The Journal of Systems & Software 204 (2023) 111798

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Using expression parsing and algebraic operations to generate test sequences.☆

Pan Liu [b,1], Yihao Li [a,1,*]

[a] School of Information and Electrical Engineering, Ludong University, Yantai, China
[b] Faculty of Business Information, Shanghai Business School, Shanghai, China

ARTICLE INFO

Article history:
Received 27 January 2023
Received in revised form 25 May 2023
Accepted 29 June 2023
Available online 4 July 2023

Keywords:
Expression parsing
Test sequence generation
Algebraic operation
Abstract syntax tree

ABSTRACT

It has become a popular trend to build software's regular expression or extended regular expression models in order to generate test sequences from these models. Such test sequences tend to have promising test coverage and fault detection capability. During this process, one critical step is expression parsing based on algebraic operations. However, the parsing can be very challenging as different algebraic systems have different algebraic operators and algebraic operations. Besides, the parsing difficulty continues to grow as software complexity increases. To address the above challenges, this paper proposes a general expression parsing framework for test sequence generation. The proposed framework consists of three stages, expression decomposition, algebraic operations, and subexpression combination. To implement the framework, an expression parsing algorithm based on abstract syntax tree is developed. Case studies based on 117 expressions collected from the literature over the past 30 years as well as 13 software systems are conducted to evaluate the effectiveness of the proposed algorithm. The results indicate that our algorithm is superior to three existing and commonly used algorithms with respect to expression parsing and software fault detection.

1. Introduction

In the past, algebraic systems such as Kleene algebra (Kozen, 1997, 1994), event algebra (Carlson and Lisper, 2004; Hinze and Voisard, 2015), concurrent regular expression system (Garg and Ragunath, 1992), and transition algebra (Liu et al., 2021), have been proposed to implement regular expression or extended regular expression modeling and test sequence generation for software. These proposed algebraic systems involve many algebraic operations for algebraic operators. According to these algebraic operations, the expression[2] model of software can always be transformed into a set of test sequences (Liu et al., 2021; Liu and Miao, 2014). Moreover, test sequences generated from the expression model have better software fault detection effectiveness (Liu et al., 2017; Wang et al., 2021) and higher test coverage (Kilincceker et al., 2019) than those generated from graphical models.

However, it is difficult to generate test sequences from an expression by using algebraic operations. On the one hand, different algebraic systems have different algebraic operators and algebraic operations. For example, the symbol "+" denotes the choice operator in Kleene algebra and the positive closure in concurrent regular expression system and transition algebra. On the other hand, as the complexity of the expression model grows, it is more difficult for the computer to automatically choose the appropriate algebraic operations to parse the model.

Although algorithms such as the Kilincceker et al.'s algorithm (Kilincceker et al., 2019) and the Polo et al.'s algorithm (Polo et al., 2020) have been proposed to parse expression for test sequence generation, these algorithms do not fully support all algebraic operations in algebraic systems to generate test sequences. The Kilincceker et al.'s algorithm only supports three basic algebraic operations including concatenation, choice, and closure. The Polo et al.'s algorithm produces test sequences from an expression by limiting the length of sequences for Kleene closure. Thus, their algorithm does not support algebraic operations of other closures in algebraic systems. Earlier, we also proposed an expression parsing algorithm based on string analysis (Liu and Xu, 2018; Zeng et al., 2014). However, it does not support constraint operation and range closure operation in algebraic systems.

To realize the parsing of the expression model based on algebraic operations, the paper proposes a general expression parsing

☆ Editor: W. Eric Wong.
* Corresponding author.
E-mail address: yihao.li@ldu.edu.cn (Y. Li).
1 Both Pan Liu and Yihao Li are first authors.
2 In the paper, expression represents regular expression and extended regular expression.

https://doi.org/10.1016/j.jss.2023.111798
0164-1212/

framework, in which the expression parsing process is divided into four steps. In the first step, an expression model is decomposed into a set of subexpressions. In the second step, we look for a suitable algebraic operation from an algebraic operational set for each subexpression parsing. In the third step, these algebraically operated subexpressions in the second step are combined into a new expression. In the last step, a set of test sequences can be obtained from the new expression according to the choice operation, otherwise the new expression needs to be decomposed again until the sequence set is obtained.

To achieve the proposed general expression parsing framework, we develop AST-EP, an expression parsing algorithm based on abstract syntax tree (AST). First, we construct an AST of the expression model using a Google tool RE2.[3] Then, a post-order list of nodes is extracted from the tree. Finally, by performing algebraic operations on the elements of the list, the parsing of expression is realized. Accordingly, a running example is provided for the demonstration of the proposed AST-EP. Later, cross-comparisons are conducted between AST-EP and three existing expression parsing algorithms on 117 expressions collected from the literature over the past 30 years and 13 open source software systems used in real life. The results show that our AST-EP not only has more powerful expression parsing capability, but also generates test sequences with better fault detection effectiveness.

To effectively implement our proposed method, two concerns require attention. Firstly, it is necessary to ensure the normativity of the constructed expression model, as an unnormalized expression model cannot be converted into an AST. Secondly, it is essential to pre-construct a set of algebraic operations as an input parameter for AST-EP.

The contributions of the paper are as follows.

(1) We present a general expression parsing framework that takes expression models as the input and generates test sequences as the output by using algebraic operations.
(2) We propose an AST-based expression parsing algorithm and discuss its theoretical basis.
(3) Case studies on 117 expressions and 13 open source software systems are conducted to investigate the parsing capability of the proposed algorithm as well as the fault detection effectiveness of the test sequences accordingly generated.

The rest of this paper is structured as follows: Section 2 introduces the related work of the paper. Section 3 explains the algebraic operations that are used in the paper. Section 4 proposes a general expression parsing framework. Section 5 puts forward the AST-based expression parsing method and designs an algorithm to realize the proposed method. Section 6 provides a running example to demonstrate the AST-based expression parsing algorithm. The case studies are presented in Section 7. Threats to validity are discussed in Section 8. Finally, conclusions and future work are given in Section 9.

2. Related work

The research related to this work includes algebraic operators and expression parsing.

2.1. Algebraic operators

An algebraic system consists of a set of alphabets and some algebraic operators. The expression model of the software can be built using these alphabets and operators. Generally speaking, having more algebraic operators indicates a better modeling power of the expression model.

The earliest research on algebraic operators originated from the theoretical study of the operator * in regular expressions. In 1981, Kozen (1981) studied complete deductive systems of infinite equations with *, and gave infinite * -continuity conditions and an equation induction axiom. Then, Ng (Ng, 1984) proposed the relational algebra including * in his doctoral dissertation, and proved the completeness of this algebraic system by formal methods. The work of both Kozen and Ng was groundbreaking in establishing relevant theories for *, which was used to describe the loop construction in programs.

In 1992, Garg and Ragunath (1992) extended traditional regular expression by adding some new algebraic operators, including alpha closure, interleaving, synchronous composition, and renaming. Then, they used four examples to demonstrate the implementation of these new algebraic operations. According to their research, these new operators give expressions the same modeling power as Petri nets.

In 1994, Kozen (1994) proposed a completeness theorem for Kleene algebra on the basis of their previous work. The proposed algebra consists of a finite alphabet set Σ and a set of operators, including +, ·, *, 0, and 1, where + denotes the disjunction operation, · denotes concatenation, * denotes transitive closure operation, the number 0 denotes a null relation, and the number 1 denotes an identity relation. The main contribution of Kozen's work is to propose new operators +, 0, and 1 to model programs. Then, Kozen (1997), Kozen and Smith (1996) presented Kleene algebra with tests. This proposed algebra contains a new operator NOT, denoted as ¯. Kozen used this operator to describe mutually exclusive behavior in programs. The operator ¯ can simplify the construction of the expression model. For example, there is $bp\bar{b}+\bar{b}pb=0$.

To describe concurrent events in software, Hoare et al. (2009, 2011) proposed concurrent Kleene algebra, and then developed a tool to support the modeling of concurrent Kleene algebra (Hoare et al., 2014). In the proposed algebraic system, they used four symbols *, ;, ‖, and [] to respectively denote concurrent composition, weak sequence combination, disjoint parallel combination, and interleaving. Different from the operator * in Kleene algebra, the operator * in concurrent Kleene algebra has different meanings.

In 2015, Jipsen and Moshier (2015) also proposed concurrent algebra with tests. They defined the operator ‖ to describe concurrent events in software systems. Additionally, they gave a partial order operator $\leq$ to process the concurrent operation. For example, there is $(x \parallel y)(a \parallel b) \leq (xa) \parallel (yb)$ in Jipsen (2014). Thus, we can use the partial order operator $\leq$ to simplify the expression model.

