Automatic Construction of Formal Syntax Tree Based on Regular Expressions

Nazir Ahmad Zafar

***Abstract*—The main functionality of a compiler is to translate source code to an executable machine code correctly and efficiently. Compiler construction is an advanced research area due to size and complexity of the code generated from the source program. Design and construction of error-free and verified compiler will remain a challenge of the current century. Verification of a source program does not assure that the generated code is correct because the compiler may lead to an incorrect target program due to bugs and errors in itself. Hence verification of a compiler is more important than verifying the source program. Lexical analyzer is a main part of compiler used for scanning input stream of characters and grouping into tokens. In this paper, formal construction of syntax tree is described directly from the regular expression to verify the lexical analyzer. At first, augmented regular expression is described then an abstract syntax tree is defined based on the regular expression. Finally formal description of some important operators checking null-ability and computing first and last positions of the internal nodes of the tree are formalized. The specification is described using Z notation then validated using Z/Eves toolset. Formal model is analyzed using powerful techniques of reduction and rewriting available in the Z/Eves toolset.**

***Index Terms*—automata theory, compiler verification, lexical analyzer, automated tools, Z specification**

I. INTRODUCTION

COMPILER is a program which translates a source program written in a high level programming language into an equivalent machine code. The higher level languages not only increase abstraction between source and resulting codes but also increase complexity when it is required to formalize these abstract structures. Now a day, compiler construction is considered as an advanced research area due to the size and complexity of the code generated. It is believed that design and construction of a verified compiler will remain a challenge of twenty first century. Although there exists much work in this area but it needs further investigation because the bugs in the compiler can lead to an incorrect machine code even the source program is verified to be correct. Further, if bugs are detected after testing the executable machine code it might be due to the source program or the compiler itself. This issue has led to verification of a compiler proving correctness of the source program before allowing it to run on the machine.

Formal methods are mathematical-based techniques used for specification, analysis, proving and verification of software and hardware systems [1]. The process of formal verification means applying formal techniques to verify the properties of a system. Formal verification of software targets the source program in which the semantics of a language gives the precise meanings to the program to be analyzed. On the other hand, program verification does not provide any guarantee that the executable machine code is correct as described by the semantics of the source program. This is because compiler may lead to an incorrect target program because of bugs in the compiler itself and can invalidate the guarantees ensured by the formal techniques. It concludes that verification of a compiler is much more important than verification of a source program to be compiled.

In this paper, formal construction of syntax tree is described directly from the regular expression to verify the lexical analyzer. Lexical analyzer is an important part of compiler which scans input stream of characters making groups into tokens. Tokens are sequences of characters having meanings in collective format. Few preliminary results of this research were presented in [2] by formalizing some important concepts of context-free grammar useful for parsing analysis. In this paper, regular expression (input program) is described by defining all of its possible symbols and operators. Relationship among the components of expression is specified to prove its well-defined-ness. The regular expression is augmented by joining a special symbol at the end of the program. An abstract syntax tree is defined based on the regular expression including its internal and terminal nodes. Finally formal description of three important operators checking null-ability and computing first and last positions of all the internal nodes of the syntax tree are described. Formal specification of the algorithm is described using Z notation and model analysis is provided using Z/Eves toolset. The results of this paper will be used in our ongoing project on verification of compiler. The major objectives of our research are: (i) linking automata and formal techniques, (ii) preparing a synthesis of approaches to be useful in the development of automated tools, (iii) identifying and proposing an integration of traditional and formal approaches and (iv) practicing syntactic and semantic relationship of Z and automata in compiler verification.

Currently, it is not possible to develop a complete software program using any single formal technique and, hence, integration of approaches is required. Although integration of approaches is a well-researched area [3-5] but there does not exist much work on verification of compiler by linking formal techniques and automata theory. Dong et al. have described an integration of timed automata and Object Z [6-

Manuscript received March 05, 2012; revised April 01, 2012. This work was supported in part by the Deanship of Scientific Research, King Faisal University, Saudi Arabia.

