Parser of Input Data in Reliability Analysis based on Logical Differential Calculus

Patrik Rusnak

***Abstract*—**One of the principal data that are used as an input of many algorithms in reliability analysis is structure function, which defines the correlation between the system performance and performance of its components. It has been shown in several papers that this function can also be viewed as a Multiple-Valued Logic (MVL) function. This idea allows us to use methods related to the investigation of MVL functions in reliability analysis. One of them is logical differential calculus, which can be used to find circumstances under which degradation of a specific system component results in a decrease in system operation. MVL functions can be represented in several ways, e.g., truth table, graphic form, symbolic form. Computer processing of MVL functions requires a specific parser that is able to transform a given representation of a MVL function into a form that can be easily processed on the computer. In this paper, the symbolic representation is considered primarily. Parsing symbolic expressions can be done using several universal algorithms. One of them is shunting-yard algorithm invented by Edsger Dijkstra. Implementation of this algorithm for parsing MVL functions but also general mathematical expressions is presented in this paper.

***Keywords*—**reliability, logical differential calculus, shunting-yard algorithm.

I. Introduction

Reliability is an important characteristic of many, not only technical, systems. One of the current issues of reliability analysis is investigation of complex systems [1]. Such systems are composed of many components that are very different in their behavior. Typical examples of complex systems are healthcare systems, which consist of components of different nature that can be classified as hardware, software, and human factor [2], or distribution networks, which are composed of hardware elements with very different behavior [3]. Investigation of such systems requires development of new methods that take this diversity into account. One of the possible ways of how this diversity can be modeled is application of multi-state models.

A multi-state model of a system allows defining several performance levels at which the system or its components can operate. These levels are known as system/components states. A map that defines the dependency between components states and state of the system is known as structure function, and it has the following form [4]:

$$\phi(x_1, x_2, \dots, x_n) = \phi(\boldsymbol{x}):\ \{0,1, \dots, m-1\}^n \rightarrow \{0,1, \dots, m-1\}, \quad (1)$$

where n denotes number of system components, m agrees with the number of system/ components states (where state 0 means complete failure while $m-1$ agrees with perfect functioning), x_i is a variable representing state of the i-th system component, and $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ is a vector of components states (state vector).

The definition of the structure function agrees with the formal definition of Multiple-Valued Logic (MVL) function. This fact allows using some tools related to the analysis of MVL functions in reliability analysis of systems modeled using multi-state approach [5]. One of these tools is logical differential calculus.

Logical differential calculus has been developed for investigation of dynamic properties of MVL functions. Its central term is logic derivative. Several types of logic derivatives exists [6] but, in reliability analysis, the most useful one is Direct Partial Logic Derivative (DPLD). With

P. Rusnak, University of Zilina, Faculty of Management Science and Informatics, Zilina, Slovakia (e-mail: pattrik.rusnak@gmail.com).

respect to MVL function $\phi(\boldsymbol{x})$, this derivative is defined as follows [6]:

$$\partial\phi(j \to h)/\partial x_i(s \to r) = \begin{cases} 1, & \text{if } \phi(s_i, \boldsymbol{x}) = j \text{ AND } \phi(r_i, \boldsymbol{x}) = h \\ 0, & \text{otherwise} \end{cases}, \quad \text{for } s, r, j, h \in \{0,1,\dots,m-1\}, s \neq r, j \neq h, \tag{2}$$

where $(a_i, \boldsymbol{x}) = (x_1, x_2, \dots, x_{i-1}, a, x_{i+1}, \dots, x_n)$ for $a \in \{s, r\}$. As we can see, a DPLD allows identifying situations in which a given change of a MVL variable results in a given change of the investigated MVL function. In reliability analysis, this permits finding state vectors at which degradation (improvement) of component i from state s to r results in degradation (improvement) of system state from j to h. Knowledge of such state vectors plays a key role in many fields of reliability analysis because it allows evaluating influence of a considered component on system operation, what can be used in optimization of system reliability or in planning system maintenance. Theoretical background for these ideas has been developed in several works. In [5, 7], it has been shown how DPLDs can be used to investigate importance of individual system components or their states. Works [8, 9] presented application of logical differential calculus in finding minimal scenarios whose occurrence results in system degradation (improvement).

The aforementioned papers have introduced a complex framework for reliability analysis based on logical differential calculus. However, the detailed computer implementation of that framework has not yet been considered. This problem is considered in this paper. Our goal is to develop a complex tool that will implement all of the methods proposed in the previously mentioned papers. The tool has to be efficient and universal, i.e. it has to be able to run efficiently on any type of input data. In our case, the input data is a structure function represented using a MVL function. MVL function can be expressed in many forms, e.g., tabular or symbolic forms. If we want to work with symbolic forms, then we need a parser that will be able to transform symbolic expression in the form that can be processed on a computer. One of the possible solutions to this problem is use of shunting-yard algorithm invented by Edsger Dijkstra [10, 11]. Practical implementation of this algorithm is considered in the rest of the paper.