In 2017, Kozen (2017) proposed two axioms of the partial order operator $\leq$. Then, they put forward Kleene Coalgebra with Tests (KCT) and the Brzozowski derivative. In addition, Kozen gave two axioms for two operators 0 and 1. Kozen's work established a theory for operators $\leq$, 0, and 1.

In 2019, Smolka et al. (2019) presented guarded Kleene algebra with tests and established its basic properties. In the proposed algebra, they gave an operator $\diamond$, a partial fusion product, to realize the composition of guarded strings. In addition, they gave a set of axioms for the proposed algebra by operators +, ·, and (b), (0), (1), and (c). The new partial fusion product operator $\diamond$ can describe more complex software behavior. For example, the expression $xa\diamond yb$ can represent xay if $a{=}b$ or an undefined otherwise.

In 2020, Polo et al. (2020) introduced five operators, including union, concatenation, Kleene closure, positive closure, option

[3] https://github.com/google/re2/blob/main/re2/re2.cc

Table 1
Algebraic operators and their descriptions in algebraic systems.

Operator	Description
+	Positive closure (Garg and Ragunath, 1992; Liu et al., 2021) Choice (Kozen, 1994; Kozen and Smith, 1996; Kozen, 2017; Wagemaker et al., 2022)
·	Concatenation (Kozen, 1994; Garg and Ragunath, 1992; Liu et al., 2021)
;	Weak sequence combination (Hoare et al., 2009, 2014) Sequential composition (Wagemaker et al., 2022)
|	Choice (Garg and Ragunath, 1992; Liu et al., 2021; Kilincceker et al., 2019)
||	Concurrent (Jipsen and Moshier, 2015; Jipsen, 2014) Disjoint parallel combination (Wagemaker et al., 2022) Interleaving (Garg and Ragunath, 1992; Liu et al., 2021)
[]	Interleaving (Hoare et al., 2009, 2011) synchronous composition (Garg and Ragunath, 1992) Parallel (Liu et al., 2021)
*	Kleene closure (Kozen, 1994; Garg and Ragunath, 1992; Liu et al., 2021; Kozen and Smith, 1996; Kozen, 2017; Wagemaker et al., 2022)
$\leq$	Partial order (Kozen, 1997; Jipsen and Moshier, 2015; Kozen, 2017)
$expand(r^*)$	Restrict the length (Polo et al., 2020)
$\diamond$	Partial fusion product (Smolka et al., 2019)
$\partial_a(s{\cdot}t)$	Derivation (Schmid et al., 2023)

operator. Then, they gave a new operation $expand(r^*)$ to restrict the length of r^*. Their work was able to convert an infinite length sequence into a finite length sequence. Different from the work of Polo et al. this paper proposes a new formula to achieve this purpose.

In 2021, Liu et al. (2021) proposed transition algebra to support the extended regular expression modeling of software. Specifically, nine operators and their algebraic operations were designed and the closure and regularity of these algebraic operations are proved. The main contribution of Pan et al. was to develop the concept of constraints on expression models.

In 2022, Wagemaker et al. (2022) introduced concurrent NetKAT. This proposed algebra contains four operators, such as sequential composition ';', iteration '*', non-deterministic choice '+', and parallel composition '|| '. Thus, the operator ; in concurrent NetKAT has a difference function than that in concurrent Kleene algebra.

In 2022, Zhang et al. (2022) found that Kleene algebra with tests cannot support statements of incorrectness logic by a counterexample. In addition, they reviewed all operators and operation properties of Kleene algebra with tests.

In 2023, Schmid et al. (2023) discussed the properties of algebraic operations in guarded Kleene algebra with tests. Then, they introduced some operations to generate trees from an algebra. Some operators, i.e. $(s{\cdot}t)(a)$ and $\partial_a(s{\cdot}t)$, were designed for this goal. These new operators can be used to describe complex software behavior, i.e., $(s\cdot t)(a) = \begin{cases} t(a) & if\ s(a) = 1 \\ s(a) & otherwise \end{cases}$.

By comparing algebraic operators in the above algebraic systems, we find that some of them have different meanings in the different algebraic systems as shown in Table 1. Although previous researchers have made significant contributions in constructing algebraic systems and proposing new algebraic operators, the expressive modeling capabilities of each algebraic system are limited by its own algebraic operators and algebraic operations which are mutually independent. To break the barrier, this paper proposes a general expression framework that is compatible with the parsing of expression models constructed by mainstream and commonly used algebraic systems.

2.2. Expression parsing

In the past, the purpose of parsing regular expressions was to determine whether a string matches a regular expression (Neamtiu et al., 2005), and a common method was to construct an abstract syntax tree (Zhang et al., 2019; Liu et al., 2020b; Cui et al., 2010) for the regular expression. In 1992, Myers (1992) designed a four Russians algorithm for regular expression matching and set the priority for three operators ·, |, and *. Then, he presented the construction method of the parsing tree of regular expression. The leaf nodes of the parse tree are symbols in the regular expression or the empty ε, and non-leaf nodes are three operators ·, |, and * in the regular expression. This proposed parse tree is a binary tree, where a non-leaf node has left and right subtrees when it is | or ·, and has only one subtree when it is *.

In 2014, Liu and Miao (2014) proposed some expression parsing rules of extended regular expression and used these rules to generate test sequences from the extended regular expression model of a program. Using these test sequences as test cases, they found a bug in the program (Liu et al., 2017).

In 2018, Liu and Xu (2018) developed a tool for test modeling and test generation that can parse some complex expression with concurrent operation and parallel operation.

Kilincceker et al. (2018) presented an algorithm to parse regular expression. They first converted a regular expression into an abstract syntax tree. Then, they used the method of preorder traversal to visit the abstract syntax tree. Finally, according to operations of three operators, test sequences can be obtained from regular expression during regular expression parsing.

In 2020, Polo et al. (2020) presented a test generation process that consists of three stages. In the first stage, they constructed annotated regular expressions to describe the system under test. In the second stage, expressions were parsed based on some operations. In the last stage, they process test sequences to generate test cases with oracle. As a major contribution, they proposed a method to limit the length of test sequences.

In 2021, Borsotti and Trofimovich (2021) proposed a new algorithm to improve expression matching efficiency. Their proposed algorithm is faster than the Kuklewicz algorithm (Kuklewicz, 2007), but requires more storage space. One advantage is that their algorithm is able to output an expression's parse tree incrementally.

In 2021, Gibney and Thankachan (2021) investigated the parsing tree structure of regular expression, and proved that there is a data structure which can make text matching regular expressions in $O\left(\frac{|T||p|}{\tau}\right)$, where $\tau \in [1, |T|]$. Cardoso et al. (2021) used formal methods to describe a regular expression parsing algorithm based on derivatives of Antimirov (1996) and Brzozowski (1964). The proposed algorithm can construct a parse tree for the regular expression and can verify that an input string can be accepted by the RE language. In 2022, Borsotti and Trafimovich (2022) proposed an algorithm for regular expression parsing and subexpressions matching.

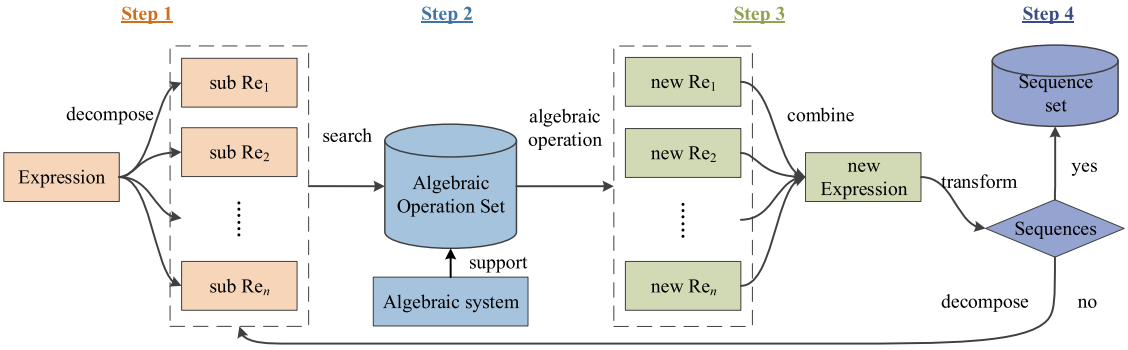

Fig. 1. A general expression parsing framework, where Re_i is a subexpression.

The above expression parsing researches can well realize the expression analysis, expression matching, and expression operation. However, few of these researches involve the test sequence generation using algebraic operations in an algebraic system. Therefore, a general method needs to be proposed to realize test sequence generation by using algebraic operations.

