Parsing Electronic Circuits in a Logic Grammar

Takushi Tanaka

Abstract—**Understanding circuits is a prerequisite for circuit design and trouble shooting. We consider circuit understanding by engineers to be a process which starts with a structural analysis and then proceeds to a causal analysis. As a step toward automatic circuit understanding, we present a new method for analyzing circuit structures. A circuit is viewed as a sentence and its elements as words. Circuit structures are defined by rules written in a logic grammar called Definite Clause Set Grammar (DCSG). Given circuits are decomposed into parse trees by the DCSG top–down parsing mechanism. These parse trees represent hierarchical structures of functional blocks. This representation is presented as one step in the process of automatic understanding of circuit structures.**

Index Terms—**Circuit structure, electronic circuit, expert system, knowledge representation, logic programming, logic grammar, syntactic processing, understanding circuits.**

I. Introduction

WHEN an engineer first looks at a circuit schematic, he tries to partition the circuit into familiar subcircuits with known goals. He then tries to pursue the causality of electrical events through those subcircuits to determine if and how it achieves the overall goal of the circuit. This is based on the fact that electronic circuits are designed as goal-oriented compositions of basic circuits with specific functions. Therefore, understanding a circuit means finding a hierarchical structure of functional blocks and rediscovering the designer's original intentions.

The work reported here is a step toward the automation of this type of circuit understanding based on the idea that circuits share a feature with language: they both carry information based on the speaker/designer's intentions mapped onto their structures. As a step toward automatic circuit understanding, we present a new method for analyzing circuit structures. We view circuits as free word order sentences, and their elements as words. Circuit structures are defined by a DCG-like logic grammar formalism [7], called Definite Clause Set Grammar [13] (**DCSG**), developed for analyzing free word order languages. A set of grammar rules, when translated into Prolog clauses, forms a logic program which will execute top–down parsing when interpreted in a straight forward manner.

When an unknown circuit is given, such a logic program will analyze the circuit and derive a parse tree from the circuit. This contrasts with earlier circuit analysis programs based on circuit theory such as SPICE [14], which functions as a circuit simulator to derive voltages and currents from the circuit (Fig. 1). Since a parse tree shows a hierarchical structure of functional blocks composing a circuit, it can help an engineer identify which elements contribute to which subfunctions and how the total function of the circuit is achieved by its subfunctions for trouble shooting, redesigning, or modifying the circuit. That is, a circuit parser could advise engineers how to interpret circuit organization. This differs from a circuit simulator, which works as an oscilloscope to measure voltages and currents.

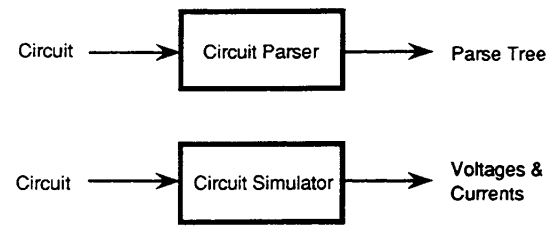


Fig. 1. Circuit parser versus circuit simulator.

A grammatical approach for circuit structures was originally developed using **Plex** grammars as an example of syntactic pattern recognition [4]. Plex grammars assume languages with symbols having an arbitrary number of attachment points for connection to other symbols, as opposed to symbols in a natural languages, which have two attachment points, a left one and a right one. Unlike ordinary grammars, rewrite rules in Plex grammars require additional fields that explicitly define interconnections of attachment points. Therefore, parsing mechanisms for Plex grammars were difficult to implement.

DCSG rules for circuit structures do not have such fields, and therefore, are closer to ordinary grammar rules. Circuit elements are represented by terminal symbols, and functional blocks by nonterminal symbols. As DCSG syntax allows terminal and nonterminal symbols with variables, connections of circuit elements are implicitly defined by shared variables. These variables are bound to shared nodes connecting those elements. The shared variable method enables analyzing circuit structures by parsing free word order languages defined by DCSG rules. A set of grammar rules written in DCSG syntax are directly translated into a logic program which decomposes given circuits into parse trees. We can analyze electronic circuits by simply defining circuit grammars. Thus there is no need to build special parsing mechanisms for circuits.

Another important study which inspired many ideas in this area was De Kleer's work on human reasoning in circuit recognition [2]. This study focused on qualitative simulation of circuit behaviors. The qualitative simulation pursues causality of electrical events to find a causal chain which explains circuit

Manuscript received December 1, 1989; revised December 27, 1991 and August 3, 1992. This work was supported in part by the Japan Ministry of Education under Grant-in-Aid for Scientific Research B-6346025.

The author is with the Department of Computer Science, Fukuoka Institute of Technology, Fukuoka, 811-02, Japan.

IEEE Log Number 9207074.

1041–4347/93$03.00 © 1993 IEEE

functions. Since his work was concerned with explaining circuit behavior from electrical principles, knowledge of hierarchically organized circuit structures was explicitly avoided. Instead of using knowledge of circuit structures, the study used heuristics and teleologies to disambiguate the causal chain.

In actual design and analysis of electronic circuits, engineers use not only basic principles of electricity, but also knowledge of circuit structures. Therefore, our paper focuses on circuit structure as functional blocks and formalizes this knowledge as circuit grammars. Since we view designed circuits as a kind of language, we may consider the circuit behaviors as the meanings of the circuit. That is, circuits can be viewed as sentences which are designed to implement specific behaviors as their meanings. From this point of view, De Kleer's work can be seen as being concerned with circuit physics, while this paper concerns circuit syntax.

This paper began as a project to develop symbolic expressions for electronic circuits. We first represented a circuit as a set of assertions [10]. Each assertion, which corresponded to a circuit element, stated its connections in the circuit. However this method had the disadvantage that it could not represent relations between circuits, e.g., equivalence or subcircuit relationships. Next, we represented circuits as a set of composite terms by replacing predicate symbols with function symbols. This new method enabled us to treat circuits as a kind of language [11]. Here, we formalize this method in terms of DCSG. We show how knowledge of circuit structures is coded into grammar rules and how circuits are decomposed into parse trees. Then, we discuss problems of context dependent circuits and semantic information in grammar rules. Finally, we examine performance of circuit grammars developed so far.

II. A Logic Grammar for Free Word Order Languages

Since circuit structures will be defined in a logic grammar called DCSG, we briefly introduce DCSG, a formalism originally developed for analyzing free word order languages [13]. In contrast with the DCG formalism [7], DCSG is based on the concept of using difference sets, as opposed to the difference *lists* used by DCG, to define grammar rules.

A. Free Word Order Languages

A free word order language $\boldsymbol{L}(\boldsymbol{G}')$ can be defined by modifying the definition of a formal grammar. We define a context-free free word order grammar $\boldsymbol{G}'$ to be a quadruple $\langle V_N, V_T, P, S\rangle$ where V_N is a finite set of nonterminal symbols, V_T is a finite set of terminal symbols and P is a finite set of grammar rules of the form:

$$A \longrightarrow B_1, B_2, \cdots, B_n, \qquad n \geq 1$$

$$A \in V_N, \quad B_i \in V_N \cup V_T, \quad i = 1, \cdots, n$$

and $S(\in V_N)$ is the starting symbol. The above grammar rule means rewriting a symbol A not with the string of symbols "$B_1, B_2, \cdots, B_n$", but with the set of symbols $\{B_1, B_2, \cdots, B_n\}$. A sentence in the language $\boldsymbol{L}(\boldsymbol{G}')$ is a set of terminal symbols which is derived from S by successive application of grammar rules. Here the sentence is a multiset which admits multiple occurrences of elements taken from V_T. Each nonterminal symbol used to derive a sentence can be viewed as a name given to a subset of the multiset. This modified formal grammar is the base of DCSG formalism.

B. DCSG

A free word order sentence is, therefore, a multiset of elements taken from V_T. Each nonterminal symbol used to derive the sentence can be viewed as a name given to a subset of the multiset. Therefore, grammar rules represent relationships between these subsets and their elements. Using the predicates "*subset*" and "*member*," we can translate grammar rules directly into a Prolog program. As with DCG parsing, this results in a Prolog program that will parse the DCSG defined language in a top–down manner. Each grammar rule is translated into a definite clause by the DCSG translation procedure.

In the present paper, both terminal and nonterminal symbols in grammar rules are written as strings of characters beginning with a lower case letter. Each terminal symbol in a grammar rule is surrounded by "[" and "]," so that the translation procedure can distinguish terminal from nonterminal symbols.

The general form of the translation from a grammar rule to a definite clause is shown by (1) and (1(a)).

$$A \longrightarrow B_1, B_2, \cdots, B_n. \tag{1}$$

$$\begin{aligned} subset(A, S_0, S_n) \; :- \; & subset(B_1, S_0, S_1), \\ & subset(B_2, S_1, S_2), \\ & \cdots \\ & subset(B_n, S_{n-1}, S_n). \end{aligned} \tag{1a}$$

Here, all symbols in the grammar rule (1) are assumed to be nonterminal symbols. That is, A and $B_1, \cdots, B_n$ are names given to subsets of elements composing a free word order sentence. The definite clause (1a) explicitly represents relationships between these subsets. The arguments $S_0, S_1, \cdots, S_n$ in (1a) are substituted by elements composing a free word order sentence. The predicate *subset* is used to refer to a subset of an object set which is given as the second argument, while the first argument is the name of its subset. The third argument is a complementary set which is the remainder of the second argument less the first; e.g., "$subset(A, S_0, S_n)$" states that A is a subset of S_0 and that S_n is the remainder. That is, the predicate *subset* defines A as the difference between two sets S_0 and S_n. The clause (1a) states if B_1 is a subset of S_0, B_2 is a subset of $S_1, \cdots$, and B_n is a subset of S_{n-1}, then A is a subset of S_0.

