Mechanisms for converting circuit grammars to definite clauses

Takushi Tanaka

Department of Computer Science and Engineering,
Fukuoka Institute of Technology,
3-30-1 Wajiro-Higashi Higashi-ku,
Fukuoka 811-0295, Japan
Email: tanaka@fit.ac.jp

Abstract: The circuit grammar is a logic grammar developed for knowledge representation of electronic circuits. Knowledge on circuit structures and their functions are coded into the grammar rules. Those grammar rules, when converted into definite clauses, form a logic programme which can parse given circuits and derive electrical behaviour. This paper shows mechanisms for converting circuit grammar rules into definite clauses.

Keywords: circuit grammar; knowledge representation; electronic circuits; definite clause grammar; logic grammar; reasoning-based system.

Reference to this paper should be made as follows: Tanaka, T. (2011) 'Mechanisms for converting circuit grammars to definite clauses', *Int. J. Reasoning-based Intelligent Systems*, Vol. 3, No. 2, pp.100–107.

Biographical notes: Takushi Tanaka received the BEng, MEng. and Dr.Eng. degrees 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 and Engineering at the Fukuoka Institute of Technology, where he has been since 1988. His research interests include artificial intelligence, language processing, logic programming, electronic circuits, robot soccer and small artificial satellite.

1 Introduction

As a step toward automatic circuit understanding, we developed a new method for analysing circuit structures described in Tanaka (2009). We view designed circuits as grammatical sentences, and their elements as words. Electrical behaviour and functions are meaning of the sentences. Knowledge on circuit structures and their electrical behaviour are coded into grammar rules. A set of grammar rules, when converted into definite clauses, forms a logic program which executes top-down parsing.

The circuit grammar is a descendant of the logic grammar called DCSG (Definite Clause Set Grammar) by Tanaka (1991) which was developed for analysing word-order free language. The circuit grammar consists of several extensions to the DCSG which are useful to analysing electronic circuits by Tanaka (1993).

In this paper, we first introduce the DCSG, then show the mechanism for converting grammar rules into definite clauses. Next, we introduce extensions for circuit analyses and show mechanisms for converting circuit grammars.

2 Logic grammar DCSG

2.1 Word-order free language

Most implementations of computer language Prolog provide a mechanism for parsing context-free languages called DCG (Definite Clause Grammar) by Pereira and Warren (1980). A set of the grammar rules, when converted into definite clauses, forms a logic program which executes top-down parsing. While, a logic grammar DCSG by Tanaka (1991) was developed for word-order free languages similar to the method of DCG.

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:

$$A \rightarrow B_1, B_2, ..., B_n \qquad (n \geq 1)$$

$$A \in V_N, \quad B_i \in V_N \cup V_T \qquad (i = 1, ..., n)$$

Copyright © 2011 Inderscience Enterprises Ltd.

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 $\{B_1, B_2, \dots, B_n\}$. 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.

2.2 DCSG conversion

The general form of the conversion procedure from a grammar rule

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

to a definite clause is:

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

Here, all symbols in the grammar rule are assumed to be non-terminal symbols. If '[B_i]'($1 \le i \le n$) 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 $S_0, S_1, \dots, S_n$ 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 definite 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 definite 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.

DCSG allows the symbol ';' as abbreviation of two grammar rules with the same left-hand side. The following rule (4) which generates B or C_1, C_2 from A is converted to the definite clause (4)' as:

$$A \rightarrow B; C_1, C_2. \tag{4}$$

$$\begin{aligned} subset(A, S_0, S_2) :- & \ subset(B, S_0, S_2); \\ & subset(C_1, S_0, S_1), \\ & subset(C_2, S_1, S_2). \end{aligned} \tag{4'}$$

3 Mechanisms for DCSG-conversion

The following List 1 shows a basic DCSG-converter written in Prolog. The line 01 defines the main predicate 'dcsgConv' which converts a grammar rule into a definite clause. It separates the grammar rule into a left-hand side and a right-hand side, and generates a definite clause from a head part and a body part. The first subgoal 'conv(Lhs, Head, S0, S1)' generates the head part from the left-hand side using the definition of line 10, and the second subgoal 'conv(Rhs, Body, S0, S1)' generates the body part from the right-hand side. If the right-hand side consists of more than two grammar symbols, the predicate 'conv' defined by the line 02 and 03 separates these symbols into the first one and others. The line 02 generates a conjunctive subgoals from grammar symbols connected by ',', while the line 03 generates a disjunctive subgoals from ';'. The line 09 defines the conversion of a single terminal symbol. The line 10 defines the conversion of a single non-terminal symbol, which is used both for converting the left-hand side and the right-hand side.