3. Algebraical operations

Generally, an algebraic system for testing consists of a symbol set Σ describing software behavior and some algebraic operators. We can construct an expression model of software behavior using both Σ and algebraic operators. Because each algebraic operator has one or more algebraic operations in any algebraic system for testing, we can always perform algebraic operations on the expression model of software. In Kozen (1994), Liu et al. (2021), Liu and Miao (2014), researchers have been shown that an expression model can always be transformed into a set of test sequences after a finite number of algebraic operations. Therefore, how to perform algebraic operations on the expression model is the key to generate test sequences. In this section, we introduce several basic algebraic operators and their algebraic operations, which are applied to the examples in the following sections.

The concatenation operator is denoted by & in the paper. For example, $a\&b$ denotes that b starts after a ends for $\forall a, b\in\Sigma$. For simplicity of expression description, the paper omits the symbol & in expressions. The algebraic operation of & is as follows:

$$\forall a, b\in\Sigma\bullet abc = (ab)c = a(bc) \tag{1}$$

$$\forall a\in\Sigma\bullet 1a = a1 = a \tag{2}$$

The choice (or union) operator is denoted by | in the paper, and its algebraic operations are as follows:

$$\forall a\in\Sigma\bullet a|a = a \tag{3}$$

$$\forall a, b\in\Sigma\bullet a|b = a\vee b \tag{4}$$

$$\forall a, b\in\Sigma\bullet a|b = b|a \tag{5}$$

$$\forall a, b\in\Sigma\bullet a|b|c = (a|b)|c = a|(b|c) \tag{6}$$

$$\forall a, b\in\Sigma\bullet a(b_1|\cdots|b_n) = ab_1|\cdots|ab_n \tag{7}$$

$$\forall a, b\in\Sigma\bullet (a_1|\cdots|a_n)b = a_1b|\cdots|a_nb \tag{8}$$

$$\forall a, b\in\Sigma\bullet (a_1|\cdots|a_n)(b_1|\cdots|b_m) = a_1b_1|\cdots|a_nb_m \tag{9}$$

Note that we can obtain three sequences ad, bc, and e from $ad|bc|e$ by Eq. (4). Therefore, if an expression can be converted to a new expression consisting of a set of sequences with the choice relationship, we can always get a set of sequences from the expression.

Kleene closure is denoted by * in the paper, and its algebraic operations are as follows:

$$\forall a\in\Sigma\bullet a^* = 1|a^1|a^2|\cdots \tag{10}$$

$$\forall a, b\in\Sigma\bullet (a|b)^* = a^*(ba^*)^* \tag{11}$$

$$\forall a\in\Sigma\bullet (a^*)^* = a^* \tag{12}$$

The interleaving operator is denoted by $\|$ in the paper, and its algebraic operations are as follows:

$$\forall a, b\in\Sigma\bullet a||b = ab|ba \tag{13}$$

$$\forall a, b_1, \cdots, b_n\in\Sigma\bullet a||(b_1|\cdots|b_n) = (a||b_1)|\cdots|(a||b_n) \tag{14}$$

$$\forall a, b_1, \cdots, b_n\in\Sigma\bullet (b_1|\cdots|b_n)||a = (b_1||a)|\cdots|(b_n||a) \tag{15}$$

For stricter definitions of algebraic operations and the introductions of more algebraic operations can be found in Kozen (1997), Carlson and Lisper (2004), Garg and Ragunath (1992), Liu et al. (2021).

4. A general framework

Before introducing our expression parsing framework, we first discuss the process of parsing an expression manually. For example, for the expression $(a|b)c^*(d|e)$, we need to process it according to some algebraic operations in Section 3 to generate test sequences. The manually parsing steps of this expression are as follows:

$(a|b)c^*(d|e)$
$= (ac^*|bc^*)(d|e)$ Eq. (8)
$= ac^*d|ac^*e|bc^*d|bc^*e$ Eq. (9)
$= ac^*d\vee ac^*e\vee bc^*d\vee bc^*e$ Eq. (4)

Finally, we can obtain a sequence set $\{ac^*d, ac^*e, bc^*d, bc^*e\}$ from the expression $(a|b)c^*(d|e)$. In the above parsing process, we perform two algebraic operations on the expression $(a|b)c^*(d|e)$. One algebraic operation is $(a|b)c^* = ac^*|bc^*$, and the other algebraic operation is $(ac^*|bc^*)(d|e)$. Thus, if we want the computer to realize the above process, the expression $(a|b)c^*(d|e)$ needs to be decomposed into two subexpressions $(a|b)c^*$ and $(d|e)$ first. Then, the computer needs to select some proper algebraic operations to respectively process two subexpressions. Finally, the computer needs to combine those processed subexpressions into a new expression, and then determines whether it needs to be performed algebraic operation again. Based on this idea, we propose a general expression parsing framework, shown in Fig. 1.

From Fig. 1, the general expression parsing framework consists of four steps. In Step 1, an expression is decomposed into a set of subexpressions. In Step 2, we need to search the algebraic operation set supported by an algebraic system to find a proper algebraic operation for the parsing of a subexpression. A set

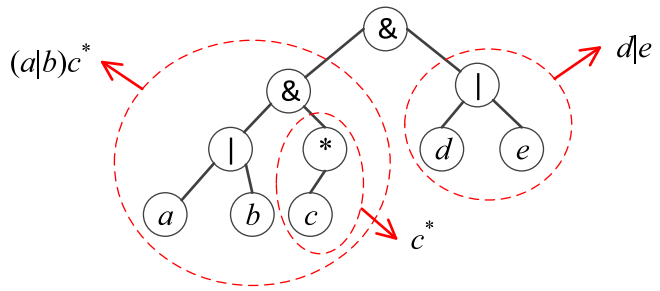

Fig. 2. The AST of expression $(a|b)c^*(d|e)$.

of new subexpressions will be obtained correspondingly. Note that, in this step, an algebraic operational set is composed of all the algebraic operations in an algebraic system. For example, Eqs. (1)–(15) in Section 2 can form a set of algebraic operations. In Step 3, all new subexpressions need to be combined into a new expression. In Step 4, if the new expression can be directly converted to a set of sequences, the set of sequences is outputted. Otherwise, the new expression needs to be decomposed again.

To implement this framework, three questions need to be answered.

RQ1: How to decompose an expression into a set of subexpressions in Step 1?

RQ2: How to combine those algebraically operated subexpressions into a new expression in Step 3?

RQ3: How to solve the problem of the infinite number of sequences in Step 4?

To answer the above three questions, an expression parsing method based on abstract syntax tree is developed.

5. The proposed AST-EP

5.1. Abstract syntax tree

In the past, abstract syntax trees, aka AST, have been widely used to parse regular expressions. For a nonempty expression, its AST structure is presented as follows:

- The AST is a binary tree;
- The leaves of the tree are the symbols in the expression;
- The non-leafs of the tree are the operators in the expression;
- The node that is a closure operator has one child node in the AST;
- The node that is a not-closure operator has two child nodes in the AST.

For example, we can construct an AST, as shown in Fig. 2, for expression $(a|b)c^*(d|e)$ by using Google tool ER2.

In Fig. 2, the root of the AST is operator &, its left subtree corresponds to subexpression $(a|b)c^*$, its right subtree corresponds to subexpression $d|e$, and the node * has one and only one subtree. Thus, in practice, the AST realizes the decomposition process of the expression. To parse expression $(a|b)c^*(d|e)$, we always process the left subtree with respect to subexpression $(a|b)c^*$ first, then the right subtree with respect to subexpression $d|e$, finally the root node of the AST in Fig. 2. Thus, the parsing process of an expression is similar to the post-order traversal process of its AST. The parsing steps in detail are as follows.

(1) Get a post-order list S = <a, b, |, c, *, &, d, e, |, &> from the AST in Fig. 2.

(2) Find the operator | in S and perform its algebraic operation on a and b. Next, we obtain $a|b$ and replace a, b, and | with $a|b$ in S, resulting in a new list S_1 = <$a|b$, c, *, &, d, e, |, &>.

(3) Find the operator * in S_1 and perform its algebraic operation on c. Next, we get c^* and replace c and * with c^* in S_1, resulting in a new list S_2= <$a|b$, c^*, &, d, e, |, &>.

(4) Find the operator & in S_2 and perform its algebraic operation on $a|b$ and c^*. Next, we obtain $(a|b)c^* = ac^*|bc^*$ and replace $a|b$, c^*, and & with $ac^*|bc^*$ in S_2, resulting in a new list S_3= < $ac^*|bc^*$, d, e, |, &>.

(5) Find the operator | in S_3 and perform its algebraic operation on d and e. Next, we obtain $d|e$ and replace d, e, and | with $d|e$ in S_3, resulting in a new list S_4= < $ac^*|bc^*$, $d|e$, &>.