If "$[B_i]$" $(1 \leq i \leq n)$ is found in the right-hand side of grammar rules, where B_i is assumed to be a terminal symbol, then "$member(B_i, S_{i-1}, S_i)$" is used instead of "$subset(B_i, S_{i-1}, S_i)$" in the translation, because the terminal symbol B_i is an element composing a free word order sentence.

The predicate *member* is defined by the following definite clauses:

$$\left.\begin{array}{l} member(M,[M|X],X). \\ member(M,[A|X],[A|Y]) : -member(M,X,Y) \end{array}\right\}. \quad (2)$$

The predicate *member* has three arguments. The first is an element of a set. The second is the whole set. The third is the complementary set of the first. That is, both terminal and nonterminal symbols are represented as differences of the last two arguments of these predicates.

DCSG syntax allows terminal and nonterminal symbols with variables. When the grammar rules are used in parsing sentences, grammar rule variables are instantiated with specific terms in the object sentences.

C. The Parsing Process

Parsing consists of the iterative application of grammar rules controlled by the Prolog backtracking mechanism. Grammar rules, such as (1a), contains only nonterminal symbols, and grammar rules that generate terminal symbols differ slightly. When the clause (1a) is used in parsing, an object set (multiset) is substituted into S_0, and the nonterminal symbol A is identified as a subset of S_0. During parsing, the elements in S_0 are successively replaced to form the sequence $S_1, \cdots, S_n$. Procedurally, the clause (1a) can be read as: in order to show A to be a subset of S_0, show B_1 to be a subset of S_0, show B_2 to be a subset of S_1 which is the remainder when B_1 is removed from $S_0, \cdots$, show B_n to be a subset of S_{n-1}, and set the remainder to be S_n.

Nonterminal symbols finally generate terminal symbols such as:

$$B \longrightarrow [C].$$

This type of rule is translated into the following definite clause:

$$subset(B,S_0,S_1) : -member(C,S_0,S_1).$$

This clause can be read as: in order to show that B is a subset of S_0, show C to be a member of S_0, and set the remainder to S_1. Here, the variable S_0 will be bound to a current context, which is the unidentified part of the object set. The goal "*member*(C,S_0,S_1)" searches the context S_0 for the terminal symbol C using the rule (2). If C is found, the goal returns the remainder in the variable S_1. Thus the nonterminal symbol B is identified as a subset of the current context. When the starting symbol S is finally identified as a subset of the object set, the top–down parsing succeeds. Details of this parsing are explained in Section III-C with examples.

D. Extensions to DCSG

Several extensions to the basic DCSG formalism are necessary for actual use. One extension introduces context dependent features into grammar rules by extending DCSG syntax. The context dependent conditions "*test* C" and "*not* C" in the grammar rules

$$A \longrightarrow B_1, \cdots, B_i, test\ C, B_{i+1}, \cdots, B_n$$
$$A \longrightarrow B_1, \cdots, B_i, not\ C, B_{i+1}, \cdots, B_n$$

are translated into

$$subset(C,S_i,\_)$$
$$not\ subset(C,S_i,\_)$$

respectively. The "_"s are the anonymous variables which may be ignored. In the process of parsing the nonterminal symbol A, "*test* C" and "*not* C", respectively, demand the existence and the absence of subset C in the current parsing context. These conditions will apply to S_i after identifying $B_1, \cdots, B_i$. When the condition succeeds, the process of identifying $B_{i+1}, \cdots, B_n$ continues on S_i. However, the translation differs for terminal symbols. Conditions of the form "*test* $[C]$" and "*not* $[C]$" will be translated into

$$member(C,S_i,\_)$$
$$not\ member(C,S_i,\_)$$

which, respectively, demand the existence and the absence of element C in the current context S_i. These conditions are used in grammar rules for parsing context dependent circuits (Section IV-B). Another extension which enable coupling syntactic and sematic information is explained in Section V.

III. Parsing Electronic Circuits

In this section we define a circuit grammar for analyzing electronic circuits consisting of hierarchically organized functional blocks. Circuit elements are represented by terminal symbols, and functional blocks by non terminal symbols. Structures of functional blocks are defined by grammar rules. Each grammar rule is translated into a definite clause, a fragment of Prolog program, by the DCSG translation procedure. A circuit grammar defines a set of circuits in the same way that an ordinary grammar defines a language. If a given circuit is in the set, the circuit can be analyzed by the Prolog translation of the circuit grammar.

A. Circuits Represented as Sentences

We will consider the class of circuits consisting only of resistors, bipolar transistors, and external terminals, e.g., bipolar integrated circuits. In order to represent this class of circuits, we introduce four types of terminal symbols with variables:

$$\begin{aligned} V_T = \{ & resistor(Name,Node1,Node2), \\ & npnTr(Name,Base,Emitter,Collector), \\ & pnpTr(Name,Base,Emitter,Collector), \\ & terminal(Name,Node)\}. \end{aligned}$$

The first argument of each terminal symbol is a name of specific element, and the others are connected nodes of the element. Connections of circuit elements are defined by binding terminal symbol variables to shared nodes. For example, the circuit *cd*42 (Fig. 2) which is an analog IC circuit

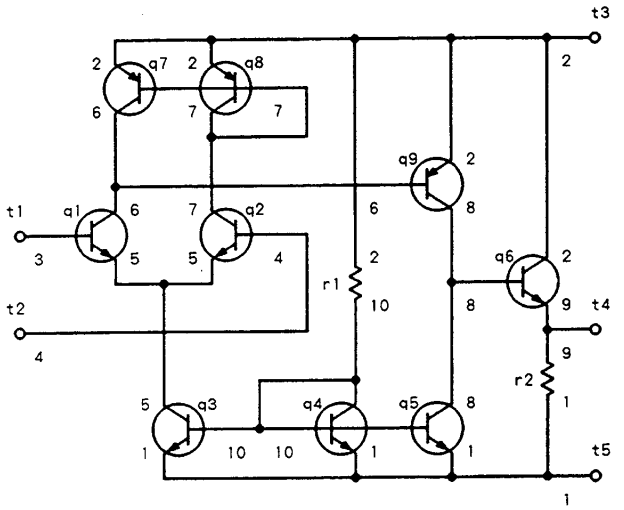


Fig. 2. Operational amplifier $cd42$.

(operational amplifier [15]) is represented by the following assertion:

$$\begin{aligned} cd42([&resistor(r1,2,10), resistor(r2,9,1),\\ &npnTr(q1,3,5,6), npnTr(q2,4,5,7),\\ &npnTr(q3,10,1,5), npnTr(q4,10,1,10),\\ &npnTr(q5,10,1,8), npnTr(q6,8,9,2),\\ &pnpTr(q7,7,2,6), pnpTr(q8,7,2,7),\\ &pnpTr(q9,6,2,8), terminal(t1,3),\\ &terminal(t2,4), terminal(t3,2), terminal(t4,9),\\ &terminal(t5,1)]). \end{aligned} \tag{3}$$

The terminal symbol "$resistor(r1,2,10)$" denotes a resistor named $r1$ connecting node 2 and node 10. The node order is arbitrary because a resistor does not have polarity. "$npnTr(q1,3,5,6)$" denotes an s named $q1$ with the base connected to node 3, the emitter to node 5, and the collector to node 6, respectively. "$terminal(t1,3)$" denotes an external terminal named $t1$ connected to node 3. The list of these terminal symbols surrounded by "[" and "]" is a free word order sentence denoting the circuit. As the list is used to represent a multiset, the order of the elements is not important. The assertion "$cd42([\cdots])$" states that the list "$[\cdots]$" is the circuit $cd42$.

B. Grammar Rules

In designing electronic circuits, engineers use many basic circuits with specific functions, and view these circuits as macro-elements or macrodevices [15]. Such functional blocks as "diode-connected transistor," "voltage regulator," "current source," "active load," and "differential amplifier" are found in the circuit $cd42$. The circuit $cd42$ itself can be viewed as the functional block "operational amplifier," Here, we are considering a class of circuits which consist of hierarchically organized functional blocks, and define the class in DCSG as a free word order language. Each functional block is defined as a nonterminal symbol, with the circuit elements as terminal symbols, as defined in the previous section.

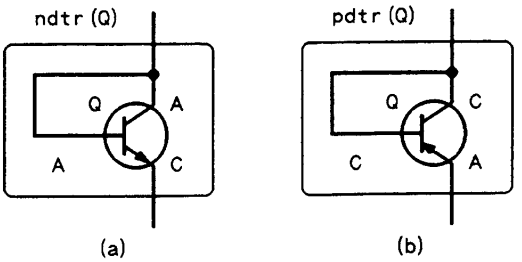


Fig. 3. Diode connected transistor.

The simplest functional block, the "diode-connected transistor," is defined by (4) and (5) as a nonterminal symbol. The rule (4) means that an NPN-transistor Q with the base and the collector connected to the same node A works functionally as a diode (Fig. 3(a)). Here, "$ndtr(Q)$" is a name given to the diode-connected transistor. The rule (5) is defined for PNP-transistor similar to the rule (4):

$$dtr(ndtr(Q),A,C) \longrightarrow [npnTr(Q,A,C,A)] \tag{4}$$

$$dtr(pdtr(Q),A,C) \longrightarrow [pnpTr(Q,C,A,C)]. \tag{5}$$

Translating the grammar rule (4) into a definite clause gives (4a), as follows:

$$\begin{aligned} &subset(dtr(ndtr(Q),A,C),S_0,S_1) :-\\ &\qquad member(npnTr(Q,A,C,A),S_0,S_1). \end{aligned} \tag{4a}$$

When the clause (4a) is used in parsing, an object circuit is substituted into S_0, and the functional block "$dtr(ndtr(Q),A,C)$" is identified in the circuit, and the remainder of the circuit is bound to the variable S_1. Procedurally, the clause (4a) can be read as: in order to identify the nonterminal symbol "$dtr(ndtr(Q),A,C)$" in S_0, find the terminal symbol "$npnTr(Q,A,C,A)$" in the circuit S_0, then set the remainder into S_1. More formally, it can also be read as: "$npnTr(Q,A,C,A)$" being a member of the multiset S_0 implies "$dtr(ndtr(Q),A,C,A)$" to be a subset of the multiset.

