A Digital System Design Language (DDL)

JAMES R. DULEY AND DONALD L. DIETMEYER, MEMBER, IEEE

Abstract—Successful design and manufacture of future digital systems will depend upon the availability of a suitable design language. A concise, precise language is presented which facilitates the specification of complex digital systems. The language 1) is independent of any particular technology, design procedure, machine organization, etc., 2) allows specification at different levels of detail from architecture to detailed Boolean equations, and 3) may be compiled into manufacturing information. Its syntax and semantics permit documents with an organization which parallels the block structure of the systems they specify.

Syntax and semantics of the language are defined. Pragmatics of the language are demonstrated throughout the paper with examples.

Index Terms—Automata, Boolean equations, computer design, declarations, design automation, design language, digital systems, syntax and semantics, system design, system model.

I. INTRODUCTION

SPECIFYING, documenting, and controlling the design of digital systems are problems of increasing severity as such systems continue to grow in size and complexity. Wilkes and Stringer [2] first recognized that a suitable design language could greatly reduce the magnitude of these problems and lead to a complete, precise, yet concise description of digital systems. Unfortunately, their contribution is mostly oriented toward the machine that they were developing at the time and is not generally useful.

Reed's register transfer language [3]–[5] has received wider distribution. It is easily learned, rather generally applicable, and its statements associate directly with hardware. However, a complete description of a system cannot be expressed in this language, no provisions exist for partitioning systems, and the small vocabulary of the language necessitates the use of many symbols. Determining the sequence of events which are to take place in a system from a Reed-language description may not be an easy task.

The LDT language of Burroughs [6]–[9] and the Sequence Chart of IBM [8] supplement Reed's language so that a complete description of a system can be given. The Sequence Chart displays timing and sequencing information in a graphic manner. Both languages are backed by computer programs which generate logic equations from Reed-like statements. The LDT system uses prescribed logic structures to an extent, but does timing analysis. The graphic nature of the Sequence Chart is awkward for most widely available input-output equipment.

Iverson's language [9]–[12] has a large vocabulary and is meant to be universal. It has been and is being used as a programming language and to describe system performance from a programmer's point of view. It offers a very concise and accurate notation for describing interregister transfers. However, a large part of the language is not pertinent to machine design, some of the symbols are not easily associated with hardware, and timing and sequencing are not easily described.

Others have augmented the ALGOL 60 language with hardware-design-oriented features. These languages are less than satisfying: symbols and syntax oriented toward expressing computational algorithms do not always provide a good means for describing hardware [13]–[15].

The Digital systems Design Language of this paper, DDL, was constructed with several goals in mind. First, a design language should not be restricted to any particular hardware technology, machine organization, timing mode, design procedure, etc. All of these have and will continue to change. Second, a design language should be useful to all concerned with a digital system, from those who specify the architecture in block form to those concerned with individual gates. Communication between architects and logic designers is possible only if both groups speak the same language. Third, increased emphasis on design automation suggests that design languages must serve as the source language of future design automation programming complexes. It must be possible to transform the language statements into manufacturing information. Finally, DDL has been constructed so that the document which specifies a system has much the same organization as the system itself.

The remainder of the paper presents the DDL language. In Section II, the terminology is defined and a system model is described. The model parallels the language's block structure and is flexible enough to specify digital systems of arbitrary structure. Sections III, IV, and V define the syntax and semantics of facility, operations, and system declarations, respectively. While syntax is presented in a modified Backus Normal form, it is not divorced from semantics for the benefit of readers with a greater interest in systems than in languages.

Finally, a second-generation computing system with parallel mode of operation, single-address instruction format, multilevel indirect addressing, index registers,
high-speed magnetic core storage, and 1's complement arithmetic is specified in the Appendix. It provides significant meaningful examples throughout the main text and illustrates the concise, precise expression that can be achieved using DDL. While this system is not of major concern, the interested reader may desire to investigate its details after digesting the syntax and semantics presented in the body of the paper.

II. System Model

Often many of the familiar logic circuits of a digital system perform very similar functions and it is advantageous to think of them collectively as "facilities." Memories, registers, sets of terminals, and blocks of combinational circuitry are obvious examples. Other components such as clocks, delay elements, and special circuits and devices are also essential.

Frequently, facilities are grouped into somewhat autonomous units which will be known here as "automata." Both data handling and control facilities are usually found in an automaton, so each automaton of a system will be considered to contain one finite-state machine which exerts control over data facilities\(^1\) [16]. The memory elements of the finite-state machine will be referred to collectively as the "state-sequencing" register of the automaton. The "state" of each automaton is determined by the content of its state-sequencing register. The timing mode of an automaton may be synchronous, asynchronous, or a combination of the two.

If the flow of information to or from a facility is controlled entirely by a single automaton, then that facility is "private" and considered to be a part of the controlling automaton. If two or more automata exert control over a facility, it is a "public" facility. All communication between automata will take place via public facilities.

Fig. 1 illustrates this model of a digital system. Each of \(k\) automata is shown to encompass private facilities and possess input and output terminals for purposes of communicating with public facilities, each other, and the environment of the system. Communications, in particular with the environment, need not be electronic. Console lights, depressed switches, and print on paper are common nonelectronic forms of man–machine communication.

III. Facility Declarations

Every facility of a system described in DDL must be defined to exist and be given a name—"identifier"—via a "declaration" statement. Any sequence of upper- or lower-case letters or digits which begins with a letter serves as an identifier. Facility declaration statements all have the common external syntax:

\[\langle NN\rangle BDD.\]

TT is the type tag and denotes the type of facility being declared to exist. REGISTER, MEMORY, TERMINAL, etc., may be used although only the first two letters are required to distinguish the declaration type. Thus RE, ME, TE, etc., will appear most often in the remainder of this paper. RE\(\beta\), ME\(\beta\), TE\(\beta\), etc., will refer to the first two letters followed by any sequence of letters. The special brackets \(\langle\) and \(\rangle\) always begin a declaration and the period \(\) always terminates a declaration. BDD, representing the body of the declaration, names and describes facilities. The following discussion defines the different facilities which can be declared in DDL.