(6) Find the operator & in S_4 and perform its algebraic operation on $ac^*|bc^*$ and $d|e$. Then, we obtain $(ac^*|bc^*)(d|e)$= ac^*d | ac^*e | bc^*d | bc^*e. Finally, four test sequences ac^*d, ac^*e, bc^*d, and bc^*e are obtained by Eq. (4).

The above parsing process is the core idea of our method. To ensure the validity of our method, we give some definitions and theorems in the following section.

5.2. Theoretical foundation

Definition 1 (*Post-Order List*)**.** A list obtained by searching an AST with the post-order traversal is called the post-order list, denoted by *Seq*.

Definition 2 (*Symbol Set*)**.** Let Σ denote the set of all symbols that are not-operators in expressions.

Definition 3 (*Operator Set*)**.** Let $\mathrm{Op} = \mathrm{Op}_1 \cup \mathrm{Op}_2$ denote the set of all operators in expressions, where

- Op_1 is the set of operators that are designed to handle symbols on either side of them in expressions, and
- Op_2 is the set of operators that are designed to handle symbols on the left side of them in expressions.

Generally, closure operators belong to Op_2 and other operators belong to Op_1. For example, in expression $(a|b)^*c$, there are $|,\&\in\mathrm{Op}_1$, $^*\in\mathrm{Op}_2$, and $a, b, c\in\Sigma$.

Definition 4 (*i-element*)**.** Let $Seq((i)$ represent the ith element in the post-order list.

We let i in i-element start at 1. Then, there is Seq = <$Seq(1)$, …, $Seq(n)$>.

Definition 5 (*First Operator*)**.** In *Seq*, if there are $Seq((i) \in \mathrm{Op}$ and $Seq(j) \in\Sigma$ for $1 \le j < i$, Seq$((i)$ is called the first operator.

Definition 6 (*Expression Segment*)**.** In expression *Re*, if *Re*' is a part of *Re*, *Re*' is called the expression segment of *Re*, which is marked as *Re*' *in Re*.

Note that referring to the description of a program segment (Rao et al., 2021), we give the definition of an expression segment. For example, $a|\ b$ is an expression segment of expression $(a|\ b)c$, which can be described as $a|b$ *in* $(a|b)c$.

Definition 7 (*Sequence Segment*)**.** In *Seq*, if *Seq*' is a part of *Seq*, *Seq*' is called the sequence segment of *Seq*, which is marked as *Seq*' *in Seq*.

Note that if *Re*' *in Re* exists, there is a *Seq*' with respect to *Re*' such that *Seq*' *in Seq* and *Seq* with respect to *Re*.

Theorem 1. *Let Seq be a post-order list with respect to expression Re. If* $Seq(i)\in Op_1$ *is the first operator in Seq, there is* $Seq(i-2)Seq(i)Seq(i-1)$ *in Re.*

Algorithm: AST-EP

Input: An AST of an expression, a hash table T for algebraic operators, and a vector V representing an algebraic operation set;

Output: A set of finite number of sequences.

```
1   Get Seq = <Seq(1), ···, Seq(n)> of the AST.
2   for (i = 1; i ≤ n; i++)//Search the first operator
3       if(Seq(i) ∈ Op1)
4           k1, k2=T.getKey(Seq(i));
5           Find an algebraic operation from V[k1] and V[k2];
6           x = R(<Seq(i-2), Seq(i-1), Seq(i)>;
7           Replace Seq(i-2), Seq(i-1), and Seq(i) with x in Seq, and then get Seq1;
8           i = i - 2;
9           Seq = Seq1;
10      end if
11      if(Seq(i) ∈ Op2)
12          x = ParsingClosure(Seq(i-1), Seq(i));
13          Replace Seq(i-1) and Seq(i) with x in Seq, and then get Seq1;
14          i = i - 1;
15          Seq = Seq1;
16      end if
17  end for
18  return a set of sequences in Seq;
```

Fig. 3. The design of AST-EP.

Proof. Because $Seq((i)$ is the first operator of Seq, there is $Seq(j) \in \Sigma$ for $1 \leq j < i$ by Definition 5. Because $Seq(i) \in Op_1$ exists, there are $x \in \Sigma$ and $y \in \Sigma$ in Re such that $xSeq(i)y$ in Seq according to the composition of Seq and Definition 1. Assume $Re' = xSeq(i)y$. Then, there is Re' in Re by Definition 6. Thus, according to the AST's structure of Re and Definition 1, there are $Seq\ (i-2) = x$ and $Seq\ (i-1) = y$. Thus, there is $Seq(i\text{-}2)Seq(i)Seq\ (i-1)$ in Re because of Re' in Re and $Re' = xSeq(i)y$. □

Theorem 2. *Let Seq be a post-order list with respect to expression Re. If $Seq(i) \in Op_2$ is the first operator of Seq, there are $Seq(i-1)Seq(i)$ in Re.*

The proof of Theorem 2 is similar to that of Theorem 1, and its proof is ignored.

Definition 8 (*Result Set*)**.** Let Ω be the set of results of algebraic operations on Σ.

Lemma 1. *Any expression supported by Kleene algebra and transition algebra must be converted to a set of sequences.*

The proof of Lemma 1 can be found in Liu et al. (2021), Liu and Miao (2014), Kozen and Smith (1996). According to Lemma 1, Ω is a set of sequences. Thus, suppose $\Omega = \{a_1a_2\ldots\ a_n,\ b_1b_2\ldots\ b_n, \ldots\}$, where $a_i,\ b_i \in \Sigma$. Let be $x = a_1a_2\ldots\ a_n$ and $y = b_1b_2\ldots\ b_n$. Then, $\Omega = \{x, y, \ldots\}$ is a set of symbols. Thus, there is $\Omega \subseteq \Sigma$ by Definition 2. Then we can perform algebraic operations on Ω.

Note that Lemma 1 only involves Kleene algebra and transition algebra. Since several algebraic systems, including Kleene algebra with tests, concurrent Kleene algebra, NetKAT, and Concurrent NetKAT, are built upon Kleene algebra, they also fulfill Lemma 1. Unfortunately, not all algebraic systems can provide a proof similar to Lemma1. Therefore, we propose prioritizing the use of algebraic systems that satisfy Lemma 1 for software modeling.

Definition 9 (*Operational Result*)**.** Let $R(A)$ denote the result of algebraic operations on expression Re, where A is the post-order list of an AST with respect to Re.

Theorem 3. *Assume that A = <Seq(1), Seq(2), …, Seq (i − 2), Seq (i − 1), Seq(i), …, Seq(n)> is a post-order list of an AST with respect to Re, and Seq(i) is the first operator in A. Then, R(A) satisfies the following two properties:*

(1) If $Seq(i) \in Op_1$, C = <Seq (i − 2), Seq (i − 1), Seq(i)>, and B = <Seq(1), Seq(2), …, Seq (i − 3), R(C), Seq(i + 1), …, Seq(n)> exist, there is $R(A) = R(B)$.

(2) If $Seq(i) \in Op_2$, C = <Seq (i − 1), Seq(i)>, and B = <Seq(1), Seq(2), …, Seq (i − 2), R(C), Seq(i + 1), …, Seq(n)>, there is $R(A) = R(B)$.

Proof. for (1):

Because $Seq(i) \in Op_1$ is the first operator in A, there are $Seq\ (i-2)$ and $Seq\ (i-1)$ such that $Seq(i\text{-}2)Seq(i)Seq\ (i-1)$ in Re by Theorem 1. Assume $Re' = Seq(i\text{-}2)Seq(i)Seq\ (i-1)$. Then, there is Re' in Re. Because there are C = <Seq (i − 2), Seq (i − 1), Seq(i)> and $Seq((i)$ is the first operator in A, $R(C)$ is the first step of $R(A)$. After we get $R(C)$ from A, A is converted to B. Thus, there is $R(A) = R(B)$. □

The proof for (2) in Theorem 3 is similar to that for (1) in Theorem 3, and its proof is ignored. In Theorem 3, there is $R(C) \in \Omega$. If we assume $x = R(C)$, then there is B = <Seq(1), Seq(2), ···, x, Seq(i+1), ···, Seq(n)>. The first operator in B must be found in <Seq(i+1), ···, Seq(n)>. Thus, we can process B again by Theorem 3 until there is no operator in B. Finally, we can always get a set of sequences from $R(B)$ according to Lemma 1.

5.3. Algorithm

Based on theoretical analysis in Section 5.2, we design an algorithm (named as AST-EP), shown in Fig. 3, to realize expression parsing based on AST.

