4-2014

Automated Oracle Generation via Denotational Semantics

Liang Cao
University of Nebraska at Omaha

Automated Oracle Generation via Denotational Semantics

A thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Liang Cao

April 2014

Supervisory Committee:

Dr. Haifeng Guo

Dr. Harvey Siy

Dr. Matt Germonprez

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Automated Oracle Generation via Denotational Semantics

Liang Cao, MS

University of Nebraska, 2014

Advisor: Dr. Haifeng Guo

Abstract

Software failure detection is typically done by comparing the running behaviors from a software under test (SUT) against its expected behaviors, called test oracles. In this paper, we present a formal approach to specifying test oracles in denotational semantics for systems with structured inputs. The approach introduces formal semantic evaluation rules, based on the denotational semantics methodology, defined on each productive grammar rule. We extend our grammar-based test generator, GENA, with automated test oracle generation. We provide three case studies of software testing: (i) a benchmark of Java programs on arithmetic calculations, (ii) an open source software on license identification, and (ii) selenium-based web testing. Experimental results demonstrate the effectiveness of our approach and illustrate the success of the application on the software testing.

Acknowledgments

I would like to thank all those who helped me complete my thesis. First and foremost, I would like to recognize Dr. Haifeng Guo, my supervisory committee chairman and advisor, for his guidance, patience and academic visions in the completion of this thesis. I really appreciate him very much for being there every step of the way.

I would also like to thank the rest of my thesis committees, Dr. Matt Germonprez for helping me develop the graduate background in the methodology studies, and for his great suggestions, Dr. Harvey Siy for his encouragement, good questions, and insightful comments.

I want to thank for Dr. Zongyan Qiu, who spent time to help me to discuss the research questions and provide very helpful suggestions.

Especially, I would like to thank for Yu-Shu Song who cooperated the implementation of our approach and provided very helpful experimental results for the research.

Lastly but not least, I would like to thank my parents for their understanding, unconditional support and encouragement to pursue my education.

Table of Contents

Multimedia Objects

Introduction

1.1 Background

A program fails when it does not do what it is supposed to do [24] and software testing is the most popular means for practitioners to check the correctness of programs in order to improve software quality and reliability [23]. Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended [1]. In an ideal world, a program is supposed to be tested in every possible permutation. However, in most cases this is not possible because creating test cases for all possibilities is impractical and completing testing of a complex application would need huge human resources and time. It is not an economically feasible practice if all (as mentioned above, this simply is not possible) or most cases are generated and executed manually.

Since software testing is a very labor intensive and hence very expensive process, the cost of developing software could be dramatically reduced if the testing process can be automated [4]. Programmers get assistance from test data generator tools in the generation of test data for a software program.

After test data are executed and results of the testing are captured, we still cannot claim the software testing as a successful one before the test results are validated in order to determine the correctness of the software behavior. The comparison of results can be viewing results by human eyeball to determine if they are what we expect for manual tests. However, it is more complicated with automated tests as each automated test data provides a set of inputs to the software under test (SUT) and compares the returned results against what is expected. The results produced by the SUT that need to be verified are called actual outputs, and the correct results that are used to evaluate actual outputs are called expected outputs [12]. Expected outputs are generated using a mechanism called a test oracle. The term oracle may be used to mean some different things in testing—the expected outputs themselves, the procedure of generating expected outputs, and the judgment of whether or not the actual outputs are what we expected [11]. In this article, the term oracle is used to mean an expected output that can be used to determine whether the software is executed correctly.

Having an oracle is especially important in automatic generation. Effective oracle approaches try to automate the related generation processes as much as possible. However, oracle challenges encountered during the process of generating an automated test oracle need to be addressed. Shahamiri and his colleagues [5]

suggested these challenges are output domain generation, input domain to output domain mapping, and using a comparator to decide on the accuracy of the actual output. The first challenge is how to provide the output domain automatically because it can be difficult and expensive to provide the expected outputs manually. An automated oracle needs automatic output domain generation. The second challenge is to map the input domain to the output domain automatically. The final challenge is using the automated comparator to compare expected and actual outputs and decide whether there is a fault or not.

1.2 Related work on oracles

In the following, some popular oracle generation approaches, which are engaging these challenges, are reviewed here. Prior studies focus on cause-effect graphs methods, decision tables methods, artificial intelligence methods, artificial neural network (ANN) methods and formal methods [5]. These studies show these approaches can partly or fully address and overcome the challenges of oracle generation.

Cause-effect graphs and decision tables [13] can be applied to address the challenge of the mapping from input domain to output domain by fetching logical rules from

specifications. Even though there are some tools to create the required structures to generate the oracles automatically, they still need some human observations and improvements to achieve the best oracle.

There have been several attempts to apply artificial intelligence methods in order to make test oracles automatically [5]. As an example, Last and his colleagues [19][20] introduced a fully automated black-box tester using info fuzzy network (IFN), which is an approach developed for knowledge discovery and data mining. The method is designed for a regression test that is inapplicable of a fresh testing and inapplicable for verifying the newly inserted functionalities.

There also have been several attempts to use ANN to generate test oracles [5]. As an illustration, Shahamiri and his colleagues proposed a Multi-Networks Oracle based on to address the mapping challenge and Input/Output Relationship Analysis to overcome the issue of output domain [21]. Their approach was evaluated using mutation testing and all of the testing activities were performed automatically. Almost all of the previous ANN-based oracle studies considered a supervised learning paradigm to model the software application as test oracles. There are not many studies investigating unsupervised learning and reinforced learning paradigms [21].

Formal oracles may address all the oracle automation challenges and provide a reliable oracle in case an accurate and complete formal model of the SUT exists. Pascale Le Gall and his colleagues [18] proposed a formal relation between testing and program correctness on the level of institutions. They suggested providing an oracle institution as an intermediate level between programs and requirement specifications. This oracle framework interprets the program behavior in order to extract semantics from programs dedicated to deal with correctness. There is some prior research that shows the approaches generating oracles from semantic of programs are reasonable. Robinson proposed a semantic test process [16] that generates tests and test oracles using models of the software [17]. Day and Gannon [14] have described a system that translates a formal specification of input and output files into an automated oracle. The specifications from which Day and Gannon extracted test oracles are divided into a syntax section and semantics section [15]. The syntax uses BNF grammars to specify the format of input and output files, respectively. The semantics defines rules that specify the relationship the output must have with the input. The syntax and semantics sections are compiled together to obtain an oracle program for checking consistency of an output text with the corresponding input text. Although all of these studies on semantic oracles show the

significance of possibility in generating oracles based on semantics, especially Day and Gannon's system shows that semantics sections can be compiled to syntax sections in order to generate an oracle, none of them fully addresses the question of how expected outputs can be produced to make the oracle in semantics in automated framework.

1.3 Our approach

Our study proposes a new automated oracle approach using formal specification. Our approach targets those SUTs, which require grammar-based structured input data, including compilers [27], reactive systems [33] and software product lines [28]. Normally, these systems need complex inputs that can be difficult to be tested systematically [29]. To specify the semantics of the inputs of those SUTs, which are specified languages, the input grammars need to be extended [34]. The approach in this paper is strongly tied to the power provided by denotational semantics to achieve this problem. Denotational semantics is a formal methodology for defining language semantics. It has been widely used in language development and practical applications [35] [36], and has been proved to be an approach for precisely defining the meaning of a language [22].

Our approach assigns semantic meaning to structured inputs in a recursive manner, applies denotational semantics [22] on these semantic meanings to specify expected outputs in order to satisfy the first challenge about output domain generation mentioned above. Furthermore, we define valuation functions associated with grammatical structures of input data to map an input directly to its meaning as the expected output in order to address the mapping challenge between input domain and output domain. Our approach in this article is implemented as follows: taking a context-free grammar, its denotational semantics, in the form of valuation functions, and the definition of associated methods used in valuation functions as its input; our automatic test data and oracle generation framework generates test cases and their oracles based on those inputs. In detail, we use the leftmost derivation strategy for test generation, meanwhile a semantic tree is built simultaneously with the procedure of test case generation. The value generated by evaluating the semantic tree where every derived variable from the structured input is bound with a corresponding semantic node using defined valuation functions is produced as expected testing output, also serving as test case's oracle. As a result, our framework generates a test case along with its oracle automatically.