List 1: Basic DCSG-converter

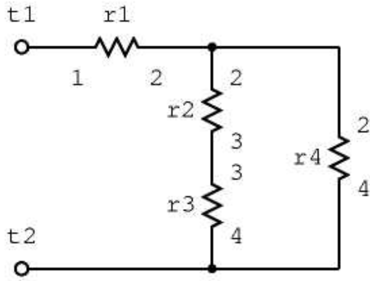
```
01  dcsgConv((Lhs --> Rhs),
             (Head :- Body)) :-
        conv(Lhs,Head,S0,S1),
        conv(Rhs,Body,S0,S1).
02  conv((CompoA,CompoB),
          (CA,CB),S0,S1) :-
        !,
        conv(CompoA,CA,S0,S),
        conv(CompoB,CB,S,S1).
03  conv((CompoA;CompoB),
          (CA;CB),S0,S1) :-
        !,
        conv(CompoA,CA,S0,S1),
        conv(CompoB,CB,S0,S1).
09  conv([Component],
          member(Component,S0, S1),
          S0,S1)  :-  !.
10  conv(Component,
         subset(Component,S0, S1),
         S0,S1)  :-  !.
```

4 Context-dependent features

4.1 Condition for absence

When we define grammar rules for actual circuits, several extensions for context-dependent features are needed. The circuit *ca*39 in Figure 1 is represented by the fact (5). Here, the list '[*resistor*(*r*1, 1, 2), *resistor*(*r*2, 2, 3), ...]' is a word-order free sentence. The compound terms such as '*resistor*(*r*1, 1, 2)' and '*terminal*(*t*1, 1)' are words which represent the resistor *r*1 connected to the nodes 1 and 2, and the external terminal t_1 connected to the node 1, respectively.

Figure 1 Circuit *ca*39



$$ca39([resistor(r1,1,2),\ resistor(r2,2,3),\ resistor(r3,3,4),\ resistor(r4,2,4),\ terminal(t1,1),\ terminal(t2,4)]). \quad (5)$$

First we define a non-terminal symbol '*res*(*R*, *A*, *B*)' by the rule (6) which enables to refer a resistor regardless its node order, because the resistor is a non-polar element. The rule is converted to the definite clause (6)'.

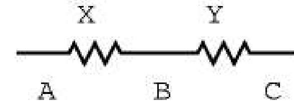
$$res(R,A,B)\ \rightarrow\ [resistor(R,A,B)];\ [resistor(R,B,A)]. \quad (6)$$

$$subset(res(R,A,B),S0,S1)\ :-\ member(resistor(R,A,B),S0,S1);\ member(resistor(R,B,A),S0,S1). \quad (6)'$$

The following conjunctive goal attempts to find a series connection of resistors (Figure 2) in the circuit *ca*39. The first subgoal *ca*39(*CT*0) binds the circuit to the variable *CT*0. The second subgoal *subset*(*res*(*X*, *A*, *B*), *CT*0, *CT*1) finds the non-terminal symbol *res*(*r*1, 1, 2) as a subset of *CT*0. The variable *CT1* is substituted by the difference set which does not contain *resistor*(*r*1, 1, 2). The third subgoal *subset*(*res*(*Y*,*B*,*C*),*CT*1,_) finds the non-terminal symbol *res*(*r*2, 2, 3) in the circuit *CT*1 as:

```
?- ca39(CT0),
   subset(res(X,A,B),CT0,CT1),
   subset(res(Y,B,C),CT1,_).
X = r1
Y = r2
A = 1
B = 2
C = 3
```