AST-EP uses a hash table T for algebraic operators and a vector V representing an algebraic operation set to quickly obtain an algebraic operation of an operator. The relation between T and V is shown in Fig. 4. From Fig. 4, the value of T is an operator and the key of T contains two subscripts k and k+m of the vector T, where $V[k]$, …, and $V[k+m]$ store all algebraic operations for the

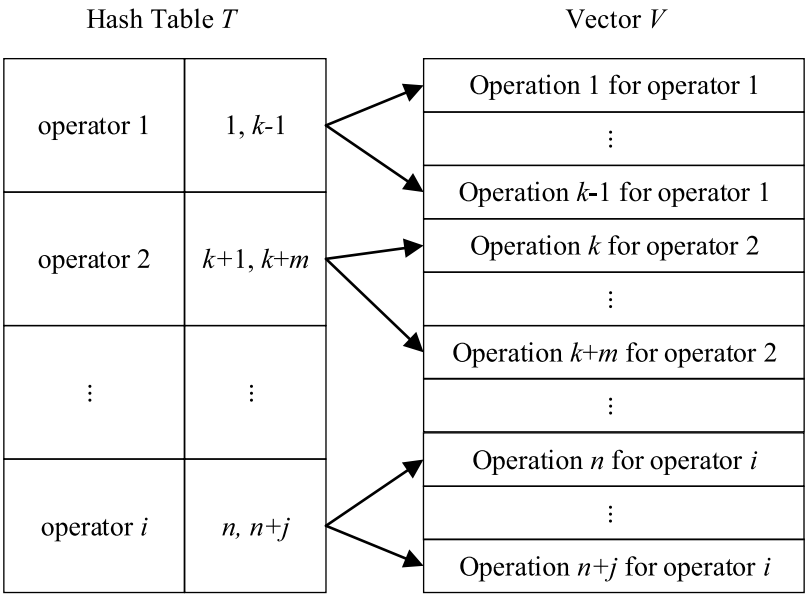

Fig. 4. The relation between T and V.

```
string ParsingClosure(string Seq1, string Seq2)// Seq1 is an expression and Seq2 is a closure operator.
1   if (Seq2 == "*" && ! Seq1.contain(|))
2       Get a string st according to Equations (16) and (18);
3   else if (Seq2 == "*" && Seq1.contain(|))
4       Get a string st according to Equations (17) and (18);
5   else if (Seq2 == "+") //"+" denotes a positive closure
6       st = ParsingClosure(Seq1, "*");
7       st = Seq1 + st;//concatenation of Seq1 and st
8   else if (Seq2 is the other closure)
9       Find an algebraic operation of Seq2 in the algebraic operation set
10      Perform closure operation for Seq(i-1) by Seq(i) and then get the result st;
11  endif
12  return st;
```

Fig. 5. The design of *ParsingClosure*.

operator. Thus, we can get a suitable algebraic operation from V by the key of T to process expression.

Additionally, AST-EP contains a closure parsing function ***ParsingClosure***. For Kleene Closure and positive closure, there is a problem of the infinite number of sequences, i.e., $a^* = 1 \mid a^1 \mid a^2 \mid \ldots$ in Eq. (8). If this problem is not properly addressed, we cannot perform software testing by the infinite number of test sequences. To solve this problem, we assume

$$a^* \approx 1|a|a^2|\cdots|a^k \tag{16}$$

By Eq. (16), we can transform an infinite number of sequences into a set of finite number of sequences. However, for some complex expressions, i.e., $(a|b|c)^*$, existing algebraic systems do not provide an algebraic operation to process them for generating a set of finite number of sequences. Thus, we give a following equation:

$$(a_1|\cdots|a_n)^* \approx 1|a_1|\cdots|a_n|(a_1|\cdots|a_n)^2|\cdots|(a_1|\cdots|a_n)^k \tag{17}$$

To determine the value of k in Eqs. (16) and (17), we present an empirical equation as follows:

$$k = \begin{cases} 3 & if\ a = a_1 a_2 \cdots a_n \wedge a_i \in \sum \\ n & if\ a = a_1|a_2|\cdots|a_n \wedge a_i \in \sum \end{cases} \tag{18}$$

By Eqs. (16), (17), and (18), we design the closure parsing function ***ParsingClosure***, shown in Fig. 5.

The core idea of AST-EP is to parse an expression according to Theorem 3. The time complex of this algorithm is $O(n \times m)$, where n denotes the number of all symbols and all operators in the expression, m denotes the size of the algebraic operation set.

In a word, to answer **RQ1**, the AST-based expression parsing method gets a post-order list from the AST of an expression. To answer **RQ2**, the process of replacing part of the post-order list with the result of an algebraic operation on subexpressions is introduced. To answer **RQ3**, we present equations (16)–(18), and design a closure parsing function ***ParsingClosure***.

6. An example

In this section, we use expression $(a|b)c(d|\ e^*)(f\|gh)$ to demonstrate the implementation of AST-EP. In expression $(a|b)c(d|\ e^*)(f\|gh)$, there are four operators &, |, *, and ||, where operator * needs to be parsed by the function ***ParsingClosure***. To parse expression $(a|b)c(d|\ e^*)(f\|gh)$, we first construct an AST, shown in Fig. 6. Then, AST-EP can get a post-order list Seq = <a, b, |, c, &, d, e, *, |, &, f, g, h, &, ||, &> from the AST in Fig. 3. The operational process of expression $(a|b)c(d|\ e^*)(f\|gh)$ in Seq is shown in Table 2.

Table 2
The operational process of expression $(a|b)c(d|e^*)(f||\ gh)$.

No.	Sequence	First Op.	Algebraic operation	New sequence
1	<a, b, |, c, &, d, e, *, |, &, f, g, h, &, ||, &>	|	$a\|b = a\|b$	<a|b, c, &, d, e, *, |, &, f, g, h, &, ||, &>
2	<a|b, c, &, d, e, *, |, &, f, g, h, &, ||, &>	&	$(a\|b)c=ac\|bc$	<ac|bc, d, e, *, |, &, f, g, h, &, ||, &>
3	<ac|bc, d, e, *, |, &, f, g, h, &, ||, &>	*	$e^*\approx 1\|e\|e^2\|e^3$	<ac|bc, d, $1\|e\|e^2\|e^3$, |, &, f, g, h, &, ||, &>
4	<ac|bc, d, $1\|e\|e^2\|e^3$, |, &, f, g, h, &, ||, &>	|	$d\|(1\|e\|e^2\|e^3)=d\|1\|e\|e^2\|e^3$	<ac|bc, $d\|1\|e\|e^2\|e^3$, &, f, g, h, &, ||, &>
5	<ac|bc, $d\|1\|e\|e^2\|e^3$, &, f, g, h, &, ||, &>	&	$(ac\|bc)\ (d\|1\|e\|e^2\|e^3)=acd\|ac\|ace\|ace^2\|ace^3\|bcd\|bc\|bce\|bce^2\|bce^3$	<acd|ac|ace|ace^2|ace^3|bcd|bc|bce|bce^2|bce^3, f, g, h, &, ||, &>
6	<acd|ac|ace|ace^2|ace^3|bcd|bc|bce|bce^2|bce^3, f, g, h, &, ||, &>	&	$gh=gh$	<acd|ac|ace|ace^2|ace^3|bcd|bc|bce|bce^2|bce^3, f, gh, ||, &>
7	<acd|ac|ace|ace^2|ace^3|bcd|bc|bce|bce^2|bce^3, f, gh, ||, &>	||	$f\|\|\ gh=fgh\|gfh\|ghf$	<acd|ac|ace|ace^2|ace^3|bcd|bc|bce|bce^2|bce^3, fgh|gfh|ghf, &>
8	<acd|ac|ace|ace^2|ace^3|bcd|bc|bce|bce^2|bce^3, fgh|gfh|ghf, &>	&	$(acd\|ac\|ace\|ace^2\|ace^3\|bcd\|bc\|bce\|bce^2\|bce^3)(fgh\|gfh\|ghf)=acdfgh\|acdgfh\|acdghf\|acfgh\|acgfh\|acghf\|acefgh\|acegfh\|aceghf\|ace^2fgh\|ace^2gfh\|ace^2ghf\|ace^3fgh\|ace^3gfh\|ace^3ghf\|bcdfgh\|bcdgfh\|bcdghf\|bcfgh\|bcgfh\|bcghf\|bcefgh\|bcegfh\|bceghf\|bce^2fgh\|bce^2gfh\|bce^2ghf\|bce^3fgh\|bce^3gfh\|bce^3ghf$	<acdfgh|acdgfh|acdghf|acfgh|acgfh|acghf|acefgh|acegfh|aceghf|ace^{2}fgh|ace^{2}gfh|ace^{2}ghf| ace^{3}fgh|ace^{3}gfh|ace^{3}ghf|bcdfgh|bcdgfh|bcdghf|bcfgh|bcgfh|bcghf|bcefgh|bcegfh|bceghf|bce^{2}fgh|bce^{2}gfh|bce^{2}ghf|bce^{3}fgh|bce^{3}gfh|bce^{3}ghf>

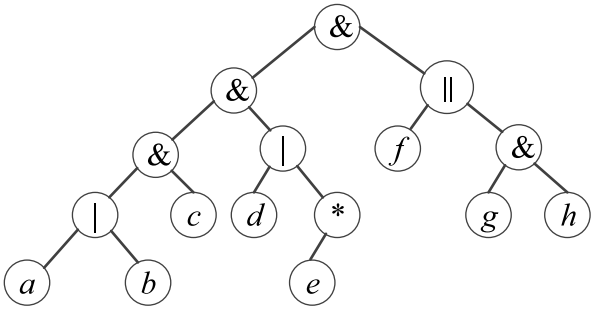

Fig. 6. The AST of expression $(a|b)c(d|\ e^*)(f\|gh)$.