Once the test cases are generated, they can be executed and the actual outputs are compared with oracles to detect software faults. The Figure 1.1 shows the flow

including three parts: *Test data and oracle generating*, *software testing*, and *validating and analyzing*. By extending Gena [2] with our oracle generator, we build an automatic test data and oracle generator framework. The framework, along with input and output domain, is shown in *Test data and oracle generating*. By using the generated oracles, we apply the test cases on testing subject applications in *software testing* and detect the fault for these applications in *validating and analyzing*. *Software testing* and *validating and analyzing* are used to obtain our experimental results. Because they are not the work in our oracle generation work, we do not give the details of the procedure of them here.

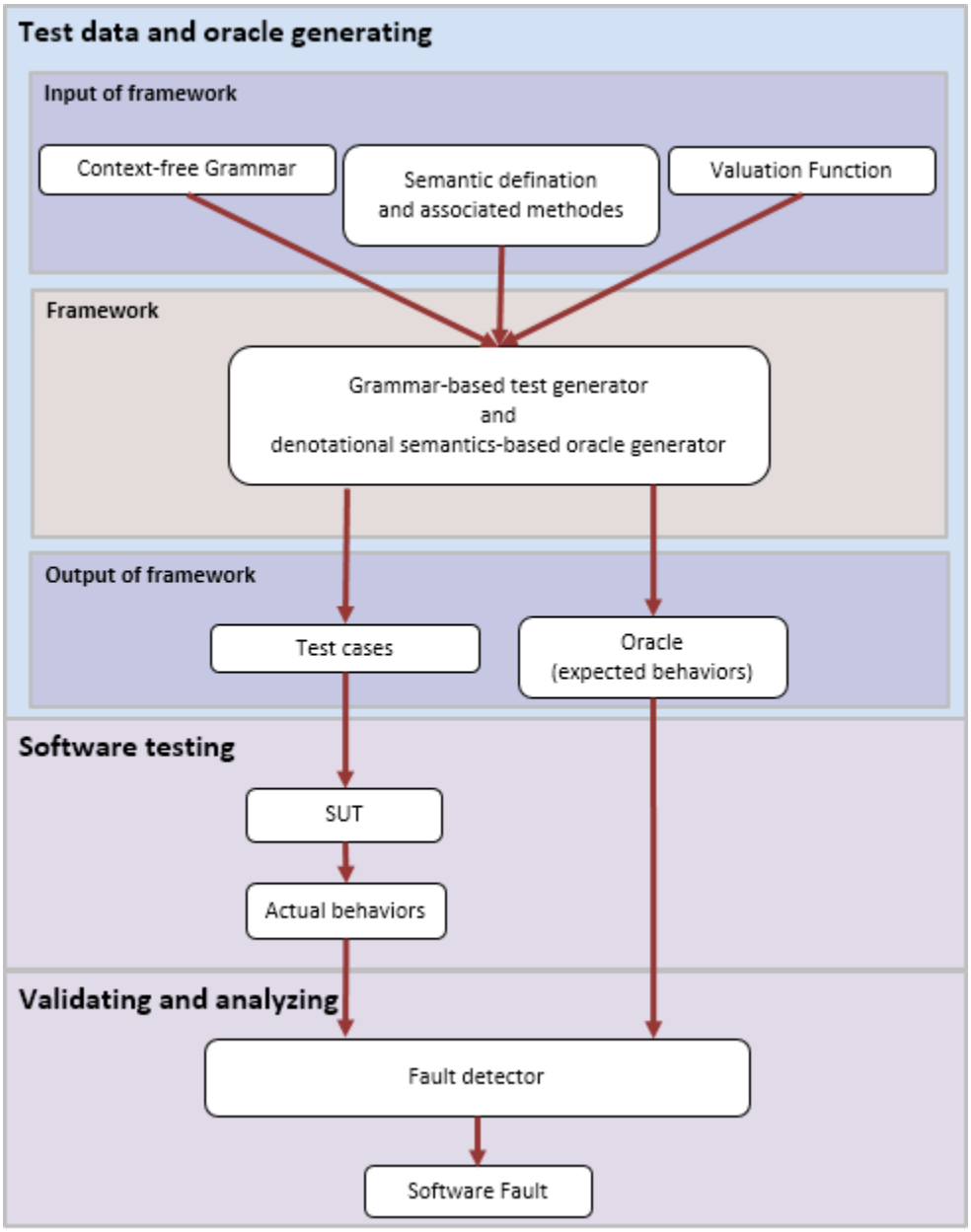


Figure 1.1: Diagram of oracle generation and usage in our applications

This paper makes the following contributions:

1) We introduce a new formulation of grammar based automated test data generation in which the goal is to generate test data from grammar, while simultaneously generating an oracle from semantics, which assigns meanings

to grammatically structured input. The syntax, semantics, and valuation functions of the input data is extracted from software's specifications.

2) We introduce an algorithm for addressing this extended oracle generating problem for automated test data generation.
3) We present the results of three empirical studies to illustrate the effectiveness of the algorithm. The algorithm was applied to three programs, which are a license scanning system, a grading system, and an online parking fee calculating system.

1.4 Organization

The rest of the paper is organized as follows. Section 2 addresses the main challenges on grammar-based test generation and the approach we adopt to generate the test case. Section 3 introduces denotational semantics, which is the approach we used to generate the oracle. Section 4 presents our approach for oracle automation. Section 4.1 introduces an application of the approach. Section 4.2 illustrates the process of automating an application of our approach. Underived string, dynamically growing semantic tree and an example are included. Section 4.3 introduces the evaluation functions for a semantic tree. Section 4.4 addresses the algorithm of evaluation of the semantic tree. Section 5 presents a Java-based implementation and our experimental

results of testing on a license scanning system, a grading system and a web testing system, respectively. Conclusions and future work are given in Section 6.

Section 2 Grammar-based test generation

2.1 Background

Grammar-Based Test Generation (GBTG) is an approach to test generation that employs context-free grammars to create sets of test cases [6]. The context-free grammar (CFG) describes the syntax of the input to the SUT. GBTG takes generative context-free grammars as an input and produces strings that conform to the syntax of the inputs of the SUT.

2.2 Related work

The work of Hanford who generated PL/1 programs for compiler testing [1] was the earliest known application of CFGs to testing; years later, Bird and Munoz applied GBTG to compiler testing, sort/merge utilities, and graphical output applications [6][7]. Burgess utilized grammars for automatically generating test sets for optimizing Fortran compilers [6][8][9]. Sirer developed a language named lava to test Java Virtual Machine [10]. Then much of the later work in GBTG focuses on network protocol testing [6].

2.3 The approach we use

In our paper, a stochastic grammar-based test generation approach is used to perform automated test case generation. In order to generate our oracle with test cases that can be terminated appropriately with good diversity, we adopt Guo and his colleagues' approach [2], which is a Java-based system named Gena based on their

grammar-based test generation algorithm to produce well-distributed test cases while taking a symbolic grammar as input, requiring zero control input from users. Gena utilizes a dynamic stochastic model, which guarantees the termination of a single test case generation. In this model, each variable is associated with a tuple of probability distributions, which are dynamically adjusted along the derivation. The approach provides various implicit balance control mechanisms to generate the balanced distribution of generated test cases over grammatical structures [2]. In the following sector, an example is used to show the abilities of the approach in termination and distribution aspects. We apply the leftmost derivation to input variables.

