A Model for the High-Level Description and Simulation of VLSI Networks

We present a new applicative model for the description and analysis of synchronous and asynchronous VLSI networks at the top levels of abstraction. The model uses two powerful paradigms that provide an elegant, fast method for the high-level description and simulation of VLSI networks.

A.J. van der Hoeven
A.A.J. de Lange
E.F. Deprettere
P.M. Dewilde

Delft University of Technology

Conventional VLSI (very large scale integration) modeling techniques[1,2] derive from a data-driven simulation concept. Any value change at an input of a design entity (node) results in the generation of new values at the outputs of the node. These values are irrelevant in the high-level design stages. The generation of these values leads to a collection (heap) of output data in which the significant values are often hard to find. In contrast, the applicative state transition (AST) model we propose explicitly represents the flow of information through a network. We do not evaluate a leaf node in a network until its (selected) input ports have received new (significant) data and the data at its (selected) output ports has been processed. Other aspects that contribute to the power of the model are:

- It allows the description of designs from the level of abstract functional or algorithmic behavior down to the register-transfer level. The model can describe both synchronous and asynchronous designs.
- In design descriptions, the model logically separates state, function, and function control. This separation increases the clearness of the descriptions and simplifies the development and application of silicon compilers.
- An AST node is a good high-level abstraction of general hardware. It can execute different instructions (functions) with different inputs and can write different outputs. It contains a controller (next-function selection) that selects the next mode of operation. Unless otherwise indicated in our discussion, the term AST denotes an individual node.

Backus[3] first introduced the AST concept to "liberate" programming languages from the classical von Neumann style. According to Backus, an AST is a "self-modifying function," that is, the input-output mapping

An earlier version of this article appeared in the *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.

0272-1732/90/0800-0041$01.00 © 1990 IEEE

Table 1.
Two versions of a delay AST.

Delay	Formula
For integer numbers	
The types of the AST	Z is the set of integer numbers
The set of named types	U = {in: Z, out: Z, state: Z}
The ports of the AST	D = in, R = out, S = state
The set of functions F	$\{f\}$
The I/O part of f	f': state $\rightarrow$ out, $o = f'(s) = s$
The control part of f	f'': in $\rightarrow F \times$ state, $(fs,s) = f''(i) = fs = f$, $s = i$
For Boolean numbers	
The types of the AST	B = {0, 1}
The set of named types	U = {in: B, out: B}
The ports of the AST	D = in, R = out, $S = \emptyset$
The set of functions F	$\{f_0, f_1\}$
The I/O part of f_0	f_0': $\emptyset \rightarrow$ out, $o = f_0'() = 0$
The control part of f_0	f_0'': in $\rightarrow F$, $fs = f_0''(i)$ = if $i = 0$, then $fs = f_0$ if $i = 1$, then $fs = f_1$
The I/O part of f_1	f_1': $\emptyset \rightarrow$ out, $o = f_1'() = 1$
The control part of f_1	f_1'': in $\rightarrow F$, $fs = f_1''(i)$ = if $i = 0$, then $fs = f_0$ if $i = 1$, then $fs = f_1$

changes in time. Consequently, an AST can operate as a set of functions with one function active at any time. Each time an AST is evaluated, the currently selected function f maps its input values onto its output values and selects the next function f: $I_f \rightarrow O_f \times F$. F is the set of functions that make up the AST.

We added a number of refinements to make the AST concept suitable for VLSI modeling. We based the communication between different ASTs on the petri net, single-token-pass mechanism as defined in the theory of Condition/Event (C/E) systems.[4-7] We extended this communication mechanism to allow for the distribution or broadcast of tokens. We discuss this subject later and also explain how the AST model enables the use of synchronous communication between AST nodes.

Concept definition

We define an AST node as a set F of functions f_i, in which each function has a subset of F as part of its range. By and large, this definition follows the AST structure introduced by Backus.

Each function of an AST can be decomposed into two functions that execute in parallel. Furthermore, we expand the definition of an AST by introducing a data state apart from the already introduced function state.

Let U be a set of named types. Each named type u: $T \in U$ has a name u that is unique in U and a type T. U^x denotes the Cartesian product that is made up from the elements of U. We sometimes refer to a named type by its name only.