From Table 2, after eight algebraic operations, AST-EP outputs 30 sequences *acdfgh*, *acdgfh*, *acdghf*, *acfgh*, *acgfh*, *acghf*, *acefgh*, *acegfh*, *aceghf*, *ace^{2}fgh*, *ace^{2}gfh*, *ace^{2}ghf*, *ace^{3}fgh*, *ace^{3}gfh*, *ace^{3}ghf*, *bcdfgh*, *bcdgfh*, *bcdghf*, *bcfgh*, *bcgfh*, *bcghf*, *bcefgh*, *bcegfh*, *bceghf*, *bce^{2}fgh*, *bce^{2}gfh*, *bce^{2}ghf*, *bce^{3}fgh*, *bce^{3}gfh*, and *bce^{3}ghf* from expression $(a|b)c(d|\ e^*)(f\|gh)$ by Eq. (4).

We also incorporated AST-EP into MTTool (Liu and Xu, 2018), a test modeling and test generation tool we developed earlier. Fig. 7 shows the construction of an AST and test sequence generation for the tree according to AST-EP.

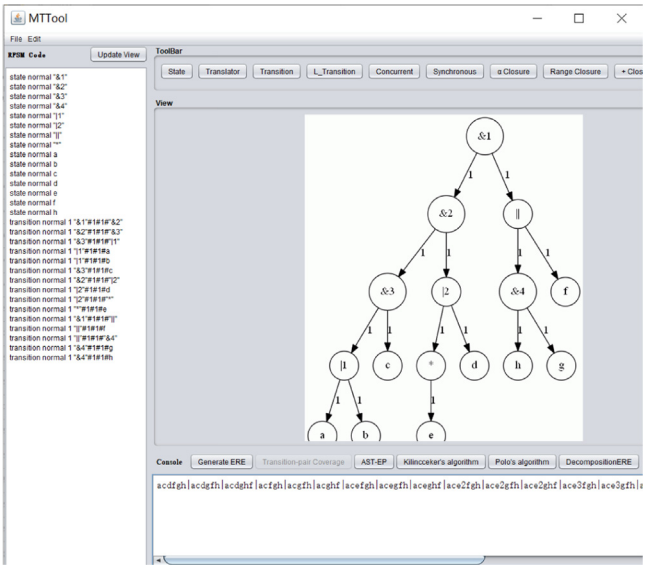

Fig. 7. A support tool for AST-EP.

7. Case studies

7.1. Experiment design

Experiments are designed to evaluate the performance between the proposed AST-EP and three existing algorithms, including DecompositionERE (Liu and Xu, 2018), Kilincceker et al.'s algorithm (Kilincceker et al., 2018), and Polo et al.'s algorithm (Polo et al., 2020). The evaluation is two folded: (1) the expression parsing capability and (2) the fault detection effectiveness.

To evaluate the expression parsing capability, the following three steps are involved:

Step 1: We use keywords such as "regular expression", "extended regular expression", and "algebraic system for test" in Google Scholar,[4] and collect 117 distinct expressions from 47 related papers over the past 30 years.

Step 2: We use Google tool RE2 to generate the ASTs of these expressions.

Step 3: All four algorithms are applied to parse each of the 117 expressions directly (for DecompositionERE and Polo et al.'s algorithm) or the corresponding ASTs from Step 2 (for Kilincceker et al.'s algorithm and AST-EP).

Definition 10 (*Expression Parsing Rate*)**.** The expression parsing rate of an algorithm is defined as

$$EPR = N_i/N \tag{19}$$

where N_i denotes the number of expressions that can be parsed by the algorithm and N denotes the number of all expressions.

To evaluate the fault detection effectiveness, the following three steps are involved:

Step 1: We recruited 66 students as volunteers to perform crowdsource testing (Liu and Li, 2022; Liu et al., 2020a) on 13 open source software. We first trained volunteers to download 13 open source software and test them using random testing

[4] scholar.google.com.

Table 3
The information of 13 open-source software.

Software	Version	Size (MB)	Release time	Description
SurveyKing[a]	1.4.0	11.2	2022–12–11	A questionnaire and examination system
ThuThesis[b]	7.3.1	1.9	2022–10–05	A Latex typesetting software
Jenkins[c]	2.375.1	95.1	2022–11–30	An open source automation server
VSCodium[d]	1.74.2	109.7	2022–12–22	A cross-platform, free source code editor
7-zip[e]	7z2200	111	2022–07–15	A file filer with a high compression ratio
Putty[f]	0.78	75.3	2022–10–29	A Telnet, SSH, rlogin, pure TCP, and serial interface connection software
Blender[g]	3.4	175,493.12	2022–12–20	A free open source 3D graphics image software
DBeaver[h]	22.3.1	112.72	2022–12–26	A common database management tool and SQL client
VideoLAN[i]	3.0.9.2	104,919.04	2022–11–08	A video player and video server
Finallshell[j]	3.9.7.6	83.61	2022–08–09	An open source remote service connection software
JMeter[k]	5.4.3	71	2021–12–24	A Java-based development stress testing tool
GeekDesk[l]	2.5.13	3.57	2022–09–02	A compact desktop quick startup management tool
FreeCAD[m]	0.20.2	455.9	2022–12–30	An open source CAD/CAE tool

[a] https://www.oschina.net/news/206097.
[b] https://mirrors.tuna.tsinghua.edu.cn/github-release/tuna/thuthesis/.
[c] https://mirrors.tuna.tsinghua.edu.cn/jenkins/.
[d] https://mirrors.tuna.tsinghua.edu.cn/github-release/VSCodium/vscodium/.
[e] https://mirrors.nju.edu.cn/7-zip/.
[f] https://mirrors.nju.edu.cn/putty/.
[g] https://mirrors.tuna.tsinghua.edu.cn/blender/.
[h] https://mirrors.nju.edu.cn/github-release/dbeaver/dbeaver/.
[i] https://mirrors.nju.edu.cn/videolan-ftp/.
[j] http://www.hostbuf.com/downloads/finalshell_install.exe.
[k] https://mirrors.tuna.tsinghua.edu.cn/apache/jmeter/binaries/.
[l] https://github.com/BookerLiu/GeekDesk/releases.
[m] https://mirrors.tuna.tsinghua.edu.cn/github-release/FreeCAD/FreeCAD/.

methods to find software bugs. These bugs have different manifestations. They are related to the performance, design, and execution of the software. Table 3 shows software information, the number of software bugs, and software function descriptions associated with software bugs.

Step 2 We designed 13 expression models for software functions that involve software bugs in Table 3. Then, we used four algorithms AST-EP, DecompositionERE, Kilincceker et al.'s algorithm, and Polo et al.'s algorithm to parse 13 expression models respectively. Next, four sets of test sequences were constructed for each software.

Step 3: We take a test sequence as an execution path of software, and then execute software according to test sequences and record bugs in the software's execution.

Definition 11 (*Software Fault Detection Effectiveness*)**.** The software fault detection effectiveness (*SFDE* for short) of test sequences generated by an algorithm is defined as

$$SFDE = \frac{\sum_{i=1}^{n} B_i}{\sum_{i=1}^{n} C_i} \tag{20}$$

where B_i denotes the number of bugs detected by test sequences on software i, and C_i denotes the number of bugs detected in the crowdsource testing on software i.

7.2. Results

Regarding the expression parsing capability, the results are shown in Fig. 8. As observed from the figure, the *EPR* of AST-EP is 100%, the *EPR* of DecompositionERE is 46%, the *EPR* of Polo et al.'s algorithm is 37%, and the *EPR* of Kilincceker et al.'s algorithm is 32% according to Eq. (17). Therefore, compared with the other three algorithms, the *EPR* of AST-EP is highest on 117 expressions. Except for AST-EP, the *EPR* of the other three algorithms is less than 50%, which indicates that these algorithms do not well support expressions constructed by algebraic systems.