Strictly speaking, this definition is incomplete because it cannot eliminate odd diode-connected transistors which short their anode and cathode. In order to make this grammar rule complete, we must add a condition that nodes A and C must be different. This can be done by adding the inequality condition "not $A = C$" into the right-hand side of the definite clause (4a). However, this condition is unnecessary as long as we are considering the class of well-designed circuits. Therefore, we generally omit this type of condition. Though we usually omit these conditions, different variables in grammar rules are assumed to be substituted by different values.

The following grammar rule (6) enables us to refer to resistors independently of their node order, e.g., a resistor represented by a terminal symbol either "$resistor(r1,2,10)$" or "$resistor(r1,10,2)$" can be identified by a nonterminal symbol "$res(r1,2,10)$".

$$\begin{aligned} res(X,A,B) \longrightarrow\ &[resistor(X,A,B)];\\ &[resistor(X,B,A)]. \end{aligned} \tag{6}$$

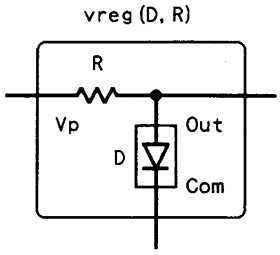


Fig. 4. V_{be} voltage regulator.

Here, the symbol ";" is used for abbreviation of two grammar rules with the same left-hand side. This grammar rule is translated into the following clause:

$$\begin{aligned} subset(res(X,A,B),S_0,S_1) \ :- \\ member(resistor(X,A,B),S_0,S_1); \\ member(resistor(X,B,A),S_0,S_1). \end{aligned} \quad (6a)$$

The symbol ";" in the clause is the logical connective "or." When an object circuit is substituted into S_0, the first subgoal is to identify "$resistor(X,A,B)$" as an element of the circuit. If it succeeds, this subgoal returns the remainder of the circuit in S_1. If it fails, the second subgoal tries "$resistor(X,B,A)$" instead of "$resistor(X,A,B)$". This kind of rule is defined for all nonpolar elements such as capacitors and inductors.

The following grammar rule defines a nonterminal symbol "V_{be} voltage regulator" (Fig. 4) which has V_{be} $(0.6 \sim 0.7$ v) as its output:

$$\begin{aligned} vbe\ Reg(vreg(D,R),Vp,\mathrm{Com},\mathrm{Out}) \longrightarrow \\ dtr(D,Out,Com), \\ res(R,Vp,Out). \end{aligned} \quad (7)$$

The grammar rule is translated into a definite clause as:

$$\begin{aligned} subset(vbeReg(vreg(D,R),Vp,Com,Out),S_0,S_2) :- \\ subset(dtr(D,Out,Com),S_0,S_1), \\ subset(res(R,Vp,Out),S_1,S_2). \end{aligned} \quad (7a)$$

Procedurally, the clause (7a) can be read as: in order to find a V_{be} voltage regulator in the circuit S_0, fist find a diode-connected transistor "$dtr(D,Out,Com)$" in S_0, then find a resistor "res(R,Vp,Out)" in S_1 which is the remainder of "$dtr(D,Out,Com)$" from S_0. The remainder of the circuit is held in S_2 from which the V_{be} voltage regulator has been removed.

A simple current source (sink-type) is defined by the following grammar rule (8). The current source consists of a V_{be} voltage regulator and an NPN-transistor (Fig. 5):

$$\begin{aligned} cSink(sink(V,R,Q),In,Com) \longrightarrow \\ vbeReg(VR,\_,Com,B), \\ [npnTr(Q,B,Com,In)]. \end{aligned} \quad (8)$$

However, this definition is also incomplete. Since the second argument of "$vbeReg$" is an anonymous variable, the definition does not specify the connections of the input node of the V_{be} voltage regulator, though the input node must be connected to a power supply.

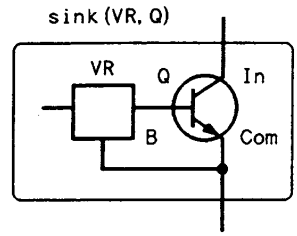


Fig. 5. Simple current source.

One method to make this rule complete is to include a power supply as a component of the current source. As a result, a single power supply in an actual circuit is shared with many parts of the circuit as a voltage source. This method is developed for context-dependent circuits in Section IV. More sophisticated methods would use electrical conditions as sematic information, but in this section we focus on grammar rules specifying topological conditions. We will discuss extending DCSG by introducing semantic information in Section V.

The "emitter-coupled pair" (Fig. 6), "active load (source type)" (Fig. 7), and "single-ended differential amplifier" (Fig. 8) functional blocks are defined as follows:

$$\begin{aligned} eCoupledPair(ecup(Q1,Q2),B1,B2,E,C1,C2) \longrightarrow \\ [npnTr(Q1,B1,E,C1)], \\ [npnTR(Q2,B2,E,C2)] \end{aligned} \quad (9)$$

$$\begin{aligned} activeLoad(al(D,Q),Ref,Vp,Ld) \longrightarrow \\ dtr(D,Vp,Ref), \\ [pnpTr(Q,Ref,Vp,Ld)] \end{aligned} \quad (10)$$

$$\begin{aligned} sDiffAmp(sdAmp(EC,AL,CS),B1,B2,C1,Vp,Vm) \longrightarrow \\ eCoupledPair(EC,B1,B2,E,C1,C2), \\ activeLoad(Al,C2,Vp,C1), \\ cSink(CS,E,Vm). \end{aligned} \quad (11)$$

These are the nonterminal symbols we have introduced so far:

$$\begin{aligned} V_N = \{&dtr(Q,A,C), res(R,A,B), \\ &vbeReg(VR,Vp,Com,Out), cSink(CS,In,Com), \\ &eCoupledPair(EC,B1,B2,E,C1,C2), \\ &activeLoad(AL,Ref,Vp,Ld), \\ &sDiffAmp(DA,B1,B2,C1,Vp,Vm)\}. \end{aligned}$$

C. Top–Down Parsing

Since all nonterminal symbols are functional blocks, any nonterminal symbol may be viewed as a starting symbol. In top–down parsing of free word order languages, grammar rules are applied to a starting symbol until it generates terminal symbols, and the terminal symbols are found in the object sentence. When all of the terminal symbols are found, and no

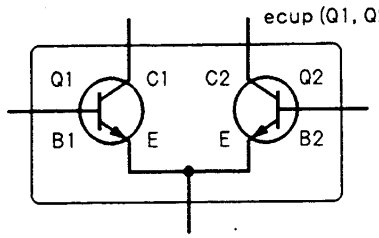


Fig. 6. Emitter coupled pair.

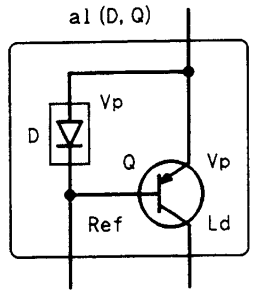


Fig. 7. Active load (source type).

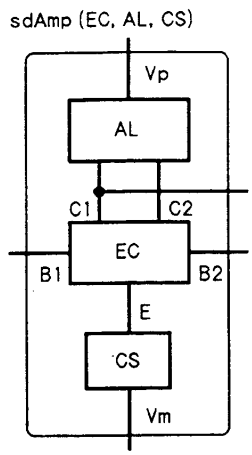


Fig. 8. Single-ended differential amplifier.

unknown terminal symbols remain in the sentence, then the top–down parsing succeeds. If a given circuit is a grammatical one, the following goal clause successfully parses the circuit. Although we have not introduced a single starting symbol, the variable S in the following goal clause works as the starting symbol which generates all of the nonterminal symbols:

$$? - subset(S, ObjectCircuit, [\,]).$$

Since the third argument of "*subset*" is a null circuit, the goal clause asks whether the whole *ObjectCircuit* is identified as a nonterminal symbol. Since the variable S matches all the nonterminal symbols defined by grammar rules, the system tries all of the grammar rules one after another until the remaining part of "*subset*" becomes the null circuit. In this case, the top–down parsing mechanism does not work efficiently, because the first argument S does not contain any information about how to parse the object circuit. However, if the object circuit is a grammatical one, the goal clause eventually parses the circuit (Section VI).

Even if the given circuit is not grammatical, namely, even if the structure of the whole circuit is not defined as a nonterminal symbol, the following goal clause identifies all known structures in the given circuit:

$$? - subset(S, ObjectCircuit, Rest).$$

The goal clause tries to identify all kinds of the nonterminal symbols in the object circuit. If a nonterminal symbol is identified in the object circuit, the remainder of the circuit is bound to the variable *Rest*. The goal clause separates the object circuit into a known structure and an unknown part.

The top–down mechanism works efficiently when a specific nonterminal symbol is given as the starting symbol. The following goal clause executes the top–down parsing for circuit $cd42$:

$$\begin{aligned} &?-cd42(Circuit),\\ &\quad subset(sDiffAmp(DA, In1, In2, Out, Vp, Vm)\\ &\qquad Circuit, Rest). \end{aligned} \tag{12}$$

The first subgoal "$cd42(Circuit)$" binds circuit $cd42$ to the variable *Circuit*. The second subgoal "*subset*$(\cdots)$" identifies a functional block "*sDiffAmp*$(\cdots)$" (single-ended differential amplifier) in the circuit $cd42$. Here, the non terminal symbol "*sDiffAmp*$(\cdots)$" works as the starting symbol. According to these grammar rules, the initial goal (12) is repeatedly decomposed into subgoals until each "*subset*" goal reaches "*member*" goals which identify terminal symbols. Each time a "*member*" goal succeeds, an element is identified as a part of a functional block, and the unknown part of the object circuit decreases.