We now define an AST as follows:

- Let D be a subset of U, $D = \{d_1, \ldots, d_m\}$. D is the set of input ports of an AST.
- Let R be a subset of U, $R = \{r_1, \ldots, r_n\}$, such that $D \cap R = \emptyset$. R is the set of output ports of an AST.
- An AST F is a set of functions $\{f_i\}$, such that f_i: $D_i^x \rightarrow R_i^x \times F_i$. D_i is a subset of D, R_i is a subset of R, and F_i is a subset of F.

It follows from this definition that a function f_i can be decomposed into two functions $[f_i', f_i'']$, such that

- Range (f_i') $\subseteq R_i^x$, Range (f_i'') = F_i, and
- Domain (f_i') $\cup$ Domain (f_i'') = Domain (f_i).

The hardware realization maps the functions f_i'' of an AST onto the controller path. The functions f_i' form the high-level model of the data path. The decomposition into two functions also allows for the differentiation between Mealy and Moore functions in an AST. We define the functions as:

- Mealy: Domain (f_i') = Domain (f_i'')
- Moore: Domain (f_i') = $\emptyset$ (constant function).

If the domain of a function f_i' is empty, the function f_i is a Moore function as opposed to a Mealy function. In case of a Moore function, we allow the execution of f_i' prior to—and independent from—f_i''. A Mealy function allows only the combined execution.

The differentiation between Mealy and Moore functions in an AST allows the creation of proper loops in a network without creating deadlocks.

For the introduction of a data state, we redefine the functions of F as follows. Let S be a subset of U, $S = \{s_1, \ldots, s_p\}$. Then the functions f_i convert into

$$f_i: D_i^x \times S^x \rightarrow R_i^x \times S^x \times F_i.$$

Apart from the fact that S significantly reduces the number of functions that need to be specified, S can represent the data state of a hardware module (for

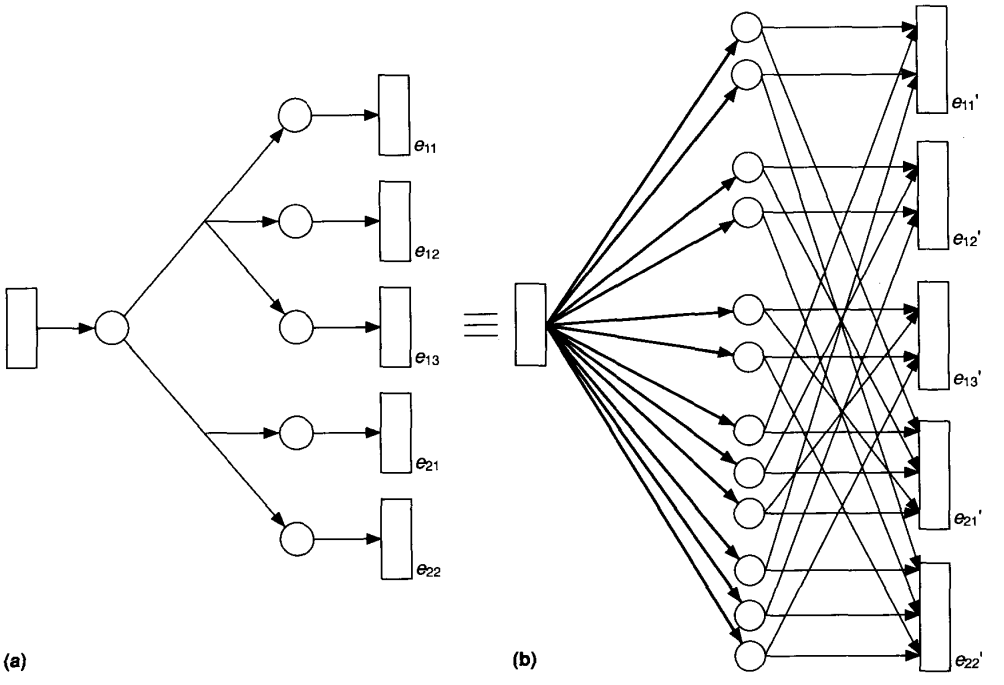

Figure 1. Typical example of token distribution (a) and its C/E-system equivalent (b).

example, the memory of a counter). S is usually implemented as a set of registers.

