In: Circuit Analysis
Editor: Virginia E. Wright
ISBN: 978-1-61728-106-8

Chapter 5

ANALYZING CIRCUIT STRUCTURES AS LANGUAGE

Takushi Tanaka[*]
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan

Abstract

As a step toward automatic circuit understanding, we present a new method for analyzing circuit structures. We view circuits as sentences, and their elements as words. The electrical behavior and functions are the meaning of the sentences. Circuit structures are defined by a logic grammar called DCSG. A set of grammar rules, when converted into Prolog clauses, forms a logic program which perform top-down parsing. When an unknown circuit is given, this logic program will analyze the circuit and derive a parse tree for the circuit.

We first present the basic concepts of the logic programming using examples of circuits, then introduce the logic grammar DCSG (Definite Clause Set Grammar) which was developed for word-order free languages. Circuits are represented as sentences in the language. Circuit structures are defined as grammar rules for functional blocks composing bipolar analog ICs. A given circuit is parsed as a grammatical sentence,

[*]Email address: tanaka@fit.ac.jp

and its hierarchical structure of functional blocks is derived. An extension to DCSG uses additional fields to hold semantic terms. Using the fields, electrical behavior and functions can be defined for the syntactic structures. After a circuit is parsed, not only its syntactic structure, but also its electrical behavior and functions can be derived as the meaning of the circuit structure.

Keywords: electronic circuit, circuit analysis, circuit structure, circuit grammar, logic grammar, knowledge representation, logic programming, DCG, DCSG

1. Introduction

When an engineer first looks at a circuit schematic, he tries to partition the circuit into familiar sub-circuits with known goals. He then tries to trace the causality of electrical events through those sub-circuits to determine if and how the overall goal of the circuit is achieved. 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 the hierarchical structure of its functional blocks and rediscovering the designer's original intentions.

As a step toward automatic circuit understanding, we present a new method for analyzing circuit structures. We view circuits as sentences, and their elements as words. The electrical behavior and functions are the meaning of the sentences. Circuit structures are defined by a logic grammar called DCSG[5]. A set of grammar rules, when converted into Prolog clauses, forms a logic program which perform top-down parsing when executed.

When an unknown circuit is given, this logic program will analyze the circuit and derive a parse tree for the circuit. This contrasts with earlier circuit analysis programs based on circuit theory such as SPICE[12], which function as a circuit simulators to derive voltages and currents from the circuit. Since the parse tree shows a hierarchical structure of functional blocks composing the circuit, it can help an engineer to identify which elements contribute to which sub-functions and how the total function of the circuit is achieved by its sub-functions for trouble shooting, redesigning, or modifying the circuit. That is, a circuit parser could advise engineers how to interpret circuit structures.

Probably, most readers of this book are not familiar with logic programming, so we first present the basic concepts of the logic programming using examples of circuits, then introduce the logic grammar DCSG (Definite Clause Set Grammar) which was developed for word-order free languages. Circuits are represented as sentences in the language and circuit structures are defined as grammar rules. Several examples show advantages of using DCSG.

As an extension to DCSG, new circuit grammar[9] uses additional fields to hold semantic terms. Using the fields, electrical behavior and functions can be defined for the syntactic structures. As an example, we will define grammar rules for functional blocks consisting bipolar analog ICs[13]. After a circuit is parsed as a grammatical sentence, not only its syntactic structure, but also its electrical behavior and functions can be derived as the meaning of the circuit structure.

2. Circuit Representation in Predicate Logic

2.1. Prolog Language

Prolog is a programming language based on predicate logic. It is well suited for symbolic computations that handle problems concerning objects and their relations. Programs written in ordinary programming language define procedures to solve problems, while Prolog programs define problems themselves and relationships between objects. Prolog programs are executed by a mechanism of theorem proving. Solving problems using the mechanism of theorem proving, instead of defining the procedure explicitly, is called logic programming.

Using electrical circuits as examples, we introduce the basic concepts of logic programming. We first show how circuits are represented in terms of predicate logic. A given circuit is defined by a set of Prolog facts. Circuit structures are defined by Prolog rules. Finding structures in the given circuit is realized by Prolog goals. The examples presented here also show the problems and limitations of this method.

2.2. Facts

All objects in circuits are represented by logical nouns called terms, and all relationships between those objects are represented by logical predicates. The terms and the predicates are combined to form atomic formulas, which are logical sentences. The following seven atomic formulas represent the circuit $ca40$ shown in Figure 1.

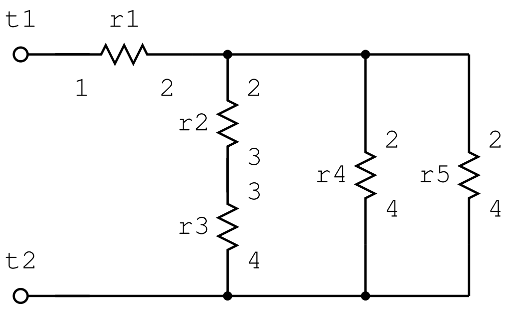

Figure 1. Circuit ca40

$resistor(r1, 1, 2).$
$resistor(r2, 2, 3).$
$resistor(r3, 3, 4).$
$resistor(r4, 2, 4).$
$resistor(r5, 2, 4).$
$terminal(t1, 1).$
$terminal(t2, 4).$

The atomic formula "$resistor(r1, 1, 2)$" consists of a predicate symbol "$resistor(...)$" and three constant terms "$r1$","1" and "2". The atomic formula states that the resistor named $r1$ is connected to node 1 and node 2. The atomic formula "$terminal(t1, 1)$" states that the external terminal named $t1$ is connected to node 1. These seven logical sentences describe whole circuit topology of $ca40$. When we consider the circuit in Figure 1, these sentences are called "facts", and placed in the Prolog database, which holds true sentences.

2.3. Goal

Since we are considering the circuit $ca40$ in Figure 1, all the formulas that represent the circuit are already placed in the Prolog database. The following atomic formula prefixed by "$?-$" is called a goal clause, which asks the Prolog system whether the resistor $r1$ is connected to the nodes 1 and 2. Here, "$?-$" is a prompt for goal clause generated by the Prolog system. When the goal clause is given, the Prolog system looks for the goal in the Prolog database. Since $resistor(r1, 1, 2)$ is in the database, the goal becomes $true$, and the Prolog system answers yes.

$$?-\ resistor(r1, 1, 2).$$
$$yes$$

We can find the nodes to which the resistor "$r2$" is connected by the following goal clause with variables A and B. These variables are also terms which construct atomic formulas. Here, the variables are represented by a string beginning with an upper case letter.

$$?-\ resistor(r2, A, B).$$

The variables A and B in the goal clause are assumed to be bound by existential quantifiers in predicate logic. Namely, the Prolog system is asked to prove the sentence

$$(\exists A)(\exists B)resistor(r2, A, B)$$

which states that "there exist nodes A and B connected to the resistor $r2$". The Prolog system looks in the database and finds $resistor(r2, 2, 3)$ as an instance for which $resistor(r2, A, B)$ becomes $true$, and the system outputs the variable-value bindings as:

$$A = 2$$
$$B = 3$$
$$yes$$

These variable-value bindings are made by a mechanism called unification. Unification discovers a substitution of variables for the two formulas $resistor(r2, 2, 3)$ and $resistor(r2, A, B)$ that makes them equal.

The following goal clause finds a resistor "X" connected to the nodes 3 and 4 as:

$$\begin{array}{l} ?-\ resistor(X, 3, 4). \\ X = r3 \\ yes \end{array}$$

Note, however, that if the node order is reversed, the goal fails to find the resistor connected to nodes 4 and 3:

$$\begin{array}{l} ?-\ resistor(X, 4, 3). \\ no \end{array}$$

Since resistors are non-polar elements, we want to refer to resistors regardless of their node order. This can be done by defining a new predicate using rules in Prolog.

2.4. Rules

A rule is a conditional sentence called a definite clause, which is also placed in the Prolog database. A rule consists of a left-hand side called the head, the special symbol "$:-$", and a right-hand side called the body. The head consists of an atomic formula which is the result of the conditional sentence. The symbol "$:-$" is the logical connective of implication "$\leftarrow$", although the order of the condition and result are reversed. The body, which is the conditional part, consists of atomic formulas. Facts can be viewed as a special case of rules which do not have conditions.