When the goal clause (12) succeeds, we will acquire the following values for the variables in (12):

$$\begin{aligned} DA &= sdAmp(ecup(q1, q2), al(pdtr(q8), q7),\\ &\qquad sink(vreg(ndtr(q4), r1), q3))\\ In1 &= 3\\ In2 &= 4\\ Out &= 6\\ Vp &= 2\\ Vm &= 1\\ Rest &= [resistor(r2, 9, 1), npnTr(q5, 10, 1, 8),\\ &\qquad npnTr(q6, 8, 9, 2), pnpTr(q9, 6, 2, 8),\\ &\qquad terminal(t1, 3), terminal(t2, 4), terminal(t3, 2),\\ &\qquad terminal(t4, 9), terminal(t5, 1)]. \end{aligned}$$

The value of the variable DA in the starting symbol keeps track of successful goals. It forms a hierarchical structure of functional blocks and can be viewed as a parse tree for the circuit corresponding to a syntactic structure of a sentence (Fig. 9). The variables $In1, \cdots, Vm$ are bound to specific nodes in the circuit. The variable *Rest* holds the remainder of circuit $cd42$, less the differential amplifier.

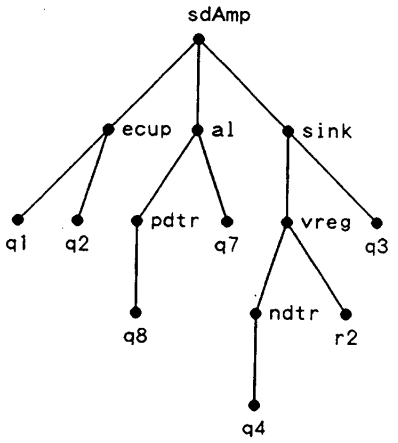


Fig. 9. Parse tree for a differential amplifier in *cd*42.

In order to parse the whole circuit of *cd*42, we further define nonterminal symbols for "common emitter," "emitter follower," and "operational amplifier with five external terminals" as:

$$commonEmitter(pnpCE(Q, CS), In, Out, Vp, Vm) \longrightarrow [pnpTr(Q, In, Vp, Out)], \; cSink(CS, Out, Vm). \quad (13)$$

$$emitterFollower(npnEF(Q, R), In, Out, Vp, Vm) \longrightarrow [npnTr(Q, In, Out, Vp)], \; res(R, Out, Vm). \quad (14)$$

$$\begin{aligned} &opAmp(opAmp1(DA, CE, EF), \\ &\quad In1, In2, Out, Vp, Vm) \longrightarrow \\ &\quad sDiffAmp(DA, In1, In2, O1, Vp, Vm), \\ &\quad commonEmitter(CE, O1, O2, Vp, Vm), \\ &\quad emitterFollower(EF, O2, Out, Vp, Vm), \\ &\quad [terminal(T1, \mathrm{In1})], \\ &\quad [terminal(T2, \mathrm{In2})], \\ &\quad [terminal(T3, \mathrm{Out})], \\ &\quad [terminal(T4, Vp)], \\ &\quad [terminal(T5, Vm)]. \end{aligned} \quad (15)$$

Although we have defined grammar rules for all functional blocks which constitutes *cd*42, the grammar rules are not adequate to parse the circuit *cd*42 because the circuit has context dependent structures. We will discuss this problem in the next section.

IV. Context Dependent Circuits

If an electronic circuit consists only of hierarchically organized functional blocks, its structure is defined by context-free rules. However we have found structures which cannot be defined by context-free rules in actual circuits. In order to analyze the circuits, we introduce context dependent conditions into grammar rules.

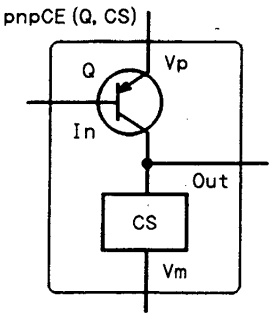


Fig. 10. Common emitter (PNP-transistor).

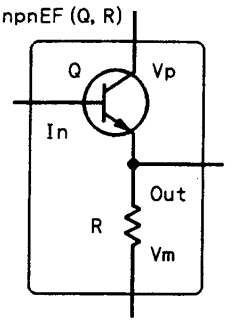


Fig. 11. Emitter follower (NPN-transistor).

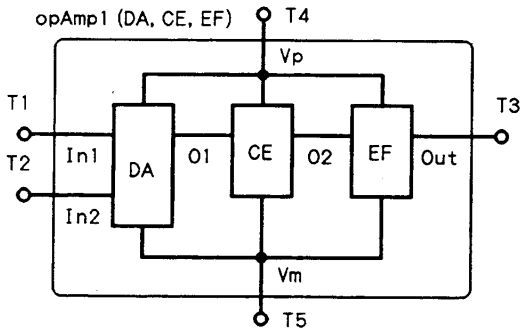


Fig. 12. Operational amplifier with external terminals.

A. Combined Circuits

Consider the circuit design process in contrast with sentence generation. Suppose a circuit goal generates two current sources as subgoals. Each current source needs a voltage source, so two voltage sources are generated. When one of the voltages is derived from the other, an engineer may combine two voltage sources into one voltage source for simplicity. That is, he has the ability to use context dependent circuit generation rules, while we have so far only considered context-free rules. Fig. 13 shows an example of this kind of circuit, which we call voltage-type degenerate. Two current sources $(q2, q3)$ share one V_{be} voltage regulator $vreg(ndtr(q1), r1)$ as a voltage source.

When a goal clause tries to identify two current sources in the circuit of Fig. 13, after one current source is identified by the grammar rule (8), the remainder part of the circuit has no

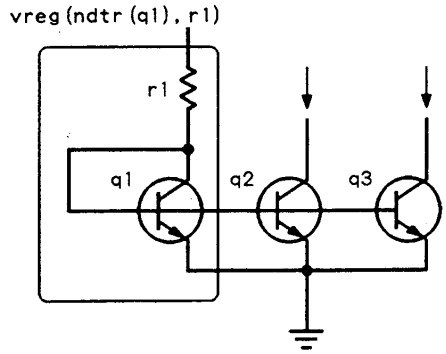


Fig. 13. Voltage type degenerate.

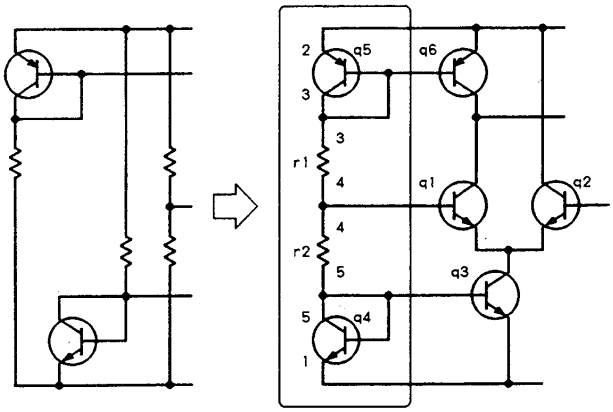


Fig. 14. Current type degenerate.

voltage source. Therefore, another current source cannot be found, since it lacks a voltage source. We solve this problem by introducing context dependent features into grammar rules.

We also found another type of combined circuit which we call current-type degenerate. Fig. 14 shows an example of this type of circuit. Here three voltage sources use almost the same current as their inputs and the inputs are connected in series to share the current. Analyzing this type of circuit is more difficult than the previous one, because most of the connections in the original circuit are changed by the degeneration.

B. Grammar Rules for Combined Circuits

Using the extended DCSG of Section II-D, we can define grammar rules for parsing context dependent circuits. The following grammar rule defines a current source in addition to (8) for voltage-type degeneration:

$$cSink(sink(VR, Q), In, Com) \longrightarrow test\ vbeReg(VR, \_, Com, B), [npnTr(Q, B, Com, In)]. \quad (16)$$

This rule does not remove the V_{be} voltage regulator in the process of identifying a current source. In order to identify two current sources in Fig. 13, the first current source must be identified by (16), and the second current source must be identified by (8). This can be done by backtracking mechanism of Prolog.

Now, we can parse the whole circuit of *cd42* in Fig. 2 by the following goal clause. The parse tree is shown in Fig. 15. All of the grammar rules used in this parsing are shown in

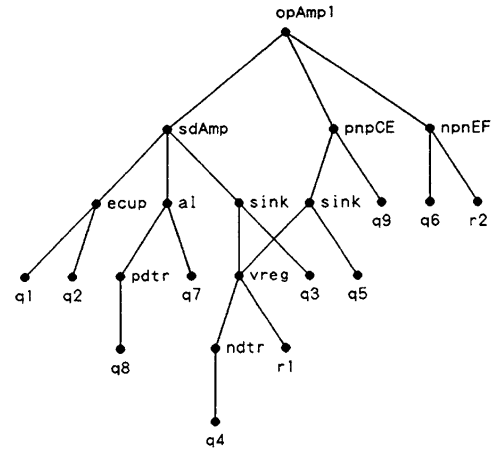


Fig. 15. Parse tree for circuit *cd42*.