An example. Table 1 specifies two versions of a delay AST. The first one is a delay for integer numbers that consists of only one function and a data-state $S = Z$. The second version is a delay for Boolean numbers that consists of two functions and no data state. Note that the functions are Moore functions.

Communication hierarchy

We can embed the AST concept into the theory of petri nets. By doing so, we can define the operational aspects of a graph that consists of interconnected AST nodes. This process allows us to construct a graph and to define an AST as a graph. By using petri net theory to define the communication protocol between nodes, we emphasize the information flow through the network instead of the raw dataflow of conventional, data-driven simulation models.

We chose the C/E system as the basic type of petri net[4,7] because

- The communication protocol derives from the single-token-pass mechanism, which closely resembles the dataflow through a VLSI network.
- We could define the AST concept in terms of conditions and events of a C/E system.
- Using the theory of timed petri nets and C/E-systems, we could easily define an appropriate timing model.

In short, a C/E system consists of a set of conditions (places) C and a set of events (transitions) E. Furthermore, each element of E has a subset of C as its input conditions and a subset of C as its output conditions. An element of C can hold at most one token (item of information). An event $e \in E$ can occur if all its input conditions hold a token and all its output conditions do not. If e occurs, the tokens at its input conditions are removed, and tokens at its output conditions are created—all in one atomic action.

Token distribution. To simplify the design and representation of AST graphs, we allow tokens to be distributed or multiplied. Allowing conditions (or ports) to have (sets of) subconditions (or subports) provides for the distribution of tokens. We informally describe the concept of token distribution as follows.

- When a condition receives a token, so do its subconditions.
- As long as one of its subconditions contains a token, the condition does too.
- When all its subconditions become devoid of a token, so does the condition.

Figure 1 depicts a typical case of token distribution

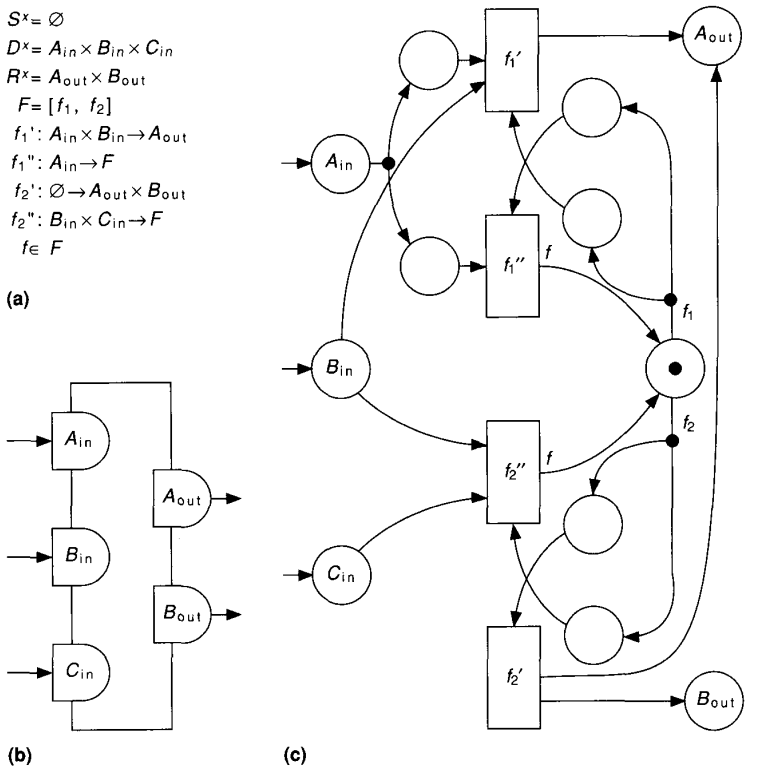

Figure 2. Functional outline of an AST (a), its C/E-system-equivalent black box representation of a node with ports (b), and its graphical representation (c).

and its equivalent C/E system. Rectangles represent the events and circles represent the conditions. In this diagram, either the events e_{11}, e_{12}, and e_{13} occur, or the events e_{21} and e_{22} occur. Figure 1b expresses this mutual exclusivity.