The following rule defines the new predicate "$res(R, A, B)$" which can refer either $resistor(R, A, B)$ or $resistor(R, B, A)$. Here, the symbol ";" works as a logical connective "or". This rule can be read if $resistor(R, A, B)$ or $resistor(R, B, A)$ is $true$, $res(R, A, B)$ becomes $true$. Procedurally, in order to show that $res(R, A, B)$ is $true$, the Prolog system tries to show that either $resistor(R, A, B)$ or $resistor(R, B, A)$ is $true$.

$$\begin{aligned} res(R,A,B)\ :-\ & resistor(R,A,B);\\ & resistor(R,B,A). \end{aligned}$$

The variables R, A, and B in facts and rules are assumed to be bound by universal quantifiers in predicate logic. Namely, this rule means the following conditional sentence.

$$(\forall R)(\forall A)(\forall B)(resistor(R,A,B) \vee resistor(R,B,A) \rightarrow res(R,A,B)).$$

When the following goal is given, the goal is unified with the head of this rule, and the body of the rule becomes a new goal.

$$?-\ res(X,4,3).$$

The body generates the following disjunction as the new goal.

$$?-\ resistor(X,4,3);\ resistor(X,3,4).$$

The first sub-goal of disjunction fails because there is no fact that can be unified with $resistor(X,4,3)$ in the database, but the second sub-goal $resistor(X,3,4)$ succeeds, and outputs:

$$\begin{aligned} & X = r3\\ & yes \end{aligned}$$

Similar rules can be defined for other non-polar elements such as capacitors and inductors:

$$\begin{aligned} cap(C,A,B)\ :-\ & capacitor(C,A,B);\\ & capacitor(C,B,A). \end{aligned}$$

$$\begin{aligned} ind(L,A,B)\ :-\ & inductor(L,A,B);\\ & inductor(L,B,A). \end{aligned}$$

The following rule defines the predicate "$anyElm(X,A)$" which refers to any element X connected to node A. Here, we assume the external terminal "$terminal(X,A)$" to be a kind of any element. The underscores "_" in atomic formulas are anonymous variables which are not of concern in the rule.

$$\begin{aligned} anyElm(X,A)\ :-\ & terminal(X,A);\\ & res(X,A,\_\,);\\ & cap(X,A,\_\,);\\ & ind(X,A,\_\,). \end{aligned}$$

2.5. Predicates for Circuit Structures

In order to find resistors connected in series (Figure 2) in the circuit $ca40$, we attempt to satisfy the following conjunctive goal. Here, "," between two atomic formulas works as the logical connective "and".

$$?-\ res(X,A,B),\ res(Y,B,C).$$

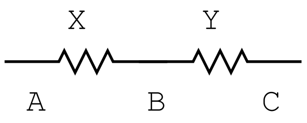

Figure 2. Resistors connected in series

This conjunctive goal successfully finds resistors connected in series. The first sub-goal $res(X,A,B)$ finds the resistor $r2$ and its connecting nodes 2 and 3, then the second sub-goal $res(Y,B,C)$ finds the resistor $r3$ connecting node 3 as follows:

$A = 2$
$B = 3$
$C = 4$
$X = r2$
$Y = r3$;

The ";" after "$Y = r3$" is typed by the user and directs the Prolog system to find another answer. The Prolog system discard the first answer, and tries to find another answer using the Prolog backtracking mechanism. The conjunctive goal also outputs unexpected answers such as:

$A = 1$

$B = 2$
$C = 1$
$X = r1$
$Y = r1$

In the series circuit shown in Figure 2, neither elements X and Y nor nodes A and C may be the same. Since the condition "$not\ A = C$" topologically includes the condition "$not\ X = Y$", we next try proving the following goal:

$$?-\ res(X, A, B),\ res(Y, B, C),\ not\ A = C.$$

The Prolog system ceases to output undesired answers of the above type, but it still outputs another type of unexpected answers as follows.

$A = 1$
$B = 2$
$C = 3$
$X = r1$
$Y = r2$

No element other than X and Y may be connected to the central node B of the series circuit. This can be expressed by first defining an element Z other than X and Y connected to B as:

$$\begin{aligned} otherElm(Z, B, X, Y) :-\ & anyElm(Z, B), \\ & not\ Z = X, \\ & not\ Z = Y. \end{aligned}$$

Now, we can define the new predicate $rSeries(sr(X, Y), A, C)$ for series circuit of resistors by adding the conditions "$not\ A = C$" and "$not\ otherElm(\_, B, X, Y)$" to reject undesired answers.

$$\begin{aligned} rSeries(sr(X, Y), A, C) :-\ & res(X, A, B), \\ & res(Y, B, C), \\ & not\ A = C, \\ & not\ otherElm(\_, B, X, Y). \end{aligned}$$

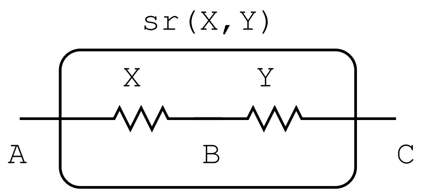

Figure 3. Series connection of resistors

The first argument "$sr(X, Y)$" of the predicate $rSeries(...)$ is the name given to the series circuit of resistors X and Y (Figure 3). The name is a form of compound term made of the function symbol $sr(...)$ and variables X and Y. Namely, the name is given by depending on the value of X and Y. This is a technique to give a unique name to a new object, and the function is called a Skolem function.

The predicate for parallel connections is also defined in the same manner:

$$\begin{aligned} rParallel(pr(X, Y), A, C) :- \; & res(X, A, B), \\ & res(Y, A, B), \\ & not\; X = Y. \end{aligned}$$

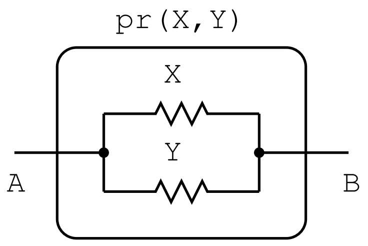

Figure 4. Parallel connection of resistors

2.6. Difficulties in Circuit Representation Using Predicate

First we defined terms and predicates for circuit elements, and then we defined predicates for abstract elements and circuit structures. If we could define predicates for all concepts in circuits in the same manner, we would have an axiomatic system of circuits which can automatically derive true sentences on circuits.

However, our circuit representation faces difficulties when we try to define predicates for relationships between two circuits such as equivalent circuits. In order to refer to relationships between circuits, we need a mechanism to identify a set of facts for a specific circuit.

One method is to introduce a new argument for circuit identification into each circuit predicate as follows:

$resistor(ca40, r1, 1, 2).$
$resistor(ca40, r2, 2, 3).$
$resistor(ca40, r3, 3, 4).$
$resistor(ca40, r4, 2, 4).$
$resistor(ca40, r5, 2, 4).$
$terminal(ca40, t1, 1).$
$terminal(ca40, t2, 4).$

The first argument $ca40$ of each predicate indicates that each element belongs to the circuit $ca40$.

Although we can refer to relationships between two circuits using identification, we have another problem, namely rewriting circuits. In circuit analysis, we often rewrite circuits into equivalent circuits. Rewriting facts in the Prolog database can be done using the built-in predicates "$assert$" and "$retract$". However using $retract$ is problematical from a logical point of view, because using $retract$ means erasing facts which were given as true, and the erased facts can never be used again. In the following sections, we resolve these problems by assuming circuits to be a kind of formal language.

2.7. Changing Circuit Representation

We have already introduced two kinds of terms, constants and variables, to represent objects in circuits. These terms are viewed as nouns to make logical sentences. Predicate logic has another kind of term, called a compound term, which consists of a function symbol and other terms. In the previous sections, the atomic formula $resistor(r1, 1, 2)$ was a logical sentence which states that "$r1$" is a resistor connected to node 1 and node 2. Here, the $resistor(...)$

worked as a predicate symbol. As predicates and functions are the same in style, we will change $resistor(...)$ from a predicate symbol to a function symbol. As a result, $resistor(r1, 1, 2)$ becomes a compound term instead of an atomic formula. Unlike mathematics, in logic, functions are not computed. The compound term works as a noun phrase, while the atomic formula works as a simple sentence. An atomic formula states a relationship between objects while a compound term represents a new object using other objects. Therefore, the compound term $resistor(r1, 1, 2)$ can be read "resistor $r1$ connected to node 1 and 2" as though it were a noun phrase.

Prolog supports a special data structure called "list". A list is a sequence of any number of terms surrounded by "[" and "]". We can use a list as a new circuit representation for the circuit $ca40$ as follows:

$$\begin{aligned}&[resistor(r1,1,2),\ resistor(r2,2,3),\ resistor(r3,3,4),\\ &\ resistor(r4,2,4),,\ resistor(r5,2,4),\ terminal(t1,1),\\ &\ terminal(t2,4)].\end{aligned}$$

Here, the list is used to represent the set of all the elements that make up the circuit $ca40$. In the next section, we develop a mechanism called DCSG to treat these lists as a word-order free sentence.