Appendix A:

$$?-cd42(Circuit), subset(opAmp(X, \_, \_, \_, \_, \_), Circuit, [\,]). \quad (17)$$

$$X = opAmp1(sdAmp(ecup(q1, q2), a1(pdtr(q8), q7), sink(vreg(ndtr(q4), r1), q3)), pnpCE(q9, sink(vreg(ndtr(q4), r1), q5)), npnEF(q6, r2)).$$

Grammar rules for parsing current-type degeneration are harder to define because most connections in the original circuit are changed. In order to parse the circuit in Fig. 14, we define a non terminal symbol "multiple voltage divider" which derives multiple voltages from a single power supply. The divider consists of two "*seriesRD*"s which are resistors and diode-connected transistors connected in series as shown in Fig. 16 "*resDtr*" refers to either a resistor or a diode-connected transistor:

$$mvDivider(mvdiv(X, Y), In, Out, Com) \longrightarrow seriesRD(X, In, Out), seriesRD(Y, Out, Com) \quad (18)$$

$$seriesRD(X, A, C) \longrightarrow resDtr(X, A, C) \quad (19)$$

$$seriesRD(srd(X, Y), A, C) \longrightarrow resDtr(X, A, B), seriesRD(Y, B, C) \quad (20)$$

$$resDtr(X, A, C) \longrightarrow res(X, A, C); dtr(X, A, C). \quad (21)$$

Here, the definition of "seriesRD" lacks the condition of series connection, namely, that no element other than X and Y must be connected to the center node B, since the output voltages of the multiple voltage divider are derived

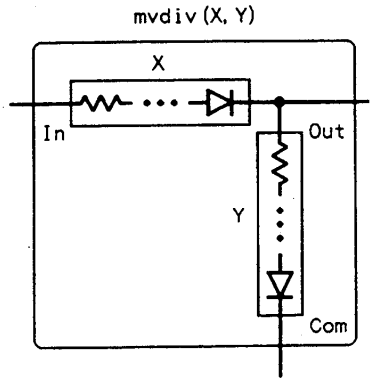


Fig. 16. Multiple voltage divider.

from the center node. Electrically, each output current in the multiple voltage divider is much smaller than the current in main branch of the divider, so we can view this circuit as a series connection. In order to parse the circuit in Fig. 14, three voltages sources must be identified. First, two voltage sources must be identified by "*test mvDivider*$(\cdots)$". Then, the third must be identified by "*mvDivider*$(\cdots)$".

These top–down methods for parsing context dependent circuits are inefficient, because the selection of the grammar rules is based on backtracking form the last stage of parsing which requires that no elements remain unknown. Two methods can be considered to overcome this problem. One method is to rewrite the circuit into an equivalent circuit bottom–up, before top–down parsing. Here, the equivalent circuit is the original circuit before degeneration. We will discuss the bottom–up method in another paper. Another method is to introduce semantic information into syntactic parsing. For example, if a voltage source X is identified between nodes A and B, then generate an electrical constraint that the voltages between nodes A and B are controlled by X. The later process succeeds in parsing this structure by using these electrical constraints instead of a physical voltage source. We will discuss this method in the next section.

V. Electrical Conditions

In this section, we extend DCSG to introduce semantic information into grammar rules. This extension enables us to use electrical conditions in parsing circuit structures. The electrical conditions reduce ambiguity in parsing circuit topology and improve parsing efficiency. Electrical conditions are also useful in parsing context dependent circuits.

A. Coupling Syntactic and Semantic Information

We introduce semantic terms into grammar rules, so that information other than circuit topology is available in parsing. The semantic terms are surrounded by "{" and "}" in any position of the right-hand side of grammar rules. For example, the following rule has a semantic condition:

$$A \longrightarrow B_1, \cdots, B_n, \{[C]\}. \tag{22}$$

When this rule is used in parsing, not only subcircuits $B_1, \cdots, B_n$ must be identified, but also the semantic condition "C" must be filled. A DCSG rule using this extension is translated into the following clause:

$$\begin{aligned} ss(A, S_0, S_n, E_0, E_n) :- & \; ss(B_1, S_0, S_1, E_0, E_1), \\ & \cdots \\ & ss(B_n, S_{n-1}, S_n, E_{n-1}, E_n), \\ & member(C, E_n, \_). \end{aligned} \tag{22a}$$

The first three arguments of the predicate "$ss(\cdots)$" are the same arguments as "*subset*$(\cdots)$" (Section II-D). That is, an object circuit is bound to S_0, then, the subcircuit "A" is identified in that circuit. The remainder of the circuit is substituted into S_n.

The last two arguments of the predicate "$ss(\cdots)$" are used for semantic information. The fourth argument is used as an input and the last argument as an output of the predicate. Usually, electrical environments or constraints on the circuit are given in the fourth argument. The information is used by semantic terms in the right-hand side of the grammar rules. Similar to ordinary terminal symbols in grammar rules, the semantic term "$\{[C]\}$" requires that the current semantic information "E_n" contains "C" as a member. However, unlike ordinary terminal symbols, it does not remove "C" from the current semantic information in transferring to the next goal.

When we define another free word order language to represent semantic information, semantic terms of the form "$\{C\}$" become available. It is translated into "$ss(C, E_n, \_, \_, \_)$". Here, "C" must be defined by grammar rules as a nonterminal symbol of the language for semantic information. The grammar rule is translated into a definite clause using the same mechanism as extended DCSG. For example, the nonterminal symbol "*powerNode*(N)" is defined by three terminal symbols as:

$$\begin{aligned} powerNode(N) \longrightarrow & \; [ground(N)]; \\ & [positivePowerSupply(N)]; \\ & [negativePowerSupply(N)]. \end{aligned} \tag{23}$$

The nonterminal symbol "*powerNode*(N)" represents an electrical property of nodes that either the node N is grounded, connected to a positive power supply, or connected to a negative power supply.

The following forms of grammar rules are also available. The first rule demands "C" must not be a member of the semantic information, and the second one demands that "C" must not be derived from semantic information:

$$A \longrightarrow B_1, \cdots, B_n, \{not\ [C]\}$$
$$A \longrightarrow B_1, \cdots, B_n, \{not\ C\}.$$

The above semantic terms are respectively translated into:

$$not\ member(C, E_n, \_)$$
$$not\ ss(C, E_n, \_, \_, \_, \_).$$

The following form of grammar rules adds the semantic term "C" into the current semantic information:

$$A \longrightarrow B_1, \cdots, B_n, \{add\ [C]\}.$$

The rule is translated into:

$$ss(A, S_0, S_n E_0, E_{n+1}) :- ss(B_1, S_0, S_1, E_0, E_1),$$
$$\cdots$$
$$ss(B_n, S_{n-1}, S_n, E_{n-1}, E_n),$$
$$E_{n+1} = [C|E_n].$$

We use this form of rules to append an electrical constraint "C" defined by the circuit topology "A" into the semantic information. Subcircuits "$B_1, \cdots, B_n$", specific functional blocks, also add their own electrical constraints into the semantic information. That is, the last two arguments of the predicate "$ss(\cdots)$" incrementally transfer semantic information defined by circuit syntactic structures. Each time a subcircuit is identified in the object circuit, electrical constraints are added into the semantic information.

B. Removing Undesired Interpretations

Using electrical conditions, we can remove undesired interpretations in parsing circuit topology. The electrical conditions prune incorrect selections generated by topological conditions, and improve parsing efficiency. For example, the following rule which defines emitter followers, has an electrical condition that the emitter must not be a power-node, i.e., a node connected to a power-supply (Fig. 11).

$$emitterFollower(npnEF(Q, R), In, Out, Vp, Vm) \longrightarrow$$
$$[npnTr(Q, In, Out, Vp)],$$
$$res(R, Out, Vm),$$
$$\{not\ powerNode(\text{Out})\}. \quad (24)$$

If a grammar rule without this electrical condition were used, four emitter-followers would be found in the circuit $cd42$ (Fig. 2): $npnEF(q3, r2)$, $npnEF(q4, r2)$, $npnEF(q5, r2)$, and $npnEF(q6, r2)$. However, the first three of them really do no work as emitter followers. As these undesired interpretations do not contribute to the final goal of the object circuit, they would be removed by backtracking mechanisms of top–down parsing.

If a terminal symbol "$ground(1)$" which states that the node 1 is to be grounded is given in the sematic information, the nonterminal symbol "$powerNode(1)$" is derived by (23). Thus undesired interpretations which could not satisfy the electrical condition "$not\ powerNode(1)$" are rejected.

C. Parsing Context Dependent Circuits

We developed a method for parsing context dependent circuits in Section IV. In order to parse a degenerate circuit, two grammar rules were applied. However, the method was inefficient, because the selection of the two grammar rules was determined by backtracking from the last stage of top–down parsing.

Using semantic information concerning electrical constraints, we can efficiently parse context dependent circuits. The key idea of this method is to transfer electrical constraints defined by preceding subgoals to the following subgoals, so that the following subgoals can use the constraints to identify functional blocks in spite of the lack of topological conditions.

TABLE I
PARSING SPEEDS FOR CIRCUIT $cd42$

grammar	goal (27)	goal (28)
A	0.183 s	0.033 s
B	0.117 s	0.017 s

First we define grammar rules for degenerate circuits. For example, the following grammar rule is defined for "V_{be} voltage regulator" instead of (7):

$$vbeReg(vreg(D, R), Vp, Com, Out) \longrightarrow$$
$$dtr(D, Out, Com),$$
$$res(R, Vp, Out),$$
$$\{add\ [vbeControlled(vreg(D, R), Out, Com)]\}. \quad (25)$$

When a structure of "V_{be} voltage regulator" is found in object circuits, the grammar rule appends the electrical constraints "$vbeControlled(\text{vreg}(D, R), Out, Com)$" into the semantic information. It means that the voltage between nodes Out and Com is controlled by $vreg(D, R)$ and fixed at V_{be} volts. That is, after a voltage regulator is identified, the current parsing context does not have the regulator, but the semantic information has an electrical constraint which is derived from the regulator.

Next, we define grammar rules for functional blocks which have degenerate circuits as components. The following grammar rule "$cSink$" is defined for sink-type current sources in addition to (8):

$$cSink(sink(VR, Q), In, Com) \longrightarrow$$
$$\{[vbeControlled(VR, B, Com)]\},$$
$$[npnTr(Q, B, Com, In)]. \quad (26)$$

When the grammar rule (8) fails due to lacking the topological condition "$cSink$," the alternative rule (26) succeeds using the sematic condition. Using this method, we can efficiently parse degenerate circuits. The complete set of grammar rules with semantic conditions necessary to parse the circuit $cd42$ is shown in Appendix B.