Figure 2 Resistors connected in series



But, this is not a right answer because the central node *B*(= 2) of series connection is also connected to another resistor *r*4. The central node of series connection must not be connected to any other elements. This is realised by introducing the condition of absence into grammar rules such as:

$$anyElm(X,A)\ \rightarrow\ [terminal(X,A)];\ res(X,A,_). \quad (7)$$

$$rSeries(rs(X,Y),A,C)\ \rightarrow\ res(X,A,B),\ res(Y,B,C),\ not\ anyElm(_,B). \quad (8)$$

The grammar rule (7) defines the non-terminal symbol '*anyElm*(*X*, *A*)' which represents any element *X* connected to the node *A*. The grammar rule (8) defines the non-terminal symbol '*rSeries*(*rs*(*X*, *Y*), *A*, *C*)' which represents two resistors *X* and *Y* connected in series with a condition that any other elements must not be connected to the central node *B*. Here, '*rs*(*X*, *Y*)' is the name given to the series connection of resistors (Skolem function). These grammar rules (7) and (8) are converted to the definite clauses (7)' and (8)'.

$$subset(anyElm(X,A),S0,S1)\ :-\ member(terminal(X,A)S0,S1);\ subset(res(X,A,_),S0,S1). \quad (7)'$$

$$subset(rSeries(rs(X,Y),A,C),S0,S2)\ :-\ subset(res(X,A,B),S0,S1),\ subset(res(Y,B,C),S1,S2),\ not\ subset(anyElm(_,B),S2,_). \quad (8)'$$

The following goal successfully fined the series connection of resistors in the circuit *ca*39 as:

```
?- ca39(CT0),
   subset(rSeries(X,A,C),CT0,_).
X = rs(r2,r3),
A = 2,
C = 4
```

The conversion with this extension '*not*' is realised by adding the following two lines 04 and 05 after the line 03 of List 1. Here, 'not' must be declared as a prefix-operator.

```
04  conv(not [Component],
         not member(Component, S0,_),
         S0, S0)  :-  !.
05  conv(not Component,
         not subset(Component, S0,_),
         S0, S0)  :-  !.
```

4.2 Condition for existence

Consider the circuit design process in contrast with sentence generation. Suppose a circuit goal generates two current sources as its sub-goals. Each current source needs a regulated voltage source, so two voltage regulators are generated. When one of the voltages is derived from the other, an engineer may combine two voltage regulators into one voltage regulator for simplicity. That is, he has the ability to use context dependent circuit generation rules.

In our system, the V_{be}-voltage regulator (Figure 3) and the sink-type current source (Figure 4) are defined by the grammar rules (9) and (10), respectively.

$$vbeReg(vreg(Q,R),In,Com,Out) \rightarrow \quad res(R,In,Out), \quad [npnTr(Q,Out,Com,Out] \tag{9}$$

$$cSource(sink(VR,Q),In,Com) \rightarrow \quad vbeReg(VR,_,Com,B), \quad [npnTr(Q,B,Com,In)]. \tag{10}$$

Figure 5 shows a part of an analogue IC circuit. Two transistors q3 and q5 form two current sources (sink-type) sharing one V_{be}-voltage regulator *vreg*(q4, r1). When a goal needs to identify two current sources in parsing, the voltage regulator is used to identify one current source, and no voltage regulator remains to identify another current source. So the goal fails.

Figure 3 V_{be}-voltage regulator

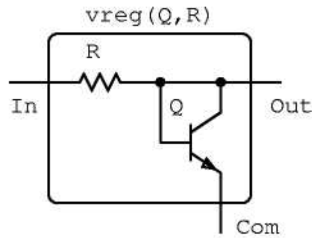


Figure 4 Current source (sink-type)

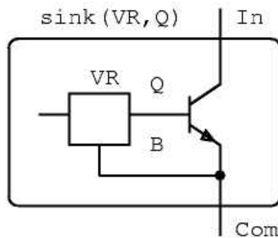
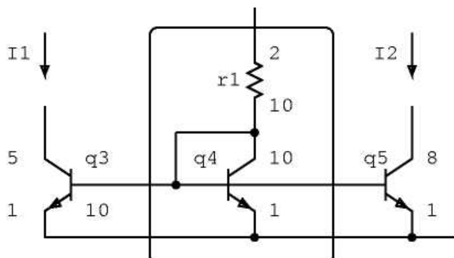