2.8. Lists

Since we will use lists to represent circuits, we discuss lists further here. A list can be viewed as a compound term made by iteratively applying the special function ".". For example, the list $[a,\ b,\ c]$ consists of the function "$.(a,\ .(b,\ .(c, [\ ])))$". Here, a list without elements is called the empty list and is simply written as "[ ]". Using the empty list, the special function "$.(c, [\ ])$" makes the list "$[c]$". The function "$.(b,\ [c])$" makes the list "$[b,\ c]$". And the function "$.(a,\ [b,\ c])$ makes the list "$[a,\ b,\ c]$".

The first element of a list is called the head of the list. The remaining part is another list and is called the tail. That is, a list consists of "$.(Head,\ Tail)$". The special symbol "|" also separates a list into a head and a tail and composes a list as shown in the following goal clauses:

$$?-\ [a] = [Head \mid Tail].$$

$Head = a$
$Tail = [\,]$.

$?-\ [a,\ b,\ c] = [Head \mid Tail]$.
$Head = a$
$Tail = [b,\ c]$

$?-\ X = [a,\ b \mid [c,\ d]]$.
$X = [a,\ b,\ c,\ d]$

$?-\ X = [a \mid [b \mid [c \mid [\,]]]]$.
$X = [a,\ b,\ c]$

3. Logic Grammar DCSG

3.1. Word-Order Free Language

Most Prolog systems provide a mechanism for parsing context-free languages called DCG(Definite Clause Grammar)[3]. A set of the grammar rules, when converted into Prolog clauses, forms a logic program which executes top-down parsing. Here, we develop a logic grammar DCSG(Definite Clause Set Grammar)[5] for word-order free language similar to the method of DCG.

First, we will introduce the concept of word-order free language. A word-order free language **L(G')** is defined by modifying the definition of a formal grammar. We define a context-free word-order free grammar **G'** to be a quadruple $< V_N, V_T, P, S >$ where: V_N is a finite set of non-terminal symbols, V_T is a finite set of terminal symbols, P is a finite set of grammar rules of the form:

$$\begin{array}{ll} A \ \longrightarrow\ B_1, B_2, ..., B_n. & (n \geq 1) \\ A \in V_N, \quad B_i \in V_N \cup V_T & (i = 1, ..., n) \end{array}$$

and $S(\in V_N)$ is the starting symbol. The above grammar rule means that the symbol A is rewritten not with the string of symbols "$B_1, B_2, ..., B_n$", but with the set of symbols $\{B_1, B_2, ..., B_n\}$. A sentence in the language **L(G')** is a set of terminal symbols which is derived from S by successive application of grammar rules. Here the sentence is a multi-set which admits multiple occurrences of elements taken from V_T. Each non-terminal symbol used to derive a sentence can be viewed as a name given to a subset of the multi-set.

3.2. DCSG Conversion

When a set of grammar rules is given to a Prolog system, the DCG mechanism is used to convert the grammar rules into Prolog clauses. We now develop a conversion for word-order free languages that is analogous to DCG conversion[11]. The general form of the conversion procedure from a grammar rule

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

to a Prolog clause is:

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

Here, all symbols in the grammar rule are assumed to be non-terminal symbols. 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 conversion.

The arguments $S_0, S_1, ..., S_n$ in $(1)'$ are multisets of V_T, represented as lists of elements. 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.

The predicate "$member$" is defined by the Prolog clauses (2) and (3) below. It 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 argument.

$$member(M, [M|X], X). \tag{2}$$
$$member(M, [A|X], [A|Y]) :- \ member(M, X, Y). \tag{3}$$

When the clause (1)' is used in parsing, an object sentence (multiset of terminal symbols) is given to the argument S_0. In order to find the subset A in S_0, the first sub-goal finds the subset B_1 in S_0 then put the remainder into S_1, the next sub-goal finds B_2 in S_1 then put the remainder into S_2, ..., and the last sub-goal finds B_n in S_{n-1} then put the remainder into S_n. That is, when a grammar rule is used in parsing, each non-terminal symbol in the grammar rule makes a new set from the given set by removing itself as its subset. While, each terminal symbol used in the grammar rule also makes a new set from the given set by removing itself as its member.

DCSG uses the predicates $subset$ and $member$ to convert grammar rules into Prolog clauses, but the differences between DCG and DCSG are minimal. If we replace the predicate $subset$ with $substring$ and remove the clause (3) from the definition of $member$, the conversion will be equivalent to DCG conversion, although ordinary DCG does not use the predicate of $substring$ for simplification.

3.3. Backward Chaining and Top Down Parsing

Usually, a Prolog program consists of two kinds of definite clauses, called facts $\{F_1, F_2, ..., F_n\}$ and rules $\{R_1, R_2, ..., R_m\}$ both viewed as axioms. The execution of a Prolog program can be viewed as a process of deriving a theorem by backward chaining from the axioms. The top-down parsing of a word-order free sentence somewhat resembles the process of backward chaining. The object sentence is given as a set of terminal symbols $\{W_1, W_2, ..., W_n\}$. The starting symbol "S" is decomposed into terminal symbols using grammar rules $\{G_1, G_2, ..., G_m\}$ until they coincide with the given sentence. That is, the set of facts corresponds to the sentence, and the set of backward chaining rules corresponds to the set of grammar rules. Deriving theorems in backward chaining corresponds to identifying non-terminal symbols in the top-down method.

There is an important difference between backward chaining and top-down parsing. Backward chaining allows multiple use of the same fact to derive a theorem, while in a context-free language, each terminal symbol in a sentence contributes only once to the reduction of non-terminal symbols. This characteristic is very useful for avoiding a common looping problem in backward chaining, a problem which is caused by multiple use of the same fact.

3.4. The Looping Problem

When problems to be solved are formalized and expressed in Prolog, we often encounter a certain kind of looping problem. We will clarify a cause of this looping problem using voltage derivation as an example.

Assume that the voltages in a circuit are those given in Figure 5. We might consider representing this voltage data by the following facts:

$$\begin{aligned}&voltage(1,2,12).\\&voltage(3,2,10).\\&voltage(3,4,7).\end{aligned}$$

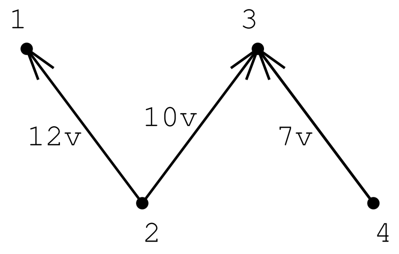

Figure 5. Voltages on a circuit

"$voltage(1,2,12)$" states that the voltage between node 1 and node 2 is 12 volts. In order to derive the voltage data independently of the node order, we could consider defining "$volt$" by the following rules:

$$\begin{aligned}&volt(A,B,V)\ :-\ voltage(A,B,V).\\&volt(B,A,-V)\ :-\ voltage(A,B,V).\end{aligned}$$

Furthermore, we will define the predicate "v" that derives voltages between two arbitrary nodes A and C as

$$\begin{aligned}&v(A,C,V)\ :-\ volt(A,C,V).\\&v(A,C,V+W)\ :-\ volt(A,B,V),\ v(B,C,W).\end{aligned}$$

It turns out, however, that these definitions will not work as intended. In order to derive the voltage between 1 and 4, we will attempt to execute the following goal clause:

$$?-\ v(1,4,X).$$

As the voltage between 1 and 4 is not given, the goal is decomposed into sub-goals by the second definition of "$v(...)$". The first sub-goal succeeds by "$volt(1,2,12)$" binding node B with 2. The second sub-goal "$v(2,4,W)$" is also decomposed into sub-goals by the second definition of "$v(...)$" again. The first sub-goal succeeds as "$volt(2,1,-12)$" using the same voltage data, and the second sub-goal becomes the same as the initial goal. Thus, the system loops.

One method to avoid this problem is to erase voltage data as they are used, so that the same datum is not used twice. This can be done by replacing the definition of "$volt(...)$" with

$$\begin{array}{ll} volt(A,B,V)\ :- & voltage(A,B,V). \\ & retract(voltage(A,B,V)). \\ volt(B,A,-V)\ :- & voltage(A,B,V). \\ & retract(voltage(A,B,V)). \end{array}$$

But the erased data cannot be recovered in backtracking.

Another common method keeps track of the data used, so that the same datum is not used twice. In this method, the definition of "$v(...)$" is replaced with the following clauses, and we acquire the voltage between nodes 1 and 4 by the goal clause "$?-\ v(1,4,X,[\,])$". The fourth argument of the goal is a list of nodes which have already been used to calculate the voltages and must not be used twice. This method has the disadvantage of requiring the overhead of explicitly keeping track of the data used:

$$\begin{array}{ll} v(A,C,V,\_)\ :-\ volt(A,C,V). & \\ v(A,C,V+W,T)\ :- & volt(A,B,V), \\ & not\ member(B,T,\_), \\ & v(B,C,W,[A|T]). \end{array}$$

In the next section, we show how to avoid this problem by viewing problem solving as a generalized parsing problem.

