Formal specifications and decomposition of logic systems for purposes of analysis, synthesis and diagnostics

Ján Bača, Juraj Giertl, Vladimír Chladný

Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovakia
Phone: +421 55 602 2573, Fax: +421 55 633 0115
E-mails: Jan.Baca@tuke.sk, Juraj.Giertl@tuke.sk, Vlado.Chladny@tuke.sk

Abstract

The contribution deals with different formal specifications of logic system that are used for solving of analysis, synthesis and diagnostics tasks. Particular attention is given to analysis of applicability of separate description for the purpose of system decomposition. The data structure for algebraic expressions with context-free grammar utilization is also defined in the contribution. We also propose algorithm of de/composition of logical systems specified by this expression and finally a procedure for identical and isomorphic circuit search.

Introduction

In relation to logic systems the tasks of synthesis, analysis and diagnostics are being solved. In separate tasks the systems are engaged by different descriptions. The definitions of the system can be divided into two basic groups. In the first group the system is defined upon its function what means functional dependence between its values. In the second group the system is defined upon its structure what means upon the connections between the system components. In these cases the function of the system can be derived from the functions of separate components and connections between them. To understand the dependence between separate system descriptions it is essential to realize, that the given function can be realized by different structures, however the given structure of the circuit realizes a unique determined function. So that the transformation between different systems description could be distinct, sometimes additional information is needed.

If the complexity $z(n)$ of the solution of the task doesn't rise with its size n in linear $c(n) \neq O(n)$, but polynomial $c(n) = O(n^k)$, or exponential $c(n) = O(g^n)$ way, then to reducing of the total solution complexity the decomposition of the system $S(n)$ into subsystems $S_1(n_1)$, $S_2(n_2)$, ..., $S_p(n_p)$, $n_i < n$, is used with great advantage. Decomposition makes sense if lowering of the solution time in decomposed systems is greater than increase of the solution time related with decomposition t_{ds} and reverse composition t_{cs} of the system.

$$t(S(n)) > \sum_{i-i}^{p} t(S_i(n_i)) + t_{ds} + t_{cs}$$

1 Formal systems specification

The starting specification of systems is often being expressed by the function of the system described in common spoken language. This form of description has a disadvantage in the fact that for an exact description of a function a very complex verbal description is needed. This is the reason why for exact function description formal description as input – output sequences (normal form), regular expressions, programs, Petri nets, finite state automata are used [1].

The task of system synthesis is related to the system specification by function that means determination of a system structure, which realizes given function. Decomposition of the given function is being made in relation to this task with the aim to simplify the synthesis.

Structure of the designed systems is mostly represented by a structural scheme. This scheme represents graphical elements, which the system consist of and connections between them. Inner representation of the graphical presentation is very complex and is hidden to the user therefore the text description of the scheme – netlist is used where the lists of elements and relations between them are mentioned.

The task of system analysis is related to the system description by its structure; that means the determination of a system function that is equivalent to the assigned structure. Other task is the system diagnostics, where also the system decomposition is used with great advantage.

Between function assignment and description of its structure there exists a whole row of other possible system descriptions that describe the structure of a system in various ways. The structure of sequential logic system is often described by a set of excitation and output functions that represent combinational section of the system. Combinational logic systems present specific case of sequential systems without memory section and are fully described by a set of output functions.

Netlist and a set of excitation and output functions represent descriptions that are suitable for presentation of logic systems by solution of basic tasks of their analysis, synthesis and diagnostics. That's why we will deal with these specifications later.

Transformation of a structure to algebraic expression of circuit functions expressed by netlist is described in [2]. Reverse transformation of algebraic function expressions of a circuit into the circuit's structure is described in [3]. Due to this fact we can obtain algebraic expression of the circuit's excitation and output functions independently to the fact the circuit was expressed by its function or by its structure. This expression represents the strings of symbols that meet the rules defined by the grammar. We can solve many tasks related to analysis, synthesis and diagnostics of logic circuits by analyzing the mentioned strings.