2.4 Balance Results

Table 2.1 shows a statistic report of the first 1000 generated arithmetic expressions, which is an example in [2] by Gena, given a symbolic grammar as follows:

E :: = F | E + F | E – F
F :: = T | F * T | F / T
T :: = [N] | (E)
[N] ::= 1..1000

The grammar has only one terminal exit, E -> F -> T -> [N], but the rest are full of recursive rules.

Table 2.1: Statistic report for test cases of arithmetic expressions

Operators	Total Frequencies
+	2191
-	2165
*	4438
/	4402

()	1859
[N]	14196

By comparing the total frequencies among operators, we can identify how balanced test case generation is overall. The total frequencies of the operators + and – are close, which indicates the balanced distribution between two recursive rules under the same variable E; similar reasons apply on the frequencies observation between * and /. Also, the total frequencies of the operators indicate the recursive rules are terminated at a reasonable level.

Section 3 Denotational Semantics

In this section, we give a brief introduction on denotational semantics. The denotational semantics approach maps a notation specification directly to its meaning, called its denotation [22]. The denotation is usually a mathematical value, such as a number or a function. No interpreters are used; a valuation function maps the notation specification directly to its meaning.

Since denotational semantics provides an approach for precisely defining the meaning of a notation specification [22], we adopt denotational approach to generate oracles, which equal an input language's execution results. The approach has three parts:

- *Syntax*: the appearance and structure of input notation specification, specified as a context-free grammar;
- *Semantics*: the assignment of meanings to the input;
- *Valuation function*: the function of mapping syntax and semantics parts to generate the expected output.

Normally, a SUT's input specification in context-free grammars is a formal language when it takes grammar-based structured inputs. The valuation function, which connects syntax and semantics parts, is defined structurally and its domain is the set of derivation trees of the language. It determines the meaning of a derivation tree by determining the meanings of its subtrees and combining them into a meaning for the

entire tree, which is the expected result of the input language, serving as the oracle of the SUT.

In the following sector, two examples are used to show the approach.

3.1 Binary numeral example

The following illustrations show the example of binary numerals based on an example in [22]:

Binary numeral's syntax definition:

B :: = D | B D

D :: = 0 | 1

Binary numeral's semantics definition:

Domain N = Integer (0, ∞)

Operations

0, 1, 2, … : N

+: N + N -> N

*: N * N -> N

The following tree depicts the binary numerals "101":

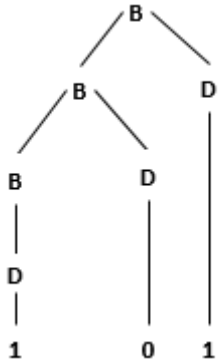


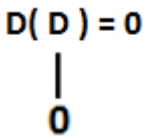
Figure 3.1: Tree depicting the binary numerals “101”

The tree’s internal nodes represent non-terminals of the syntax definition.

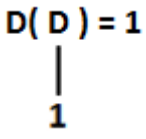
The meaning of the digit subtree:

is the number 0.

We might state this as:



That is, the D valuation function maps the tree to its meaning, 0. Similarly, the meaning of the other binary digits in the tree is one; that is:



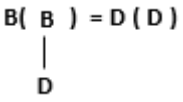
We use the following one-dimensional form to represent these two-dimensional equations by using double brackets. The double brackets surrounding the subtrees are used to clearly separate the syntax pieces from the semantic notation.

ValueD[[0]] = 0

ValueD[[1]] = 1

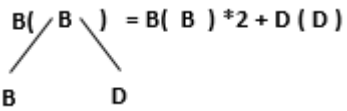
Furthermore, we use the same way to determine the meanings of the binary numeral trees. Looking at the leftmost B-tree, we see it has the form:

The meaning of this tree is just the meaning of its D-subtree, that is, 1. In general, for any unary binary numeral subtree



we have ValueB[[D]] = ValueD[[D]].

The principle of binary arithmetic dictates that the meaning of this tree must be the meaning of the left subtree doubled and added to the meaning of the right subtree.



We write this as ValueB[[BD]] = (ValueB[[B]] * 2) + ValueD[[D]]. Using this definition we complete the calculation of the meaning of the tree.

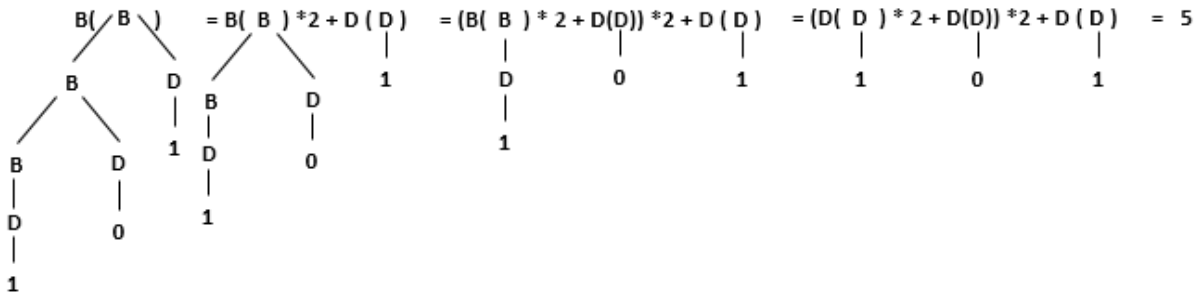


Figure 3.2: Denotational definitions of binary numerals “101”

Figure 3.2 shows complete denotational definitions of the above binary numeral

example

Syntax:

B :: = D | B D

D :: = 0 | 1

Semantics:

Domain N = Integer (0, ∞)

Operations

0, 1, 2, … : N

+: N + N -> N

*: N * N -> N

Valuation functions:

B:

ValueB[[BD]] = (ValueB[[B]] * 2) + ValueD[[D]]

ValueB[[D]] = ValueD[[D]]

D:

ValueD[[0]] = 0

ValueD[[1]] = 1

When we determine the meaning of the tree in the above diagram, we represent the tree in its linear form [[101]], using the double brackets to remind us that it is indeed a tree. We mimic the tree transformation in the leftmost derivation manner and begin with:

ValueB[[101]] = (ValueB[[10]]* 2) + ValueD[[1]]

The ValueB[[BD]] equation of the B function divides [[101]] into its subparts. We continue:

(ValueB[[10]]* 2) + ValueD[[1]]

= (((ValueB[[1]]* 2) + ValueD[[0]] * 2) + ValueD[[1]]

= (((ValueD[[1]]* 2) + ValueD[[0]] * 2) + ValueD[[1]]

= (((1 * 2) + 0) * 2) + 1

= 5

In the above example, the valuation functions are applied to mapping the above syntax and semantics, and we know binary numeral “101”’s meaning is 5. In our grammar-based test generation, the binary numeral “101” is the test case, and its meaning, 5, is the oracle.

3.2 Arithmetic expressions example

The following illustrations show the example of taking an arithmetic expression and performing its integer evaluation in a Java application using the denotational semantics approach.

The syntax of the input language represented by integer arithmetic expressions is given as the following:

E :: = F | E + F | E – F

F :: = T | F * T | F / T

T :: = [N] | (E)

[N] ::= 1..1000

where [N] is an abstract notation from a finite domain of integers. We will generate the oracle for the arithmetic expression “3 * (4 + 5) - 6” from denotational semantics.