3.5. Solution of the Looping Problem

To solve the above looping problem, we introduce a change of representation which involves viewing the voltage derivation problem not as a backward search problem, but as a parsing problem. The node-voltage data are not represented by a set of facts. Each expression "$voltage(A, B, V)$" forms a compound term. The voltages in Figure 5 are represented by a set of those terms using a list:

$$vData([voltage(1, 2, 12),\ voltage(3, 2, 10),\ voltage(3, 4, 7)]).$$

Accordingly, we represent the voltage derivation not as clauses for backward chaining but as grammar rules for parsing. The following grammar rules correspond to the clauses of "$volt(...)$":

$$\begin{array}{l} volt(A, B, V) \ \longrightarrow\ [voltage(A, B, V)]. \\ volt(B, A, -V) \ \longrightarrow\ [voltage(A, B, V)]. \end{array}$$

"$voltage(A, B, V)$" surrounded by "[" and "]" is a terminal symbol, while "$volt(A, B, V)$" is a nonterminal symbol. Here, we have introduced universally quantified variables $(A, B,$ and $V)$ into the grammar rules. These variables are instantiated when they are applied to object sentences. According to the DCSG conversion procedure, the grammar rules are converted into the following clauses:

$$\begin{array}{l} subset(volt(A, B, V), S0, S1)\ :-\ member(voltage(A, B, V), S0, S1). \\ subset(volt(B, A, -V), S0, S1)\ :-\ member(voltage(A, B, V), S0, S1). \end{array}$$

In order to derive the voltage between two arbitrary nodes, we define the following grammar rules corresponding to the clauses of "$v(...)$":

$$\begin{array}{l} v(A, C, V) \ \longrightarrow\ volt(A, C, V). \\ v(A, C, V + W) \ \longrightarrow\ volt(A, B, V), v(B, C, W). \end{array}$$

These grammar rules are converted into the following clauses:

$$\begin{array}{ll} subset(v(A, C, V), S0, S1)\ :- & subset(volt(A, C, V), S0, S1). \\ subset(v(A, C, V + W), S0, S2)\ :- & subset(volt(A, B, V), S0, S1), \\ & subset(v(B, C, W), S1, S2). \end{array}$$

Deriving the voltage X between nodes 1 and 4 is accomplished by identifying the nonterminal symbol "$v(1, 4, X)$" in the word-order free sentence of voltage data as follows:

$$?-\ vData(VD),\ subset(v(1,4,X),VD,\_).$$

$$X\ =\ 12\ +\ (-10\ +\ 7)$$

Terminal symbols associated with the nonterminal symbol "$v(1,4,X)$" are removed sequentially from the object sentence. Therefore, the looping problem due to using the same data repeatedly does not occur. This method is similar to removing data using the predicate "$retract(...)$" described in the previous section. But the removed data can be recovered by backtracking.

We have overcome a common looping problem in backward chaining by simply rewriting backward chaining rules as grammar rules and by viewing a set of facts as a word-order free sentence. The looping problem caused by multiple use of the same fact is avoided by using DCSG to view circuit analysis as a generalized parsing problem.

4. Finding Structures in Circuits

4.1. Circuits Represented as Sentences

Since we have a mechanism for parsing word-order free languages, we will treat a circuit represented as a list as a word-order free sentence. The circuit $ca49$ in Figure 6 is represented by the fact (4) shown below. The predicate $ca49([...])$ states that the word-order free sentence "[...]" represents circuit $ca49$. The compound term $battery(b1,1,2)$ represents the battery $b1$ connected its positive terminal to the node 1 and its negative terminal to the node 2.

$$\begin{aligned}&ca49([resistor(r1,1,3),\ resistor(r2,2,3),\ resistor(r3,1,4),\\&\qquad resistor(r4,2,4),\ resistor(r5,3,4),\ battery(b1,1,2)]).\end{aligned} \tag{4}$$

4.2. Grammar Rules without Recursion

Since a resistor is a non-polar element, we need a way to refer to a resistor regardless of its node order. The non-terminal symbol $res(R,A,B)$ can refer either $resistor(R,A,B)$ or $resistor(R,B,A)$. Here, DCSG allows the symbol ";" as abbreviation of two grammar rules with the same left hand side. The

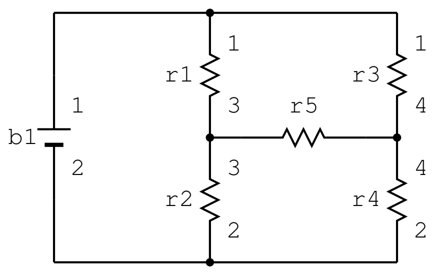

Figure 6. Circuit ca49

non-terminal symbol $batt(...)$ enables us to refer the same battery with different names. If a battery is referred to with reverse node-order, the name of battery is prefixed by a minus symbol. The non-terminal symbol $anyElm(X, A)$ can refer to any element X connected to the node A.

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

$$\begin{aligned} batt(E, A, B) &\longrightarrow\ [battery(E, A, B)]. \\ batt(-E, B, A) &\longrightarrow\ [battery(E, A, B)]. \end{aligned}$$

$$\begin{aligned} anyElm(X, A) \longrightarrow\ & [terminal(X, A)]; \\ & res(X, A, \_\,); \\ & batt(X, A, \_\,). \end{aligned}$$

4.3. All Elements Connected to a Node

In circuit analysis, Kirchhoff's current law (KCL) requires that the sum of all branch currents into a node be zero. In order to apply KCL at a specific node, we must find all elements connected to the node. The non-terminal symbol $allElm(X, A)$ successfully finds these elements connected to the node A. Here, X is replaced by a list of the all elements. If the same rules were written in backward chaining by replacing the symbol "$\longrightarrow$" with "`:-`", a looping problem due to using the same data repeatedly would occur.

$$allElm([\,], A) \longrightarrow\ not\ \ anyElm(\_\,, A).$$

$$\begin{aligned} allElm([X|Y], A) \longrightarrow\ & anyElm(X, A), \\ & allElm(Y, A). \end{aligned}$$

The following goal finds all elements connected to node 1 in the circuit $ca49$ as:

$$?-\ ca49(CT), subset(allElm(X, 1), CT, \_).$$

$$X = [r1, r3, b1]$$

4.4. Paths and loops

According to Kirchhoff's voltage law (KVL), the sum of branch voltages along to a loop must be zero. In order to find loops in a circuit, we first define the non-terminal symbol $path(X, A, B)$ which finds routes between two nodes A and B.

$$\begin{aligned} & path([X], A, B) \longrightarrow anyElm(X, A, B). \\ & path([X, B|Y], A, C) \longrightarrow\ anyElm(X, A, B), \\ & \qquad\qquad\qquad\qquad\qquad path(Y, B, C). \end{aligned}$$

The variable X in $path(X, A, B)$ is substituted for by elements and nodes from the starting node A to the ending node B. The following goal attempts to find all routes from the starting node 1 to the ending node 2 in the circuit $ca49$.

$$?-\ ca49(CT), subset(path(X, 1, 2), CT, \_).$$

$$\begin{aligned} & X = [b1]; \\ & X = [r1, 3, r2]; \\ & X = [r1, 3, r5, 4, r4]; \\ & X = [r1, 3, r5, 4, r3, 1, b1] \end{aligned}$$

The first answer $[b1]$ shows the path from the node 1 via element $b1$ to the node 2. The second answer shows the path from the node 1 via $r1$, node 3, and $r2$ to the node 2. The first three answers show paths from the node 1 to the node 2, but the fourth answer is not desired. It goes back to the starting node 1 and then goes to the ending node 2 via $b1$. Since parsing sentence does not use the same word twice, no element in the circuit appeared twice in the answer. The problem is that our definition does not inhibit the use of the same node twice.

Since we want to get correct answers which do not include loops in the paths, we modify the grammar rules as follows:

$$\begin{aligned} &path([X], A, B, \_) \longrightarrow anyElm(X, A, B).\\ &path([X, B|Y], A, C, T) \longrightarrow anyElm(X, A, B),\\ &\qquad\qquad quote\ not\ member(B, T, \_),\\ &\qquad\qquad path(Y, B, C, [A|T]). \end{aligned}$$

The "$quote$" in the grammar rule is a command to the DCSG converter. It directs the converter to insert the following strings as is. Namely, "$not\ member(B, T, \_)$," is inserted as a Prolog clause in the DCSG conversion. The last argument of $path$ is substituted for by a list of nodes which are already in the path and not to be used twice as a relay node in the process of finding path.