One of the main tasks is the decomposition of string to substrings. According to tasks that are to be solved, specific features have to be kept by created substrings. For example factorization demands searching of identical substring in a string corresponding to one function, pre-realization - searching of identical substrings in a string corresponding to more functions, diagnostics – searching of isomorphic substring in a string corresponding to one function or more functions.

System $S_q (u_1, u_2, \dots, u_m)$ is isomorphic to system $S_p (v_1, v_2, \dots, v_m)$ if S_q can be obtained from S_p by substitution $u_i \leftarrow v_j$, $i,j \in \langle 1, m \rangle$. Sets $U = (u_1, u_2, \dots, u_m)$, $V = (v_1, v_2, \dots, v_m)$ represent input, inner and output values of the systems.

Systems $S_p (v_1, v_2, \dots, v_m)$ and $S_q (u_1, u_2, \dots, u_m)$ are identical, if they are isomorphic to each other and $u_i \equiv v_j$.

Excitation and output functions represent Boolean functions (B-functions). Algebraic expressions of B-functions represent a string of symbols that represent operands, operators and other symbols defining the priority of operations.

2 Definition of data structure for algebraic expression of a logic system function

By decomposition of logic systems it is necessary to differentiate between combination and sequential logic circuits. In the case of combination circuits the algebraic expression – disjunctive normal form (DNF), conjunctive normal form (CNF), Sheffer's form (realization by the NAND elements), Pierce's form (realization by the NOR elements), the bracket expression (result of factorization), expression by non-equivalence (realization by the XOR elements) or their combination is fully sufficient. Function of combination circuit can by expressed as follows:

identifier_of_function = algebraic_expression

For the function identifier, usually the symbol ***F***, or other upper case letter of alphabet is used. Algebraic expression is a string consisting of variable identifiers, operators belonging to individual operations and brackets that designate operations priority. For variable identifiers lower case letters are used that can be supplemented by numbers or other symbols.

For combination circuit with more outputs, the equal expression as mentioned before is used, but with the difference, that functions of separate outputs are written in individual lines. Function identifiers are usually supplemented with the output number.

Typical model of a sequential circuits function is the finite state automat, or its transition and output table. Structure of the circuit representing the finite state automat is given by its combinational section and a memory section. The memory section is decomposed into elementary automats matching to individual inner variables of the sequential circuit. Only the combinational section represented by excitation and output functions will be decomposed, and due to this fact the algebraic expression can be used for its description. The equal expression to the description of combination circuit with many outputs is used. Every function is written in individual line, where for identifier of the excitation function symbols ***D, T, R, S, J, K,*** are used depending on the used type of an elementary automat, and for the output function identifier symbol ***Y*** is used. Number of the elementary automat, or the output number is supplemented to every identifier.

2.1 Context-free grammar for the language of B-functions

Structure defined in this way can be formally defined with context-free grammar [4] consisting of the following rules (terminal symbols are written in normal type, non-terminal in italics):

START → *START NEW_LINE FUNCTION_ID = EXPR* | *FUNCTION_ID = EXPR*

EXPR → *EXPR* + *EXPR* | *EXPR* ↑ *EXPR* | *EXPR* ↓ *EXPR* | *EXPR* ⊕ *EXPR* | *EXPR EXPR* | (*EXPR*) | ¬ *EXPR* | *VAR_ID*

FUNCTION_ID → A *NUMBER* | … | Z *NUMBER* | A | … | Z

VAR_ID → a *NUMBER* | … | *z NUMBER* | a | … | z

NUMBER → *NUMBER* 0 | ...| *NUMBER* 9 | 1| ...| 9

NEW_LINE → \n

2.2 Lexical analysis of input

The objective of the lexical analysis is to find out if only allowed symbols are at input. In order to achieve simpler implementation it is suitable to predefine some symbols used for operator marking. Instead of Sheffer's algebra operator '↑' symbol '|' is used, instead of Pierce's algebra operator '↓'symbol '!' is used, instead of the non-equivalency operator '⊕', symbol '#' is used and instead of the operator of negation '¬', symbol '\' is used.

