with port *a*. These critical signal transitions relate directly to original events. There are also auxiliary transitions, so called because the designer has free choice as to where to put them into the specification. Only some local ordering relations between critical signal transitions and their auxiliary associates constrain addition of such transitions.

An example of an auxiliary signal transition is the release of $a_{req}-$. It should be related to $a_{req}+$ so that they can nev-

er be activated simultaneously, and thus should form a sequentially ordered pair. Sometimes, the entire acknowledgment part ($a_{ack}+$, $a_{ack}-$) of a handshake pair associated with a single abstract event can be regarded as auxiliary. The local ordering requirement would be adherence to the sequential protocol of actions: $a_{req}+ \rightarrow a_{ack}+ \rightarrow a_{req}- \rightarrow a_{ack}-$. Note that auxiliary signals can be relocated in the STG for speed or area optimization during logic synthesis.

FIFO buffer expansion

We expanded the FIFO buffer of our example, represented by the LPN shown in Figure 7d. We used a handshake expansion for memory ports *W* and *R*, as well as for *a* and *b*. We expanded ports *a* and *b* into request and acknowledgment signals between the buffer and its environment: R_{in} and A_{in} for *a*; R_{out} and A_{out} for *b*. We used signal casting to expand remaining events R_1, R_2, G_1, G_2, D_1, D_2, U_r, U_a, D_r, D_a. The overall signal expansion uses a two-phase signaling discipline, in which all signals first change from 0 to 1 and then from 1 to 0. Both phases are significant to control flow. The main advantage of two-phase signaling is speed.[12] We also benefit from a similar signaling scheme in the internal interfacing of the request-grant-done (RGD) arbiter, memory unit, and U/D counter.

In the buffer controller structural model, shown in Figures 6c and 6d, we explicitly use sequencers to control activation of the other components. We can simplify the design by interconnecting the modules directly, without explicit sequencers, so that they operate according to the LPN specification, shown in Figure 7d.

The combination of two-phase and four-phase signaling schemes takes place during the effect of the conditions Not Full (NF) and Not Empty(NE) on the control flow. This is the main difference between our design and other FIFO designs.[12,13] Combining signaling schemes makes our design more simple than those mentioned.

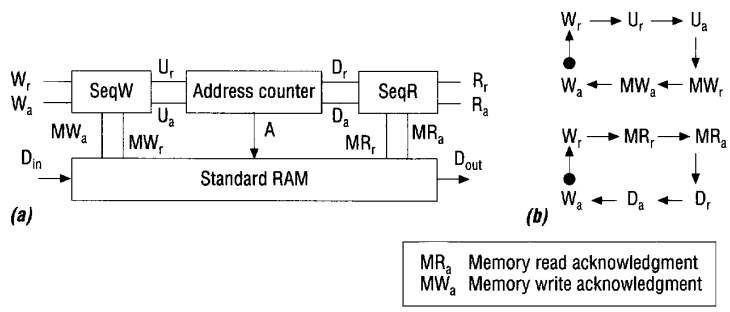


Figure 10. LIFO (stack) memory structural representation (a) and LPN (b).

Figure 11 shows our implementation's block diagram. Most parts of this diagram are intuitively clear, as they correspond to the LPN in Figure 7d. Note that the implementation uses transparent latches that combine the 2-phase and the 4-phase signaling schemes safely (that is, without metastability problems). We assume that the two wires indicated by dotted lines in Figure 11 have less delay than the time between the departure of handshake output signals A_{in} or R_{out} and the arrival of input signals R_{in} or A_{out}.

The circuit is correct and independent of speed under the indicated delay assumptions. It is almost inevitable that the combination of the two signaling strategies affects the pure delay insensitivity. Our design also guarantees freedom from hazards by speed independence.

We synthesize all the components in Figure 11 using an STG-based method.[1]

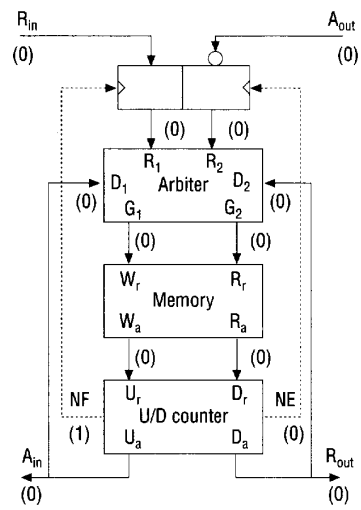


Figure 11. Counter-based buffer control circuit.