The following goal successfully finds all paths from the node 1 to the node 2 in the circuit $ca49$. The last argument of $path$ is substituted for by "[2]" which shows the ending node 2 must not be used as a relay node.

$$?-\ ca49(CT), subset(path(X, 1, 2, [2]), CT, \_).$$

$X = [b1];$
$X = [r1, 3, r2];$
$X = [r1, 3, r5, 4, r4];$
$X = [r3, 4, r4];$
$X = [r3, 4, r5, 3, r2];$
No

If the ending node is equal to the starting node, the goal enumerates all loops through the node as follows:

$$?-\ ca49(CT), subset(path(X, 1, 1, [\,]), CT, \_).$$

$X = [r1, 3, r5, 4, r3];$
$X = [r1, 3, r5, 4, r4, 2, -b1];$

$X = [r1, 3, r2, 2, -b1];$
$X = [r1, 3, r2, 2, r4, 4, r3];$
$X = [r3, 4, r4, 2, -b1];$
$\ldots$

4.5. Series-Parallel Circuit

Recursively defined circuits do not appeared in actual IC-designs[13], but circuit theory has the concepts of such circuits. The following grammar rules define the circuit which consists only of series and parallel connections of resistors.

$$\begin{aligned}
&spCircuit(R, A, B) \longrightarrow res(R, A, B).\\
&spCircuit(sr(X, Y), A, C) \longrightarrow spCircuit(X, A, B),\\
&\qquad\qquad spCircuit(Y, B, C),\\
&\qquad\qquad not\ anyElm(\_, B, \_).\\
&spCircuit(pr(X, Y), A, B) \longrightarrow spCircuit(X, A, B),\\
&\qquad\qquad spCircuit(Y, A, B).
\end{aligned}$$

The first rule defines a single resistor R as a series-parallel circuit $spCircuit(R, A, B)$. The second rule defines a series connection of series-parallel circuits as a series-parallel circuit $spCircuit(sr(X, Y), A, C)$. Here, $sr(X, Y)$ is the name given to the connection. The grammar rule has a condition "$not\ anyElm(\_, B, \_)$" that the central node B of the series connection must not be connected to other elements. The third rule defines a parallel connection of series-parallel circuits as a series-parallel circuit $spCircuit(pr(X, Y), A, B)$. "$pr(X, Y)$" is the name given to the connection. Unlike the definitions of series and parallel connections in Section 2.5, the conditions such as "$not\ X = Y$" are not needed because no elements are used twice in parsing.

In order to identify a series-parallel circuit connected to nodes 1 and 4 in Figure 1, we attempt the following goal:

$$?-\ ca40(CT), subset(spCircuit(X, 1, 4), CT, \_).$$

But the second goal loops. The top-down mechanism of DCSG decomposes the starting symbol into terminal symbols iteratively until it generates a set of terminal symbols which coincides with the object data set. Since the series-parallel connection is defined by left recursive rules, the system infinitely decomposes the starting symbol with the same symbol when the generated elements do not coincide.

In top-down parsing, we must avoid grammar rules with left recursion. In ordinary context-free grammars, this can be done by introducing additional non-terminal symbols to change rules into Greibach normal form[1]. Each grammar rule in the normal form first generates a terminal symbol then generate non-terminal symbols. This means that each terminal symbol is a component of some non-terminal symbols with other non-terminal symbols. So, we have to define the non-terminal symbol "series-parallel circuit" with the terminal symbol "resistor" and other circuits. This can be realized by introducing a concept of three-terminals circuit as a non-terminal symbol. This method is discussed in the paper[8]. Another method to solve this problem is to use bottom up parsing. The bottom-up method is discussed in the paper[5].

5. Circuit Grammar for Functional Blocks

This section extends DCSG to create a new circuit grammar for functional blocks of electronic circuits[9]. This circuit grammar has fields for semantic terms, and defines not only syntactic structures but also the relationships between those structures and their meaning. Here we assume circuit functions as meaning of circuit structres.

5.1. Semantic Field in Left-Hand Side

Semantic terms are placed in curly brackets in grammar rules as follows.

$$A, \{F_1, F_2, ..., F_m\} \;\longrightarrow\; B_1, B_2, ..., B_n. \tag{5}$$

This grammar rule can be read as stating that the symbol A with meaning $\{F_1, F_2, ..., F_m\}$ consists of the syntactic structure $B_1, B_2, ..., B_n$. This rule is converted into a Prolog clause as follows.

$$\begin{aligned}
&ss(A, S_0, S_n, E_0, [F_1, F_2, ..., F_m|E_n]) :- \\
&\qquad\qquad\qquad\qquad ss(B_1, S_0, S_1, E_0, E_1), \\
&\qquad\qquad\qquad\qquad ss(B_2, S_1, S_2, E_1, E_2),
\end{aligned}$$

$$\begin{aligned} & \ldots, \\ & ss(B_n, S_{n-1}, S_n, E_{n-1}, E_n). \end{aligned} \qquad (5)'$$

Since the conversion differs from that used in DCSG, we use the predicate "ss" instead of "$subset$". When the rule is used in parsing, the goal $ss(A, S_0, S_n, E_0, E)$ is executed, where the variable S_0 is replaced by an object set (object circuit) and the variable E_0 is replaced by an empty set. The subsets (sub-circuits) "$B_1, B_2, ..., B_n$" are successively identified in the object set S_0. After all of these subsets are identified, the remainder of these subsets (the complementary set) is put into S_n. While, the semantic information of B_1 is added with E_0 and put into E_1, the semantic information of B_2 is added with E_1 and put into E_2,..., and the semantic information of B_n is added with E_{n-1} and put into E_n. Finally, the semantic information $\{F_1, F_2, ..., F_m\}$, which is the meaning associated with symbol A, is added and all of the semantic information is put into E.

Terminal symbols are surrounded by square brackets in grammar rules. The symbol "$[B_i]$" is converted to $member(B_i, S_i, S_{i+1})$ which identifies the element B_i in the object set S_i, and put the remainder into S_{i+1}. The terminal symbol "$[B_i]$" does not change the current semantic information E_i, but the technique in Section 6.3 enables us to add semantic informations to terminal symbols.

5.2. Semantic Terms in the Right-Hand Side

Semantic terms in the right-hand side define the semantic conditions for the grammar rule. For example, the following rule (6) is converted into the Prolog clause (6)' as follows.

$$A \longrightarrow B_1, \{C_1, C_2\}, B_2. \qquad (6)$$

$$\begin{aligned} ss(A, S_0, S_n, E_0, E_n) \ :- \ & ss(B_1, S_0, S_1, E_0, E_1), \\ & member(C_1, E_1, \_\,), \\ & member(C_2, E_1, \_\,), \\ & ss(B_2, S_1, S_2, E_1, E_2). \end{aligned} \qquad (6)'$$

When the clause (6)' is used in parsing, the conditions C_1 and C_2 are tested to see if the semantic information E_1 meets these conditions after identifying the symbol B_1. If it succeeds, the parsing process goes on to identify the symbol B_2.

6. Grammar Rules for Functional Blocks

6.1. Electrical Dependencies

When a designed circuit does not work, the engineer tries to localize the fault. He first checks the power supply to confirm that the correct voltage is being applied, since the power supply voltage is a prerequisite for correct behavior throughout the whole circuit. Next, he traces causal chains of voltage and current relationships in the circuit. The location of the fault is often determined by finding a place where causal chains fail to connect as intended.

Since inputs and outputs are clearly separated in logic circuits, problems in deriving causal chains do not occur at the logic level. In contrast, it is much harder to derive the causal chains from the circuit topology at the transistor level without knowledge of how circuits are organized. A current through a resistor causes a voltage across that resistor, while, inversely, a voltage applied to a resistor causes a current through the resistor. Although it is difficult to determine which is the cause and which is the effect from the standpoint of the physics of electrical devices, engineers use this kind of causal reasoning to form causal chains that explain how a given circuit works.

The new circuit grammar has fields for semantic terms. Using these semantic terms, we can define relationships between circuit structures and their functions. The circuit functions we consider are the electrical behaviors that are useful to circuit designers or users. These electrical behaviors are defined on the voltages and currents occurring in the circuit. In particular, electrical dependencies such as causality and conditions are useful to understand how circuits work. We show how voltage and current dependencies are coded in the new circuit grammar, and how these dependencies are derived through parsing circuit structures.

6.2. Object Circuit

We now develop grammar rules using the functional blocks appearing in the circuit $cd15$, which is a type of operational amplifier called a transconductance amplifier (Figure 7). The amplifier receives a voltage input and produces a current output. The circuit $cd15$ is represented as the following word-order free sentence. Here, the compound term $npnTr(q1, 3, 5, 6)$ is a terminal symbol which represents the NPN-transistor named $q1$ with the base connected to node 3, the emitter to node 5, and the collector to node 6 respectively.