Figure 5 Two current sources sharing one voltage regulator



In order to solve this problem, we introduce a new mechanism which tests an existence of a symbol but does not reduce to upper non-terminal symbols as:

$$cSource(sink(VR,Q),In,Com) \rightarrow \quad test\ vbeReg(VR,_,Com,B), \quad [npnTr(Q,B,Com,In)]. \tag{11}$$

The grammar rule (11) is converted to (11)'.

$$subset(cSource(sink(VR,Q),In,Com),S0,S1) :- \quad subset(vbeReg(VR,_,Com,B),S0_), \quad member(npnTr(Q,B,Com,In),S0,S1). \tag{11'}$$

Though the current source has two definitions (10) and (11), an appropriate rule is selected in parsing by non-deterministic mechanism of logic programming.

This extension is realised by adding the following two lines 06 and 07 into List 1. The 'test' must be declared as a prefix-operator.

```
06  conv(test [Component],
         member(Component,S0,_),
         S0, S0)  :-  !.
07  conv(test Component,
         subset(Component,S0,_),
         S0, S0)  :-  !.
```

5 Extension for equivalent circuits

In circuit analyses, we often rewrite object circuits into equivalent circuits such as DC equivalent circuits and small signal equivalent circuits. This can be done by combining a parsing process which removes some elements and a generating process which adds some elements. Usually parsing programmes also work generating by exchanging input and output in logic programming, so we can consider an operator *invert* which exchanges input and output of the predicate *subset*. The following grammar rule defines a rewriting process *A* which first identifies the non-terminal *B* in the object circuit *S*0 and removes *B* to make *S*1, then adds *C* to the circuit *S*1 to make *S*2.

$$A \rightarrow B, invert\ C. \tag{12}$$

$$subset(A,S0,S1) :- \ subset(B,S0,S1), \quad subset(C,S2,S1). \tag{12'}$$

But this method is not so good. As we use Prolog lists to represent word-order free sentences, the same sentence has many different expressions of lists consisting permutation of wards which cause useless backtracking.

Instead of the method, we introduce a simple mechanism for adding elements as (13).

$$A \rightarrow B, add\ [C]. \tag{13}$$

$$subset(A,S0,S2) :- \ subset(B,S0,S1), \quad S2 = [C|S1]. \tag{13'}$$

This simple method becomes useful, because equivalent circuits of devices usually consist of a small number of elements. As this extension enables to rewrite circuits during parsing, the predicate *subset*(A, $S0$, $S2$) no longer means 'A is a subset of $S0$'. This extension is realised by adding the following line 08 to the List 1 with the declaration of prefix operator 'add'.

The whole DCSG-converter is shown in Appendix A. In the appendix, the predicate 'read_grammar_assert(G)' reads the file G of grammar rules. Each rule is converted into a definite clause, and asserted.

```
08 conv(add  [Component],
        S1 = [Component|S0],S0, S1) :- !.
```

6 Extensions for circuit functions

6.1 Circuit structures and functions

We assume that circuit functions are the meaning of the syntactic structure of the circuit. The circuit functions we consider are the electrical behaviours that are useful to circuit designers or users. These electrical behaviours 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.

In order to separate these semantic information from syntactic structures, the new circuit grammar has additional fields for the semantic information. In the paper of Tanaka (2010), dependencies on voltages and currents are coded using the semantic fields, then these electrical dependencies are derived through parsing circuit structures.

The electrical dependencies are represented by a set of compound terms which can also be viewed as a word-order free sentence. If we define grammar rules for the language describing circuit functions, we can analyse the derived dependencies using the grammar rules.