Arithmetic expressions' semantics, like their expected result in the Java application, are typically integrated as integers with a set of standard arithmetic operators, such as "+", "-", "*" and "/". The semantics definition is given as the following:

Domain N = Integer $(0, \infty)$

Operations

0, 1, 2, … : N

+: N + N -> N

-: N - N -> N

*: N * N -> N

/: N / N -> N

The denotational semantics is defined by four types of valuation functions: ValueE, ValueF, ValueT, and ValueN, which map their corresponding grammatical structures to their respective semantics. The full valuation functions are given as the following:

ValueE[[F]] = ValueF[[F]]

ValueE[[E+F]] = ValueE[[E]] + ValueF[[F]]

ValueE[[E-F]] = ValueE[[E]] - ValueF[[F]]

ValueF[[T]] = ValueT[[T]]

ValueF[[F*T]] = ValueF[[F]] * ValueT[[T]]

ValueF[[F/T]] = ValueF[[F]] / ValueT[[T]]

ValueT[[[N]]] = ValueN[[[N]]]

ValueT[[(E)]] = (ValueE[[E]])

ValueN[[[N]]] = N

where double brackets are used to represent grammatical structures, a derivation subtree in practice. And, the symbolic terminal [N] is treated as a terminal, which is substituted by a random element from its domain in practice.

The following derivation tree depicts the arithmetic expression “3 * (4 + 5) - 6” in the leftmost derivation manner:

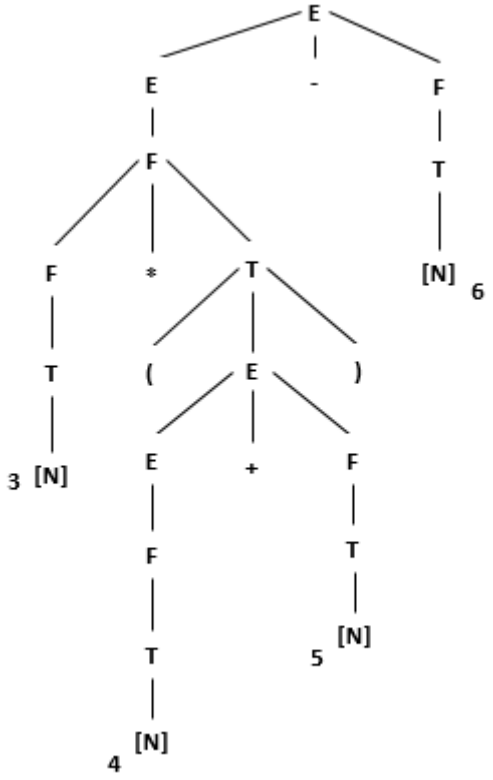


Figure 3.3: Tree depicting arithmetic expression “3 * (4 + 5) - 6”

When we determine the meaning of the input expression 3*(4+5)-6 in the above diagram, we represent in its linear form ValueE[[3*(4+5)-6]]. We mimic the tree transformation in the leftmost derivation manner:

ValueE[[3*(4+5)-6]] = ValueE[[3*(4+5)]] – ValueF[[6]]

= ValueF[[3 * (4+5)]] – ValueT[[6]]

= ValueE[[3]] * ValueT[[(4+5)]] – ValueT[[6]]

= ValueF[[3]] * (ValueE[[4+5]]) – ValueT[[6]]

= ValueT[[3]] * (ValueE[[4]]+ ValueF[[5]]) – ValueT[[6]]

= ValueT[[3]] * (ValueF[[4]]+ ValueT[[5]]) – ValueT[[6]]

= ValueT[[3]] * (ValueT[[4]]+ ValueT[[5]]) – ValueT[[6]]

= ValueN[[3]] * (ValueN[[4]]+ ValueN[[5]]) – ValueN[[6]]

= 3 * (4 + 5) – 6

= 21

The denotational semantics approach maps input data directly to its expected results of a SUT, which provides a solution for the challenges of oracle output generation and mapping between input domain and output domain. Based on this observation, we adopt the denotational semantics for automated test oracle generation.

Section 4 A denotational semantic approach for oracle automation

The essential challenge in oracle automation is how to decide output domain and map input domain to output domain. Our approach utilizes an automated framework that generates test oracles based on denotational semantics, which addresses the output domain and the mapping challenge described in the previous section. Our framework extends denotational semantics on Gena [2], which is an automatic grammar-based test generator with good termination and distribution aspects introduced in sector 2. To adopt the denotational semantics approach on Gena, we mainly work in the following parts:

- Implementing semantic domain along with associated operations
- Specifying semantic valuation functions along with the CFG input
- Automating the application of valuation functions along with test generation

Semantic domains are determined by the SUT. Our framework provides an interface for users to define a semantic domain and its associated operations as Java class and methods, respectively. We introduce the specifications of our approach on an example in the following section.

4.1 An application of the approach

Considering an arithmetic expressions example, we define a semantic domain in a domain Java class. We also define an integer instance variable, which will eventually hold the semantic result of the input and a set of methods (including “plus”, “sub”, “mul”, and “div”) supporting the standard integer arithmetic operations. We extend CFG input with LISP-like notation to define denotational semantics. Furthermore, we compute semantic values by using lambda calculus. One reason for this is that denotational semantics expresses its definition using the higher-order functions of the lambda calculus; another reason is that lambda calculus’ uncomplicated syntax and semantics provide the power to represent all computable functions.

The CFG input with valuation functions for a subset of arithmetic expressions is shown in Figure 4.1:

(1) E :: = F @@ (F)
(2) E :: = E + F @@ (plus E F)
(3) E :: = E – F @@ (sub E F)
(4) F :: = T @@ (T)
(5) F :: = F * T @@ (mul F T)
(6) F :: = F / T @@ (div F T)
(7) T :: = [N] @@ ([N])
(8) T :: = (E) @@ (E)
(9) [N] ::= 1..1000

Figure 4.1: CFG input with valuation functions for a subset of arithmetic expressions

Given the above semantic definitions, each production rule is equipped with valuation functions by a delimiter “@@”. In the case of the production rule in line (2), the input

data contains a grammar structure E + F, and its semantic value can be computed by lambda expression λE.λF. (plus E F). Here the value of the expression is the evaluation of applying associated operation “plus”, which is defined in the domain class on the formal argumentλE.λF., which are omitted in valuation functions due to their implication in the production rules. Similarly, in the case of the production rule (E :: = F @@ (F)) formal argument λF. is omitted. Furthermore, because there is a singleton argument listed in the valuation function, the value of the expression is the result of the singleton.

4.2 Automating the application of valuation functions along with test generation

4.2.1 Underived String

Our automated test data framework generates a test case using the strategy of the leftmost derivation. The application of the leftmost derivation is illustrated here. Given a symbolic grammar G = (V, T, P, S), where V is a set of variables, T is a set of terminals that include symbolic terminals, P is a set of production rules that represent the relations from V to (V ∪ T)*, and S is the start variable. The derivation is in the form of $E \Rightarrow^{Ri} \omega$, where E is a variable in V and ⇒Ri is a single leftmost derivation applying the i-th production rule of E, $\omega \in (V \cup T)*$.

We define the production rule index for given grammar and semantic rules as follows:

Table 4.1: Production rule index of arithmetic expressions

Variable for derivation: V	Production Rule Index: Ri	Production rule
E	E1	F
E	E2	E + F
E	E3	E - F
F	F1	T
F	F2	F * T
F	F3	F / T
T	T1	[N]
T	T2	(E)

We include underived variables in underived string, which initially starts from root “E” in the above example. When we generate test case “3*(4+5)-6/2”, the underived string will be updated during the test case generation process as shown in Figure 4.2:

$$
\begin{aligned}
E &\Rightarrow^{E3} E - F \\
&\Rightarrow^{E3} F - F \\
&\Rightarrow^{F2} F * T - F \\
&\Rightarrow^{F1} T * T - F \\
&\Rightarrow^{T1} [N] * T - F \\
&\Rightarrow^{T2} [N] * (E) - F \\
&\Rightarrow^{E2} [N] * (E + F) - F \\
&\Rightarrow^{E1} [N] * (F + F) - F \\
&\Rightarrow^{F1} [N] * (T + F) - F \\
&\Rightarrow^{T1} [N] * ([N] + F) - F \\
&\Rightarrow^{F1} [N] * ([N] + T) - F \\
&\Rightarrow^{T1} [N] * ([N] + [N]) - F \\
&\Rightarrow^{F3} [N] * ([N] + [N]) - F / T \\
&\Rightarrow^{F1} [N] * ([N] + [N]) - T / T \\
&\Rightarrow^{T1} [N] * ([N] + [N]) - [N] / T \\
&\Rightarrow^{T1} [N] * ([N] + [N]) - [N] / [N]
\end{aligned}
$$

Figure 4.2: Underived string during test case generation process

where each [N] is automatically substituted with a random integer from its domain during the generation.