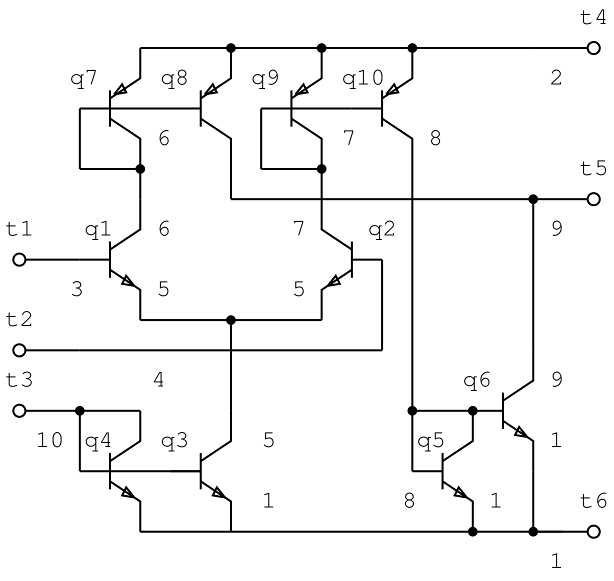

Figure 7. Circuit $cd15$

$$\begin{aligned}&cd15([npnTr(q1, 3, 5, 6), npnTr(q2, 4, 5, 7), npnTr(q3, 10, 1, 5),\\ &\quad npnTr(q4, 10, 1, 10), npnTr(q5, 8, 1, 8), npnTr(q6, 8, 1, 9),\\ &\quad pnpTr(q7, 6, 2, 6), pnpTr(q8, 6, 2, 9), pnpTr(q9, 7, 2, 7),\\ &\quad pnpTr(q10, 7, 2, 8)]).\end{aligned} \tag{7}$$

6.3. Rules for Devices

The grammar rule (8) defines an NPN-transistor Q in active state. Although "$npnTr(Q, B, E, C)$" is a terminal symbol, it is also defined as a non-terminal with semantic information. The compound term $gt(v(C, E), vst)$ represents the fact that the collector-emitter voltage $v(C, E)$ is greater than the collector saturation voltage vst. The compound term $equ(v(B, E), vbe)$ represents the fact that the base-emitter voltage $v(B, E)$ is equal to the forward voltage of PN-junction vbe. The compound term $cause(v(B, E), i(B, Q), Q)$ represents the fact that the base-emeitter voltage $v(B, E)$ causes the base current $i(B, Q)$ by the operation of the NPN-transistor Q. Here, $i(B, Q)$ represents the branch current from node B to transistor Q.

Grammar rules for the saturated state and the cutoff state are also defined. When a terminal symbol $[npnTr(Q, B, E, C)]$ is found in parsing a circuit, one of these rules is selected, and its semantic terms are derived as a meaning of the symbol non-deterministically. Similar rules are also defined for PNP-transistors.

$$\begin{aligned}
&npnTr(Q, B, E, C),\\
&\{\ state(Q, active),\\
&\quad gt(v(C, E), vst),\\
&\quad equ(v(B, E), vbe),\\
&\quad gt(i(B, Q), 0),\\
&\quad gt(i(C, Q), 0),\\
&\quad cause(v(B, E), i(B, Q), Q),\\
&\quad cause(v(B, E), i(Q, E), Q),\\
&\quad cause(i(B, Q), v(B, E), Q),\\
&\quad cause(i(Q, E), v(B, E), Q),\\
&\quad cause(i(B, Q), i(C, Q), Q)\} \longrightarrow [npnTr(Q, B, E, C)].
\end{aligned} \tag{8}$$

6.4. Rules for Functional Blocks

A transistor in which the base and the collector are connected together works as a diode (Figure 8). The following grammar rule (9) defines the diode-connected transistor "$dtr(dtr(Q), A, C)$" in the $conductive$ state as a non-terminal symbol. The syntactic part of the right-hand side defines either an NPN-transistor Q or a PNP-transistor Q whose base and collector are connected to the same node. The semantic term $state(Q, active)$ in the right-hand side is an electrical condition which requires that the transistor Q must be in the $active$ state. Here, $dtr(Q)$ is a name given to the diode (Skolem function).

The semantic terms in the left-hand side represent the electrical behavior in the conductive state. E.g. "$gt(i(A, dtr(Q)), 0)$" represents the current flows from A to $dtr(Q)$. Here, "$i(A, dtr(Q))$" is the branch current from the node A to the functional block $dtr(Q)$. Since we assume that branch current flows not only from a node to an element but also from a node to a functional block, the same current can have different expressions, for example $i(A, dtr(Q))$ and $i(A, Q)$. The semantic term $equiv(i(A, dtr(Q)), i(A, Q))$ is added to form the equivalence relations for these currents. The grammar rule for diode-connected transistor in reverse bias is also defined in the same manner.

$$\begin{aligned}
&dtr(dtr(Q), A, C),\\
&\{\ state(dtr(Q), conductive),\\
&\quad gt(i(A, dtr(Q)), 0),\\
&\quad cause(v(A, C), i(A, dtr(Q)), dtr(Q)),\\
&\quad cause(v(A, C), i(dtr(Q), C), dtr(Q)),\\
&\quad cause(i(A, dtr(Q)), v(A, C), dtr(Q)),\\
&\quad cause(i(dtr(Q), C), v(A, C), dtr(Q)),\\
&\quad equiv(i(A, dtr(Q)), i(A, Q)),\\
&\quad equiv(i(dtr(Q), C), i(Q, C))\} \longrightarrow \ (\ npnTr(Q, A, C, A);\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad pnpTr(Q, C, A, C)\ ),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{state(Q, active)\}.
\end{aligned} \tag{9}$$

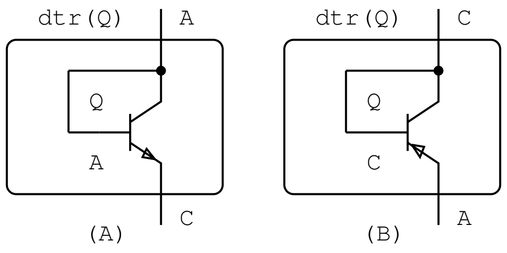

Figure 8. Diode-connected transistor

Figure 9 shows two current mirror circuits. Both circuits generate the same current as their reference current. The grammar rule (10) is defined for the source-type current mirror shown in Figure 9(A). The semantic terms in the right-hand side are the electrical conditions that operate the circuit. The semantic term in the left-hand side "$cause(i(cmo(D, Q), Ref), i(cmo(D, Q), So),$ $cmo(D, Q))$" is related to the main function of this current mirror. That is, the external current from $cmo(D,Q)$ to Ref causes another external current from $cmo(D,Q)$ to So by the circuit $cmo(D,Q)$. The next "$cause(i(cmo(D,Q),Ref),i(D,Ref),cmo(D,Q))$" represents the external current $i(cmo(D,Q),Ref)$ causes the internal current $i(D,Ref)$ of the functional block. This causal relationship connecting external and internal aspects of functional block enables us to explain the main function of the functional block togather with equivalence relations between currents.

A similar rule is also defined for the sink-type current mirror shown in Figure 9(B).

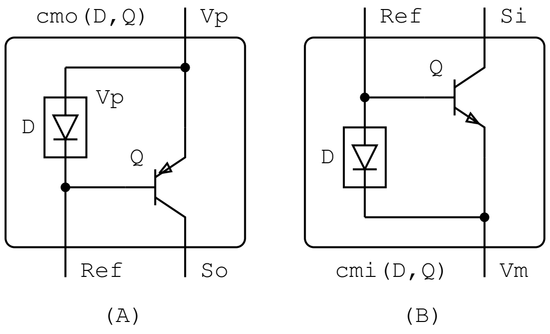

Figure 9. Current mirror

$$
\begin{aligned}
&currentMirrorSource(cmo(D,Q),Ref,Vp,So),\\
&\{\ cause(i(cmo(D,Q),Ref),i(cmo(D,Q),So),cmo(D,Q)),\\
&\ \ cause(i(cmo(D,Q),Ref),i(D,Ref),cmo(D,Q)),\\
&\ \ equiv(i(cmo(D,Q),So),i(Q,So))\} \longrightarrow\\
&\qquad\qquad\qquad dtr(D,Vp,Ref),\\
&\qquad\qquad\qquad \{state(D,conductive)\},\\
&\qquad\qquad\qquad pnpTr(Q,Ref,Vp,So),\\
&\qquad\qquad\qquad \{state(Q,active)\}.
\end{aligned}
\tag{10}
$$