VI. EXPERIMENTS

We will examine parsing speeds of the Prolog translation of the circuit grammars developed so far. Table I shows CPU times for parsing circuit $cd42$ by C-Prolog$_+$ [8] on a SPARK-Station 2 (Sun Microsystems). The first row shows experiments using grammar $\boldsymbol{A}$ in Appendix A which does not have semantic conditions. The second row shows experiments using grammar $\boldsymbol{B}$ in Appendix B which has semantic conditions. In order to investigate the effects of semantic conditions, both of the grammar rules were translated into logic programs as described in Section V-A.

The first column shows CPU times by the following goal clause which executes a top–down parsing from a starting

symbol S:

$$
\begin{aligned}
?-\ & cd42(CT) \\
& T0\ is\ cputime, \\
& ss(S, CT, [\,], [ground(1)], \_), \\
& T\ is\ cputime - T0.
\end{aligned} \tag{27}
$$

As the starting symbol S does not have information of how to parse the circuit, the goal tries all nonterminal symbols as a starting symbol by backtracking until no element remains unknown (Section III-C). The fourth argument of the predicate "$ss(S, \cdots)$" is used as an input of semantic information and substituted by "$[ground(1)]$" which states the node 1 is grounded. Grammar $\boldsymbol{B}$ uses this information, but grammar $\boldsymbol{A}$ does not. The variables T and $T0$ are introduced to measure CPU times of parsing.

The second column shows that CPU times of top–down parsing form a specific starting symbol as:

$$
\begin{aligned}
?-\ & cd42(CT), \\
& T0\ is\ cputime, \\
& ss(opAmp(X, \_, \_, \_, \_, \_, \_), CT, [\,], [ground(1)], \_), \\
& T\ is\ cputime - T0.
\end{aligned} \tag{28}
$$

As the starting symbol "$opAmp(X, \cdots)$" is given, the clause efficiently parses the circuit $cd42$. The semantic information improves parsing efficiency in each case of the goal clauses (27) and (28).

The parsing process for DCSG can be viewed as a kind of top–down depth-first goal search based on backtracking. The search space increases combinatorially according to the number of grammar rules with the same left-hand sides and the same kinds of elements from the object circuits. Semantic information is more important in parsing larger circuits for early pruning of incorrect paths. This situation becomes clearer in the following experiments.

Table II shows the CPU times for parsing the circuit $cd67$ in Fig. 17. The additional grammar rules C and D used for parsing $cd67$ are shown in Appendix C and Appendix D. Grammar C does not have semantic conditions but grammar D does. The first row shows experiments without semantic conditions and the second row shows experiments with semantic conditions. The first column shows the CPU times for the goal (27a) which executes top–down parsing from the starting symbol S. The second column shows the CPU times for the goal (28a) which has the specific starting symbol "$phaseLockedLoop(\cdots)$":

$$
\begin{aligned}
?-\ & cd67(CT), \\
& T0\ is\ cputime, \\
& ss(S, CT, [\,], [ground(0)], \_), \\
& T\ is\ cputime - T0.
\end{aligned} \tag{27a}
$$

$$
\begin{aligned}
?-\ & cd67(CT), \\
& T0\ is\ cputime, \\
& ss(phaseLockedLoop(X, \_, \_, \_, \_, \_, \_), \\
& \quad CT, [\,], [ground(0)], \_), \\
& T\ is\ cputime - T0.
\end{aligned} \tag{28a}
$$

TABLE II
PARSING SPEEDS FOR CIRCUIT $cd67$

grammar	goal (27a)	goal (28a)
$A \cup C$	501.250 s	2.683 s
$B \cup D$	62.633 s	0.550 s

The effect of the use of semantic information on parsing speed is about a factor of 2 for the circuit in the Table I, but the effect increases to factors of 5 and 8 for the experiments in Table II. Semantic information effectively suppresses search space growth due to increasing numbers of grammar rules and circuit elements. The effect of the use of specific starting symbol on parsing speed increases to more than a factor of 100 in Table II, while it is less than a factor of 10 in Table I. Namely, when we parse larger circuits, the existence of a specific starting symbol becomes more important, since the symbol directs how to read the circuit. A parse tree obtained by the goal clause (28a) is shown in Fig. 18.

VII. CONCLUSIONS

We have viewed electronic circuits as a kind of language, and developed methods for parsing electronic circuits in a logic grammar. Knowledge of circuit structures as functional blocks was coded into grammar rules. The performance of our system depends on the defined grammar rules. As more grammar rules are defined, more circuits can be parsed. As opposed to ordinary circuit analysis based on circuit theory, our system cannot analyze circuits which consist of arbitrary connected circuit elements. As is true for natural language, we cannot understand structures not included in the grammar. That is, all circuits which can be parsed are sentences defined by given circuit grammars. If an object circuit has unknown structures, our system cannot parse the whole circuit, but can separate the object circuit into known parts and others.

Circuit grammars are defined for bipolar analog integrated circuits in [15]. This reference work contains 101 circuits, 90% of which have less than 14 transistors. Each circuit has a specific function as a standard building block for IC's. Each circuit can also be considered as a hierarchical composition of subfunctional blocks such as current sources and emitter-coupled pairs. These hierarchically organized circuits can be defined by grammar rules in DCSG.

Our method is useful for IC's, which have definite design rules rather than circuits of arbitrary connected discrete elements. When we design circuits using discrete elements, we use many variations derived from basic circuits. Grammar rules which directly analyze all of these variations may not be provided. In fact, our approach is to identify those circuits by rewriting and simplifying into equivalent circuits. We will discuss the problem of rewriting circuits in another paper.

The objective of our paper is to build a model of understanding electronic circuits When a circuit schematic is given, say, for trouble shooting, an engineer will first read the circuit schematic to understand how circuits are organized. Only then will he try to pursue the causality of electrical events to understand how they work. If the first stage of understanding

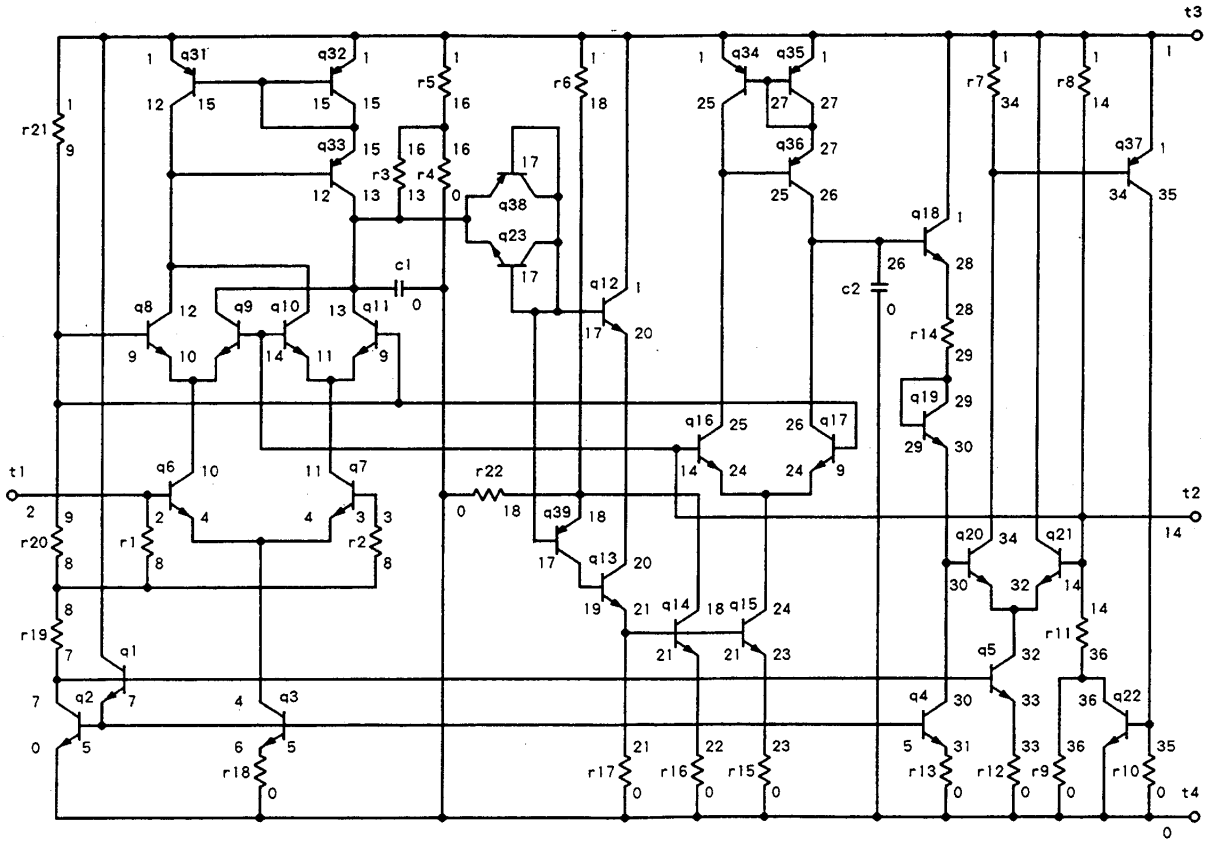


Fig. 17. Phase-locked loop *cd67*.

the circuit is not attained, the second stage becomes very difficult. Usually, a change of voltages or currents in the input terminal causes changes of voltages or currents in connected elements. These changes cause other changes of adjacent elements and finally cause changes of the output terminal. Thus these changes in a circuit are connected by many causal chains. However, only one of those causal chains is important and directly contributes to the circuit function. When we handle large hierarchically organized circuits, finding this causal chain is almost impossible unless we have knowledge of the circuit structures involved. This paper represents this first stage in understanding electronic circuits.