II. SHUNTING-YARD ALGORITHM

The main principle of the shunting-yard algorithm is to process mathematical expressions specified in infix notation (e.g.: $2^3 + 4 * 5$) to the form of a reverse polish notation (i.e., $23 \wedge 45 * +$) [12, 13]. In our implementation of this algorithm, we will use two stacks: the first stack named output stack will store the output set of tokens, the second stack named temporary stack will store functions, operators, parentheses, and function arguments separators to maintain the priority of every operation.

Token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. We will often refer to a token by its token name [12].

The stack used in the shunting-yard algorithm is an abstract data type, which is a collection of elements with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that has not been removed. Additionally, a peek operation may be defined, which gives access to the first element of the stack without modifying it. The order in which elements are removed from the stack (last in, first out) is the basis for its alternative name LIFO [14].

The algorithm will recognize all tokens in input text and insert them into one of the two stacks by following these rules:

- If the token is a number or constant, then push it onto the output stack.
- If the token is an operation, then push it onto the temporary stack. But before that happens, you need to check the priority of the operation on the top. If on the top of the temporary stack is operation with higher or equal priority and inserting operation is left associative or when inserting operation is right associative and on the top of the temporary stack is operation with higher priority, then the operation at the top is pulled from the temporary stack, and it is pushed to the output stack. This checking process is repeated, until checking condition is no longer valid or the temporary stack is empty.
- If the token is a function token, then push it onto the output stack. However, when token is a function parameters separator, then tokens from the temporary stack are popped and pushed to the output stack until at the top of the temporary stack is the beginning of the function. If the beginning of the function is not found, then the function and the input text are invalid.
- If the token is a left parenthesis, then push it onto the output stack. But if the token is a right parenthesis, then tokens from the temporary stack are popped and pushed to the output stack until at the top of the temporary stack is a left parenthesis. Pop the left parenthesis from the temporary stack, but not onto the output stack. If the temporary stack runs out without finding a left parenthesis, then there are mismatched parentheses.
- When there are no more tokens to read, pop all tokens from the temporary stack and push them to the output stack.

An illustrative example of parsing input text using the shunting-yard algorithm can be seen in Fig. 1 where input string $2 \wedge 3 + 4 * 5$ is parsed.

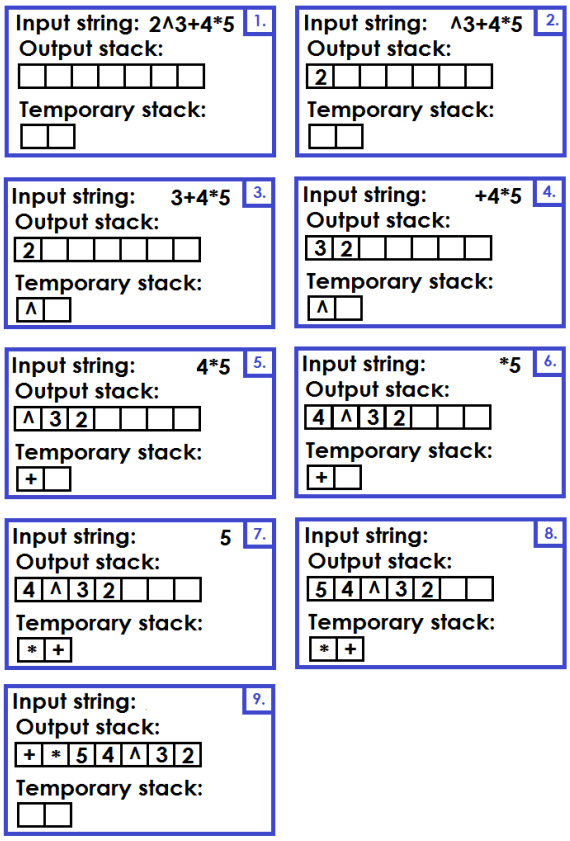

Fig. 1 Illustration of the shunting-yard algorithm

III. Building multi-way tree

After we successfully execute shunting-yard algorithm, we get all the tokens in reverse polish notation in the output stack. But if we want to perform some specific action with output stack, e.g., symbolic computations such as simplification, then it will be difficult. We need to figure out a better way to keep output tokens and take advantage from the output stack. We concluded that the best solution will be in building multi-way tree from the output stack.