4.2.2 Dynamically growing semantic tree associated with test generation

In our framework, test data is generated by using the strategy of the leftmost derivation. When the derivation travels through these production rules, a semantic tree is built dynamically along with the procedure of test case generation to support oracle generation by applying the associated valuation functions. During the test case generation, derived variables are bound with corresponding semantic nodes. There are two types of semantic nodes, a regular node and a λ-node. A regular node includes the following three parts:

- A derived variable V or a semantic terminal
- A link to a semantic subtree that presents the semantic value of V
- A link to a peer semantic node that appears in valuation functions

A λ-node includes the following three parts:

- A derived variable V
- A link to the formal argument part of lambda expression
- A link to the body part of lambda expression

In Figure 4.3, a semantic subtree with the valuation function specified in production rule (E :: = E – F @@ (sub E F)) is presented:

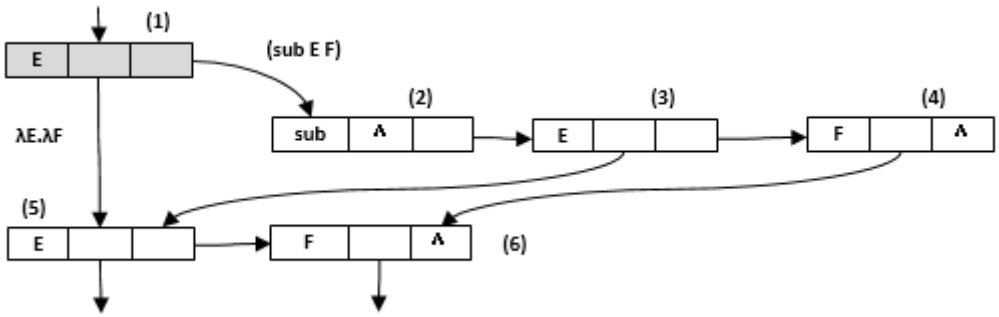


Figure 4.3: Internal structure of storage of valuation function in (sub E F)

where node (1) is a λ-node, which is colored by gray and the rest are regular nodes. A special symbolic "Λ" is used to denote a null link. In the expression body part, nodes (2), (3), and (4) represent the valuation function (sub E F). "sub" in node (2) is a built-in function in Java, defined in domain class; the semantics of E at node (3) and F at node (4) will be obtained from the associated formal argument part, node (5) and node (6), respectively; while the semantics of E at node (5) and F at node (6) in the formal argument part will be extended recursively when E and F are further derived during test generation.

To perform such a recursive extension on the semantic tree with test generation, our framework binds every underived variable of test generation with a corresponding regular semantic node. Once this variable is derived by applying a production rule, its bound semantic node will be extended with a semantic subtree based on its associated valuation function, rooted by a λ-node representing this derived variable. We still use the above figure as an example. Consider the underived variable E of test case in body part of E ::= E – F, which is bound to node(5). When grammar rule (E ::= F) is

applied to variable E in test case generation process, its bound semantic node, node(5), is extended with subtree rooted by λ-node, node(7), based on equipped valuation function (F) in production rule E ::= F. Also the body part of lambda expression is presented as node (8), and the formal argument part is presented as node (9). The extended semantic tree is shown as follows:

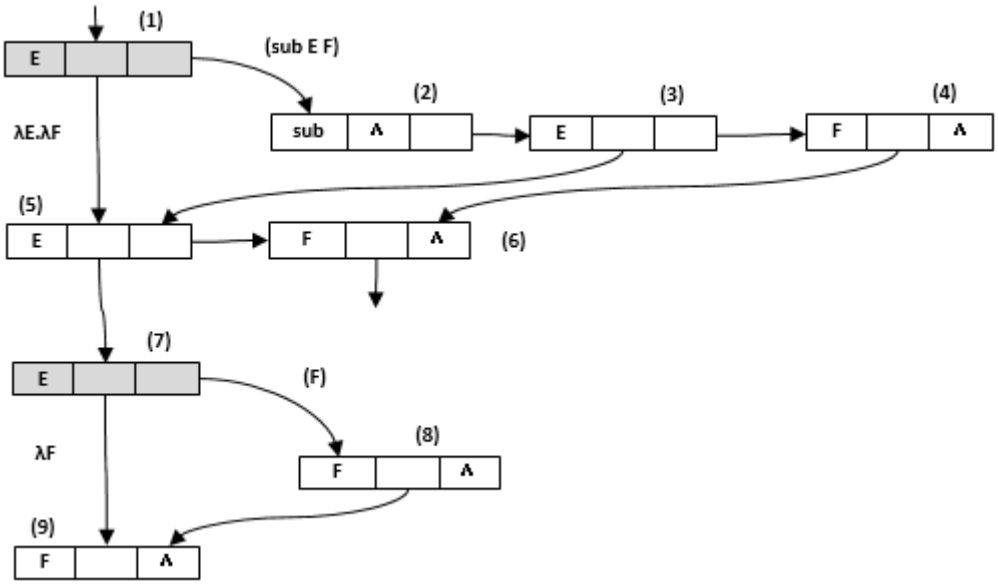


Figure 4.4: Internal structure of storage of valuation function extending to (F)

Since the associated semantic tree is extended simultaneously along the process when the derivation path is traveled, the built semantic tree is accurately mapping with the derivation path, which represents as a test case. We refer to the underived string, which includes the underived variable information mentioned in section 4.2.1 to extend the semantic tree in our application.

4.2.3 Example

Given the symbolic grammar in section 4.2.1, the Figure 4.5 shows a complete sequence of the extension procedure of the semantic tree for the test case “3*4 - 2”.

A variable with a superscript (e.g. $E^{(1)}$) indicates that the variable is bound with a semantic node where the number in superscript is shown.

Starting from root E, the underived string is “$E^{(1)}$”. The semantic node, node (1), is established to associate this underived variable E.