Terminal and Memory Declarations

The type tags of the terminal and memory (or register, a one-dimensional memory) declarations are TE\(\beta\) and ME\(\beta\) (or RE\(\beta\)), respectively. The bodies of terminal and memory declarations have identical syntax:

\[\text{BDD} \equiv \text{ID}_1, \text{ID}_2, \ldots, \text{ID}_m \quad \text{for } m \geq 1.\]

Each entry ID\(_i\) consists of at least a unique identifier BID, a sequence of arbitrary length of English letters and decimal digits which begins with a letter, by which a set of wires or flip-flops are to be known, and can be of two types.

Type 1: ID\(_i\) :: BID \([F_n, F_{n-1}, \ldots, F_1]\) for \(n \geq 1\)

where

a) \(F_j \equiv I_i^n; I_i^n\)
b) \(F_j \equiv I_j\) and \(I_j\) is a decimal integer.

Type 2: ID\(_i\) :: BID.

"A\[18:2\]" is an example of a Type 1a) ID\(_i\): it may be used in a terminal or memory declaration to indicate the existence of 17 terminals or a 17-bit register, respectively, named A and with bit positions numbered consecutively left to right from 18 to 2. When a special numbering of terminals is not important, the syntax of Type 1b), "A\[17\]," for example, may be used to declare the same facility with bit positions numbered from 1 to

\(^1\) Our unusual definition of an automaton is felt to be both efficacious and pragmatic.
17. Brackets need not appear (Type 2) if one terminal or flip-flop is to be declared. Higher-dimensional arrays may also be declared with Type 1 by using additional arguments of the same syntax within the brackets. For example,

\[(\text{ME}) \text{ IR}[0:7, 15].\]

declares eight 15-bit registers which are to be thought of collectively as a memory.

The concatenation operator \(\cup\) and equal sign \(=\) may be used to partition a set of terminals or a register \(\text{ID}_t\) and associate a name \(\text{ID}_{t_i}\) with each part via the syntax:

\[
\text{ID}_t = \text{ID}_{t_1} \cup \text{ID}_{t_2} \cup \cdots \cup \text{ID}_{t_k}.
\]

For example, if a 24-bit command register is naturally thought of as composed of a 6-bit operation code sub-register, a 3-bit index designator, and a 15-bit address register, then subregisters \(\text{OPN}, \text{IX}, \text{ADDR}\) may be declared and their relationship to register \(\text{COM}\) indicated in the register declaration:

\[(\text{RE}) \text{ COM}[24] = \text{OPN}[0:5] \cup \text{IX}[3:1] \cup \text{ADDR}[15].\]

Other examples of memory and terminal declarations can be found in lines 3, 4, 16–18, 20, and 48 of the Appendix.

After a facility such as a set of terminals or a register has been declared to exist, it or its components may be “referenced” in other syntax. An unbracketed identifier refers to an entire register or set of terminals. Thus “IX” in a statement following the register declaration above refers to either the input or output terminals of three flip-flops. How IX is used determines whether the input or output terminals are being referenced. Use of “\(\text{COM}[7] \cup \text{COM}[8] \cup \text{COM}[9]\)” or “\(\text{COM}[7:9]\)” are equivalent references. Both “\(\text{COM}[10]\)” and “\(\text{ADDR}[1]\)” reference the same single flip-flop of register COM.

The appearance of “\(\text{IR}[1X]\)” following the above example declarations references one of the eight registers of memory IR. The contents of register IX determine which register is being referenced. Such a reference declares the existence of a decoding network by implication. “\(\text{IR}[1X, 1:4]\)” may be used if selected bits of a word of memory IR are to be referenced.

**Time and Delay Declarations**

Time and delay declarations, type tags \(\text{TIB}\) and \(\text{DEB}\), respectively, offer a means of specifying periodic clocks and delay elements, components frequently used to achieve synchronization in digital systems. One or more periodic clocks may be declared in the body of a time declaration. Each clock \(C_t\) is named with a unique identifier \(\text{BID}\) followed by its period \(T\), expressed in seconds and enclosed in parentheses.

\[
\text{BDY} \doteq C_1, C_2, \cdots, C_n \quad \text{for } n \geq 1
\]

where

\[
C_t \doteq \text{BID} (T)
\]

Modified FORTRAN exponent notation is used to express the period \(T\). Thus

\[(\text{TI}) P(10E-6), PP(1.6E-9).\]

defines a 10-\(\mu\)s clock \(P\) and a 1.6-\(\mu\)s clock \(PP\). No synchronization between these clocks is assumed. Identifiers \(P\) and \(PP\) without the parenthetical arguments may be used in subsequent statements to reference the signals generated by these clocks.

The delay declaration specifies transport delay elements. The signal at the output terminal of such a unit has the value which appeared at the input terminal \(\tau\) seconds earlier. Syntax of the delay declarations is the same as that of the time declaration, but with \(\tau\) replacing \(T\). \(\tau\) may be expressed in terms of a declared clock’s period using \(\ast\) for a delimiter between the clock identifier and number.

\[(\text{DE}) \text{ DLY}1(2E-6), \text{ DLY}2(5 \ast P).\]

specifies a 2-\(\mu\)s delay element DLY1 and a 5-\(\mu\)s unit DLY2, assuming \(P\) was declared to be a 10-\(\mu\)s period in a previous time declaration. As with register and memory references, the context in which the delay element identifier appears determines whether its input or output terminal is being referenced. Further examples can be found in lines 2 and 49 of the Appendix.

**Boolean Declaration**

The body of a Boolean declaration, type tag \(\text{BOB}\), consists of Boolean equations \(\text{BE}\) separated by commas. This declaration type then offers a means of specifying the combinational logic circuitry which is to interconnect terminals. The equal sign \(=\) will be referred to later as the “connection” operator: it specifies that the output terminals of the combinational logic specified on its right by a Boolean expression \(\text{BEX}\) are to be connected to the terminals specified by the identifier on its left. The formal syntax is

\[
\text{BDY} \doteq \text{BE}_1, \text{BE}_2, \cdots, \text{BE}_n \quad \text{for } n \geq 1
\]

and

\[
\text{BE}_j \doteq \text{ID}_j = \text{BEX}.
\]

As an example, two Boolean equations appear in the Boolean declaration below:

\(\langle T E \rangle R[15], C[15], Q1[15].\)

\(\langle R E \rangle A[15], X[15].\)

\(\langle B O \rangle Q1 = A \oplus R, C = A \lor X.\)

Actually, 30 logic circuits are being specified since all of the registers and terminals have a dimension of 15. Thus the equation \(Q1 = A \oplus R\) is a brief expression of
15 equations:
\[ \vdots \]
The Boolean declaration is used in lines 6, 21, and 50 of the Appendix.