The author Dr. Nazir Ahmad Zafar is with the Department of Computer Science, College of Computer Sciences and Information Technology (CCSIT), King Faisal University, Saudi Arabia. (phone: +966-3-5898139; e-mails: nazafar@ kfu.edu.sa; nazafar@gmail.com).

7]. R. L. Constable has presented formal description of few important concepts of automata [8-9]. A formal relationship is explored between Petri-nets and Z notation in [10]. Formal analysis of UML is presented in [11-12] using B. An introduction to algebraic structures is investigated using fuzzy automata in [13]. A formal procedure of fuzzy automata and language theory is discussed in [14]. An important notion of algebraic and automata theories is presented in [15]. Rest of the paper is organized as follows:

In section 2, an introduction to Z notation is given. In section 3, reasoning to construct verified complier is provided. Formal construction of syntax tree is given in section 4. Model analysis is done in section 5. Conclusion and future work are discussed in section 6.

II. An Introduction to Z notation

In requirements engineering, there exists various traditional methods which typically are used for expressing software specifications using computer tools for checking properties of systems. Such methods require a full commitment because the specification must be used to construct a complete and consistent model which will be assumed as a baseline for the further development. For complex and incomplete model such methods are not very effective. However, for the complete validation and verification of large scale software specification, it is needed to apply mathematics-based techniques to overcome the weaknesses of the traditional approaches. Experience of applying formal methods shows that it is one of the best options for modeling, particularly, safety critical and complex systems for checking and verifying the safety and other properties.

Formal methods are notations based on discrete mathematics used for describing and analyzing properties of software systems using computer tools. Usually these techniques are based on discrete structures such as sets, relations, functions, graphs and automata. Formal approaches may be classified as property oriented and model descriptive. Property based methods are used to describe software in terms of properties and invariants. Model oriented methods are used to construct model of a system emphasizing both on statics and dynamics of a system. Although there are various notations of formal methods but at the current stage of development, it needs an integration of formal and existing approaches for complete and consistent description of a system.

Z notation is a model centered approach based on sets, sequences, bags, relations and predicate logic [16]. The Z is a specification language used at an abstract level of modeling and specification of systems. Z is usually used for specifying behavior of sequential programs by the abstract data types. The Z has standard set operators, for example, union, intersection, comprehensions, Cartesian products and power sets. The Z allows organizing a system into its smaller components using a powerful data structure named schema. The schema defines a way in which state of a system can be described and refined. Schema has two parts one for definitions and other for defining properties. Refinement is a promising way of Z supporting verifiable transformation from an abstract specification into an executable code. Formal specification described using Z notation can further be refined and transformed to an implemented system.

III. Verifying Lexical analyzer

Compiler verification is a branch of software engineering which deals to prove that compiler behaves exactly as its language specification. Testing and formal methods are two most common techniques for validation and verification in development of a compiler. Compiler testing has various disadvantages similar to other computer programs testing. For example, it is hard to prove that compiler is completely error-free and optimized. The primary objective of writing a compiler is to prove that it is correct and error-free. There exists much research work referring that many tested compilers have bugs and errors in the code [17]. An alternative way is to apply formal methods in compiler verification to find proofs reducing complexity and ensuring correctness of construction procedure. Due to required accuracy, reduction in complexity and optimization needed, compiler construction has become very important and advanced area of research in computer science. Moreover, it is realized that construction of a fully verified compiler is a challenge of the twenty first century.

The main task of a compiler is to translate a source code to an executable and optimized machine code. Of course, an accuracy in compiler construction has much importance because the bugs in the compiler can lead to an incorrect machine code generated from the source code even the source program is fully verified. Constructing and verifying lexical analyzer is an important phase of a compiler whose functionality is to scan the input stream of characters from left to right and grouping it into tokens. The tokens are sequences of characters having meanings in a grouped format. There are two primary methods for implementing the lexical analyzer. The first one is a hard coded program to perform the scanning tasks and the second one uses regular expression and automata theory to model the scanning process. In the first method a main loop in the program reads characters one by one from the input program and uses a switch statement to process it. The output of the procedure is a sequence of tokens from the source program.