6.2 Semantic term in left-hand side

The semantic information such as electrical states and voltage-current dependencies associated with circuits are placed in curly brackets as:

$$A, \{F_1, F_2, ..., F_m\} \rightarrow B_1, B_2, ...B_n. \tag{14}$$

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

$$\begin{aligned} & ss\left(A, S_0, S_n, E_0, \left[F_1, F_2, ..., F_m \middle| E_n\right]\right) :- \\ & \qquad ss(B_1, S_0, S_1, E_0, E_1), \\ & \qquad ss(B_2, S_1, S_2, E_1, E_2), \\ & \qquad \ldots, \\ & \qquad ss(B_n, S_{n-1}, S_n, E_{n-1}, E_n). \end{aligned} \tag{14'}$$

Since the conversion differs from that used in DCSG, we use the predicate '*ss*' instead of '*subset*'.

When the rule (14)' is used in parsing, the goal $ss(A, S_0, S_n, E_0, E)$ is executed, where the variable S_0 is substituted by an object set (object circuit) and the variable E_0 is replaced by an empty set. The subsets '$B_1, B_2, \ldots, B_n$' 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 $\{F_1, F_2, \ldots, F_m\}$, which is the meaning associated with symbol A, is added and all of the semantic information is put into E.

As each variable of $S_0, S_1, \ldots, S_n$ is substituted by unknown part of the object circuit, they decrease according to identifying $B_1, B_2, \ldots, B_n$. While, each variable of $E_0, E_1, \ldots, E_n$ is substituted by known semantic informations through parsing. So they increase. Namely, circuit structures change to semantic informations through parsing.

6.3 Semantic term in right-hand side

Semantic terms on the right-hand side define the semantic conditions such as electrical states of transistor which enables the circuit function. For example, the following rule (15) is converted into the definite clause (15)' as follows.

$$A \rightarrow B_1, \{C\}, B_2. \tag{15}$$

$$\begin{aligned} & ss(A, S_0, S_2, E_0, E_2) :- \\ & \qquad ss(B_1, S_0, S_1, E_0, E_1), \\ & \qquad member(C, E_1, _), \\ & \qquad ss(B_2, S_1, S_2, E_1, E_2). \end{aligned} \tag{15'}$$

When the clause (15)' is used in parsing, the semantic condition C is tested to see if the semantic information E_1 satisfies this condition after identifying the symbol B_1. If it succeeds, the parsing process goes on to identify the symbol B_2.

Appendix B shows the whole program list of the converter for new circuit grammar with semantic terms.

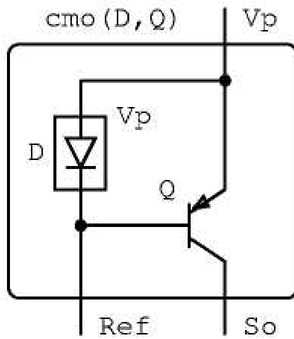
6.4 Examples on circuit grammar

The following grammar rule (16) defines the non-terminal symbol '*currentMirrorSource*(*cmo*(*D*, *Q*), *Ref*, *Vp*, *So*)' for the source type current-mirror circuit shown in Figure 6. The circuit generates the same current with a reference current. The compound term *cmo*(*D*, *Q*) is the name given to the circuit which consists of a diode-connected transistor D and a PNP-transistor Q. The term *i*(*cmo*(*D*,*Q*), *Ref*) represents the reference currents from the circuit *cmo*(*D*, *Q*) to the node *Ref*. The term *i*(*cmo*(*D*,*Q*), *So*) represents the generated current from the circuit *cmo*(*D*, *Q*) to the node *So*. The semantic term '*cause*(*i*(*cmo*(*D*, *Q*), *Ref*), *i*(*cmo*(*D*, *Q*),

So), *cmo*(*D*, *Q*))' in the left-hand side of the grammar rule shows a causality of these two currents supported by the circuit *cmo*(*D*, *Q*).

$$
\begin{aligned}
& currentMirrorSource(cmo(D,Q), Ref, Vp, So), \\
& \{cause(i(cmo(D,Q), Ref), \\
& \quad i(cmo(D,Q), So), cmo(D,Q)), \\
& cause(i(cmo(D,Q), Ref), \\
& \quad i(D, Ref), cmo(D,Q)), \\
& equiv(i(cmo(D,Q), So), i(Q, So))\} \rightarrow \\
& \qquad dtr(D, Vp, Ref), \\
& \qquad \{state(D, conductive)\}, \\
& \qquad pnpTr(Q, Ref, Vp, So), \\
& \qquad \{state(Q, active)\}.
\end{aligned} \tag{16}
$$

Figure 6 Current mirror (source type)



The semantic term *state*(*D*, *conductive*) in the right-hand side of the grammar rule shows an electrical condition that the diode-connected transistor *D* must be in the conductive state. The semantic term *state*(*Q*, *active*) shows another electrical condition that the transistor *Q* must be in the active state. These electrical conditions enable the current-mirror function.