Our second goal was to understand circuit behaviors using information obtained by parsing circuit structures. As a first step, we introduced semantic terms into grammar rules, so that we could use electrical conditions in parsing. So far, we have only used the semantic terms to increase parsing efficiency and to help parsing context dependent circuits. The semantic terms in grammar rules can also be used to define relations between circuit structures and their electrical constraints which directly concern their functions. These electrical constraints on object circuits are incrementally obtained as semantic information each time a functional block is identified in parsing. When the whole circuit is parsed, the semantic information contains a set of electrical constraints which contributes to the final goal of the circuit. Using this set of electrical constants, we will be able to find the single causal chain which constitutes the circuit's function. This, we hope, will provide the basis for our next step in this research, which will be the formalization of circuit behavior knowledge as meanings of syntactic structures.

Appendix

A. Grammar Rules for Parsing Circuit *cd42*

$$dtr(ndtr(Q), A, C) \longrightarrow [npnTr(Q, A, C, A)].$$
$$dtr(pdtr(Q), A, C) \longrightarrow [pnpTr(Q, C, A, C)].$$

$$res(X, A, B) \longrightarrow [resistor(X, A, B)]\ [resistor(X, B, A)].$$

$$* \quad vbeReg(vreg(D, R), Vp, Com, Out) \longrightarrow . \\ dtr(D, Out, Com), \\ res(R, Vp, Out).$$

$$cSink(sink(VR, Q), In, Com) \longrightarrow \\ vbeReg(VR, \_, Com, B), \\ [npnTr(Q, B, \mathrm{Com}, \mathrm{In})].$$

$$* \quad cSink(sink(VR, Q), In, Com) \longrightarrow \\ test\ vbeReg(VR, \_, Com, B), \\ [npnTr(Q, B, Com, In)].$$

$$* \quad eCoupledPair(ecup(Q1, Q2), \\ B1, B2, C1, C2) \longrightarrow \\ [npnTr(Q1, B1, E, C1)], \\ [npnTr(Q2, B2, E, C2)].$$
$$activeLoad(al(D, Q), Ref, Vp, Ld) \longrightarrow$$

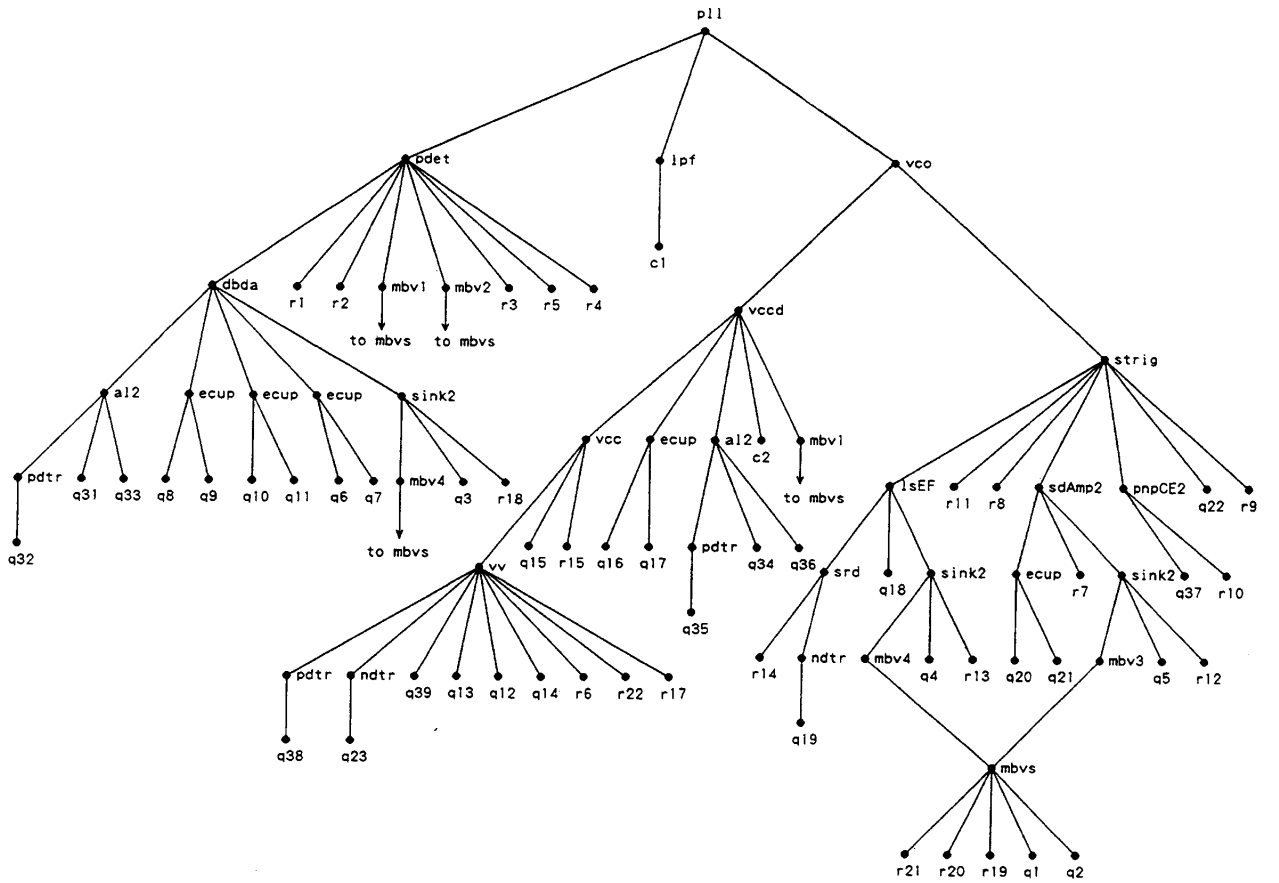


Fig. 18. Parse tree for circuit *cd*67.

$$dtr(D, Vp, Ref),$$
$$[pnpTr(Q, Ref, Vp, Ld)].$$

$$sDiffAmp(sdAmp(EC, AL, CS)$$
$$B1, B2, C1, Vp, Vm) \longrightarrow$$
$$eCoupledPair(EC, B1, B2, E, C1, C2),$$
$$activeLoad(AL, C2, Vp, C1),$$
$$csink(SC, E, Vm).$$

$$* \quad commonEmitter(pnpCE(Q, CS),$$
$$In, Out, Vp, Vm) \longrightarrow$$
$$[pnpTR(Q, In, Vp, Out)],$$
$$cSink(CS, Out, Vm).$$

$$emitterFollower(npnEF(Q, R), In, Out, Vp, Vm) \longrightarrow$$
$$[npnTr(Q, In, Out, Vp)],$$
$$res(R, Out, Vm).$$

$$opAmp(opAmp1(DA, CE, EF),$$
$$In1, In2, Out, Vp, Vm) \longrightarrow$$
$$sDiffAmp(DA, In1, In2, O1, Vp, Vm),$$
$$commonEmitter(CE, O1, O2, Vp, Vm),$$
$$emitterFollower(EF, O2, Out, Vp, Vm),$$
$$[terminal(T1, In1)],$$
$$[terminal(T2, In2)],$$
$$[terminal(T3, Out)],$$
$$[terminal(T4, Vp)],$$
$$[terminal(T5, Vm)].$$

*B. Grammar Rules with Semantic Conditions for Parsing Circuit cd*42

Only the rules different from corresponding rules in A are listed. To make grammar B, replace the rules marked with a $*$ symbol in grammar A with the following rules.

$$vbeReg(vreg(D, R)Vp, Com, Out) \longrightarrow$$
$$dtr(D, Out, Com),$$
$$res(R, Vp, Out),$$
$$\{add\ [vbeControlled(vreg(D, R)Out, Com)]\},$$

$$cSink(sink(VR, Q), In, Com) \longrightarrow$$
$$\{[vbeControlled(VR, B, Com)]\},$$
$$[npnTr(Q, B, Com, In)].$$

$$eCoupledPair(ecup(Q1, Q2),$$
$$B1, B2, E, C1, C2) \longrightarrow$$
$$[npnTr(Q1, B1, E, C1)],$$
$$[npnTr(Q2, B2, E, C2)],$$
$$\{not\ powerNode(E)\}.$$

$$emitterFollower(npnEF(Q, R),$$
$$In, Out, Vp, Vm) \longrightarrow$$
$$[npnTr(Q, In, Out, Vp)],$$

$res(R, Out, Vm),$
$\{not\ powerNode(Out)\}.$

$powerNode(N) \longrightarrow [ground(N)];$
$[positivePowerNode(N)];$
$[negativePowerNode(N)].$

C. Additional Grammar Rules for Parsing Circuit cd67

$cap(X, A, B) \longrightarrow [capacitor(X, A, B)];$
$[capacitor(X, B, A)].$

$low - passFilter(lpf(X), A, B) \longrightarrow cap(X, A, B).$

$cSink(sink2(VS, Q, R), In, Com) \longrightarrow$
$[npnTr(Q, B, E, In)],$
$res(R, E, Com),$
$vSource(VS, B, Com).$

$activeLoad(al2(D, Q1, Q2), Ref, Vp, Ld) \longrightarrow$
$dtr(D, Vp, B),$
$[pnpTr(Q1, B, Vp, Ref)],$
$[pnpTr(Q2, Ref, B, Ld)].$

$sDiffAmp(sdAmp2(EC, R, CS)$
$B1, B2, C1, Vp, Vm) \longrightarrow$
$eCoupledPair(EC, B1, B2, E, C1, Vp),$
$res(R, Vp, C1),$
$cSink(CS, E, Vm).$

$commonEmitter(pnpCE2(Q, R),$
$In, Ot, Vp, Vm) \longrightarrow$
$[pnpTr(Q, In, Vp, Ot)],$
$res(R, Ot, Vm).$

$levelshiftEF(lsEF(Q, LS, CS), \mathrm{In}, Ot, Vp, Vm) \longrightarrow$
$[npnTr(Q, In, E, Vp)],$
$seriesRD(LS, E, Ot),$
$cSink(SC, Ot, Vm).$