In the second method, the source program is read character by character beginning with the start state. After reading each character, the transition function is used to move from current state to the next state. If the final state is reached, it is checked if the token read is reserved word it is passed to the token stream as output. If it is not a reserved word, its name is put in the symbol table if does not exit already. Once a final state is reached an associated action is performed and the same process is continued. If we are not able to reach a final state an error is encountered and error handling routine is called upon. In this method, input is a program which is a regular expression and output is a collection of tokens identified by the finite automata. In this paper a part of verification of lexical analyzer to construct syntax tree from regular expression is described.

There are various other applications of automata theory in addition to compilers construction and verification. Software engineering and maintenance, pattern identification, robotics and speech recognition are some examples of it [18]. In software engineering, test cases can be generated if the system is described by models using automata theory [19]. Applications of automata theory in pattern recognition increase an accuracy of the patterns to be recognized. This is because it can provide a higher level of abstraction by

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

defining the semantics rules for patterns as compared to other specifications techniques. The applications of pattern recognition are found everywhere from language processing to computer networks.

IV. Formal Specification

Main objective of this research is to construct deterministic finite automata from a regular expression directly. For this purpose, at first, we describe formal specification of a given regular expression using Z notation. Then regular expression is extended by adding a special symbol as an end character. End character is used to show the end of input string given to lexical analyzer. Internal node is defined to have information including left position, right position and nullable variables. In next, the syntax tree is described based on the regular expression. Finally, functions for computing left positions, right positions and nullable operator for every node of the syntax tree are described. Although we are well-acquainted with regular expressions but a brief review is given below before its formal description.

Symbol: is an abstract entity. Letters, digits and punctuation are examples of symbols.

Alphabet: is a finite set of symbols used to build larger structures. In automata theory, alphabet is usually denoted by the Greek letter sigma $\sum$. For example, $\sum = \{a, b, c\}$ is an alphabet, where a, b, c are symbols, and abcb is a structure.

Empty String: consists of zero symbols and is denoted by ε.

$\sum^*$: is a set of all possible strings that can be generated from a given alphabet $\sum$.

Regular Expression: is a rule that defines the set of words that are valid tokens in a formal language. The regular expressions (rules) are usually built up from three operators named as concatenation, alternation and repetition.

A. Regular Expressions

In formal specification of regular expression, four variables are assumed as listed in the schema *RE* given below. The first one is *symbols* which is a collection of all alphabets and operators. It represents internal nodes in the syntax tree. The second one is *terminals* which is a finite set of alphabets representing children in the syntax tree. The third one is *operators* having values concatenation, alternation and repetition. The fourth one component is regular expression representing *re* and is a sequence of alphabets and operators. The *Symbol*, *Terminal* and *Operator* are sets at an abstract level of specification over which operators, for example, union, intersection and complement cannot be defined.

[*Symbol*]; *Terminal == Symbol; Operator == Symbol*

Formal definition of regular expression is given below using *RE* schema. In first part of the schema, definitions of variables defining regular expression are given. Invariants over the variables and their relations are defined in second part of the schema in terms of properties. In fact, invariants prove the well-defined-ness of the variables in Z notation. In definition of variables *symbols* has a type of power set of *Symbol*. The set of alphabets *terminals* has a type of power set of *Terminal*. The third variable, *operators*, has a type of power set of *Operator*. The last one regular expression has a sequence type consisting of alphabets and operators. In the schema, *star* and *plus* symbols are used to represent repetition. The symbol *or* is used for alternation. The symbols *lp* and *rp* represent to left and right parenthesis.