It is suitable to use the tool called "Lex" to create a lexical analyzer. It can generate the source code in "C" language based on specification file. This can be later used either standalone, or in combination with a syntax analyzer. The listing of specification file example created upon the principles described in [5] is the following:

```
DIGIT       [0-9]
PDIGIT      [1-9]
NUMBER      {PDIGIT}|{PDIGIT}({DIGIT})+
FID         [A-Z]
VAR         [a-z]

%%
" "         ;
"\n"        {return NEW_LINE;}
{NUMBER}    {return NUMBER;}
{FID}       {return FID;}
{VAR}       {return VAR;}
"="         {return ASSIGN;}
"("         {return L_BRACKET;}
")"         {return R_BRACKET;}
"+"         {return OR;}
"|"         {return NAND;}
"!"         {return NOR;}
"#"         {return XOR;}
"\\"        {return NOT;}
.           {printf("lexical error: wrong input:\"%s\"\n",yytext);}
%%
```

2.3 Syntactical analysis of input

The syntactical analyzer task is to find out, if the symbols recognized by lexical analyzer are ordered in accordance with the formal grammar described in section 2.2.

It is suitable to use the tool called "Yacc" to create a syntax analyzer. It can generate the source code in "C" language based on specification file. It is necessary then to compile this code into executable form using "C" language compiler. The compiler produces the executable program that can determine whether the given input belongs to the language described by the formal grammar, or can transform the input. In our case the transformation of input means decomposition into normal forms. The listing of part of specification file example is the following, where formal grammar based upon principles described in [5] is

written. This file has to include a code in “C” language, that will provide an input file opening, syntax analysis, semantic analysis, code generation, or other functions related to the task of system decomposition.

```
%token VAR NUMBER FID
%token ASSIGN L_BRACKET R_BRACKET NEW_LINE
%token NAND NOT OR NOR XOR
%start START

%%
START       : START NEW_LINE FUNCTION_ID ASSIGN EXPR
            | FUNCTION_ID ASSIGN EXPR
            ;
EXPR        : EXPR OR EXPR
            | EXPR NAND EXPR
            | EXPR NOR EXPR
            | EXPR XOR EXPR
            | EXPR EXPR
            | L_BRACKET EXPR R_BRACKET
            | NOT EXPR
            | VAR_ID
            ;
FUNCTION_ID : FID
            | FID NUMBER
            ;
VAR_ID      : VAR
            | VAR NUMBER
            ;
%%
```

3 De/composition of logical systems

3.1 Decomposition of algebraic expression into substrings

Decomposition of algebraic expression comes out from an expression described by the grammar (section 2.1), which is analyzed in bottom-up way. The result of decomposition is the decomposition of an algebraic expression into substrings that are realizable with only one elementary logic gate.

Algorithm of decomposition of an algebraic expression into substrings can be described in the following steps:

For a function of every primary output do following:

1. Search substring realizable with one elementary logic gate. Gates NOT, AND, OR, NAND, NOR, XOR, are considered.
2. Substitute found substring by one variable, marked by symbol ***x*** and two numeric values. First indicates the level of the substring, the second serves as an identifier for the substring within the given level. Every substring can include only primary variables and variables used for substitution of lower level substrings.
3. Express the function with substitution variables.
4. Repeat steps 1, 2, 3, until the expression is reduced to only one elementary logic gate.

Example of an application of the presented algorithm for the circuit realizing full adder circuit can be following (Figure 1). The circuit has two outputs – sum and carry into higher level:

$$s = \neg(\neg(a)b + a\neg(b))c + (\neg(a)b + a\neg(b))\neg(c)$$

$$p = ab + (\neg(a)b + a\neg(b))c$$

Function	**Substitution**
s = ¬(¬(a)b + a¬(b))c + (¬(a)b + a¬(b))¬(c)	x[1,1] = ¬(c)
s = ¬(¬(a)b + a¬(b))c + (¬(a)b + a¬(b)) x[1,1]	x[1,2] = ¬(b)
s = ¬(¬(a)b + a¬(b))c + (¬(a)b + a x[[1,2]) x[1,1]	x[2,1] = a x[1,2]
s = ¬(¬(a)b + a¬(b))c + (¬(a)b + x[2,1]) x[1,1]	x[1,3] = ¬(a)
s = ¬(¬(a)b + a¬(b))c + (x[1,3] b + x[2,1]) x[1,1]	x[2,2] = x[1,3] b
s = ¬(¬(a)b + a¬(b))c + (x[2,2] + x[2,1]) x[1,1]	x[3,1] = (x[2,2] + x[2,1])
s = ¬(¬(a)b + a¬(b))c + x[3,1] x[1,1]	x[4,1] = x[3,1] x[1,1]
s = ¬(¬(a)b + a¬(b))c + x[4,1]	x[1,4] = ¬(b)
s = ¬(¬(a)b + a x[1,4])c + x[4,1]	x[2,3] = a x[1,4]
s = ¬(¬(a)b + x[2,3])c + x[4,1]	x[1,5] = ¬(a)
s = ¬(x[1,5] b + x[2,3])c + x[4,1]	x[2,4] = x[1,5] b
s = ¬(x[2,4] + x[2,3])c + x[4,1]	x[3,2] = (x[2,4] + x[2,3])
s = ¬(x[3,2])c + x[4,1]	x[4,2] = ¬(x[3,2])
s = x[4,2] c + x[4,1]	x[5,1] = x[4,2] c
s = x[5,1] + x[4,1]	x[6,1] = x[5,1] + x[4,1]
s = x[6,1]	
p = ab + (¬(a)b + a¬(b))c	x[1,6] = ¬(b)
p = ab + (¬(a)b + a x[1,6])c	x[2,5] = a x[1,6]
p = ab + (¬(a)b + x[2,5])c	x[1,7] = ¬(a)
p = ab + (x[1,7] b + x[2,5])c	x[2,6] = x[1,7] b
p = ab + (x[2,6] + x[2,5])c	x[3,3] = (x[2,6] + x[2,5])
p = ab + x[3,3] c	x[4,3] = x[3,3] c
p = ab + x[4,3]	x[1,8] = ab
p = x[1,8] + x[4,3]	x[5,2] = x[1,8] + x[4,3]
p = x[5,2]	

3.2 Composition of logic system from substrings

A composition of logic system comes out of a system of substrings obtained by a decomposition of algebraic expression (section 3.1). The result of a composition is a system of algebraic expressions of all circuit outputs functions and algebraic expression of system modules outputs functions. Except the system of substrings the level of modules, which the composed circuit should consist of is the input for the algorithm. Determination of modules level depends upon a type of used logic elements and upon a task which is the de/composition done for.

Algorithm of logic system composition from substrings representing one-level circuits can be described in the following steps:

For every primary output function of the circuit do following:

1. Let ***n*** be level of function, let ***m*** be level of modules the designed circuit consist of. Let ***m_temp* = *n* – (*n* mod *m*)**.
2. Substitute all modules of higher level than ***m_temp*** by module composition of lower level by substitution of particular substrings into function expression until the function is not expressed only by modules outputs of ***m_temp*** be level or lower.
3. In this expression, substitute all modules with level lower than ***m_temp*** by substitution of particular substrings until the function is not expressed only by modules outputs ***x[m_temp,i]***, ***x[m_temp-m,i]***, primary variables, or modules whose level could be lower than ***m_temp***, but are not suitable for substitution as their outputs are also inputs on lower level of the circuit.
4. In this expression, for every module ***x[p,q]*** do following:
   4.1. If ***p*** ≥ ***m***, let ***m_temp* = *p* – *m***, else ***m_temp* = *p***
   4.2. Proceed to step 2

Example of an application of the presented algorithm for a full adder circuit is presented in next table. Functions of outputs ***s = x[6,1], p = x[5,2]*** gained by substitutions presented in section 3.1 are input for the algorithm. We chose ***m = 3,*** because the circuit has to be realized by logical elements NOT, AND, OR, and so gain a circuit consisting of three-level modules.