**Operator Declaration**

The operator declaration, type tag OP\( \beta \), uses the terminal and Boolean declarations to name and define blocks of combinational circuitry which may be time-shared by different facilities. Dummy identifiers not previously declared may appear in this declaration if time-sharing of the combinational circuitry is anticipated. The body of the operator declaration is headed by a list of operator identifiers \( \text{OID}_i \), each of which may be followed by a list of dummy identifiers \( X_i \) if any, enclosed in parentheses, which is followed by dimension information \( D \) enclosed in brackets. The syntax and semantics of the dimensional information are identical to those in the terminal or memory declaration.

Operator identifier syntax is

\[ \text{OID}_i \equiv \text{BID} \]
\[ \text{OID}_i \equiv \text{BID}[D] \]
\[ \text{OID}_i \equiv \text{BID}(X_1, X_2, \ldots, X_k) \]

or

\[ \text{OID}_i \equiv \text{BID}(X_1, X_2, \ldots, X_k)[D]. \]

Terminal and Boolean declarations then complete the body of the operator declaration as auxiliary declarations AUX:

\[ \text{BDY} \equiv \text{OID}_1, \text{OID}_2, \ldots, \text{OID}_n, \text{AUX} \quad \text{for } n \geq 1. \]

The identifiers BID name the output terminals of the declared logic blocks. The bracketed dimension information has the same syntax and significance as in terminal declarations. If dummy identifiers appear they refer to input terminals of the logic blocks.

A 12-bit 1's complement adder named ADD1 with input terminals named \( X \) and \( Y \) is specified below. \( C \) identifies carry terminals within the logic block.

\( \langle \text{OP} \rangle \text{ADD1}(X, Y)[0:11] \)
\( \langle \text{TE} \rangle X[0:11], Y[0:11], C[0:11]. \)
\( \langle \text{BO} \rangle \text{ADD1} = X \oplus Y \oplus (C[1:11] \circ C[0]). \)
\[ C = X \cdot Y \vee (X \vee Y) \cdot (C[1:11] \circ C[0]). \]
\( \cdot \) (end of OP)

Notice the use of the concatenation operator in specifying the end-around carry and the block structure of the language which begins to emerge. The period which terminates the operator declaration is followed by an optional comment enclosed in parentheses; such a comment may follow any terminating period. A similar example is found in lines 19–21 of the Appendix.

Use of "ADD1 (A, R)" in a subsequent statement references the output terminals of this block of logic. If \( A \) and \( R \) have been declared to be 12-bit registers, then the reference "ADD1 (A, R)" indicates that the 12 output terminals of \( A \) are to be connected to the 12 \( X \) terminals via implied combinational logic, and the \( R \) terminals are to be connected to the \( Y \) terminals. The 1's complement sum of the contents of the \( A \) and \( R \) registers appears at the ADD1 terminals. If "ADD1(U, V)" appears in still another statement, then logic equivalent to that shown in Fig. 2 is specified by implication.

**Element Declaration**

Unusual circuits and off-the-shelf logic structures may be introduced into a design with the element declaration, type tag EL\( \beta \), offering the designer unlimited versatility. The declaration uses the "black box" approach: neither the internal structure nor the function of the element is described. The element EID\( j \) is declared by naming the box BID and defining its sets of input terminals IT\( i \), and sets of output terminals OT\( i \). The body of the element declaration is then expressed with the syntax

\[ \text{BDY} \equiv \text{EID}_1, \text{EID}_2, \ldots, \text{EID}_n \quad \text{for } n \geq 1 \]

where

\[ \text{EID}_j \equiv \text{BID} (\text{OT}_1, \text{OT}_2, \ldots, \text{OT}_k : \text{IT}_1, \text{IT}_2, \ldots, \text{IT}_m) \quad \text{for } k, m \geq 0. \]

The OT\( i \) and IT\( j \) have exactly the same syntax and semantics as an ID\( j \) in a terminal declaration.

The designer must specify connections to the terminals of an element in other declarations. A magnetic core memory MCM606 with 12 sense output lines SENSE, 12 write input lines WRITET, 6 address terminals MART, a start read input line SR, and a start write line SW may be introduced with the declaration

\[ \langle \text{EL} \rangle \text{MCM606(SENSE[12]) : WRITET[12]}, \]
\[ \text{MART[6]}, \text{SR}, \text{SW}. \]

Similar declarations appear in lines 5 and 47 of the Appendix.

**Binary Strings**

Strings of 0's and 1's are important facilities even though they are usually not noticed in the final implementation of switching systems. The syntax of a binary string is

\[ \begin{array}{c}
\text{BDY} \\
\langle \text{CO} \rangle \text{BDY}, \text{where BDY is any sequence of characters except the period (.) and bracket (may be introduced at any point, and could be used to discuss the function and application of an element.}
\end{array} \]

A comment declaration with syntax (CO) BDY, where BDY is any sequence of characters except the period (.) and bracket (may be introduced at any point, and could be used to discuss the function and application of an element.
where \( nRk \) specifies the binary value of the string in one of three number systems and \( k \) specifies the length of the string in decimal. \( R \) indicates the number system in which \( n \) is expressed. Either a B for binary, \( \emptyset \) for octal, or D for decimal may be used. Thus the string 001001 may be specified by either 1001B6, 1106, or 9D6.

For added flexibility strings of all 0's, 1's, or don't care (?) are specified by the syntax

\[ nXk \]

where \( n \) is either a 0, 1, or ? and \( k \) again specifies the length of the string. Strings 0X1, 1X1, or ?X1 may be abbreviated, 0, 1, or ?, respectively.