RE

$$
\begin{aligned}
&symbols: \mathbb{F}\ Symbol \\
&terminals: \mathbb{F}\ Terminal \\
&operators: \mathbb{F}\ Operator \\
&re: \text{seq}\ Symbol \\
&star, plus: Symbol \\
&or, lp, rp: Symbol \\
&\hline \\
&star \in operators \land plus \in operators \\
&or \in operators \land lp \in symbols \land rp \in symbols \\
&\forall t: Terminal \mid t \in terminals \bullet t \in symbols \\
&\forall o: Operator \mid o \in operators \bullet o \in symbols \\
&\#\, re \geqslant 1 \land (1, re\ 1) \in re \Rightarrow re\ 1 \neq or \land re\ 1 \neq star \land re\ 1 \neq plus \land \\
&re\ 1 \neq rp\ \#\, re \geqslant 1 \land (\#\, re, re\ (\#\, re)) \in re \Rightarrow re\ (\#\, re) \neq or \land re\ (\#\, re) \neq lp \\
&star \in \text{ran}\ re \lor plus \in \text{ran}\ re \\
&\Rightarrow \#\, re \geqslant 2 \land (\forall i: \mathbb{N} \mid i \in 2\ ..\ \#\, re \\
&\quad \bullet\ (re\ i = star \lor re\ i = plus \\
&\quad\ \Rightarrow re\ (i - 1) \in terminals \lor re\ (i - 1) = rp)) \\
&or \in \text{ran}\ re \Rightarrow \#\, re \geqslant 3 \land (\forall i: \mathbb{N} \mid i \in 2\ ..\ \#\, re - 1 \\
&\quad \bullet\ (re\ i = or \Rightarrow re\ (i - 1) \neq lp \land re\ (i - 1) \neq or \\
&\quad\ \land re\ (i + 1) \neq rp \land re\ (i + 1) \neq or)) \\
&lp \in \text{ran}\ re \Rightarrow \#\, re \geqslant 3 \land (\forall i: \mathbb{N} \mid i \in 1\ ..\ \#\, re - 1 \\
&\quad \bullet\ (re\ i = lp \Rightarrow re\ (i + 1) \neq rp \land re\ (i + 1) \neq star \\
&\quad\ \land re\ (i + 1) \neq plus \land re\ (i + 1) \neq or)) \\
&rp \in \text{ran}\ re \Rightarrow \#\, re \geqslant 3 \land (\forall i: \mathbb{N} \mid i \in 2\ ..\ \#\, re \bullet (re\ i = rp \Rightarrow re\ (i - 1) \neq lp \land re\ (i - 1) \neq or))
\end{aligned}
$$

Invariants:

- The repetition symbols, star and plus, are elements of the set of operators.
- The alternation variable is an element of operators-set.
- The left and right parentheses are elements of symbols.
- Each element in the set of terminals is an element of set of symbols.
- Each element in the set of operators is also an element of set of symbols.
- If the regular expression is non-empty, then its left most symbol cannot be alternation or repetition operator. The first element cannot be right parenthesis. The right most symbol of the regular expression cannot be alternation operator or left parenthesis symbol.
- If cardinality of regular expression is more than one then for any element, excluding first element, if it is repetition then its left is terminal or right parenthesis.
- If cardinality of regular expression is more than two then for any element, excluding first and last elements, if it is an alternation element then its left cannot be alternation or left parenthesis and its right cannot be alternation of right parenthesis.
- If cardinality of regular expression is more than two then for any element, excluding last element, if it is left parenthesis then its right element cannot be right parenthesis, repetition or alternation operator.
- If cardinality of regular expression is more than two then for any element, excluding first element, if it is

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

right parenthesis then its left element cannot be left parenthesis or alternation operator.

Because our objective is to formally construct deterministic finite automata for regular expression, that is why, a language consisting of set of all possible regular expressions is described by the schema *REs* given below. The schema has only one component *res* which is a power set of Schema *RE*. In the predicate part of the schema few of the properties of regular expressions are defined using universal quantifier. Similarly, rest of the properties can be described.