If an object circuit has the syntactic structures represented by *dtr*(*D*,*Vp*,*Ref*) and *pnpTr*(*Q*, *Ref*, *Vp*, *So*), and the semantic information of the object circuit satisfies these electrical conditions, the current-mirror circuit *cmo*(*D*, *Q*) is identified in the object circuit, and electrical dependencies such as '*cause*(...)' are derived.

7 Conclusions

The DCSG-converter was originally developed using Yacc/Lex in the paper of Tanaka and Bartenstein (1999). The DCSG-converter in Yacc/Lex analyses grammar rules as a context-free language using bottom-up method, while the converter in Prolog analyses DCSG rules using pattern matching called unification. The converter in Prolog has a simple structure compared with the one in Yacc/Lex. In the DCSG conversion, terminals and non-terminals are directly converted to components of Prolog clauses without change. Therefore, inside structures of terminal and non-terminal symbols are not important for the conversion, while the converter in Yacc/Lex must analyse their structures to identify terminals and non-terminals. The converter in Prolog does not need to define a lexical analyser.

The DCSG-converter is useful not only for circuit analysis, but also for avoiding a common looping problem in Prolog programs. Usually, a Prolog program consists of two kinds of definite clauses, called facts and rules 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. That is, the set of facts corresponds to the set of words, 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. When we change problems from backward chaining to top-down parsing by simply replacing facts with words and rules with grammar rules, 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 by Tanaka (1991, 2007).

The new circuit grammar has fields for semantic information. The circuit grammars not only define syntactic structures of circuits but also define relationships to circuit functions as meaning of the structures. Therefore, if a circuit is given, not only its structure but also its functions are derived through parsing.

Circuit functions are basically defined on behaviour of voltages and currents of the circuits. So we try to define grammar rules to formalise knowledge on voltages and currents dependencies in an earlier paper (Tanaka 2010). The derived electrical dependencies through parsing will be useful for understanding electrical behaviour and troubleshooting of the circuit. The derived dependencies are represented by a set of compound terms which can also be viewed as a word-order free language. We can easily define a non-terminal symbol in the language which represents transitivity of the electrical dependencies.

The electrical dependencies, however, only describe the shallow behaviour of circuits. We are currently developing grammar rules which define circuit behaviours and functions more precisely using this converter.

Acknowledgements

Prolog programs in Appendix were originally developed using C-Prolog and MINERVA. We are also using GNU-Prolog and SWI-Prolog.

References

Pereira, F.C.N. and Warren, D.H.D. (1980) 'Definite clause grammars for language analysis', *Artificial Intelligence*, Vol. 13, pp.231–278.

Tanaka, T. (1991) 'Definite clause set grammars: a formalism for problem solving', *Journal of Logic Programming*, Vol. 10, pp.1–17.

Tanaka, T. (1993) 'Parsing circuit topology in a logic grammar', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, pp.225–239.

Tanaka, T. (2007) 'A logic grammar for circuit analysis – problems of recursive definition', *LNAI*, Vol. 4693, pp.852–860.

Tanaka, T. (2009) 'Circuit grammar: knowledge representation for structure and function of electronic circuits', *International Journal of Reasoning-based Intelligent Systems*, Vol. 1, pp.56–67.

Tanaka, T. (2010) 'Deriving electrical dependencies from circuit topologies using logic grammar', *International Journal of Reasoning-based Intelligent Systems*, Vol. 3, pp.28–33.

Tanaka, T. and Bartenstein, O. (1999) 'DCSG-converters in Yacc/Lex and Prolog', *Proceedings of the 12th International Conference on Applications of Prolog*, pp.44–49.