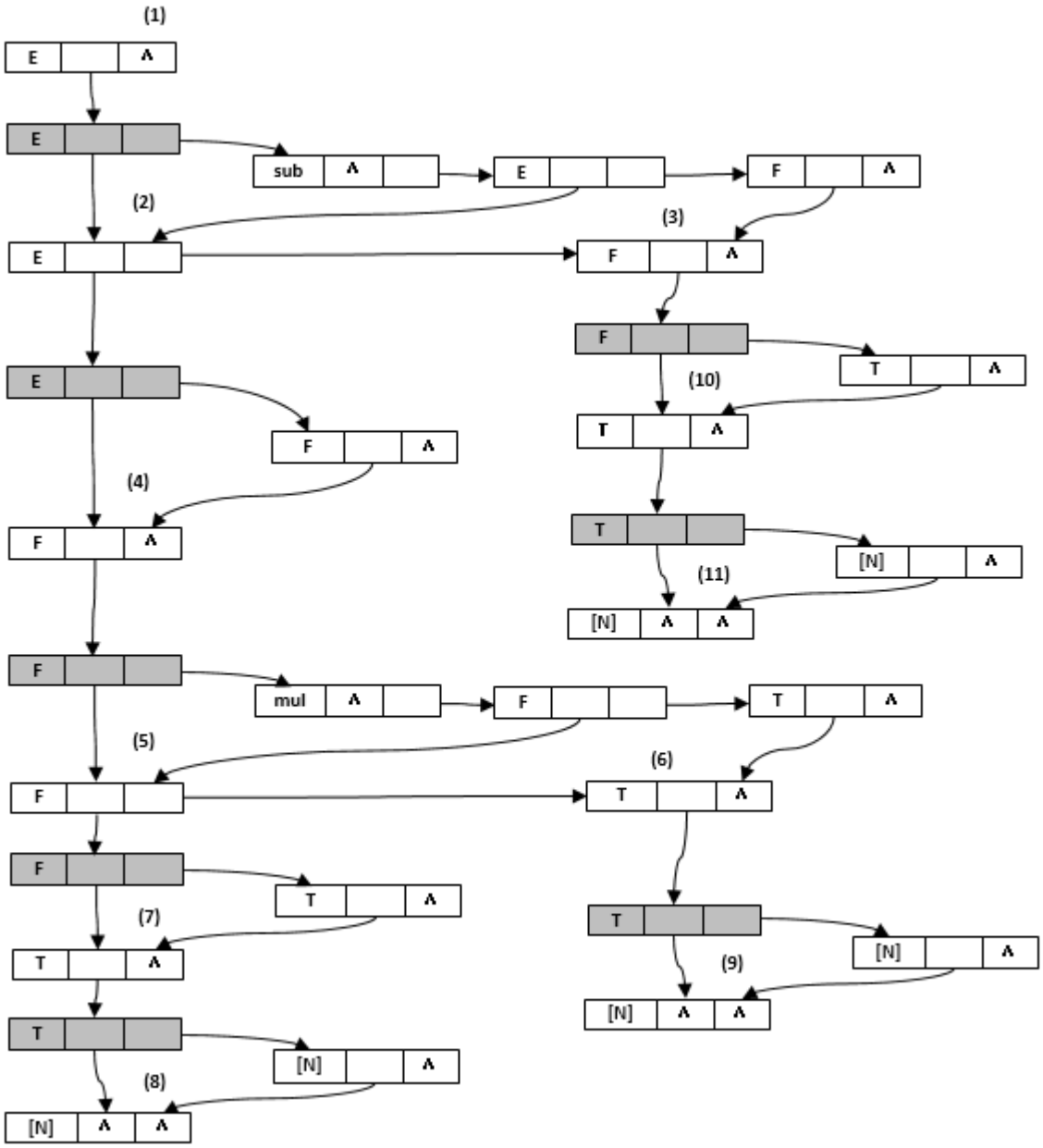


Figure 4.5: Semantic tree associated with test case “3*4 - 2”

As the derivation moves on from $E^{(1)}$ to $E^{(2)} + F^{(2)}$ and the underived string is updated to “$E^{(2)}$ $F^{(3)}$”, a subtree based on functionλE.λF. (sub E F) in 3-rd production rule of E is extended under node E, where formal argument E and F are bound to the semantic node node (2) and node (3), respectively; Then, according to the leftmost derivation strategy, the underived string is updated to “$F^{(4)}$ $F^{(3)}$” and a semantic subtree is

extended under node (2). The subsequence is executed, and eventually a symbolic terminal variable [N] is reached. An instance number is generated and stored in its corresponding semantic node. Here, the number is 3 for $[N]^{(8)}$. Similarly, the semantic tree is extended under node (3), and eventually the whole semantic tree for test case "3*4 - 2" is built.

4.3 Evaluation functions for semantic tree and the generation of the oracle

We apply evaluation functions on the following semantic tree, which is built as an example in the previous section for test case "3*4 - 2". The value of λ-node E under node (1) is the semantic value for node (1). So the next step is evaluating thisλ-node E. It is the evaluation ofλ-expression applying calculus expression based on body parts (sub E F) on formal argument node E at (2) and node F at (3). Similarly, the λ-node value will be calculated recursively.

Let Evaluation(node) be the function to evaluate the node in the semantic tree. Here we present nodes with the indices of their positions.

Evaluation($E^{(1)}$) = (sub Evaluation($E^{(2)}$) Evaluation($F^{(3)}$))

= (sub Evaluation($F^{(4)}$) Evaluation($F^{(3)}$))

= (sub (mul Evaluation($F^{(5)}$) Evaluation($T^{(6)}$)) Evaluation($F^{(3)}$))

= (sub (mul Evaluation($T^{(7)}$) Evaluation($T^{(6)}$)) Evaluation($F^{(3)}$))

= (sub (mul Evaluation($[N]^{(8)}$) Evaluation($T^{(6)}$)) Evaluation($F^{(3)}$))

= (sub (mul 3 Evaluation($T^{(6)}$)) Evaluation($F^{(3)}$))

= (sub (mul 3 Evaluation($[N]^{(9)}$)) Evaluation($F^{(3)}$))

= (sub (mul 3 4) Evaluation($F^{(3)}$))

= (sub (mul 3 4) Evaluation($T^{(10)}$))

= (sub (mul 3 4) Evaluation($[N]^{(11)}$))

= (sub (mul 3 4) 2)

= (sub 12 2)

= 10

In our grammar-based test generation, the test case "3*4-2" is generated and its associated semantic tree is evaluated as "10", the oracle of the test case, which equals the expected value of the test case.

4.4 Algorithm

We present a detailed pseudo-code for the evaluation function in our automatic oracle generation framework. To support the oracle generation, a semantic tree is gradually constructed along a derivation path traveled during a test generation. A semantic tree node contains a variable and two links. The first one is the link to a subtree that contains formal arguments to calculate the value of the variable; the second is the link

to a subtree that contains the body of the lambda expression. A link is null value if the subtree is empty. The methods associated with semantic tree evaluation are described as follows:

– int getValue: return the value of the subject node; null is returned if the value does not exist.

– void setValue(int): set the value to the subject node.

- int getValueByApplyingOperator: return the value of a node's lambda calculation. In detail, the value of formal arguments obtained from the first link of the subtree is applied to an expression in which the operators and parameters are found in the second link. The operators are defined in domain class.

- Node[] getArgumentNodes: return formal argument nodes of the input node if they exist; null is returned if no node exists.

```
1: Global: semantics G = (V, T, P, S)
2: Input: a semantic tree parent node, sNode;
3: Output: oracle
4: function int Evaluation (sNode)
5:     if (sNode is in form of [N]) then          ⊲ encounter terminal, end of a recursion
6:         Let r is random integer value in defined domain
7:         return r
8:     else
9:         nodes <- sNode.getArgumentNodes()                ⊲ get argument nodes
10:        for (node in nodes)
11:            if (node.getValue is null) then
```

12: node.setValue(Evaluation(node)) ◁ get nodes' value by recursively computing their subtree.
13: end if
14: end for
15: return sNode.getValueByApplyingOperator() ◁ apply operator on the computation of nodes
16: end if
17: end function

Figure 4.6: Algorithm Evaluation

The algorithm shows the evaluation function to generate the meaning of a semantic tree. We first check whether the recursion is end; if that is the case, a random integer value will be returned (lines 5-7). If there exists a subtree structure for the subject node, we define the nodes that are formal arguments of the subject node to obtain the value of the subject node (line 9). Then we recursively get these nodes' values (lines 10-14). Lastly, we apply these argument nodes to the operator and return the result, which is the value of the semantic tree whose root is the input node (line 15).

Section 5 Experimental Results

We have carried out three experiments to measure how generated test oracle and its associated cases are well mapped over a given symbolic grammar, and how those input test specifications can be used for automatic testing.

5.1 License scanning system

We have implemented an automatic license text and oracle generation system for an open source license scanning tool. Considering an open source license scanning tool takes a free format text as an input string, performs license identifying operations to discover open source license, and finally returns the identified license name. We generated oracle, which is an expected license name, along with our test case, which is a free format text containing license key words. The actual output from the license scanning tool and oracle are compared for every test case.