**IV. Operations**

Identifiers referencing input and output terminals of declared facilities serve as the operands of operations which express the interconnections and interactions between facilities. Operators appearing in operations determine the nature of these interconnections and interactions.

"Logic" operators are essential in a digital design language. Table I displays the logic operators of DDL and their hierarchy. With the exception of "logic reduction" and "exponentiation" these operators should be familiar. The symbols are also very common except for the NAND (\( \uparrow \)) and NOR (\( \downarrow \)) differ from the AND (\( \& \)) and OR (\( \vee \)) symbols only by the vertical lines for correlation. Two symbols are provided for specifying many logic operations so the designer may use the symbol he prefers.

The logic "exponentiation" operator (\( \langle \) \( \rangle \)) permits complementation of selective terminals of an operand. This operation requires two operands of length \( n \) (\( n \geq 1 \)), an identifier XID, and a binary string BS, and has the syntax

\[ \text{XID} \setminus \text{BS}. \]

The result of this operation is a set of \( n \) terminals, say \( v_i \), where

\[ v_i = \begin{cases} \text{XID}_i & \text{for } \text{BS}_i = 1 \\ \neg \text{XID}_i & \text{for } \text{BS}_i = 0. \end{cases} \]

Therefore, if \( A \) is a 6-bit register, "\( A \setminus 9D6 \)" references the following set of terminals:


The logic "reduction" operator (\( / \)) permits specification of logic between the \( n \) terminals of any identifier XID. The logic performed is designated by a binary operator \( \alpha \) in the following syntax:

\[ \alpha / \text{XID} \]

where

\[ \alpha \in \{ \&, \circlearrowleft, \lor, \rightarrow, \neg, \emptyset, \vee, \emptyset, \vee \}. \]

The result of logic reduction is one terminal defined by

\[ \text{XID}_1 \alpha \text{XID}_2 \alpha \cdots \alpha \text{XID}_n. \]

As an example, the notation \( / A \) expresses


For the two nonassociative operators NAND and NOR, the following definitions prevail:

\[ \downarrow / \text{XID} \triangleq \neg (\text{XID}_1 \lor \text{XID}_2 \lor \cdots \lor \text{XID}_n) \]

and

\[ \uparrow / \text{XID} \triangleq \neg (\text{XID}_1 \land \text{XID}_2 \land \cdots \land \text{XID}_n). \]

The logic reduction and exponentiation operator can drastically simplify the specification of logic. For example, the logic for detecting when register \( A \) contains the binary string 001001 is described by "\( / A \setminus 9D6 \)" (note from Table II that \( \setminus \) is executed before \( / \)).

Finally, the "arithmetic" and "relation" operators \( +, -, <, \leq, \geq, > \) may be used in a high-level logic specification for purposes of hierarchy, they may be inserted in Table I between logic reduction and AND. These operators specify very complex combinational logic networks which must ultimately be defined using operator declarations.

A "Boolean expression" is composed of identifiers which reference output terminals of facilities, binary strings, operators of Table I, and possibly parentheses to alter the hierarchy. Any such expression specifies combinational logic and generates new logic variables.
(terminals), but does not connect these terminals to input terminals of other facilities. If BE represents a Boolean expression, then

\[ ID_1 = BE \]

and

\[ ID_2 \leftarrow BE \]

define the syntax of the "connection" and "transfer" operator, respectively. The connection and transfer operators operate only on terminals and memory elements, respectively, and ID must be the identifier of a set of terminals while ID must identify a set of memory elements. The connection operator (\(=\)) signifies that the terminals of BE are to be connected directly to ID. On the other hand, the transfer operator (\(\leftarrow\)) requires that the contents of flip-flops be replaced; information is to be transferred to memory elements. Additional circuitry is usually required to actually load flip-flops. The nature of this circuitry is a function of the flip-flop type chosen for implementation; its existence is implied by the transfer operator. Also, the connection of signals to terminals can often be thought of as an immediate operator whereas loading memory elements requires some time. This distinction is fundamental.

A note about length compatibility requirements is appropriate at this point. All binary Boolean logic operators and the transfer and connection operators must have operands of equal dimension. An exception to this rule is found for outputs of one-terminal operands. A single-output terminal may be made compatible in length to other outputs or inputs of length \(k\) by fan-out. Thus, "\([0:5] \leftarrow \text{ADDR}[0]\)" means

\[
\begin{align*}
A[0] & \leftarrow \text{ADDR}[0] \\
A[1] & \leftarrow \text{ADDR}[0] \\
& \vdots \\
A[5] & \leftarrow \text{ADDR}[0].
\end{align*}
\]

Since "shifting" and "counting" operations are so frequently encountered and are so readily implemented, special symbols are warranted for their specification. The syntax of the two operations is very similar and is shown with the corresponding semantics in the first four entries of Table II in which ID identifies a register, \(k\) is an integer, and \(x\) is one of the symbols listed in Table III. These symbols are used for their mnemonic value as well as for conciseness and precision in specification.

The "state-transition" operator, denoted with a right going arrow (\(\rightarrow\)), indicates that the contents of the state-sequencing register are to be altered to the binary sequence which encodes the state whose identifier SID appears at the tip of the arrow. Thus the syntax is

\[ \rightarrow SID \]

and expresses the operation of changing the contents of the state-sequencing register to the encoding of state SID. Observe that both \(\leftarrow\) and \(\rightarrow\) call for the contents of memory elements to be changed: with \(\rightarrow\) the state-sequencing register is implied.

In connection with state sequencing, DDL has facilities to declare the state-sequencing register and to reference unknown state encodings to facilitate interrupt specification. First, the state-sequencing register is declared in a memory or register declaration, but its identifier is preceded by "\#" to distinguish this special register:

\[ ID_i \equiv \# \text{BID}[F_1]. \]

Secondly, the syntax

\[ \# \text{SID} \]

represents the binary string that is the encoding of state SID. Therefore, if

\[ \langle \text{ME} \rangle \# \text{SSR}[10]. \]

is specified, the operation "\(\rightarrow \text{P0}\)" can be written as

\[ \text{SSR} \leftarrow \# \text{P0}. \]

With SSR explicitly declared, information transfers from other registers to SSR and transfers of state codes to other registers may be specified.