Figure 10 shows the emitter-coupled pair which generates two collector currents from the voltage across two bases $B1$ and $B2$. The difference between the two collector currents is proportional to the voltage. While, the total of two collector currents is controlled by the reference current from the node Rf into the current mirror CMi. Here, the grammar rule (11) formalizes only dependencies between those voltages and currents.

$$\begin{array}{l}
eCoupledPair(ECP, B1, B2, Rf, C1, C2, Vm), \\
\{\ cause(v(B1, B2), i(C1, Q1), ECP), \\
\quad cause(v(B2, B1), i(C2, Q2), ECP), \\
\quad cause(i(E, CMi), i(Q1, E), ECP), \\
\quad cause(i(E, CMi), i(Q2, E), ECP), \\
\quad equiv(i(Rf, ECP), i(Rf, CMi)), \\
\quad equiv(i(B1, ECP), i(B1, Q1)), \\
\quad equiv(i(B2, ECP), i(B2, Q2)), \\
\quad equiv(i(C1, ECP), i(C1, Q1)), \\
\quad equiv(i(C2, ECP), i(C2, Q2))\} \longrightarrow \\
\qquad\qquad\qquad npnTr(Q1, B1, E, C1), \\
\qquad\qquad\qquad \{state(Q1, active)\}, \\
\qquad\qquad\qquad npnTr(Q2, B2, E, C2), \\
\qquad\qquad\qquad \{state(Q2, active)\}, \\
\qquad\qquad\qquad currentMirrorSink(CMi, Rf, Vm, E), \\
\qquad\qquad\qquad quote\ ECP = ecp(Q1, Q2, CMi).
\end{array} \tag{11}$$

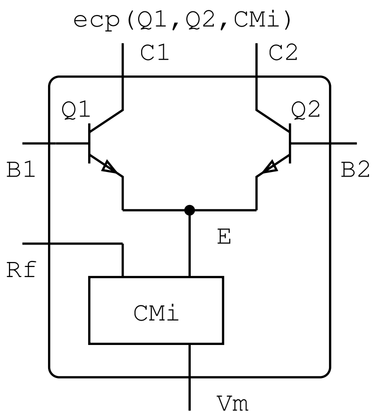

Figure 10. Emitter coupled pair

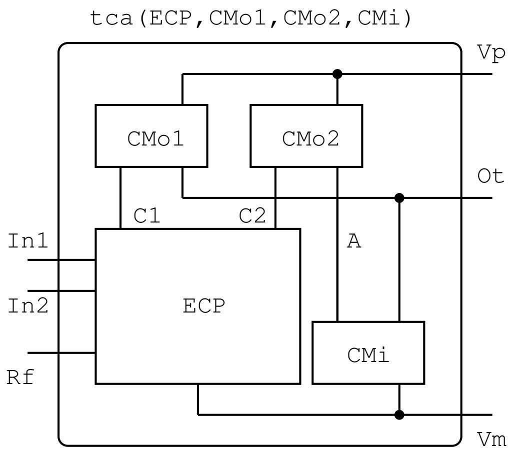

Figure 11. Transconductance amplifier

Figure 11 shows the structure of transconductance amplifire. Two collector currents generated by the emitter coupled pair are duplicated by two source-type current mirrors respectively. One of the currents forms the source current of the amplifier. Another current is duplicated by a sink-type current mirror, and forms the sink-current of the amplifier. The following grammar rule (12) is defined for the transconductance amplifier.

$$
\begin{aligned}
&transConductanceAmp(TCA, In1, In2, Rf, Ot, Vp, Vm),\\
&\{\ cause(v(In1, In2), i(TCA, Ot), TCA),\\
&\ \ cause(i(CMo1, Ot), i(TCA, Ot), TCA),\\
&\ \ cause(i(Ot, CMi), i(Ot, TCA), TCA),\\
&\ \ equiv(i(Rf, TCA), i(Rf, ECP)),\\
&\ \ cause(i(C1, ECP), i(CMo1, C1), TCA),\\
&\ \ cause(i(C2, ECP), i(CMo2, C2), TCA),\\
&\ \ cause(i(CMo2, A), i(A, CMi), TCA)\} \longrightarrow\\
&\qquad\qquad eCoupledPair(ECP, In1, In2, Rf, C1, C2, Vm),\\
&\qquad\qquad currentMirrorSource(CMo1, C1, Vp, Ot),\\
&\qquad\qquad currentMirrorSource(CMo2, C2, Vp, A),\\
&\qquad\qquad currentMirrorSink(CMi, A, Vm, Ot),\\
&\qquad\qquad quote\ \ TCA = tca(ECP, CMo1, CMo2, CMi).
\end{aligned}
\tag{12}
$$

7. Parsing Circuits

All of the grammar rules defined in the previous section are converted into Prolog clauses according to the circuit grammar conversion method described in Section 5. The clauses form a logic program that performs top-down parsing. The following goal (13) parses the circuit $cd15$ and derives the circuit structure and its electrical behaviour. The first subgoal $cd15(CT)$ substitutes the circuit $cd15$ into the variable CT. The circuit is given to the second argument of the predicate $ss(...)$. The first argument X is a functional block identified in the circuit and the third argument is the remainder of the circuit. Since the third argument is empty, the goal asks whether the whole circuit can be identified as the single non-terminal symbol X. The fourth argument [ ] means no semantic information is given at the start of parsing. Each time a functional block is identified, semantic information about the functional block is added. After the whole circuit is parsed, the value of Y has much semantic information about the circuit.

$$?-\ cd15(CT),\ ss(X, CT, [\,], [\,], Y). \tag{13}$$

$$\begin{aligned} X = transCondAmp(\, & tca(ecp(q1, q2, cmi(drt(q4), q3)), \\ & \qquad cmo(dtr(q7), q8), \\ & \qquad cmo(dtr(q9), q10), \\ & \qquad cmi(dtr(q5), q6)), \\ & 3, 4, 10, 2, 9, 1) \end{aligned}$$

$$\begin{aligned} Y = \ [& cause(v(3,4), i(tca(...), 9), tca(...)), \\ & cause(i(cmo(dtr(q7), q8), 9), \\ & \qquad i(tca(...), 9), tca(...), \\ & cause(i(9, cmi(dtr(q5), q6)), \\ & \qquad i(9, tca(...), tca(...)), \\ & equiv(i(10, tca(...)), i(10, ecp(...))), \\ & cause(i(6, ecp(...)), \\ & \qquad i(cmo(dtr(q7), q8), 6), tca(...)), \\ & cause(i(7, ecp(...), \\ & \qquad i(cmo(dtr(q9), q10), 7), tca(...)), \\ & cause(i(cmo(dtr(q9), q10), 8), \end{aligned}$$

$$\begin{aligned}
&\quad\quad i(8, cmi(dtr(q5), q6)), tca(...),\\
&cause(i(8, cmi(dtr(q5), q6)),\\
&\quad\quad i(9, cmi(dtr(q5), q6)), cmi(...)),\\
&cause(i(8, cmi(dtr(q5), q6)),\\
&\quad\quad i(8, dtr(q5)), cmi(...)),\\
&equiv(i(9, cmi(dtr(q5), q6)), i(9, q6)),\\
&state(q6, active),\\
&gt(v(9, 1), vst),\\
&equ(v(8, 1), vbe),\\
&gt(i(8, q6), 0),\\
&gt(i(9, q6), 0),\\
&gt(i(q6, 1), 0),\\
&cause(v(8, 1), i(8, q6), q6),\\
&cause(i(8, q6), i(q6, 1), q6),\\
&cause(i(q6, 1), i(9, q6), q6),\\
&state(dtr(q5), conductive),\\
&gt(i(8, dtr(q5)), 0),\\
&cause(i(8, dtr(q5)), i(dtr(q5), 1), dtr),\\
&cause(i(dtr(q5), 1), v(8, 1), dtr),\\
&equiv(i(8, dtr(q5)), i(8, q5)),\\
&equiv(i(dtr(q5), 1), i(q5, 1)),\\
&state(q5, active),\\
&gt(v(8, 1), vst),\\
&equ(v(8, 1), vbe),\\
&gt(i(8, q5), 0),\\
&gt(i(q5, 1), 0),\\
&cause(v(8, 1), i(8, q5), q5),\\
&cause(i(8, q5), i(q5, 1), q5),\\
&cause(i(q5, 1), i(8, q5), q5),\\
&\ ...\ 108\ lines\ omitted\ ...)]
\end{aligned} \qquad (14)$$

The value of X shows that the circuit $cd15$ is identified to be the operational amplifier "$transCondAmp(tca(...),\ 3, 4, 10, 2, 9, 1)$". The first argument $tca(...)$ is a name given to the identified circuit, and the rest are the connecting nodes in the circuit. The name keeps track of identified functional blocks and is viewed as a parse tree which shows the syntactic structure of the circuit (Figure 12). Each node represents a functional block identified in the circuit $cd15$.