A tree is an abstract data type or data structure that implements this abstract data type. A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root [14].

Building a multi-way tree is performed by algorithm, which illustration can be noticed in Fig. 2. Firstly, we must ensure that the output stack is not empty. If it is not empty, then we can begin the following algorithm:

1. Create a node and set this node as the root.
2. Call this this recursive operation that will operate as follows:
    2.1. Select a token from the output stack.
    2.2. If the token does not have arguments, then end.
    2.3. If the token has n arguments, then repeat these steps n times:
        2.3.1. Create a new node.
        2.3.2. Set the node as a son of the token.
        2.3.3. Run recursive operation over the node.

After the algorithm run, we get a multi-way tree, which will run demanding tasks much simpler and more practical.

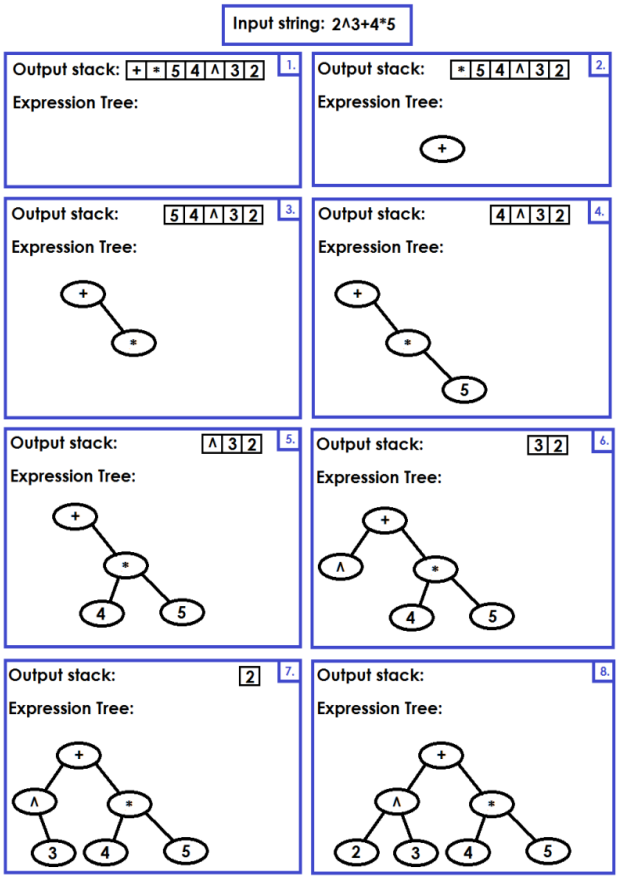

Fig. 2 Building a multi-way tree

IV. Implementation of Shunting-Yard Algorithm

In order to implement the shunting-yard algorithm, building a multi-way tree and ensure calculation for complex framework for reliability analysis based on logical differential calculus, we have to think of the class architecture for the developed tool. Basic module, responsible for parsing and evaluating expressions (can be seen in Fig. 3) has been designed so that it can process the input expression and convert it into the shape of the multi-way tree. The class diagram consists of 10 classes, whose individual meaning will now be explained.

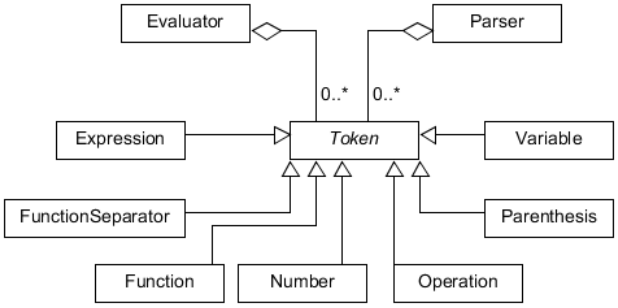

Fig. 3 Class diagram

The first one is abstract class Token, which is a superclass for any supported type of expression, such as constants, operators, functions, etc. Variable class, a subclass of class Token, represents a variable and this variable has to meet certain established rules such as variable must begin with capital or small letter, $ or break character; the rest characters must be capital or small letters, digits or break characters and variables could not be named as defined functions or operations (e.g., X1, variable_one, ThisIsAlsoVariable). Number is a class (a subclass of class Token), which represents the numeric value or defined constants. The occurrence of parentheses and their impact on the priority has also to be taken into account during parsing, so Parenthesis class, a subclass of class Token, is needed for their representation.

The previous classes allow us to use variables, constants and parentheses in the program. Another important part is support for operations and functions. For this purpose, the next classes are defined in the parser.