Often an automaton has a large state set which can be partitioned in a natural way, and it is desirable to perform such a segmentation. Parts of a state set will be known as "segments" of an automaton and will be documented as blocks contained within the automaton block. Next-state transitions from a state of one segment to a state of another segment (of the same automaton) are expressed with the basic syntax

\[ \Rightarrow \text{SEGID}(\rightarrow \text{NID}, \Rightarrow \text{RID}). \]

The identifier SEGID names the segment which contains the next state NID of the automaton. If that next state is the first state to appear in the specification of the segment, \(\rightarrow \text{NID}\) may be deleted from the parenthetical argument list. In the above example, RID identifies the state of the original segment to which the automaton is to return if the second segment does not explicitly indicate all next states. Just \(\Rightarrow\) may appear, for example, to indicate that a return is required. In all cases, the symbol \(\Rightarrow\) specifies a transition between segments and the symbol \(\rightarrow\) signifies a new state within a segment or set of states. Both symbols cause a new state encoding to be transferred to the state-sequencing register of the automaton.

Automata communicate with one another via public facilities, and activity in one automaton may influence activity in another automaton. This occurrence may be emphasized by using the "activation" operator (\(\exists\)) and the syntax
TABLE II
DDL OPERATORS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count Up</td>
<td>( \uparrow_k ) ID</td>
<td>Register ID is incremented by 1 or by ( k ), respectively</td>
</tr>
<tr>
<td>Count Down</td>
<td>( \downarrow_k ) ID</td>
<td>Register ID is decremented by 1 or by ( k ), respectively</td>
</tr>
<tr>
<td>Shift Right</td>
<td>( \Rightarrow_k ) ID</td>
<td>Register ID is shifted right by 1 or by ( k ), respectively</td>
</tr>
<tr>
<td>Shift Left</td>
<td>( \Leftarrow_k ) ID</td>
<td>Register ID is shifted left by 1 or by ( k ), respectively</td>
</tr>
</tbody>
</table>

Activation \( \equiv \text{AUID} \) (CSOP) | CSOP is a set of operations that influences automaton AUID

Connection \( ID = BE \) | The terminals ID are connected to the network defined by the Boolean expression

Transfer \( ID \leftarrow BE \) | Memory elements ID are to be loaded from the network defined by the Boolean expression

Transition Type 1 \( \rightarrow SID \) | Execute a transition to state SID where SID is in the same block; i.e., change the state-sequencing register to the encoding of state SID

Transition Type 2 \( \Rightarrow SID (\rightarrow NID, \Rightarrow RID) \) | Execute a transition to state NID in segment SEG and return to state RID upon execution of a return operation

Return Transition \( \Rightarrow \) | Return to the state specified by a transition type 2

IF-THEN \( | BE | \text{CSOP} \) | If \( BE = 1 \), then execute CSOP

IF-THEN-ELSE \( | BE | \text{CSOP}_1 ; \text{CSOP}_0 \) | If \( BE = 1 \), then execute \( \text{CSOP}_1 \). If \( BE = 0 \), execute \( \text{CSOP}_0 \)

IF-VALUE \( | BE | \text{CSOP}_1 , | BE | \text{CSOP}_0 , | \rightarrow | \text{CSOP}_i , | \rightarrow | \text{CSOP}_e , | \rightarrow | \text{CSOP}_m \) | For \( k = 0, \ldots, 2^m - 1 \)

Copy \( | \text{SID} \) | Makes \( I_2 = I_1 + 1 \) copies of DD, where DD may be any CSOP, state, segment or automaton

Sharp Type 1 \( (\text{RE}) \text{ID} \) | Distinguishes the state sequencing register

Sharp Type 2 \( \# \text{SID} \) | References the binary encoding of state SID

Star \( \ast \text{OP} \) | Frees the operation OP, of its global conditions

TABLE III
SHIFT CONTROL CHARACTERS

<table>
<thead>
<tr>
<th>( x )</th>
<th>Type of Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>Leave vacated bits of ( A ) unchanged</td>
</tr>
<tr>
<td>0</td>
<td>Enter 0's</td>
</tr>
<tr>
<td>1</td>
<td>Enter 1's</td>
</tr>
<tr>
<td>c</td>
<td>Circulate ( A )</td>
</tr>
<tr>
<td>e</td>
<td>Extend the leftmost or rightmost bit of ( A )</td>
</tr>
</tbody>
</table>

\( \equiv \text{AUID} \) (CSOP)

where AUID is the name of an automaton and CSOP is a set of operations (CSOP will be defined in the next paragraph). \( \equiv \text{AUID} \) serves only to improve the readability of a document and corresponds to no hardware. It is used to remind the designer that the operations in CSOP affect the operation of automaton AUID. For example,

\( \equiv DC (C = A \cdot B, E = F) \)

indicates that signals are to be connected to public terminals \( C \) and \( E \). \( \equiv DC \) serves only as a reminder to the reader of the document that automaton DC will be influenced by these signals.

Conditional Operations

A "compatible set of operations," or CSOP, is a list of operations \( \text{OP}_1 \) (including conditional operations to be defined) which can be executed simultaneously by the declared hardware:

\[ \text{CSOP} = \text{OP}_1, \text{OP}_2, \ldots, \text{OP}_n \quad \text{for } n \geq 1. \]

"\( A \leftarrow C, B \leftarrow D, \rightarrow \text{STATE1} \)" is a CSOP, but "\( A \leftarrow C, A \leftarrow D \)" is not since \( A \) cannot be loaded from both \( C \) and \( D \) simultaneously. ("\( A \leftarrow C \lor D \)" would be meaningful.) The order in which the \( \text{OP}_i \) are specified is inconsequential. This, in general, is not true in programming languages, but is basic for a design language since processes are performed in parallel.

In most cases in digital design, a condition must be satisfied before an operation is to be performed. To facilitate precise and concise expression, two basic formats for the specification of such conditional operations are introduced.