The FOSSology license scanning tool [26] is our software test subject in this study. To identify a sample license from a free format text by FOSSology, some contents are selected from the sample license specification and used as key words in the FOSSology license discovery module. Once these key words in sample license specifications are identified in an input string, FOSSology will “suggest” this license

is identified. During the scanning procedure, FOSSology changes all input strings to lower case to mitigate case sensitive issues.

5.1.1 Example: Adaptive Public License 1.0 license

Given open source Adaptive Public License 1.0 license (APL-1.0) specification [30] and FOSSology license scanning specifications, we summarized APL license's input specification as follows:

(1) "This License is adaptive, and the generic version" string is contained;

(2) "Adaptive Public License Version 1.0" or "Adaptive Public License v1.0" string is contained;

(3) The word spelling "license" can be "license" or "licence";

If the text string meet specification (1) and it does not meet specification (2), then an "APL" license can be identified; if the text string meet specification (1) and (2) at the same time then an "APL-1.0" license can be identified. No appearance order for specification (1) and (2) is demanded. Specification (3) is applied to specifications (1) and (2). According to these specifications, we defined the following syntax in Figure 5.1:

```
(1) APL10 ::= S1 @@ (S1)
(2) APL10 ::= S2 @@ (S2)
(3) S1 ::= 'This ' LICENSE ' is adaptive, and the generic version' @@
    ('APL')
(4) S2 ::= S1 =AND= APLTITLE @@ (assembleLicense S1 APLTITLE)
(5) APLTITLE ::= ADAPTIVE ' ' VERSION10 @@ ('APLTITLE')
(6) ADAPTIVE ::= 'adaptive public ' LICENSE
(7) LICENSE ::= 'licence' | 'license'
(8) VERSION10 ::= V NUM
(9) V ::= 'v' | 'version '
(10)    NUM ::= 1.0
```

Figure 5.1: Syntax and valuation functions of Adaptive Public License license

We established two patterns of test cases, which are the production rules at line 1 and line 2, based on corresponding specifications. The production rule at line 3 associates with specification (1); the production rule at line 5 associates with specification (2); according to specification (2), we also defined the production rules at lines 8, 9 and 10 to represent different format of version information; and we defined the production rules at lines 6, 7 based on specification (3); the production rule at line 4 was established in order to meet specification (1) and (2) at the same time. Here, we introduced operator "=AND=" to address the conjunction relationship of two variables next to "=AND=". For example, in S2 ::= S1 =AND= APLTITLE @@ (assembleLicense S1 APLTITLE)(line 5), the generated license text S2 includes license text segment S1 and APLTITLE.

A valuation function defined behind the delimiter '@@' provides semantic value for the variable defined before the delimiter. For example, in S2 ::= S1 =AND=

APLTITLE @@ (assembleLicense S1 APLTITLE)(line 5), S2's semantic value can be calculated via the λ-expressionλS1.λAPLTITLE.(assembleLicense S1 APLTITLE), where the value of argument S1 and APLTITLE is calculated from further derivation. If the semantic value is defined directly without further processing, the valuation function simply relays the result from the expression. For example, in S1 ::= 'This ' LICENSE ' is adaptive, and the generic version' @@ ('APL') (line 3), 'APL'after delimiter '@@' is the semantic value for variable S1.

The semantic domain and associated methods are defined in a domain class in our framework. In the given case, we define the operation "assembleLicense", which induces a certain license when listed variables' information meets certain license assemble rules. The following license assemble rule based on the APL 1.0 license specification [30] is defined for the method: "{APL, APLTITLE}-->APL-1.0", where "-->" indicates the license component on the right side can be induced from the license components bracketed in {} on the left side.

Table 5.1 shows license scanning results on 18 APL specified license strings, which cover all derivation paths of given input grammar, by running these license texts on the FOSSology license scanning tool [26]. We categorized the percentage of the test cases in all cases according to actual outputs and their oracles.

Table 5.1: Report of license scanning results on APL license

	Case Number	Percentage in all cases	Oracle	Actual output
1	2	11%	APL	APL
2	16	89%	APL-1.0	APL-1.0

The above table shows there is no test case where the actual output is different from its oracle.

5.1.2 Example: Apache 2.0 license

Consider a more complicated example -- the open source Apache 2.0 license. We input the following grammar and valuation function based on the specifications, which we summarized according to the Apache 2.0 license's specifications [25] and the FOSSology license scanning specifications:

```
APACHE20 ::= S1 @@ (S1)
APACHE20 ::= S2 @@ (S2)
APACHE20 ::= S3 @@ (S3)
APACHE20 ::= S4 @@ (S4)
APACHE20 ::= S5 @@ (S5)
APACHE20 ::= S6 @@ (S6)
APACHE20 ::= S7 @@ (S7)
APACHE20 ::= S8 @@ (S8)
APACHE20 ::= S9 @@ (S9)
S1 ::= REFERENCE @@ ('APACHE-2.0')
S2 ::= LICENSE =AND= VERSION @@ (assembleLicense LICENSE
VERSION)
S3 ::= URLAPACHE2 @@ (URLAPACHE2)
S4 ::= URLOPENSOURCEAPACHE2 @@ (URLOPENSOURCEAPACHE2)
S5 ::= APACHE DELIMITER V2 =AND= A1 @@
('APACHE_V2-POSSIBILITY')
S5 ::= A1 =AND= APACHE DELIMITER V2 @@
('APACHE_V2-POSSIBILITY')
```

```
S6 ::= BSD =AND= PREFIX =AND= REFERENCE @@ (assembleLicense
BSD PREFIX REFERENCE)
S7 ::= BSD =AND= LICENSE =AND= REFERENCE @@ (assembleLicense
BSD LICENSE REFERENCE)
S8 ::= ASF =AND= LICENSE =AND= VERSION2 @@ (assembleLicense ASF
LICENSE VERSION2)
S9 ::= BSD2 =AND= LICENSE =AND= VERSION2 @@ (assembleLicense
BSD2 LICENSE VERSION2)
BSD ::= 'distribution and use in source and binary forms' MOD ' ' ISARE ' '
PERMIT ' provided that' @@ ('BSD')
BSD2 ::= 'distribution of the source code in binary form must reproduce' @@
('BSD2')
BSD2 ::= 'distribution in binary form must reproduce' @@ ('BSD2')
MOD ::= '' | ' with or without modifications' | ' with or without modification'
ISARE ::= 'is' | 'are'
PERMIT ::= 'permitted' | 'permitted for any purpose'
ASF ::= 'copyrighted software available under a free-to-use- ' A1 ' by the apache
software foundation' @@ ('ASF')
LICENSE ::= APACHE ' ' A1 @@ ('APACHE')
LICENSE2 ::= APACHE ' ' SERIES ' ' A1
V2 ::= 'v2' | 'v2.0'
URLAPACHEU ::= WWW DOT 'apache' DOT 'org/licenses/' @@
('VERSION-UNKNOWN')
URLAPACHE2 ::= WWW DOT 'apache' DOT 'org/licenses/license-2.0' @@
('APACHE-2.0')
URLOPENSOURCEAPACHE2 ::= WWW DOT 'opensource' DOT
'org/licenses/apache-2.0' @@ ('APACHE-2.0')
DOT ::= '' | '.'
WWW ::= 'www' | 'http://www'
A1 ::= 'licence' | 'license'
APACHE ::= 'Apache' | 'APACHE' | 'apache'
DELIMITER ::= ' ' | '_' | '-' | ''
PREFIX ::= COPYRIGHT ' ' YEAR ' ' APACHE ' ' APACHESUFFIC    @@
('APACHE')
COPYRIGHT ::= '©' | '(c)' | 'copyright' | '&copy'
YEAR ::= YEARNUM SUFFIX
SUFFIX ::= ',' | '-' | ' '
APACHESUFFIC ::= 'group' | 'software' | 'foundation'
```

```
YEARNUM ::= '1900' | '2000' | '2025'
REFERENCE ::= REF1 ' under' =AND= LICENSE =AND= VERSION @@
('VERSION2.0')
REFERENCE ::= REF2 ' under' =AND= LICENSE2 @@ ('VERSION2.0')
REF1 ::= 'distributed' | 'offer' | 'offered' | 'released' | 'licensed' | 'available' |
'protected' | 'provided'
REF2 ::= 'distributed' |'modified'
VERSION ::= V SERIES @@ ('VERSION2.0')
VERSION2 ::= SERIES @@ ('VERSION2.0')
VERSION2 ::= V SERIES @@ ('VERSION2.0')
V ::= 'v' | 'version ' | 'v.'
SERIES ::= 20 | 2.0
```