REs

$res: \mathbb{F}\ RE$

$\forall re: RE \mid re \in res \bullet re.star \in re.operators \land re.plus \in re.operators$

$\forall re: RE \mid re \in res \bullet re.or \in re.operators \land re.lp \in re.operators \land re.rp \in re.operators$

$\forall re: RE \mid re \in res \bullet \forall t: Terminal \mid t \in re.terminals \bullet t \in re.symbols$

$\forall re: RE \mid re \in res \bullet \forall o: Operator \mid o \in re.operators \bullet o \in re.symbols$

After scanning input the lexical analyzer identifies the tokens. As there must be some special symbol at the end of a file which shows end of file in the input string. In the schema *ExtendedRE* given below, a special symbol hash (#) is joined at the end of the input string to produce augmented string. It is supposed that the hash symbol does not exist in the set of symbols of the regular expression. As regular expression is defined as a sequence of symbols that is why the special symbol is joined using concatenation operator.

ExtendedRE

ΔRE

$hash: Symbol$

$hash \notin \text{ran}\ re$

$re' = re \frown \langle hash \rangle$

B. Tree Construction

To construct deterministic finite automata directly from a regular expression, at first, syntax tree is required. The syntax tree from the augmented regular expression is described below. Then three important functions namely nullable, first position and last position are computed formally to be useful for the description of follow position function. After computing follow positions of the internal nodes of the abstract syntax tree deterministic finite automata can be easily constructed.

Before description of the syntax tree a generic definition of an internal node of the tree is given below using the Schema *Internal*. The schema consists of six variables namely *node* for an identifier of the node, *left* for left child of the tree, *right* for right child of the tree, *firstpos* for first position function, *rightpos* for right position function and *nullable* for checking nullability of the node. Nullable is a Boolean function having value true or false. First and last position functions are collection of identifiers of the node computed based on children nodes.

$NULLABLE ::= TRUE \mid FALSE$

Internal

$node: Operator$

$left: ExtendedRE$

$right: ExtendedRE$

$firstpos: \mathbb{F}\ \mathbb{N}$

$lastpos: \mathbb{F}\ \mathbb{N}$

$nullable: NULLABLE$

A formal description of the relationship between regular expression and syntax tree is given using the *REtoTree* schema. The schema consists of nine components in addition to *RE* schema. The first one *res* is a collection of all possible regular expressions based on the *ExtendedRE* schema. The second one is a set of terminals which are alphabets of the language described by a regular expression. The third variable *leafs* is collection of children of the syntax tree. The fourth variable *Internal* is a set of internal nodes of the tree. The definitions of other components are already given in the definition of regular expression.

REtoTree

RE

$res: \mathbb{F}\ ExtendedRE$

$terminals: \mathbb{F}\ Terminal$

$leafs: \mathbb{F}\ Symbol$

$internals: \mathbb{F}\ Internal$

$epsi, star, or, con: Symbol$

$ids: \mathbb{F}\ \mathbb{N}$

$\forall l: Symbol \mid l \in leafs \bullet l = epsi \lor l \in terminals$

$\forall i: Internal \mid i \in internals \bullet \exists re1, re2: ExtendedRE \mid re1 \in res \land re2 \in res \bullet i.left = re1 \land i.right = re2 \land i.node = star \lor i.node = or \lor i.node = con$

$\#ids = \#terminals$

$ids = \{ n: \mathbb{N}; i: Internal \mid i \in internals \land n \in i.firstpos \lor n \in i.lastpos \bullet n \}$

Invariants: (1) Leaf is either a terminal or null string. (2) Each internal node has two well-defined children. One of these might be null but both cannot be null strings. (3) The set of identifiers (leafs) of the tree is same as the set of terminals of the language. (4) The set of identifiers of parent is based on its children.

C. Formal Specification of Operators

Formal description of nullable, first position and last position operators is given here. The *Nullable* operator consists of four components, that is, the node itself, left child, right child and node type. The nullable variable of the node is computed based on its children. Three types of nodes are assumed that is alternation, concatenation and repetition. The definitions of the variables are given in first part of the schema and properties are described in the predicate part.