C/E-system representation. Here we express the AST concept as a C/E system. We do this informally by following an example. In Figure 2, the output of the events f_i'' is a token with a value f from the set F. If the value is equal to f_1, the destination of the token is the f_1 events. These events constitute a Mealy function. If the value is equal to f_2, the destination of the token is the f_2 events, or a Moore function. One can easily deduce that this graph is free of conflict since either the f_1 events (functions) or the f_2 events can occur. In a Mealy function f_i, the inputs of f_i'' are a subset of the inputs of f_i'. In a Moore function f_i, the set of inputs of f_i' is empty. (Refer to the previous formulas.)

The execution rules of the C/E system prescribe the operation of the AST. The execution of the graph in Figure 2c starts with a token in the common output-condition of the f_i'' events.

Hierarchy in the AST graph. The basic elements in AST graphs are the ASTs, which can be defined in terms of a C/E system as shown. However, we would like to define hierarchical constructions on top of the ASTs. This hierarchy results from simply connecting output ports of some ASTs to input ports of other ASTs and adding input or output ports to the incoming or outgoing edges of the graph. These ports then form the interface of the graph (Figure 3, described later). Now, AST graphs can function as nodes in other AST graphs. Note that the operation of such a hierarchical graph is the same as that of its flattened equivalent. By connecting nodes in this manner, one can define asynchronous systems.

An AST node defined by a graph of nodes. One can also define an AST node by specifying an AST graph with the constraint that the graph must behave as a leaf AST node. We use this method for the construction of synchronous networks. We construct the node from the graph in the following fashion.

- The current functions of the (hierarchical) nodes of the graph compose the current function of the AST.
- The function subsequently executes as if it were a function of an AST.
- The functions that have now been selected again compose the next function.

Note that the use of Moore functions in such a composite AST leads to pipelining effects in the network. The use of a composite AST results in a synchronous system with implicit clocking.

Conditional functions. A synchronous operation can have a conditional (data-dependent) dataflow, which can be modeled using operations or conditional functions. An operation or conditional function in an AST is a function composition $f = fz \circ \ldots \circ fj \circ fi$ that executes as if it were a single function. f is executed if $fj, \ldots, fz$ are selected by $fi, fj, \ldots, fy$ as their respective next functions. $fi, \ldots, fy$ are control functions, and fz is a dataflow function.

A simple example is a two-input multiplexer with control, left, and right ports for input and one output port. The multiplexer functions are

fc: control $\rightarrow \emptyset \times \{fl, fr\}$ (control function).
fl: left $\rightarrow$ out $\times \{fc\}$ (dataflow function).
fr: right $\rightarrow$ out $\times \{fc\}$ (dataflow function).

The resulting conditional functions of the AST are now $fl \circ fc$ and $fr \circ fc$. The value at the control port determines which function executes. We always consider a conditional function to be a Mealy function.

Timing in the graph

The literature defines different mechanisms to impose timing constraints onto petri nets. In principle, two ways of adding time to petri nets exist.

The first way, proposed by Ramchandani,[8] assigns a duration to each event in a petri net. An event effectively splits into a consumption event of tokens and a production event of tokens between whose respective occurrences lies the specified duration. However, this method of defining time changes the behavior of the untimed petri net because it destroys the concept of the atomic occurrence of an event.

The second way, proposed by Merlin[5] for general petri nets, is based on the duration of a token in a condition. Merlin proposes a minimal wait for an event. An event may occur only after that time has elapsed, counting from the moment at which all its input conditions initially contain a token. From the Merlin timing operation, it follows that the operation of a C/E system does not change as long as the system is conflict free.[4] Therefore, we can add timing delays to an AST graph with the Merlin mechanism.

With the timing mechanism proposed by Walter,[6] we can impose data-integrity constraints onto an AST graph. In accordance with his proposal, we add a minimum duration to each condition. A token at a condition becomes "visible" to its output events after the minimum duration has expired. We can now define the complete timing operation of a C/E system. Since for any AST graph there exists an equivalent C/E system, we thus define the timing model for the graph.