$dBalanceDA(dbda(AL, Ec1, Ec2, Ec, CS)$
$B1, B2, D1, D2, C2, Vp, Vm) \longrightarrow$
$activeLoad(AL, C1, Vp, C2),$
$eCoupledPair(Ec1, D1, D2, E1, C1, C2),$
$eCoupledPair(Ec2, D2, D1, E2, C1, C2),$
$eCoupledPair(Ec, B1, B2, E, E1, E2),$
$cSink(CS, E, Vm).$

* $multiBiasVsource(mvbs(R1, E2, R3, Q1, Q2),$
$Vp, V1, V2, V3, V4, Vm) \longrightarrow$
$[npnTr(Q1, V3, V4, Vp)],$
$[npnTr(Q2, V4, Vm, V3)],$
$res(R3, V2, V3),$
$res(R2, V1, V2),$
$res(R1, Vp, V1).$

* $vSource(VS, V, Com) \longrightarrow mvSource(VS, V, Com).$

* $vSource(VS, V, Com) \longrightarrow test\ mvSource(VS, V, Com).$

$mvSource(mbv1(MB), V, Com) \longrightarrow$
$multiBiasVsource(MB, \_ V, \_ , \_ , \_ , Com).$
$mvSource(mbv2(MB), V, Com) \longrightarrow$
$multiBiasVsource(MB, \_ , \_ V, \_ , \_ , Com).$
$mvSource(mbv3(MB), V, Com) \longrightarrow$
$multiBiasVsource(MB, \_ , \_ , \_ V, \_ , Com).$
$mvSource(mbv4(MB), V, Com) \longrightarrow$
$multiBiasVsource(MB, \_ , \_ , \_ , \_ , V, Com).$

$phaseDetector(pdet(DA, R1, R2, VS1, VS2, RL, R3,$
$R4), In1, In2, Ot, Vp, Vm) \longrightarrow$
$dBalanceDA(DA, In1, B2, BB1, In2, Ot, Vp, Vm),$
$res(R1, \mathrm{In1}, V2),$
$res(R2, B2, V2),$
$vSource(VS1, BB1, Vm),$
$vSource(VS2, V2, Vm),$
$res(RL, Ot, V3),$
$res(R3, Vp, V3),$
$res(R4, V3, Vm).$

$voltageCurrentConverter(vcc(VV, Q, R),$
$In, Sink, Vp, Vm) \longrightarrow$
$vvConverter(VV, In, B, Vp, Vm),$
$[npnTr(Q, B, E, Sink)],$
$res(R, E, Vm).$

$vvConverter(vv(D1, D2, Q1, Q2, Q3, Q4, R1, R2, R3),$
$In, Ot, Vp, Vm) \longrightarrow$
$dtr(D1, In, B),$
$dtr(D2, B, In),$
$[pnpTr(Q1, B, E, C)],$
$[npnTr(Q2, C, Ot, A)],$
$[npnTr(Q3, B, A, Vp)],$
$[npnTr(Q4, Ot, D, E)],$
$res(R1, Vp, E),$
$res(R2, E, Vm),$
$res(R3, Ot, Vm),$
$res(R4, D, Vm)$

$vctrlChargeDischarge(vccd(VC, EC, AL, C, VS),$
$I1, I2, Ot, Vp, Vm) \longrightarrow$
$voltageCurrentConverter(VC, I1, E, Vp, Vm),$
$eCoupledPair(EC, I2, B2, E, C1, Ot),$
$activeLoad(AL, C1, Vp, Ot),$
$cap(C, Ot, Vm),$
$vSource(VS, B2, Vm).$

$schmitTrig(strig(EF, DA, CE, Q, R1, R2, R3),$
$In, Ot, Vp, Vm) \longrightarrow$
$sDiffAmp(DA, B1, Ot, C1, Vp, Vm),$
$levelshiftEF(EF, In, B1, Vp, Vm),$
$commonEmitter(CE, C1, C2, Vp, Vm),$
$[npnTr(Q, C2, Vm, C3)],$
$res(R1, C3, Vm),$

$res(R2, Ot, C3),$
$res(R3, Vp, Ot).$

$voltageControlledOsc(vco(CD, TRIG),$
$In, Ot, Vp, Vm) \longrightarrow$
$vctrlChargeDischarge(CD, In, Ot, O1, Vp, Vm),$
$schmitTrig(TRIG, O1, Ot, Vp, Vm)$

$phaseLockedLoop(pll(PD, LPF, VCO),$
$In, Ot, Vp, Vm) \longrightarrow$
$phaseDetector(PD, In, Ot, O1, Vp, Vm),$
$low - passFilter(LPF, O1, Vm),$
$voltageControlledOsc(VCO, O1, Ot, Vp, Vm),$
$[terminal(T1, In)],$
$[terminal(T2, Ot)],$
$[terminal(T3, Vp)],$
$[terminal(T4, Vm)].$

$seriesRD(X, A, C) \longrightarrow resDtr(X, A, C).$
* $seriesRD(srd(X, Y), A, C) \longrightarrow resDtr(X, A, B),$
$seriesRD(Y, B, C).$

$resDtr(X, A, C) \longrightarrow res(X, A, C);$
$dtr(X, A, C).$

D. Additional Grammar Rules with Semantic Conditions for Parsing Circuit cd67

Only the rules different from corresponding rules in C are listed. To make grammar D, replace the rules marked with a * symbol in grammar C with the following rules.

$MultiBiasVsource(MB, Vp, V1, V2, V3, V4, Vm) \longrightarrow$
$[npnTr(Q1, V3, V4, Vp)],$
$[npnTr(Q2, V4, Vm, V3)],$
$res(R3, V2, V3),$
$res(R2, V1, V2),$
$res(R1, Vp, V1),$
$"MB = mvbs(R1, E2, R3, Q1, Q2)",$
$\{add[vControlled(mbv1(MB), V1, Vm)]\},$
$\{add[vControlled(mbv2(MB), V2, Vm)]\},$
$\{add[vControlled(mbv3(MB), V3, Vm)]\},$
$\{add[vControlled(mbv4(MB), V4, Vm)]\}.$

$vSource(VS, V, \text{Com}) \longrightarrow$
$\{[vControlled(VS, V, Com)]\}.$
$vSource(VS, V, Com) \longrightarrow mvSource(VS, V, Com).$

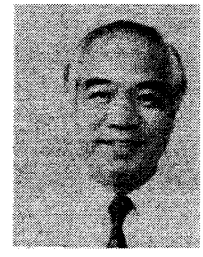
$seriesRD(srd(X, Y), A, C) \longrightarrow resDtr(X, A, B),$
$\{not\ powerNode\ (B)\},$
$seriesRD(Y, B, C).$

ACKNOWLEDGMENT

This work was started by using a deductive system called Duck. The author would like to thank Prof. Drew McDermott for Duck. He would like to extend special thanks to David J. Littleboy for technical discussions and advice.

REFERENCES

[1] H. Abramson and V. Dahl, *Logic Grammars.* New York: Springer-Verlag, 1989.
[2] J. De Kleer, "Causal and teleological reasoning in circuit recognition," Tech. Rep. 529, Artificial Intelligence Lab., MIT, 1979.
[3] ——, "How circuits work," *Artif. Intell.,* vol. 24, pp. 205–280, 1984.
[4] K. S. Fu, *Syntactic Methods in Pattern Recognition.* New York: Academic, 1974.
[5] ——, *Syntactic Pattern Recognition and Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1982.
[6] S. A. Greibach, "A new normal form theorem for context-free phrase structure grammars," *J. ACM,* vol. 12, pp. 42–52, 1965.
[7] F. C. N. Pereira and D. H. D. Warren, "Definite clause grammars for language analysis," *Artif. Intell.,* vol. 13, pp. 231–278, 1980.
[8] F. C. N. Pereira and C. Tweed, *C-Prolog User's Manual,* Edinburgh Computer Aided Architectural Design, 1987.
[9] G. J. Sussman and G. Steele Jr., "Constraints: A language for expressing almost-hierarchical descriptions," *Artif. Intell.,* vol. 14, pp. 1–39, 1980.
[10] T. Tanaka, "Representation and analysis of electrical circuits in a deductive system," in *Proc. IJCAI-83,* Karlsruhe, 1983, pp. 263–267.
[11] T. Tanaka, "Parsing circuit topology in a deductive system," in *Proc. IJCAI-85,* Los Angeles, CA, 1985, pp. 407–410.
[12] T. Tanaka, "Structural analysis of electronic circuits in a deductive system," in *Expert System Applications,* L. Bolc and M. J. Coombs, eds. Germany: Springer-Velag, 1988, pp. 257–308.
[13] T. Tanaka, "Definite clause set grammars: A fermalism for problem solving," *J. Logic Program.,* vol. 10, pp. 1–17, 1991.
[14] P. W. Tuinenga, *SPICE—A Guide to Circuit Simulation & Analysis Using PSpice.* Englewood Cliffs, NJ: Prentice-Hall, 1988.
[15] *101 Analog IC Designs,* Sunnyvale, CA: Interdesign Inc., 1976.



Takushi Tanaka was born in Fukuoka, Japan, on January 7, 1944. He received the B. Eng., M.Eng., and Dr. Eng. degree from Kyushu University, Fukuoka, Japan, in 1967, 1969, and 1987, respectively. From 1982 to 1983, he was a Post-Doctoral Fellow with the Department of Computer Science, Yale University, where he was involved with the AI project.

From 1976 to 1988, he worked on understanding natural language as a Senior Researcher at The National Language Research Institute, Tokyo, Japan. He is currently a Professor with the Department of Computer Science, Fukuoka Institute of Technology. His research interests include language understanding, qualitative reasoning, knowledge representation, logic programming, electronic circuits, and computer networks.

Dr. Tanaka is a member of AAAI, JSAI, IPSJ, and IEICE.