$NodeType ::= OR \mid CON \mid STAR$

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Nullables

$ileft?: Internal$
$iright?: Internal$
$iparent!: Internal$
$type: NodeType$

$type = OR$
$\Rightarrow ileft?\ .\ nullable = TRUE \land iright?\ .\ nullable = TRUE$
$\lor iright?\ .\ nullable = FALSE$
$\Rightarrow iparent!\ .\ nullable = TRUE$
$type = OR$
$\Rightarrow iright?\ .\ nullable = TRUE \land ileft?\ .\ nullable = TRUE$
$\lor ileft?\ .\ nullable = FALSE$
$\Rightarrow iparent!\ .\ nullable = TRUE$
$type = CON \Rightarrow iparent!\ .\ nullable = TRUE$
$\Leftrightarrow iright?\ .\ nullable = TRUE \land ileft?\ .\ nullable = TRUE$
$type = STAR$
$\Rightarrow iparent!\ .\ nullable = ileft?\ .\ nullable$
$\land iright?\ .\ nullable = ileft?\ .\ nullable$

Invariants: (1) If the node is alternation type then it is nullable if and only if one of its children is nullable. (2) If the node is concatenation type then it is nullable if and only if both of its children are nullable. (3) If node is repetition type then it is nullable if and only if it child is nullable.

The first position function of a node n is a set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language described by a part of the regular expression rooted at n. The *LeftPositions* function consists of same components as in case of *Nullables*. The left positions are described based on its children.

LeftPositions

$ileft?: Internal$
$iright?: Internal$
$iparent!: Internal$
$type: NodeType$

$type = OR$
$iparent!\ .\ firstpos = ileft?\ .\ firstpos \cup iright?\ .\ firstpos$
$type = CON$
$ileft?\ .\ nullable = TRUE$
$\Rightarrow iparent!\ .\ firstpos = ileft?\ .\ firstpos \cup iright?\ .\ firstpos$
$ileft?\ .\ nullable = FALSE \Rightarrow iparent!\ .\ firstpos = ileft?\ .\ firstpos$
$type = STAR$
$iparent!\ .\ firstpos = ileft?\ .\ firstpos$
$\land ileft?\ .\ firstpos = iright?\ .\ firstpos$

Invariants: (1) If the node is alternation type then its first position is union of first positions of its left and right children. (2) If node type is concatenation, its left child is nullable then first position of the node is union of first positions of its left and right children. If left child is not nullable then first position is equal to first position of left child. (3) If the node is repetition type the then first position of the node is left position of its child.

The last position function of the node n is the set of positions in the subtree of the syntax tree rooted at n that correspond to the last symbol of at least one string in the language described by the subexpression of the regular expression rooted at n. The last position operator consists of four components namely node identifier, left and right children and type. The formal description of last position function is described in predicate part of the *RightPositions* schema.

RightPositions

$ileft?: Internal$
$iright?: Internal$
$iparent!: Internal$
$type: NodeType$

$type = OR$
$iparent!\ .\ lastpos = ileft?\ .\ lastpos \cup iright?\ .\ lastpos$
$type = CON$
$iright?\ .\ nullable = TRUE$
$\Rightarrow iparent!\ .\ lastpos = ileft?\ .\ lastpos \cup iright?\ .\ lastpos$
$iright?\ .\ nullable = FALSE \Rightarrow iparent!\ .\ lastpos = ileft?\ .\ lastpos$
$type = STAR$
$iparent!\ .\ lastpos = ileft?\ .\ lastpos \land ileft?\ .\ lastpos = iright?\ .\ lastpos$

Invariants: (1) If the node is alternation type then its last position is union of last positions of its left and right children. (2) If the node is concatenation type and its right child is nullable then last position of the node is union of last positions of its left and right children. If right child is not nullable then last position of the node is equal to last position of its right child. (3) If the node is repetition type then last position of the node is last position of its child.

V. Model Analysis

In this section, model analysis is done for the specification. It is noted that although computer tools improve quality of software systems but, on the hand, there does not exist any real computer tool which may assure about complete correctness of a model. It means even the specification is well-written using any of the specification language it may contain potential errors. That is an art of writing a formal specification never guarantee that the system is correct, complete and consistent. However, if the specification is analyzed with computer tools it increases a confidence over the system to be developed.