First, \( | \text{BE} | \text{CSOP}_1 ; \text{CSOP}_0 \)

defines the syntax of the IF-THEN-ELSE conditional
operations where \(BE\) is a one-terminal Boolean expression. When \(BE\) has the value of 1, \(CSOP_1\) is executed; when \(BE\) has the value of 0, \(CSOP_0\) is executed. The delimiters may be translated as

\[
\text{if } \begin{array}{|c|c|c|c|}
1 & \text{then} & \text{else} & \text{end of condition}
\end{array}
\]

This syntax was chosen for power of expression as well as conciseness. The syntax is more powerful than that of ALGOL 60, for example, since it eliminates the need for parentheses around sequences of operations and removes the IF-THEN-ELSE ambiguity (i.e., \(CSOP_1\) may be another IF-THEN-ELSE operation). A variation of the IF-THEN-ELSE condition is the IF-THEN structure:

\[
\begin{array}{|c|}
\hline
\text{BE} \\
\hline
\end{array} \quad \text{CSOP}_1.
\]

In this case, no operations are to be executed when \(BE\) is 0.

The IF-THEN-ELSE and IF-THEN structures only accept a one-terminal Boolean expression. When \(BE\) names \(n\) terminals, the IF-VALUE structure is used to express the condition and what operations are to be executed for each of the \(2^n\) values which the \(n\) terminals may present. The general syntax of the IF-VALUE statement is

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{BE} & \text{of} & \text{CSOP}_0 & \text{of} & \text{CSOP}_1 & \ldots & \text{of} & \text{CSOP}_{2^n-1} \\
\hline
\end{array}
\]

If the binary value of the \(n\)-terminal \(BE\) is equal to the decimal integer contained in \(,\) then the CSOP immediately to the right is executed. This is equivalent to writing \(2^n\) IF-THEN operations

\[
\begin{array}{|c|}
\hline
\cdot \big/ \text{BE} \big/ \text{kD} \big/ \text{CSOP}_k, \\
\hline
\end{array}
\]

with \(k\) going from 0 to \(2^n-1\).

For example, let \(X\) represent a 3-terminal set, and let \(A\), \(B\), and \(C\) identify three CSOPs. Then the valid IF-VALUE operation

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{X} & \text{of} & \text{A} & \text{of} & \text{B} & \text{of} & \text{C} \\
\hline
\end{array}
\]

has the semantics of

\[
\begin{array}{|c|}
\hline
\cdot \big/ \text{X} \big/ \text{0D3} \\
\hline
\cdot \big/ \text{X} \big/ \text{1D3} \\
\hline
\cdot \big/ \text{X} \big/ \text{2D3} \\
\hline
\cdot \big/ \text{X} \big/ \text{7D3} \\
\hline
\end{array}
\]

An alternate equivalent form of the IF-VALUE statement of this example is

\[
\begin{array}{|c|c|c|c|}
\hline
\text{X} & \text{of} & \text{A} & \text{of} & \text{B} & \text{of} & \text{C} \\
\hline
\end{array}
\]

Notice the similarity between the use of the colon in the IF-VALUE and its use within the brackets of the memory and terminal declarations: the syntax is identical. In addition, the designer can use the semicolon (ELSE) to represent the remainder of the \(2^n\) values that are not specified in the delimiters (\(\{\}\)). Thus, the previous example can be written as

\[
\begin{array}{|c|c|c|c|}
\hline
\text{X} & \text{of} & \text{A} & \text{of} & \text{B} & \text{of} & \text{C} \\
\hline
\end{array}
\]

In review, Table II summarizes the basic system operations allowed in DDL. At this point the reader should be able to read and interpret most of the Appendix. The next section will define the remainder of the syntax of DDL.

V. System Block Declarations

In this section the declarations that specify the block structure of the system shown in Fig. 1 are defined. These declarations have the same general format as the facility declarations and define to exist states, automata, segments of an automaton, and systems. These declarations also assign identifiers to these parts of the system.

State Declaration

The state declaration is used to declare state identifiers and the CSOPs that are to be performed when the automaton is in each of its states. The state declaration has the syntax

\[
\langle \text{ST}\rangle \text{ SL.}
\]

where \(\text{SL}\) is a list of states

\[
\text{SL} \equiv \text{ST}_1 \text{ST}_2 \ldots \text{ST}_n \quad \text{for } n \geq 1
\]

and each state \(\text{ST}_j\) may have either of the following two formats:

\[
\begin{align*}
\text{ST}_j \equiv & \text{SID : CSOP.} \\
\text{ST}_j \equiv & \text{SID : BE : CSOP.}
\end{align*}
\]

\(\text{SID}\) is the state identifier, \(\text{CSOP}\) is a compatible set of operations to be executed when the automaton is in state \(\text{SID}\), and \(\text{BE}\) is a Boolean expression. In the first format, \(\text{CSOP}\) is executed whenever the automaton enters state \(\text{SID}\), i.e., whenever the state-sequencing register contains the encoding of state \(\text{SID}\). With the second form, the automaton remains in state \(\text{SID}\) without executing \(\text{CSOP}\) until \(\text{BE}\) is satisfied. This is very useful in describing idle or wait states.

The state declaration

\[
\langle \text{ST}\rangle \text{ P0 : GO : R } \leftarrow \text{A} \rightarrow \text{P1.}
\]

\[
\text{P1 : A } \leftarrow \text{A} \lor \text{R, } \rightarrow \text{P0.}
\]

specifies two states. State P0 is an idle state: the automaton waits in state P0 until signal GO is received. Upon receiving this signal, the automaton transfers
B to R and makes a transition to state P1. When the automaton reaches state P1, it generates the necessary control signals to cause transfer of A ∨ R to A, and simultaneously returns to state P0 to await another GO signal.

It should be noted that each state identifier SID can be regarded as naming a condition or terminal which bears a Boolean 1 when the automaton is in state SID, and a Boolean 0 otherwise. In the example, before the transfer R ← B can take place, two conditions, P0 and GO, must be satisfied.