- A timed C/E system $TCE = (CE, T_C, T_E)$ is a C/E system $CE = (C, E; F)$, in which T_C is a function from the set of conditions C to Q, and T_E is a function from the set E of events to Q. In this system, Q denotes the set of integral numbers $F \subset C \times E \cup E \times C$. F defines the interconnections between the events and the conditions.
- An event (function) $e \in E$ occurs (executes) if and only if tokens at its input conditions (ports) are visible and its output conditions (ports) do not hold a token. If $e \in$ Domain (T_E), a time $T_E(e)$ must have elapsed since the moment all its output conditions became visible. A token at a condition $c \in C$ becomes visible if and only if the condition holds a token and a time $T_C(c)$ has elapsed since the moment the condition started to hold a token.

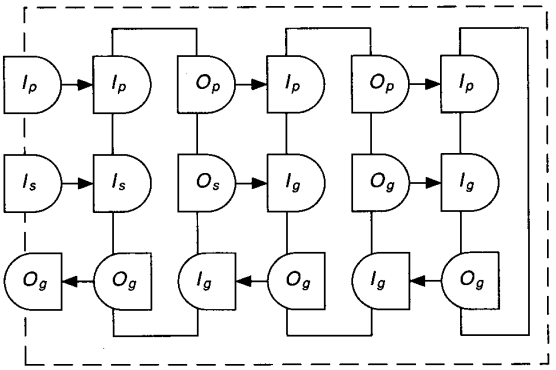

Figure 3. An asynchronous sorter network. Dashed lines and external ports denote black box representation.

One can model the computation time needed by a function with the wait time of an event. One can describe setup times and hold times with the minimum duration time of a token at a condition. If an AST is defined as an AST graph, one can compute the overall timing specification from the timing specifications of the constituting AST nodes.

An application

Here we explore the design of a simple priority queue as both a real-time systolic (synchronous) system and a real-time wavefront (asynchronous) system. This example stresses the capability of the AST graph model to handle loops in the design and to describe both synchronous and asynchronous designs. We based the priority queue on a sorter AST (Figure 3) that contains three functions.

On the basis of its input from the left (I_s), the select function either selects the put or the get function and transfers its input value to the right (O_s). The put function takes a value from the left (I_p), compares it with the current state value, outputs the larger value to the right (O_p), and stores the smaller value in its state. The function outputs its state on O_g and reads data from I_g to establish a correct dataflow. The select function becomes its successor. The get function outputs the input from I_g to O_g, stores it in its state, and chooses the select function again. Note that all functions defined are Mealy functions. The initial function of the sorter

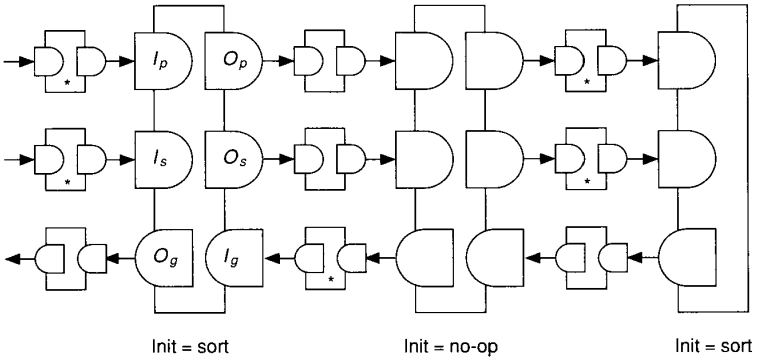

Figure 4. Design of a synchronous sorter network. Rectangles and ports denote black box representations of AST nodes.