Appendix A Converter for DCSG

```
:-  op(900, fx, not).
:-  op(900, fx, test).
:-  op(900, fx, add).
dcsgConv((Lhs --> Rhs),
          (Head :- Body)) :-
         conv(Lhs,Head,S0,S1),
         conv(Rhs,Body,S0,S1).
conv((CompoA,CompoB),
     (CA,CB),S0,S1) :-
            !,
            conv(CompoA,CA,S0,S),
            conv(CompoB,CB,S,S1).
conv((CompoA;CompoB),
     (CA;CB),S0,S1) :-
         !,
         conv(CompoA,CA,S0,S1),
         conv(CompoB,CB,S0,S1).
conv(not [Component],
     not member(Component,S0,_),
     S0,S0) :- !.
conv(not Component,
     not subset(Component,S0,_),
     S0,S0) :- !.
conv(test [Component],
     member(Component,S0,_),
     S0,S0) :- !.
conv(test Component,
     subset(Component,S0,_),
     S0,S0) :- !.
conv(add [Component],
     S1=[Component|S0],
     S0,S1) :- !.
conv([Component],
     member(Component,S0,S1),
     S0,S1) :- !.
conv(Component,
     subset(Component,S0, S1),
     S0,S1) :- !.
read_grammar_assert(G) :-
     see(G),read_assert.
read_assert :-
     read(X),
   (not X=end_of_file,
    dcsgConv(X,Y),
    assertz(Y),
    read_assert ; true).
```

Appendix B Converter for circuit grammar

```
:- op(900, fx, not).
:- op(900, fx, test).
:- op(900, fx, add).
:- op(950, fx, quote).
cgConv((Lhs --> Rhs),
        (Head :- Body)) :-
    lconv(Lhs,Head,C0,C1,E0,E1),
    rconv(Rhs,Body,C0,C1,E0,E1).
lconv((Compo,{Es}),
      ss(Compo,C0,C1,E0,E2),
      C0,C1,E0,E1) :-
    !,
    makelist(Es, E1, E2).
lconv(Compo,
    ss(Compo,C0,C1,E0,E1),
    C0,C1,E0,E1) :- !.
makelist((E,Es), E1, [E|E2]) :-
    !,
    makelist(Es,E1,E2).
makelist(E,E1,[E|E1]).
rconv((CompoA, CompoB),
      (CA,CB),
       C0,C1,E0,E1) :-
    !,
    rconv(CompoA,CA,C0,C,E0,E),
    rconv(CompoB,CB,C,C1,E,E1).
rconv((CompoA;CompoB),
      (CA;CB),
```

```
      C0,C1,E0,E1) :-
    !,
    rconv(CompoA,CA,C0,C1,E0,E1),
    rconv(CompoB,CB,C0,C1,E0,E1).
rconv(not [Compo],
     (not member(Compo,C0,_),
       C1=C0,E1=E0),
     C0,C1,E0,E1) :- !.
rconv(not Compo,
      (not ss(Compo,C0,_,E0,_),
        C1=C0,E1=E0),
     C0,C1,E0,E1) :- !.
rconv(test [Compo],
      (member(Compo,C0,_),
        C1=C0,E1=E0),
      C0,C1,E0,E1) :- !.
rconv(test Compo,
      (ss(Compo,C0,_,E0,_),
        C1=C0,E1=E0),
      C0,C1,E0,E1) :- !.
rconv(quote Compo,
      (Compo, C1=C0, E1=E0),
      C0,C1,E0,E1) :- !.
rconv(add [Compo],
      (C1=[Compo|C0], E1=E0),
      C0,C1,E0,E1) :- !.
rconv({Compo},
      (member(Compo,E0,_),
        C1=C0, E1=E0),
      C0,C1,E0,E1) :- !.
rconv([Compo],
      (member(Compo,C0,C1),E1=E0),
      C0,C1,E0,E1) :- !.
rconv(Compo,
      ss(Compo,C0,C1,E0,E1),
      C0,C1,E0,E1) :- !.
read_grammar_assert(G) :-
    see(G),read_assert.
read_assert :-
    read(X),
   (not X=end_of_file,
    cgConv(X,Y),
    assertz(Y),
    read_assert ; true).
```