The Z/Eves is a powerful tool used here for analyzing the Z specification of construction of syntax tree based on a regular expression. A snapshot of the specification analysis is presented in Figure 1. The first column on the left of the figure shows syntax checking and the second column represents the proof correctness of the specification. The symbol ‘Y’ shows that the formal specification is correct syntactically and proof is also correct while the symbol ‘N’ represents that errors exist. All the schemas are checked to prove that specification is correct in syntax and has a correct proof. Some schemas of the specification were proved using reduction techniques available in the toolset.

Summary of the analysis is presented in Table 1. In first column of the table, name of schema is given. The symbol “Y”, in column 2, indicates that all schemas are well-written and proved. Similarly, domain checking, reduction and proof by reduction are represented in columns 3, 4 and 5, respectively. The character “Y*” annotated with '*' shows that the schema is proved by performing reduction on the predicates part to make specification more meaningful.

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

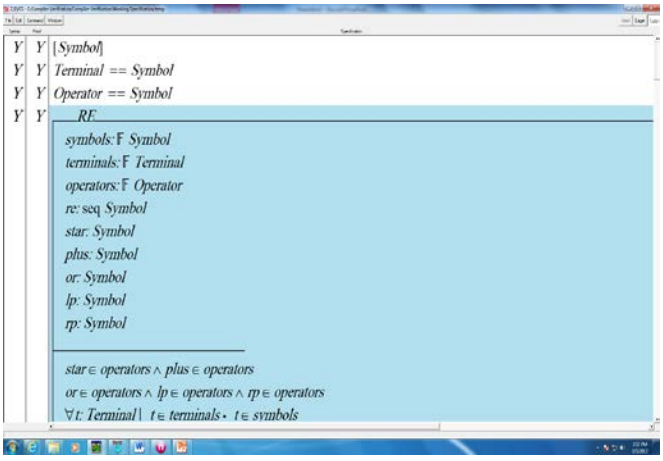

Figure 1. Snapshot of the Model Analysis.

TABLE I. RESULTS OF MODEL ANALYSIS

Schema Name	Syntax Type Check	Domain Check	Reduction	Proof
RE	Y	Y	Y	Y
REs	Y	Y	Y	Y
ExtendedRE	Y	Y	Y*	Y
Internal	Y	Y	Y	Y
REtoTree	Y	Y	Y	Y
Nullables	Y	Y	Y	Y
LeftPositions	Y	Y	Y*	Y
RightPositions	Y	Y	Y*	Y

VI. CONCLUSION

The development and verification of lexical analyzer has been an important research area. In this paper, syntax tree is directly generated from regular expression using Z notation for verifying lexical analyzer. Identification of errors and resolving of ambiguities at lexical level is important and a real challenge in compiler construction because unidentified errors may grow exponentially to the subsequent phases. The verification of lexical analyzer is benefited by linking automata to Z specification. The Z, being abstract in nature and having computer tool support, is used because it enhanced reliability and correctness of models of the system. It is observed that formal specification helped us to make it possible resolving ambiguities. The specification was verified and validated using Z/Eves toolset. Several other tools exist to support the formal specification but Z/Eves is a powerful one to analyze the specification because of rich mathematical notations which made it possible to reason about behavior of the specification more effectively.

A need for such tools is not only in the area of programming languages but other applications, for example, pattern recognition and queries in databases can also benefit. For an accurate processing system, a program must be able to correctly process a language by obey precise lexical analysis rules using formal definitions which shows importance of this research. An exhaustive survey of existing work, on integration of approaches and verification of systems, was performed. Some of the interesting and relevant works [20-28] were found but our approach is different because of abstract and conceptual level integration of automata and Z for verification. Formalization of other concepts, useful in compiler verification, is in progress and will appear soon.

REFERENCES

[1] C. J. Burgess, "The Role of Formal Methods in Software Engineering Education and Industry," Technical Report, University of Bristol, UK, 1995.

[2] K. A. Buragga, and N. A. Zafar, "Formal Parsing Analysis of Context-Free Grammar using Left Most Derivations, ICSEA, 2011.

[3] H. Beek, A. Fantechi, S. Gnesi, and F. Mazzanti, "State/Event-Based Software Model Checking," Integrated Formal Methods, Springer, vol. 2999, pp. 128-147, 2004.