Since a state is not a unit of hardware, a state identifier SID can be used to identify a terminal. An automaton is in state SID when the encoding of SID (♯ SID) is in the state-sequencing register. The Boolean equation

\[ \text{SID} = \cdot / \text{SSR} \setminus \# \text{SID} \]

defines hardware capable of detecting this occurrence. A designer may use this syntax and the Boolean declaration to define state terminals if special encodings are desirable. If state P0 must be encoded 001001, then the designer may write

\[ \langle \text{B0} \rangle \text{P0} = \cdot / \text{SSR} \setminus 9D6. \]

**Automaton Declaration**

The automaton declaration defines an automaton to exist. The declaration associates an identifier with the automaton and specifies the automaton's states, operations, and private facilities. The syntax of the automaton declaration is

\[ \langle \text{AU} \beta \rangle \text{HEAD BDY}. \]

where the body, BDY, declares all private facilities of the automaton and its states. The syntax is

\[ \text{BDY} \equiv \text{DL1DL2} \ldots \text{DLn} \quad \text{for} \ n \geq 0 \]

where DLn is a facility declaration, state declaration, or segment declaration (to be defined). The heading HEAD is of one of the following four types:

- HEAD ≡ ID : BE : CSOP
- HEAD ≡ ID : BE
- HEAD ≡ ID : CSOP
- HEAD ≡ ID :

In each case ID is the identifier by which the automaton will be known. If NCOP is the set of all operations with the exception of connection operations, then BE is a condition that must be satisfied before any NCOP operation of the automaton can be executed.\(^1\) On the other hand, CSOP in the heading of an automaton declaration represents a set of operations that are to be executed in each state of the automaton. BE is ideal for specifying a condition such as the clock signal of synchronous automata. CSOP in the heading circumvents the need to specify the same operation in each state.

Examples of asynchronous automata declarations are found in lines 8–10 and 13–14 of the Appendix where circuity is specified to insure proper functioning of the console switches. Synchronous automata CPU and MEM are declared in lines 15–45 and 46–59 of the Appendix. Notice the private facilities of these latter automata.

**Segment Declaration**

The segment declaration is almost identical to the automaton declarations; its syntax is

\[ \langle \text{SE} \beta \rangle \text{HEAD BDY}. \]

BDY of the segment declaration is the same as that of the automaton declaration except that DLn may not be a segment declaration. HEAD is exactly the same as in the automaton declaration. The BE of the heading of the segment declaration is a condition that must be satisfied before NCOP operations of the segment are executed. CSOP is a set of operations that are to be executed whenever the automaton is in a state of the segment.

Automaton CPU of the Appendix is partitioned into three segments: DECODE (lines 22–35), ADDSUB (lines 36–41), and LDASTA (lines 42–45).

**System Declaration**

The syntax of the system declaration is also nearly identical to that of the automaton declaration.

\[ \langle \text{SY} \gamma \rangle \text{HEAD BDY}. \]

The DLn of the BDY of a system declaration include all public facility declarations and all automaton declarations of the system. The DLn may not contain state or segment declarations directly; only automata or segments of automata may contain a state declaration and only automata may contain segments. The BE of the HEAD must be satisfied before any NCOP operation can be performed in the system. The CSOP is executed as the heading prescribes. Lines 1–60 of the Appendix declare system EXAMPLE with its automata and public facilities.

**Other Useful Syntax**

Any operation, state, segment, or automaton may be embedded in the "copy operation" which has syntax

\[ \{ \text{BID} = I_1 : I_2 \} \text{ DD}. \]

\(^1\) Global conditions usually do not interrupt the flow of information in combinational logic, but govern the loading of memory elements. Hence connection operations are excluded.
where DD is the embedded syntax, BID is the index identifier of the operation, and I₁ and I₂ are lower and upper limits on the index, respectively. The hardware declared or implied by DD is to be duplicated \(I₂ - I₁ + 1\) times. Index BID may be used within declaration DD to distinguish copies. Thus, if many nearly identical circuits are required in a system, the designer must specify that circuit only once. An example of the copy operator is shown in lines 7–14 of the Appendix where asynchronous circuitry is copied for the three console switches.

The "identifier declaration" also reduces the amount of writing necessary to specify a system. This declaration assigns an identifier IID to either a CSOP or sequence of concatenated system identifiers ID. Thereafter, instead of writing CSOPs or CCOs repeatedly, the designers simply reference their declared IIDIs. The syntax of the identifier declaration is

\[
(ID\delta)\ BDY.
\]

where

\[
\begin{align*}
BDY & \equiv \text{IDD}_1, \text{IDD}_2, \ldots, \text{IDD}_n \quad \text{for} \ n \geq 1 \\
\text{IDD}_1 & \equiv \text{IID} = (\text{CSOP}) \\
\text{IDD}_k & \equiv \text{IID} = \text{ID}_1 \circ \text{ID}_2 \circ \ldots \circ \text{ID}_k \quad \text{for} \ k \geq 1. 
\end{align*}
\]

Example:

\[
\text{SUBSTITUTE} = (B[1:6] \leftarrow \text{odd}, \leftrightarrow A, \# C).
\]

Finally, the star operator ( \(\ast\) ) is introduced to free operations in set NCOP of the HEAD condition of the segment, automaton and system block. The syntax is simply

\[
\ast OP_i
\]

where \(OP_i\) is the operation in a compatible set not to be influenced by the global condition. This operator can save considerable writing when a complex condition is required for almost all operations. Tables II and IV summarize DDL operators and declarations, respectively.

### VI. Conclusions

DDL can be a very useful tool for the design, documentation, and even simulation of digital systems. Its conciseness facilitates expressing, analyzing, modifying, and, in general, dealing with large digital systems in an organized manner. The language is mnemonic and fundamental in concept to facilitate design and enhance readability. The organization of a document can parallel the block structure of the anticipated hardware.

The language permits a specification of sufficient precision so that a hardware realization of the system’s logic can be obtained using programmable algorithms.

Techniques have been developed which transform a DDL document into another DDL document consisting of Boolean equations for combinational logic and next-state equations for flip-flops [1]. These techniques have been used to write an experimental software program to compile implementations of general system design documents [17]. Such programs make possible complete logic design automaton for designs specified in DDL.\(^4\) Compilation philosophy will be the subject of a future paper.