OUR METHODOLOGY PROVIDES a unified solution to the problem of synthesizing logic implementations from abstract specifications. Although easily applied to control circuits, it is suitable for all types of interface hardware. These features set our methodology apart from those presented elsewhere.

Although our prime example, a FIFO buffer controller, is rather simple, it is possible to apply the same methodology to other types of discriminators, such as a frequency differentiator, a low latency arbiter, and so on.

We are currently working on a set of graphical and algorithmic tools to support two-stage synthesis procedures based on the combination of labeled Petri nets and signal transition graphs. D&T

Acknowledgments

Engineering and Physical Science Research Council (EPSRC) grant GR/J52327 partially supported our work. The Ministero dell' Universita' e della Ricerca Scientifica e Tecnologica (MURST) partially supported L. Lavagno's work.

References

1. A.V. Yakovlev, A.M. Koelmans, and L. Lavagno, "High-Level Modeling and Design of Asynchronous Interface Logic," Tech. Report 460, University of Newcastle upon Tyne, Computing Science Dept., Newcastle upon Tyne, Nov. 1993.
2. L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits*, Kluwer Academic Publishers, Boston, 1993.
3. C.W. Moon, P.R. Stephan, and R.K. Brayton, "Synthesis of Hazard-Free Asynchronous Circuits from Graphical Specifications," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1991, pp. 322-325.
4. C.L. Seitz, "System Timing," *Introduction to VLSI Systems*, C.A. Mead and L.A. Conway, eds., Addison-Wesley, Reading, Mass., 1980, pp. 218-262.
5. M. Kishinevsky et al., *Concurrent Hardware: The Theory and Practice of Self-Timed Design*, John Wiley and Sons, London, 1993.
6. T.-A. Chu, "On the Models for Designing VLSI Asynchronous Digital Systems," *Integration: The VLSI Journal*, Vol. 4, No. 2, June 1986, pp. 99-113.
7. A.J. Martin, "Synthesis of Asynchronous VLSI Circuits," *Formal Methods for VLSI Design*, J. Staunstrup, ed., North Holland, Amsterdam, 1990, pp. 237-283.
8. L.Y. Rosenblum and A.V. Yakovlev, "Signal Graphs: From Self-Timed to Timed Ones," *Proc. Int'l Workshop Timed Petri Nets*, IEEE CS Press, 1985, pp. 199-207.
9. T. Murata, "Petri Nets: Properties, Analysis, and Applications," *Proc. IEEE*, Vol. 77, No. 4, Apr. 1989, pp. 541-580.
10. P. Vanbekbergen et al., "Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications," *Proc. ICCAD*, IEEE CS Press, 1990, pp. 184-187.
11. I. Reicher and M. Yoeli, "Net-Based Modeling of Communicating Parallel Processes with Applications to VLSI Design," Tech. Report 532, Technion, Haifa, 1988.
12. I.E. Sutherland, "Micropipelines," *Comm. ACM*, Vol. 32, No. 6, 1989, pp. 720-738.
13. D.L. Dill, S.M. Nowick, and R.F. Sproull, "Automatic Verification of Speed-Independent Circuits with Petri Net Specifications," *Proc. Int'l Conf. Computer Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp. 212-216.

Alexandre V. Yakovlev is a lecturer in the Department of Computing Science, University of Newcastle upon Tyne, United Kingdom. His research interests are in the field of modeling and synthesis of asynchronous, concurrent, and fault-tolerant systems. Yakovlev received MSc and PhD degrees in computing science from the St. Petersburg Electrical Engineering Institute, Russia.

Albert M. Koelmans lectures for the University of Newcastle's Department of Computing Science. His research interests include CAD for VLSI, functional programming applications, and design of asynchronous and fault-tolerant VLSI systems. He has written articles on hardware description languages, the VISD encryption chip, graphical representation of a CHDL, and transformational reasoning. Koelmans graduated from Groningen University, The Netherlands, with a Doktoraal degree in mathematics and computer science.

Luciano Lavagno is an assistant professor at the Politecnico di Torino, Italy, where his current research interests include synthesis of asynchronous circuits, concurrent design of mixed hardware and software systems, and formal verification of digital systems. Lavagno received the DEng from the Politecnico di Torino and his PhD in electrical engineering from the University of California, Berkeley. In 1991, he received the Best Paper award at the 28th Design Automation Conference. He is a member of the IEEE and the IEEE Computer Society.

Address questions or comments about this article to Alexandre Yakovlev, Dept. of Computing Science, University of Newcastle upon Tyne, Claremont Tower, Claremont Rd., Newcastle upon Tyne NE1 7RU, UK; alex.yakovlev@newcastle.ac.uk.