[4] O. Hasan and S. Tahar, "Verification of Probabilistic Properties in the HOL Theorem Prover," Integrated Formal Methods, Springer, vol. 4591, pp. 333-352, 2007.

[5] F. Gervais, M. Frappier, and R. Laleau, "Synthesizing B Specifications from EB3 Attribute Definitions," Integrated Formal Methods, Springer, vol. 3771, pp. 207-226, 2005.

[6] J. S. Dong, R. Duke, and P. Hao, "Integrating Object-Z with Timed Automata," pp. 488-497, 2005.

[7] J. S. Dong, et al., "Timed Patterns: TCOZ to Timed Automata," The 6th ICFEM, pp. 483-498, 2004.

[8] R. L. Constable, et al., "Formalizing Automata II: Decidable Properties," Technical Report, Cornell University, 1997.

[9] R. L. Constable, et al., "Constructively Formalizing Automata Theory," Foundations of Computing Series, MIT Press, 2000.

[10] M. Heiner and M. Heisel, "Modeling Safety Critical Systems with Z and Petri nets," International Conference on Computer Safety, Reliability and Security, Springer, pp. 361–374, 1999.

[11] H. Leading and J. Souquieres, "Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B," Asia-Pacific Software Engineering Conference, pp. 495-504, 2002.

[12] H. Leading and J. Souquieres, "Integration of UML Views using B Notation," Proceedings of Workshop on Integration and Transformation of UML Models, 2002.

[13] W. Wechler, "The Concept of Fuzziness in Automata and Language Theory," Akademic-Verlag, Berlin, 1978.

[14] N. M. John and S. M. Davender, "Fuzzy Automata and Languages: Theory and Applications," Chapman & HALL, 2002.

[15] M. Ito, "Algebraic Theory of Automata and Languages," World Scientific Publishing Co., 2004.

[16] J. M. Spivey, "The Z Notation: A Reference Manual," Englewood Cliffs, NJ, Printice-Hall, 1989.

[17] C. Lindig, Random Testing of C Calling Conventions," AADEBUG'5, ACM, 2005.

[18] J. A. Anderson, "Automata Theory with Modern Applications," Cambridge University Press, 2006.

[19] M. v. d. Brand, A. Sellink, and C. Verhoef, "Generation of Components for Software Renovation Factories from Context-Free Grammars," CRE, pp. 144-153, 2001.

[20] M. Balakrishna, D. Moldovan, and E. K. Cave, "Automatic Creation and Tuning of Context-Free Grammars for Interactive Voice Response Systems," IEEE NLP-KE, pp. 158 – 163, 2005.

[21] L. Pedersen and H. Reza, "A Formal Specification of a Programming Language: Design of Pit," Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, pp. 111-118, 2008.

[22] D. P. Tuan, "Computing with Words in Formal Methods," Technical Report, University of Canberra, Australia, 2000.

[23] A. Hall, "Correctness by Construction: Integrating Formality into a Commercial Development Process," Praxis Critical Systems Limited, Springer, vol. 2391, pp. 139-157, 2002.

[24] D. K. Kaynar and N. Lynchn, "The Theory of Timed I/O Automata," Morgan & Claypool Publishers, 2006.

[25] D. Jackson, I. Schechter, and I. Shlyakhter, "Alcoa: The Alloy Constraint Analyzer," Proceedings of The 22nd International Conference of Software Engineering, pp. 730-733, 2000.

[26] D. Aspinall and L. Beringer, "Optimisation Validation," Electronic Notes in Theoretical Computer Science, vol. 176, pp. 37–59, 2007.

[27] S. Briaisa and U. Nestmannb, "A Formal Semantics for Protocol Narrations," Theoretical Computer Science, vol. 389, pp. 484–511, 2007.

[28] L. Freitas, J. Woodcock, and Y. Zhang, "Verifying the CICS File Control API with Z/Eves: An Experiment in the Verified Software Repository," Science of Computer Programming, vol. 74, pp. 197-218, 2009.

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)