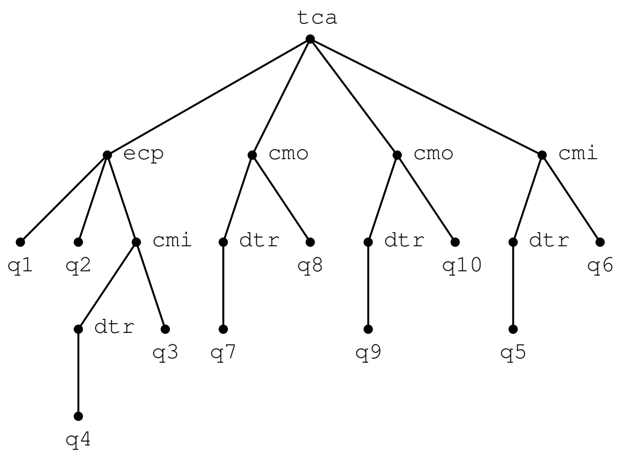

Figure 12. Parse tree for $cd15$

The semantic information substituted into Y consists of electrical conditions, causal relationships, and equivalence relations. The electrical conditions show the conditions under which all the functional blocks will work correctly as components of the given circuit. For example, the five lines from $state(q6, active)$ represent electrical conditions which keep the transistor $q6$ active. These electrical conditions on $q6$ togather with a conductive state of $dir(q5)$ enable $cmi(dtr(q5), q6)$ to work as a current mirror circuit as shown in Figure 13.

The causal relationships consist of dependencies on voltages and currents. The first one "$cause(v(3, 4), i(tca(...), 9), tca(...))$" is added after identifying the whole circuit. It is related to the main function of transconductance amplifier. Here, the structure of the transconductance amplifier is abbreviated as $tca(...)$. This main causal relationship is also supported by internal causal chains of the transconductance amplifier as shown in Figure 14. The causal relationships such as between $i(cmo, 7)$ and $i(cmo, 8)$ in the figure are also explained by internal causal chains of the source-type current mirror circuit. Equivalence relations (Section 6.3), which connect internal and external expressions of a functional block, enable us to explain details of the main causal rela-

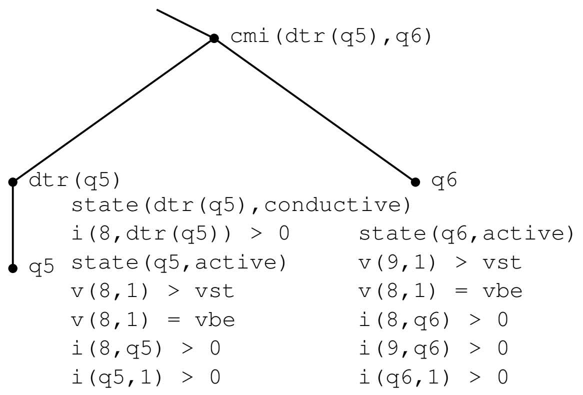

Figure 13. Electrical conditions for $cmi(dtr(q5), q6)$

tionship of the functional block.

8. Conclusions

We have viewed electronic circuits as a kind of formal language, and developed methods for parsing electronic circuits as sentences in that language. Circuit structures are defined by a logic grammar called DCSG. A set of grammar rules, when converted into Prolog clauses, forms a logic program which perform top-down parsing. When an unknown circuit is given, this logic program will analyze the circuit and derive a parse tree for the circuit. Since the parse tree shows a hierarchical structure of functional blocks composing the circuit, it can help engineers to identify which elements contribute to which sub-functions and how the total function of the circuit is achieved by its sub-functions for trouble shooting, redesigning, or modifying the circuit.

The performance of our system depends on the defined grammar rules. As more grammar rules are defined, more circuits can be parsed. Unlike 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 grammatical sentences defined by the given circuit grammars. If an object circuit has unknown structures, our system will not be

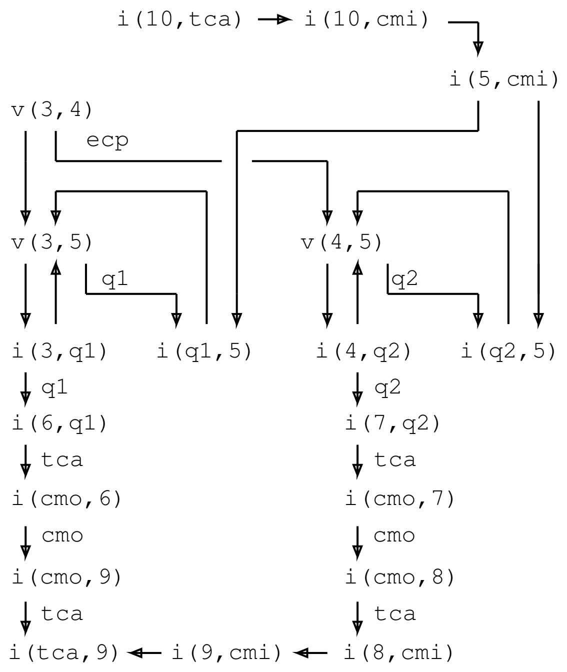

Figure 14. Causal chains on $cd15$

able to parse the whole circuit, but it can, however, separate the object circuit into known parts and the remainder.

When we consider the circuit design process in contrast with sentence generation, engineers often use context dependent grammar rules. For example, suppose a circuit design rule (goal) generates two current sources as its components (sub-goals). 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 a single voltage source for simplicity. That is, engineers have the ability to use context dependent circuit generation rules. We, however, have so far only considered context-free rules in this chapter. Grammar rules for context dependent circuits are discussed in our paper[6].

The newly developed circuit grammar includes fields for semantic terms [9]. Using these semantic terms, we defined electrical conditions to handle circuit functions and causal relationships between voltage and current. In particular, we showed how to derive electrical dependencies as the meanings of circuit structures by parsing circuits. The derived semantic terms which represent electrical conditions and causal relationships can also be viewed as forming a word-order free sentence which describes the circuit's behavior. Here, the terms such as "$cause(...)$" and "$equiv(...)$" are terminal symbols. We can easily define non-terminal symbols which implement transitivity on causality and equivalence relations on branch currents. Although these electrical dependencies only describe the surface behavior of electronic circuits, they will be useful for understanding how circuits work and for localizing faults in trouble shooting. We are currently developing a language for describing circuit behaviors and functions more precisely.

References

[1] S.A. Greibach, "A New Normal Form Theorem for Context-Free Phrase Structure Grammars", *JACM*, 1965, vol. 12, pp. 42–52.

[2] J. De Kleer, *Causal and Teleological Reasoning in Circuit Recognition*, TR-529, Artificial Intelligence Lab., M.I.T., MA, 1979.

[3] F.C.N. Pereira; D.H.D. Warren, "Definite Clause Grammars for Language Analysis", *Artificial Intell.*, 1980, Vol. 13, pp. 231–278.

[4] T. Tanaka, "Parsing Circuit Topology in a Deductive System", *Proc. IJCAI-85*, Los Angeles, CA, 1985, pp. 407–410.

[5] T. Tanaka, "Definite Clause Set Grammars: A Formalism for Problem Solving", *J. Logic Programming*, 1991, Vol.10, pp. 1–17.

[6] T. Tanaka, "Parsing Electronic Circuits in a Logic Grammar", *IEEE Trans. Knowledge and Data Eng.*, 1993, Vol.5, No.2, pp.225–239.

[7] T. Tanaka; O. Bartenstein, "DCSG-Converters in Yacc/Lex and Prolog", *Proc. 12th International Conference on Applications of Prolog*, Tokyo, Japan, 1999, pp.44–49.

[8] T. Tanaka, “A Logic Grammar for Circuit Analysis - Problems of Recursive Definition”, *LNAI*, Springer, 2007, Vol. 4693, pp.852–860.

[9] T. Tanaka, “Circuit Grammar: knowledge representation for structure and function of electronic circuits”, *Int. Journal of Reasoning-based Intelligent System*, Inderscience, 2009, Vol.1 pp.56-67.

[10] T. Tanaka, “Deriving Electrical Dependencies from Circuit Topologies Using Logic Grammar”, *LNAI*, Springer, 2009, Vol. 5712, pp.325-332.

[11] T. Tanaka, “A Mechanism for Converting Circuit Grammars to Definite Clauses”, *LNAI*, Springer, 2010, Vol. 6278, pp.190-199.

[12] P. W. Tuinenga, *SPICE - A Guide to Circuit Simulation & Analysis Using PSpice*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[13] *101 Analog IC Designs*, Interdesign Inc., Sunnyvale, CA, 1976.