Function	**Substitution**
s = x[6,1] s = x[5,1] + x[4,1] s = x[4,2] c + x[4,1] s = ¬ (x[3,2])c + x[4,1] s = ¬ (x[3,2])c + x[3,1] x[1,1] s = ¬ (x[3,2])c + x[3,1] ¬ (c)	x[6,1] = x[5,1] + x[4,1] x[5,1] = x[4,2] c x[4,2] = ¬ (x[3,2]) x[4,1] = x[3,1] x[1,1] x[1,1] = ¬ (c)
x[3,2] = (x[2,4] + x[2,3]) x[3,2] = (x[1,5] b + x[2,3]) x[3,2] = (x[1,5] b + a x[1,4]) x[3,2] = (¬ (a) b + a x[1,4]) x[3,2] = (¬ (a) b + a ¬ (b))	x[2,4] = x[1,5] b x[2,3] = a x[1,4] x[1,5] = ¬ (a) x[1,4] = ¬ (b)
x[3,1] = (x[2,2] + x[2,1]) x[3,1] = (x[1,3] b + x[2,1]) x[3,1] = (x[1,3] b + a x[1,2]) x[3,1] = (¬ (a) b + a x[1,2]) x[3,1] = (¬ (a) b + a ¬ (b))	x[2,2] = x[1,3] b x[2,1] = a x[1,2] x[1,3] = ¬ (a) x[1,2] = ¬ (b)

p = x[5,2] p = x[1,8] + x[4,3] p = x[1,8] + x[3,3] c p = ab + x[3,3] c	x[5,2] = x[1,8] + x[4,3] x[4,3] = x[3,3] c x[1,8] = ab
x[3,3] = (x[2,6] + x[2,5]) x[3,3] = (x[1,7] b + x[2,5]) x[3,3] = (x[1,7] b + a x[1,6]) x[3,3] = (¬ (a) b + a x[1,6]) x[3,3] = (¬ (a) b + a ¬ (b))	x[2,6] = x[1,7] b x[2,5] = a x[1,6] x[1,7] = ¬ (a) x[1,6] = ¬ (b)

It is obvious that modules ***x[3,1], x[3,2]*** and ***x[3,3]*** have identical structure and so are realized only once (pre-realization). We can also find out that the module ***s*** is isomorphic to the module ***x[3,1]***, what becomes interesting from the diagnostic point of view. The procedure of search of identical and isomorphic modules is described in section 4. Structural scheme of a de/composed full adder circuit is shown in figure 1.