Operation class, a subclass of class Token, represents all operations in the specified system. This class stores representation in the input, operation's priority, associativity and arity. Another important class is Function class, which is also a subclass of class Token. This class presents a general representation of the functions defined in the system. Functions can have multiple parameters. These parameters are separated by function parameters separator. In order to take function parameters separator into account when parsing, it was necessary to define a class FuncionSeparator, which is also inherit from abstract class Token.

Sometimes, it is also necessary to represent the expression, e.g. $5 * x + 6$. For this purpose, the Expression class can be used.

One of the main classes in the module is the class Parser, which executes shunting-yard algorithm based on the specified input text and stores the output stack for further usage. It also allows us to set the characters used in the input text, such as decimal point, types of parentheses and function parameters separator.

The other one of the main classes is class Evaluator. An instance of this class is responsible for creating a multi-way tree based on the stack received from an instance of class Parser. After creating the tree, instances of this class allow us to perform all symbolic calculations on the symbolic expression, such as simplification or, in case of logic functions, transformation into normal forms.

V. Conclusion

One of the current issues of reliability analysis is investigation of complex systems. Several approaches can be used in solving this problem. One of them is based on application of MVL. MVL functions and tools related to them (e.g., logical differential calculus) can be used to express the structure function of a system under consideration, investigate importance of the system components, or find minimal scenarios needed for ensuring system mission. Application of these ideas in the analysis of systems composed of many components requires creating complex software that will implement all of them. However, such software has not yet been developed. Because of that, we decided to create one.

The essential part of the previously mentioned software is a module that will be able to parse and transform complex mathematical expressions, which are easily readable by a human, into the form that can be processed on the computer. For this purpose, we decided to implement the shunting-yard algorithm and use a multi-way tree to represent the parsed expression. In this paper, we present the architecture of the module that implements this algorithm. The architecture reflects all elements that can exist in the mathematical expressions (variables, constants, functions, operators and their properties), and it is fully customizable for specific, e.g. logic, expressions, i.e., it allows defining the format of variables, possible constants, and operators with their properties, such as precedence, associativity, and arity.

References

[1] E. Zio, "Reliability engineering: Old problems and new challenges," *Reliability Engineering & System Safety*, vol. 94, no. 2, pp. 125–141, Feb. 2009.

[2] E. Zaitseva, V. Levashenko, and M. Rusin, "Reliability analysis of healthcare system," in *2011 Federated Conference on Computer Science and Information Systems, FedCSIS 2011*, 2011, pp. 169–175.

[3] P. Praks, V. Kopustinskas, and M. Masera, "Probabilistic modelling of security of supply in gas networks and evaluation of new infrastructure," *Reliability Engineering & System Safety*, vol. 144, pp. 254–264, Dec. 2015.

[4] B. Natvig, *Multistate Systems Reliability Theory with Applications*. Chichester, UK: John Wiley & Sons, Ltd, 2011.

[5] E. Zaitseva and V. Levashenko, "Multiple-valued logic mathematical approaches for multi-state system reliability analysis," *Journal of Applied Logic*, vol. 11, no. 3, pp. 350–362, Sep. 2013.

[6] S. N. Yanushkevich, D. M. Miller, V. P. Shmerko, and R. S. Stankovic, *Decision Diagram Techniques for Micro- and Nanoelectronic Design Handbook*, vol. 2. Boca Raton, FL: CRC Press, 2005.

[7] M. Kvassay, E. Zaitseva, J. Kostolny, and V. Levashenko, "Importance analysis of multi-state systems based on integrated direct partial logic derivatives," in *2015 International Conference on Information and Digital Technologies (IDT)*, 2015, pp. 183–195.

[8] M. Kvassay, E. Zaitseva, V. Levashenko, and J. Kostolny, "Minimal cut vectors and logical differential calculus," in *2014 IEEE 44th International Symposium on Multiple-Valued Logic*, 2014, pp. 167–172.

[9] M. Kvassay, E. Zaitseva, and V. Levashenko, "Minimal cut sets and direct partial logic derivatives in reliability analysis," in *Safety and Reliability: Methodology and Applications - Proceedings of the European Safety and Reliability Conference, ESREL 2014*, 2015, pp. 241–248.

[10] Edsger W. Dijkstra, "Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60" in *Research Report 35, Mathematisch Centrum, Amsterdam, 1961*. Reprint archived at http://www.cs.utexas.edu/users/EWD/MCReps/MR35.PDF.

[11] T. S. Norvell "Parsing Expressions by Recursive Descent," Spring 1999 http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm.

[12] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, & Tools, Second Edition*, Pearson Education, Inc., 2007.

[13] B. R. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*, John Wiley & Sons, 1999.

[14] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms (3rd ed.)*, MIT Press and McGraw-Hill, 2009.