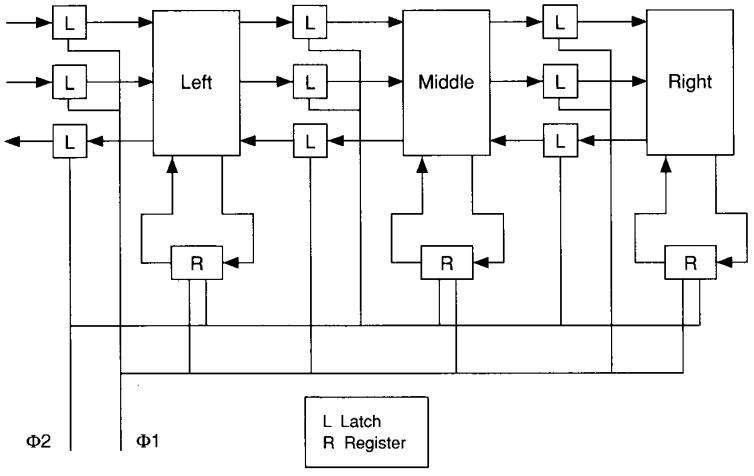

Figure 5. Realization of a synchronous sorter network. Directed edges denote data flow; other edges denote clock edges.

AST outputs the value $+\infty$ on O_g with select as its next function. Its initial state value is $+\infty$. Combining a number of sorter ASTs into a one-dimensional array, as depicted in Figure 3, simply creates a wavefront priority queue.

For the design of the systolic version, we have to take some additional steps. First, we combine the functions of the original sorter AST into two conditional functions: put ∘ select and get ∘ select. Note that these are Mealy functions. Therefore we can't just cascade the sorter ASTs because loops would be created in the network. To break up the loops, we need to separate the sorter ASTs with buffer ASTs (Figure 4).

A buffer AST consists of the following two functions.

$$f_1 : I \rightarrow S \times F, f_1(i) = (i, f_2),$$
$$f_2 : S \rightarrow O \times F, f_2(s) = (s, f_1).$$

If we properly initialize the buffer (if marked with an asterisk, init = f_2, else f_1) and simulate the graph in an asynchronous mode, we observe that of two adjacent sorter ASTs, only one is operating at any moment. Therefore, in the synchronous mode, we have to add a dummy function to the sorter AST for the no-op function. The sorter AST now switches between one of its sorter functions and its no-op function. Finally, we choose the right initial function for each AST node in the network (see Figure 4).

Given the types of data, we can now realize the sorter AST in hardware. A register that operates on a nonoverlapping, two-phase (Φ1 and Φ2) clock realizes the state of the sorter node. The buffers in the graph are realized by latches that either operate on the clock signal Φ1 or Φ2 (depending upon whether they were marked with an asterisk or not). Of any two adjacent sorter ASTs, one operates on $\Phi 1\uparrow$ and the other operates on $\Phi 2\uparrow$. (See Figure 5.)

Implementation notes

We embedded the model described into an interactive design system called HIFI (Hierarchical, I(n)tera(c)tive Flowgraph Integration). The core of the system consists of a hierarchical, interactive simulator[9] and a graphical user interface for both design and simulation of HIFI networks.[10]

In the HIFI system, one can specify both behavior and structure in a generic fashion. The concepts of the method of generic design come from the abstraction and application axioms of Lambda calculus.[11]

We are implementing a number of high-level partitioning and synthesis algorithms[12,13] and have designed the tools around a powerful, versatile, semantic database environment.[14] Construction of the HIFI system takes place in the object-oriented Objective C programming language.[15]

We have described a method for the modeling and simulation of VLSI networks at a high level of abstraction based on mathematical principles. The power of the model primarily stems from the notion that it describes the flow of information (instead of data) through a network. The model allows the description of self-timed systems, synchronous systems, and combinational ripple logic from the highest functional level onto the register-transfer level. The AST model separates data states, functional behavior, and control in a natural way that can be exploited by automatic layout-synthesis tools (silicon compilers). The model favors a hierarchical design style of stepwise structural refinement of behavioral descriptions, which is a powerful means to reduce the design complexity. To model the timing of VLSI (sub)systems in a structure (network), we use a simple model that adds delays for the assignment of functions and ports that are part of an AST. This approach allows for fast verification and analysis of the correctness of dataflow in a VLSI system as well as the determination and optimization of system throughput.

Acknowledgments

The Dutch National Science Foundation supported this work under Grant STW DEL 47.0643. The European Community supported this work via the ESPRIT NANA project.