At the University of Wisconsin, the authors and others have found DDL to be a valuable teaching and research aid. We have used this symbolic language to achieve a highly precise, concise specification of existing and hypothetical computing systems of various types. Time-sharing, multiprocessing, and data channel design are features that have been included. Designs in the synchronous, asynchronous, and synchronous-asynchronous time modes have also been expressed.

\(^4\) Transliteration of symbols is necessary when employing standard high-speed printers. A paper tape system with an ALGOL vocabulary has been employed very successfully without distorting the symbol set very much. IBM Selectric typewriters with a special head can also be used.
APPENDIX

SPECIFICATION OF A SECOND-GENERATION COMPUTING SYSTEM

<SY> EXAMPLE:

<TE> P[100E - 9].

<RE> M[6], MREG[12], READ.

<TE> R[1:3], START, STEP, RESET, SW[1:3], OUTP[1:3], RR, CW, AVAIL.

<EL> THREE NO SWITCHES(CL[1:3], OP[1:3]).

<BO> START = OUTP[1], STEP = OUTP[2], RESET = OUTP[3].

{i = 1:3}

<AU> NO BOUNCE[i]:


<AU> SWITCH[i]:

<ST> OUTP[1] = 1. . . . (end of ST, AU)

<AU> CPU : P:

<RE> IC[6], A[0:11], B[0:11], OV, CY, OP[0:3], IX[1:2], ADDR[0:5].

<ME> IM[1:2:6].

<TE> ADD.

<OP> ADDL(Y, Z) [0:11]:

<BO> ADDL = Y + Z + C[1:11] + C[0]. . . . (end of OP)

<SEG> DECODE

<ST> P0: START+RESET: MAR=IC, IC, CY+1, +P1. (begin instruction fetch)

<ST> P1: | P0 | [STEP] RST[1:2] = 1X2., | AVAIL| & MEM(RR=1), +P3; +P1.

<ST> P2: IC=0, A=0, MREG1=0, R=0, OV=0, RST = 1X3, +PO. (reset CPU)

<ST> P3: READ: IX+MREG[4:15], ADDR+MREG[1:6:11],

<BO> MREG[1:6:11], IC+0, IC, +PO. (read instruction from MREG1)

<ST> P4: | /IX | MAR+ADDR, +P1; | (/P0[0:11])+(/IX)| ADDR+ADDL

<BO> ADDL = Y + Z + C[1:11] + C[0]. . . . (end of OP)

<SEG> ADDSUB:

<ST> R0: AVAIL; = MEM(RR = 1), +R1. (initiate read)

<ST> R1: READ: R[0:OPN[3]] = MREG1; MREG1, +R2. (complement for subtraction)

<ST> R2: A=ADD(A,R), ADD = = OP[0], [A][0], +R4; +R3.

<ST> R3: OV=A[0], R[0], =. (set overflow)

<ST> R4: OV=A[0], R[0], =. . . . (end of ST, SEG)

<SEG> LDADA:

<ST> S0: AVAIL; = MEM(RR = 1), +S1.

<ST> S1: READ: A=MREG1, +. (load A)

<BO> TO: AVAIL; = MEM(CW = 1), +. (store A) . . . . (end of ST, SEG, AU)

<AU> MEM: P:

<EL> MEMORY(SENSET[12]: SR, SW, WRIT[12], MART[6]).

<RE> M[6], MREG[12].

<DE> DLY1(.8E-6), DLY2(.7E-6).

<BO> WRIT[6] = MREG, MREG = MAR, AVAIL = UO.

<ST> UO: RR v CM, [RR] = PO, READ+O, [CM] = UO,

<BO> MAR=MREG+MREG1.

<BO> DLY1 = 1, SR = 1, +P1.

<BO> DLY1: MREG+SENSET, +P2.

<BO> MREG1+MREG, READ+1, +S2. (read)

<BO> DLY2 = 1, SR = 1, +S1.

<BO> DLY1 = +S2, (cleared)

<BO> SW = 1, DLY2 = 1, +S3.

<BO> DLY2: = UO. (written or restored)... (end of ST, AU)

(end of SY)
Design of Pattern Classifiers with the Updating Property Using Stochastic Approximation Techniques

S. S. YAU, MEMBER, IEEE, AND J. M. SCHUMPERT, MEMBER, IEEE

Abstract—A nonparametric training procedure for finding the optimal weights of the discriminant functions of a pattern classifier in any optimization criterion, expressible as a convex function from an arbitrary sequence of sample patterns, is proposed. This design procedure is based on the stochastic approximation technique, and has the updating property because it processes the sample patterns whenever they become available. The procedure is used to find the optimal weights for the least-mean-square error criterion, and is shown to require very simple computation which leads to simple implementation. Both two-category and multi-category cases are considered, and an acceleration scheme to increase the rate of convergence for the training procedure is also presented. These results are demonstrated by examples.

Index Terms—Acceleration scheme, implementation, least-mean-square error criterion, nonparametric training procedures, optimal weights, pattern classifiers, stochastic approximation techniques, two- and multi-category cases, updating property.

I. INTRODUCTION

A PATTERN classifier is a decision-making system whose input is a set of measurements of an input pattern represented by a vector \( x = (x_1, x_2, \ldots, x_n) \), and whose output is a signal indicating the category to which the input pattern is classified. The set of measurements which is made in the receptor is assumed to be a description of each pattern, which is sufficient for classification purposes [1], [2]. The (Euclidean) vector space \( X \) of all possible \( x \)'s is called the “pattern space.” When the form of the probability distribution of the patterns in each category over \( X \) is known, we can easily use statistical decision theory to design an optimal (Bayesian) classifier which yields the minimum probability of misclassification [1]–[5]. However, the information available to the designer about the patterns to be classified is usually only a set of sample patterns, which is assumed to be a good representation of the patterns in various categories. In this case it is often possible to achieve a reasonably good performance by using a set of discriminant functions \( g_i(x), i = 1, 2, \ldots, R \), where \( R \) is the number of categories. The discriminant functions considered usually have a fixed form with adjustable weights. For ex-