Table 4 shows numbers of software bugs detected by test sequences on 13 open source software. As shown in Table 4, test

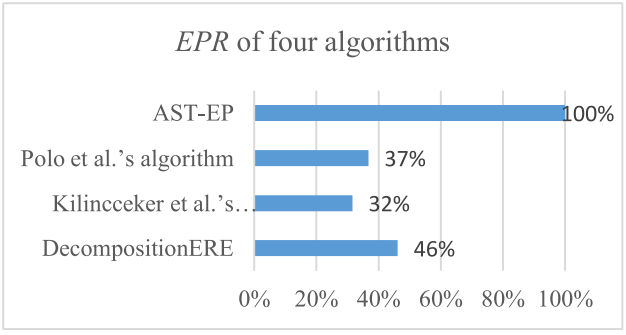

Fig. 8. The expression parsing rates of four algorithms.

sequences generated by AST-EP successfully detect all software bugs recorded by volunteers in crowdsource testing. Test sequences generated by DecompositionERE can detect bugs of 12 software except for 7-zip. With the exception of SurveyKing, test sequences generated by Kilincceker et al.'s algorithm and Polo et al.'s algorithm have the same software fault detection capability. The reason for this is that the expression model of 7-zip contains operators that DecompositionERE, Kilincceker et al.'s algorithm, and Polo et al.'s algorithm cannot parse, thus these algorithms cannot generate effective test sequences to detect software bugs in 7-zip. Furthermore, by comparing Tables 3 and 4, for VideoLAN, we found that test sequences generated by AST-EP detect two more bugs than those found by volunteers in crowdsource testing.

According to Eq. (18), we calculate *SFDE*s of test sequences generated by four algorithms as shown in Fig. 9. As observed, test sequences generated by AST-EP have the highest *SFDE*, followed by those generated by DecompositionERE, next test sequences generated by Polo et al.'s algorithm, finally by Kilincceker et al.'s algorithm. Furthermore, the *SFDE* of the test sequence generated by AST-EP is 1.04, which indicates that test sequences generated by AST-EP are more effective in terms of fault detection capability.

Table 4
The number of bugs detected by test sequences generated by four expression parsing algorithms on 13 software.

Software	Number of bugs detected by test sequences			
	DecompositionERE	Kilincceker et al.'s algorithm	Polo et al.'s algorithm	AST-EP
SurveyKing	5	4	5	**6**
ThuThesis	2	2	2	2
Jenkins	1	1	1	1
VSCodium	3	3	3	3
7-zip	0	0	0	**6**
Putty	13	9	10	13
Blender	2	2	2	2
DBeaver	1	1	1	1
VideoLAN	4	3	3	**7**
Finallshell	2	0	0	2
JMeter	3	3	3	3
GeekDesk	2	2	2	2
FreeCAD	2	2	2	2

Table 5
Comparison of four expression parsing algorithms for test sequence generation.

Item	DecompositionERE	Kilincceker et al.'s algorithm	Polo et al.'s algorithm	AST-EP
Supported Model	Extended Regular expression	Regular expression	Regular expression	Extended Regular expression
Supported algebraic operations	Concatenation, Choice, Star, concurrence, synchronization	Concatenation, Choice, Star	Concatenation, Choice, Kleene Closure, Positive Closure	All algebraic operations
Supported algebraic system	Transition algebra	Kleene algebra	Kleene algebra	All algebraic systems
Test generation method	String analysis	AST and preorder traversal	Set and constraint operation	AST, post-order traversal

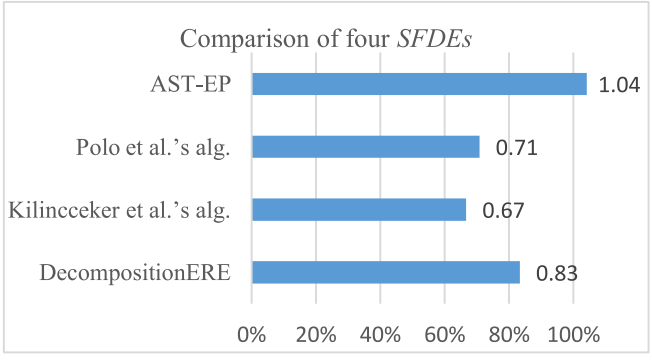

Fig. 9. The *SFDEs*' comparison of test sequences generated by four algorithms.

7.3. Discussion

We construct Table 5 to discuss the characteristic of four expression parsing algorithms. As observed from the table, algorithms DecompositionERE and AST-EP support the generation of test sequences from extended regular expressions, while both Kilincceker et al.'s and Polo et al.'s algorithms only support the generation of test sequences from regular expressions. Among the four algorithms, AST-EP supports the most algebraic operations, followed by DecompositionERE, followed by Polo et al.'s algorithm, and finally Kilincceker et al.'s algorithm. DecompositionERE can support part of algebraic operations in transition algebra, both Kilincceker et al.'s algorithm and Polo et al.'s algorithm support algebraic operations in Kleene algebra, and AST-EP supports all algebraic operations in different algebraic systems. DecompositionERE uses the method of string analysis to realize the generation of test sequences from expressions. Kilincceker et al.'s algorithm needs to convert an expression to an AST, and then get test sequences by using the preorder traversal method. Polo et al.'s algorithm uses set operations and a length constraint operation to generate test sequences. AST-EP uses the post-order traversal method to get test sequences. To sum up, AST-EP can support more algebraic operations in algebraic systems to generate test sequences from the expression model than the other three algorithms. In other words, it would be handy for researchers and practitioners to develop expression parsing systems for new algebraic systems and algebraic operations in the future, thereby promoting the industrial application of these new algebraic systems.

8. Threats to validity

Our method has two potential threats. The first threat is that existing tools, such as Neamtiu et al.'s tool (Neamtiu et al., 2005), CSS (Cui et al., 2010), ATOM (Liu et al., 2020b), pycparser (Zhang et al., 2019), javalang (Zhang et al., 2019), and re2, for generating ASTs only support regular expressions and do not support a wide range of extended regular expressions. For example, the interleaving operator $\|$ is taken as two choice operators in RE2. To mitigate this threat, we have re-developed these tools so that they support the parsing of extended regular expressions. The second threat is the redundant test sequences. Parsing expressions in terms of algebraic operations may lead to redundant sequences. For example, there can be redundant sequences between a^* and a^+ because of $a^* = 1 \mid a^1 \mid a^2 \mid \cdots$ and $a^+ = a^1 \mid a^2 \mid \cdots$. Such redundancy will not only increase the time of expression parsing, but also increase the cost of software testing. To alleviate, we have used algebraic operations to remove redundant sequences. Thus, for $(a^* \mid a^+)$ we can obtain $(a^* \mid a^+) \approx 1 \mid a^1 \mid a^2 \mid a^3 \mid a^1 \mid a^2 \mid a^3 = 1 \mid a^1 \mid a^2 \mid a^3$ by Eq. (3). In this way, the redundant sequences and operations are reduced.

9. Conclusion and future work

Algebraic systems such as Kleene algebra and transition algebra have enriched expression modeling and test generation theory. Applying algebraic operations to parse expressions is the

key to test sequence generation. However, such parsing can be very challenging as different algebraic systems have different algebraic operations. In addition, the parsing difficulty continues to grow as software complexity increases. To address these challenges, a general expression parsing framework is proposed. Specifically, we put forward AST-EP, a parsing algorithm based on abstract syntax tree and establish its theoretical basis. Case studies based on 117 expressions collected from the literature over the past 30 years as well as 13 software systems are conducted to demonstrate its superior parsing and fault detection performance compared to three existing parsing algorithms.

In addition to the two concerns mentioned earlier in Section 1, AST-EP currently lacks the ability to encompass all existing operators, such as $\diamond$ and $\partial_a(s{\cdot}t)$. This limitation arises from the fact that current conversion tools, although powerful like RE2, are unable to handle these operators and generate the corresponding AST. As a result, one of our future tasks will focus on developing an AST conversion tool that is compatible with all existing operators, in order to align with the proposed AST-EP.

Furthermore, our future plans also involve integrating program analysis tools with the constraint operators proposed in Liu et al. (2021), we previously constructed a new algebraic operator called the constraint operator in transition algebra. This operator allows for constraint operations on expression models, enabling them to generate test sequences that satisfy the given constraints. Therefore, in the future, we will use a coverage criterion as a constraint condition and utilize algebraic operations designed for constraint operators to achieve coverage-based test generation.

CRediT authorship contribution statement

Pan Liu: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing – original draft, Writing – review & editing, Supervision. **Yihao Li:** Conceptualization, Methodology, Validation, Formal analysis, Writing – original draft, Writing – review & editing.