References

1. Special Issue on VHDL: The VHSIC Hardware Description Language, *IEEE Design & Test of Computers*, Vol.3, No. 2, Apr. 1986, pp. 10-73.
2. R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. Computers*, Feb. 1984, pp. 160-177.
3. J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM*, Vol. 21, Aug. 1978, pp. 613-641.
4. W. Reisig, *Petri Nets*, Springer-Verlag, Berlin, 1985.
5. P.M. Merlin and D.J. Farber, "Recoverability of Communications Protocols," *IEEE Trans. Comm.*, Sept. 1976, pp. 1036-1043.
6. B. Walter, "Timed Petri-Nets for Modeling and Analyzing Protocols with Real-Time Characteristics," in *Protocol Specification, Testing and Verification, III*, H. Rudin and C.H. West, eds., North-Holland, Amsterdam, 1983.
7. *Petri Nets: Central Models and Their Properties*, W. Brauer, W. Reisig, and G. Rozenberg, eds., Springer-Verlag, 1986.
8. C. Ramchandani, *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*, Massachusetts Institute of Technology, Cambridge, Mass., 1974.
9. A.J. van der Hoeven, "Modeling and Implementation of Communications AST Nodes in an Object-Oriented Fashion," tech. report, EE Dept., Delft University of Technology, Delft, The Netherlands, Sept. 1988.
10. P. van Prooijen, "A User Interface for the HiFi System," tech. report, EE Dept., Delft University of Technology, Nov. 1988.
11. H.P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, 1981.
12. J. Annevelink, "HIFI, A Design Method for Implementing Signal Processing Algorithms ...," PhD thesis, Delft University of Technology, Jan. 1987.
13. K. Jainandunsing, "Parallel Algorithms for Solving Systems of Linear Equations and Their Mapping on Systolic Arrays," PhD thesis, Delft University of Technology, Jan. 1989.
14. P. van der Wolf and T.G.R. van Leuken, "Object Type Oriented Data Modeling for VLSI Data Management," *Proc. 25th Design Automation Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 351-356.
15. B.J. Cox, *Object Oriented Programming, An Evolutionary Approach*, Addison Wesley, Reading, Mass., 1986.

Van der Hoeven

De Lange

Deprettere

Dewilde

A.J. van der Hoeven is a researcher at the Delft University of Technology, The Netherlands, where he is working toward the PhD degree in electrical engineering. His primary scientific interests include high-level VLSI design modeling and analysis, high-level VLSI synthesis, and polymorphic programming languages.

Van der Hoeven received the MS degree in electrical engineering from the Delft University of Technology. He is a student member of the IEEE.

A.A.J. de Lange is currently working toward the PhD degree in electrical engineering and computer science at the Delft University of Technology. His main interests are the high-level modeling, simulation, and synthesis of massively parallel pipelined processor arrays and mapping them onto VLSI devices.

De Lange holds an MS degree in electrical engineering from the Delft University of Technology. He has authored more than 20 scientific/technical publications and is a student member of the IEEE.

E.F. Deprettere is an associate professor in the network theory section (signal processing group) of the department of electrical engineering, Delft University of Technology. His current research interests are VLSI and modern signal processing, modeling, computer graphics, VLSI array processing, and mapping signal processing algorithms, network graphs, and matrix equations onto silicon devices.

Deprettere received the MS degree from the Ghent State University in Belgium and the PhD degree from the Delft University of Technology. He coauthored a paper that received a 1989 IEEE SP award.

P.M. Dewilde is professor of network theory at the Delft University of Technology. He has held various research and teaching positions at the University of California, Berkeley, the University of Lagos in Nigeria, and the University of Leuven in Belgium. His research interests include theoretical topics such as the inverse scattering theory and its impact on computational algebra. Practical topics include fast real-time computing on dedicated VLSI devices.

Dewilde received the degree of electrical engineering from the University of Leuven, the license in mathematics from the Belgian Central Examination Commission, and the PhD degree in electrical engineering from Stanford University. He became a Fellow of the IEEE in 1981 for his work on the scattering theory. He has also been a project leader for major European projects in microelectronics.

Questions concerning this article can be addressed to A.J. van der Hoeven, Delft University of Technology, EE Department, Network Theory Section, PO Box 5031, 2600 GA Delft, The Netherlands.