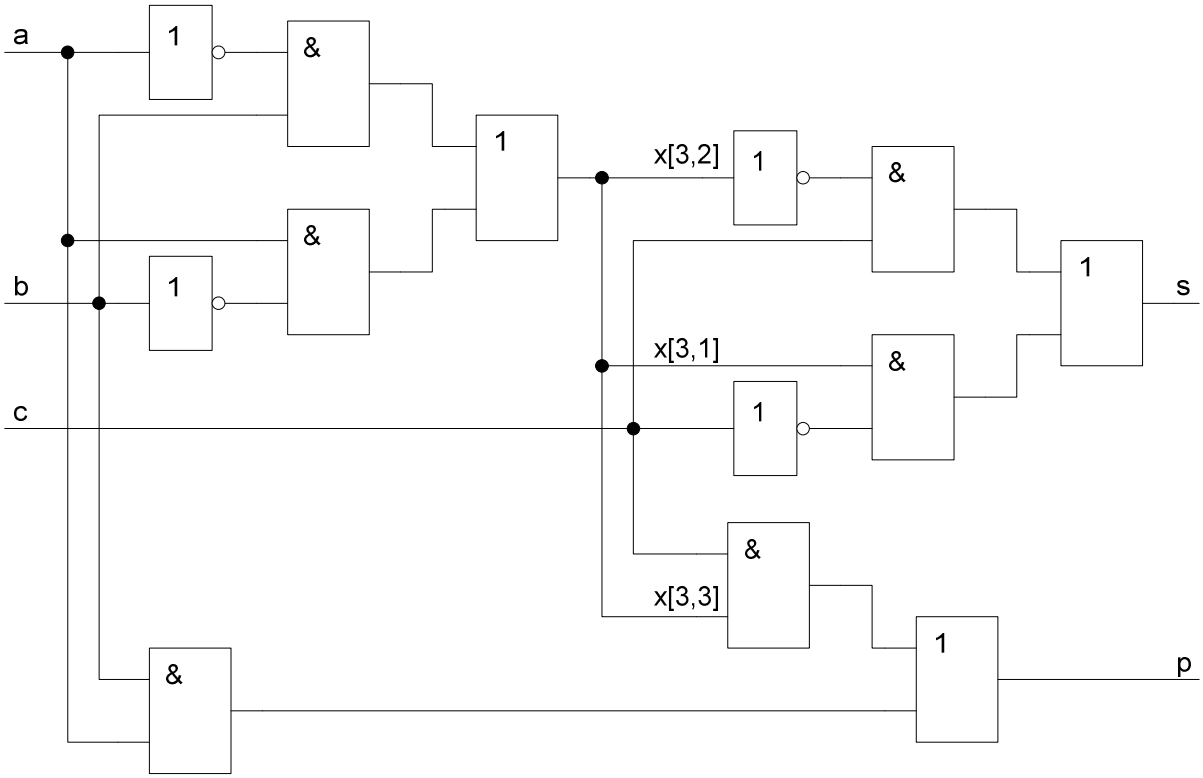

Figure 1. Structural scheme of a de/composed full adder circuit

4 Identification of identical and isomorphic circuits

Identification of identical circuits can be executed in the process of decomposition to one-stage circuits by comparison of expressions ***x[i,j]***, for ***j=1, 2, …, u***, where ***u*** is the count

of expression with the level ***i***. Identical expressions ***x[i,j]*** and ***x[i,j+v]*** are replaced by expression ***x[i,j]***.

By the comparison of expressions we have to take in account the validity of commutative rules. Expressions that differ only by permutation of variables are identical.

Detection of isomorphic circuits is executed after the substitution of identical expressions, so that the identical variables should come only with one labeling. To determination of the isomorphism of circuits it is necessary to find out, if the conditions of identity of relevant operation (type of the operator and number of operand) and also the condition of variables substitution are fulfilled. Detection of the last mentioned condition is very demanding, because the number of different substitutions is equal to the number of variables permutations. That's why the condition of substitution is detected only when all other conditions are fulfilled.

For easier detection of expressions identity and circuits' isomorphism it is useful to order the variables in them according to the alphabet and indexes and introduce the circuit's scheme evaluation [6].

5 Conclusion

Decompositions of strings and suggested possibilities of their application in different task in the areas of analysis, synthesis and diagnostics of logic circuits are proposed in the contribution. There are many other areas where the decomposition of strings is used [7]. Implementation of string decomposition into mentioned tasks exceeds the range of this contribution. Some of the mentioned tasks and also the generalization of grammar for other types of strings and processing of program systems for their lexical, syntactic and semantic analysis are in the plan for the future.

References

[1] *Bača, J.*: Logické systémy. EF TU Košice, 1992

[2] *Antalík, R., Bača, J., Beneš, B.*: Faults Detector and Locator. In.: Proceedings of the EMES 2003, Oradea-Felix Spa, Romania, May 29-31, 2003, pp. 7-12

[3] *Petrovský, B.*: Zostavovanie štruktúry logických obvodov. Diploma thesis. DCI FEI TU Košice, 2000.

[4] *Kollár, J., Havlice Z.*: Technológia jazykových systémov. Vydavateľstvo elfa, s.r.o., Košice, 2001. ISBN 80-89066-12-7

[5] *Levine, J. R., Mason, T., Brown, D.*: Lex & Yacc. O'Reilly & Associates, Inc., Sebastopol, California, 1992. ISBN 15-65920-00-7

[6] *Bača, J.*: Decomposition of Logic Circuits. In: Proceedings of International Conference Electronic Computers and Informatics '98, FEI TU Košice- Herľany, October 1998. pp.100-103. ISBN 80-88786-94-0

[7] *Tzeytlin, G. E.*: Vvedenije v algoritmiku, Sfera, Kyjev 1998, ISBN 960-7267-14-8