Declaration of competing interest

There are no conflicts of interest to declare.

Acknowledgments

This work was supported in part by National Social Science Fund General Project of China under Grant 18BTQ058 and National Natural Science Foundation of China under Grant 61502299.

References

Antimirov, V., 1996. Partial derivatives of regular expressions and finite automaton constructions. Theoret. Comput. Sci. 155 (2), 291–319.
Borsotti, A., Trafimovich, U., 2022. A closer look at TDFA. arXiv preprint arXiv: 2206.01398.
Borsotti, A., Trofimovich, U., 2021. Efficient POSIX submatch extraction on nondeterministic finite automata. Softw. - Pract. Exp. 51 (2), 159–192.
Brzozowski, J.A., 1964. Derivatives of regular expressions. J. ACM 11 (4), 481–494.
Cardoso, E.M., Amaro, M., da Silva Feitosa, S., dos Santos Reis, L.V., Bois, A., Ribeiro, R.G., 2021. The design of a verified derivative-based parsing tool for regular expressions. CLEI Electron. J. 24 (3).
Carlson, J., Lisper, B., 2004. An event detection algebra for reactive systems. In: Proceedings of the 4th ACM International Conference on Embedded Software. ACM, pp. 147–154.
Cui, B., Li, J., Guo, T., Wang, J., Ma, D., 2010. Code comparison system based on abstract syntax tree. In: 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology. IC-BNMT, IEEE, pp. 668–673.
Garg, V.K., Ragunath, M., 1992. Concurrent regular expressions and their relationship to Petri nets. Theoret. Comput. Sci. 96 (2), 285–304.
Gibney, D., Thankachan, S.V., 2021. Text indexing for regular expression matching. Algorithms 14 (5), 133.
Hinze, A., Voisard, A., 2015. EVA: An event algebra supporting complex event specification. Inf. Syst. 48, 1–25.
Hoare, C.T., Möller, B., Struth, G., Wehrman, I., 2009. Concurrent Kleene algebra. In: International Conference on Concurrency Theory. Springer, pp. 399–414.
Hoare, T., Möller, B., Struth, G., Wehrman, I., 2011. Concurrent Kleene algebra and its foundations. J. Log. Algebr. Program. 80 (6), 266–296.
Hoare, T., van Staden, S., Möller, B., Struth, G., Zhu, H., 2014. Developments in concurrent Kleene algebra. J. Log. Algebraic Methods Program. 1–18.
Jipsen, P., 2014. Concurrent Kleene algebra with tests. In: International Conference on Relational and Algebraic Methods in Computer Science. Springer, pp. 37–48.
Jipsen, P., Moshier, M.A., 2015. Concurrent Kleene algebra with tests and branching automata. J. Log. Algebraic Methods Program. 85 (4), 637–652.
Kilincceker, O., Turk, E., Challenger, M., Belli, F., 2018. Regular expression based test sequence generation for HDL program validation. In: 2018 IEEE International Conference on Software Quality, Reliability and Security Companion. QRS-C, IEEE, pp. 585–592.
Kilincceker, O., Silistre, A., Challenger, M., Belli, F., 2019. Random test generation from regular expressions for graphical user interface (GUI) testing. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion. QRS-C, IEEE, pp. 170–176.
Kozen, D., 1981. On induction vs.*-continuity. In: Workshop on Logic of Programs. Springer, pp. 167–176.
Kozen, D., 1994. A completeness theorem for Kleene algebras and the algebra of regular events. Inform. and Comput. 110 (2), 366–390.
Kozen, D., 1997. Kleene algebra with tests. ACM Trans. Program. Lang. Syst. (TOPLAS) 19 (3), 427–443.
Kozen, D., 2017. On the coalgebraic theory of Kleene algebra with tests. In: Rohit Parikh on Logic, Language and Society. Springer, pp. 279–298.
Kozen, D., Smith, F., 1996. Kleene algebra with tests: Completeness and decidability. In: International Workshop on Computer Science Logic. Springer, pp. 244–259.
Kuklewicz, C., 2007. Regular expressions/bounded space proposal. URL: http://wiki.haskell.org/index.php.
Liu, P., Ai, J., Xu, Z.J., 2017. A study for extended regular expression-based testing. In: Computer and Information Science (ICIS), 2017 IEEE/ACIS 16th International Conference on. IEEE, pp. 821–826.
Liu, D., Feng, Y., Zhang, X., Jones, J., Chen, Z., 2020a. Clustering crowdsourced test reports of mobile applications using image understanding. IEEE Trans. Softw. Eng. 1–19.
Liu, S., Gao, C., Chen, S., Yiu, N.L., Liu, Y., 2020b. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. IEEE Trans. Softw. Eng. 48 (5), 1800–1817.
Liu, P., Li, Y., 2022. Response time evaluation of mobile applications combining network protocol analysis and information fusion. Inf. Softw. Technol. 106838.
Liu, P., Li, Y., Miao, H., 2021. Transition algebra for software testing. IEEE Trans. Reliab. 70 (4), 1438–1454.
Liu, P., Miao, H., 2014. Theory of test modeling based on regular expressions. In: Structured Object-Oriented Formal Language and Method. Springer, pp. 17–31.
Liu, P., Xu, Z., 2018. MTTool: A tool for software modeling and test generation. IEEE Access 6, 56222–56237.
Myers, G., 1992. A four Russians algorithm for regular expression pattern matching. J. ACM 39 (2), 432–448.
Neamtiu, I., Foster, J.S., Hicks, M., 2005. Understanding source code evolution using abstract syntax tree matching. In: Proceedings of the 2005 International Workshop on Mining Software Repositories. pp. 1–5.
Ng, K.C., 1984. Relation Algebras with Transitive Closures Doctoral dissertation. University of California, Berkeley.
Polo, M., Pedreira, O., Places, Á.S., Garcia Rodriguez de Guzman, I., 2020. Automated generation of oracled test cases with regular expressions and combinatorial techniques. J. Software: Evol. Process 32 (12), e2273.
Rao, L., Liu, S., Liu, A., 2021. Testing program segments to detect software faults during programming. Int. J. Perform. Eng. 17 (11).
Schmid, T., Kappé, T., Silva, A., 2023. A complete inference system for skip-free guarded Kleene algebra with tests. In: Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27 2023, Proceedings. Springer, pp. 309–336.
Smolka, S., Foster, N., Hsu, J., Kappé, T., Kozen, D., Silva, A., 2019. Guarded Kleene algebra with tests: Verification of uninterpreted programs in nearly linear time. Proc. ACM Program. Languages 4 (POPL), 1–28.
Wagemaker, J., Foster, N., Kappé, T., Kozen, D., Rot, J., Silva, A., 2022. Concurrent NetKAT. In: European Symposium on Programming. Springer, Cham, pp. 575–602.
Wang, Z., Zhang, Y., Gao, P., Shuang, S., 2021. Comparing fault detection efficiencies of adaptive random testing and greedy combinatorial testing for Boolean-specifications. Int. J. Perform. Eng. 17 (1).
Zeng, M., Liu, P., Miao, H., 2014. The design and implementation of a modeling tool for regular expressions. In: Advanced Applied Informatics (IIAIAAI), 2014 IIAI 3rd International Conference on. IEEE, pp. 726–731.
Zhang, C., de Amorim, A.A., Gaboardi, M., 2022. On incorrectness logic and Kleene algebra with top and tests. Proc. ACM Program. Lang. 6, 1–30.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 783–794.

Pan Liu was born in Nanchang, Jiangxi, China in 1976. He received the M.Sc. degree in computer software and theory from Nanchang University in 2006 and the Ph.D. degree in computer application from Shanghai University in 2011. From 1997 to 2006, he was a senior software engineer and worked in several software companies in China. He has developed dozens of software products using Java language and C++ language. From 2006 to 2008, he was a senior lecturer at Nanchang University in China, and an associate professor at Shanghai Business School in China from 2011 to 2018. Now, he was a professor at college of information and computer, Shanghai Business School, Shanghai, China. His papers have been published in some well-known international Journals and IEEE conferences. His main interests include software testing, model-based testing, formal method, and data analysis and modeling. Pan Liu can be contacted at panl008@163.com.

Yihao Li received the B.S. degree in software engineering from the East China Institute of Technology, the M.S. degree in computer science from Southeastern Louisiana University, and the Ph.D. degree in Software Engineering from The University of Texas at Dallas, USA. He held a postdoctoral position at the Graz University of Technology, Austria. He is currently an Associate Professor with the School of Information and Electrical Engineering, Ludong University, China. His research interests include intelligent transport, software quality assurance, testing, and debugging.