Figure 5.2: Syntax and valuation functions of Apache version 2.0 license

The following license assemble rules based on the Apache version 2.0 license specification [25] are defined for the method “assembleLicense” in the domain class of our framework:

1. {LICENSE}-->APACHE
2. {ASF, APACHE}-->APACHE
3. {APACHE,VERSION2.0}-->APACHE-2.0
4. {BSD,APACHE,VERSION2.0}-->APACHE-2.0,BSD-style
5. {BSD,APACHE}-->APACHE,BSD-style
6. {BSD2,APACHE,VERSION2.0}-->APACHE-2.0,BSD-style

Figure 5.3: Semantic definition of the rules for assembleLicense method on Apache version 2.0 license

Table 5.2 shows license scanning results on 1000 Apache version 2.0 specified license strings, which cover all derivation paths of the given input grammar:

Table 5.2: Report of license scanning results on Apache version 2.0 license

	Case Number	Percentage in all cases	Oracle	Actual output
1	382	38%	APACHE-2.0	APACHE-2.0
2	477	48%	Apache-2.0,BSD-style	Apache-2.0,BSD-style
3	45	5%	Apache-2.0,BSD-style	Apache-2.0,BSD-style,U-Cambridge-style
4	84	8%	Apache_v2-possibility	Apache_v2-possibility
5	12	1%	Apache_v2-possibility	Apache-possibility

The Row 3 and 5 show the test cases where the actual outputs are different from their oracles, and it indicates that there may be some failed instances for Apache version 2.0 scanning module in FOSSology.

Regarding the test results in row 3, we confirmed that the key word “in source and binary forms is permitted provided” in the BSD-style license is also used as a key word in the U-Cambridge-style license scanning module. FOSSology reported the generated test cases, which meet BSD-style license specifications also meet U-Cambridge-style license specification. In the aspect of identifying the Apache 2.0 license, there is no inconsistency found and no fault is found in associated scanning modules.

Regarding the test results in Row 5, “Apachev2” is considered as one acceptable form for an Apache v2 possible license. However, the test results were inconsistent with expectations where the test cases in form of “license … apachev2” are identified as Apache-possibility instead of Apache_v2-possibility. With the power of an oracle, we

located the fault in the software successfully and significantly reduced the cost to generate expected output from massive test cases.

5.2 A Grading System

We extended our oracle generator to Gena [3] as an automatic grading system for Java programs. Consider a Java programming assignment, which takes an infix arithmetic expression as an input string, then convert the input to expression and calculate this expression to return a number. We used our framework to generate arithmetic expressions and their oracles. Then we compared returned numbers from Java program subjects, which take generated arithmetic expressions as inputs with our generated oracles to detect failing cases. The context-free grammar of the arithmetic expression and its valuation functions are defined as follows, which was introduced in Figure 4.1 and used as an example in Section 4:

(1) E :: = F @@ (F)
(2) E :: = E + F @@ (plus E F)
(3) E :: = E – F @@ (sub E F)
(4) F :: = T @@ (T)
(5) F :: = F * T @@ (mul F T)
(6) F :: = F / T @@ (div F T)
(7) T :: = [N] @@ ([N])
(8) T :: = (E) @@ (E)
(9) [N] ::= 1..1000

As we mentioned in Section 4, the operations in valuation functions are defined with standard Java arithmetic operations in the domain class in our framework. Table 5.3

shows the report of grading results on 14 Java program subjects, by running 1000 different arithmetic expressions generated by our framework. Because the generated arithmetic expressions can be very complex (e.g. 766+2*(359*840)/249/(429-184+711)-105-389+314), automated generated oracles can significantly reduce the time of calculating expected value based on test cases independently. By comparing the oracle with the actual output number from those Java programs, we collected the ratios of correctness for each subject. For example, the first subject performs correctly on 14% of the 1000 test cases. We located the failing test cases by the power of the oracle and listed the possible causes of the failure incorporated with typical causes related to processing arithmetic expressions.

Table 5.3: Report of grading results on 14 Java program subjects

	Correctness Ratio	Possible Causes
1	14%	Right-associativity
2	78%	Parenthesis not properly handled
3	100%	
4	2%	Not working at all
5	9%	Right-associativity; operator precedence ignorance
6	6%	Right-associativity; operator precedence ignorance
7	53%	[N] * [N] /[N]
8	100%	
9	68%	Partial operator precedence ignorance
10	100%	
11	14%	Right-associativity
12	54%	Operator precedence ignorance
13	4%	Operators not supported
14	10%	Right-associativity and parenthesis problem

5.3 Web testing system for online parking fee calculating system

We also applied our oracle generation approach to Song's application [31], a selenium-based web testing system. A real world web application, a parking lot calculator of Gerald Ford International Airport (http://www.grr.org/ParkCalc.php),

was used as our test subject in this application, as shown in Figure5.4.

Figure 5.4: Parking lot calculator

The parking lot calculator takes parameters including entry date and time, leaving data and time, and parking fee type, etc. as input; calculates the fee and returns a number for the parking cost. Traditionally, people generate test cases for web test by submitting parameters to servers and executing them, manually or automatically, to obtain the expected output. The procedure can be costly in time and money. We applied our oracle generation approach to generate executable JUnit test cases, which leverages the Selenium web testing framework [32] to test web applications automatically. Our framework can generate oracles for web-based tests with test cases

simultaneously, which improves the efficiency of test procedures of web applications. Because of space, only main part of the CFG and its valuation functions are attached here:

```
S ::= Round @@ (Round)

Round ::= Operation Fetch @@ ( Operation )
Operation ::= Lot Time Cal @@ (price Lot Time)
Fetch ::= 'bufferedWriter.write
          ( driver.findElement(By.cssSelector("b")).getText()+"\n");'

Lot ::= ShortTerm @@ ( ShortTerm )
Lot ::= Economy @@ ( Economy )
Lot ::= Surface @@ ( Surface )
Lot ::= Garage @@ ( Garage )
Lot ::= Valet @@ ( Valet )

Time ::= Entry Exit @@ (timeSub Exit Entry)
Time ::= Exit Entry @@ (timeSub Exit Entry)

Entry ::= EntryTime EntryDate @@ (dtimeStd EntryDate EntryTime)
Entry ::= EntryDate EntryTime @@ (dtimeStd EntryDate EntryTime)

EntryTime ::= EntryAmPm EntryTimeInput  @@ (time24Std EntryAmPm EntryTimeInput)
EntryTime ::= EntryTimeInput EntryAmPm  @@ (time24Std EntryAmPm EntryTimeInput)
EntryDate ::= 'driver.findElement(By.id("EntryDate")).clear();
              driver.findElement(By.id("EntryDate")).sendKeys("' TDate '");' @@ (TDate)

EntryAmPm ::= EntryAm @@ (EntryAm)
EntryAmPm ::= EntryPm @@ (EntryPm)
EntryAm ::= 'driver.findElement(By.name("EntryTimeAMPM")).click();' @@ (am)
EntryPm ::= 'driver.findElement(By.cssSelector
           ("input[name='EntryTimeAMPM'][value='PM']")).click();' @@ (pm)
EntryTimeInput ::= 'driver.findElement(By.id("EntryTime")).clear();
                    driver.findElement(By.id("EntryTime")).sendKeys("' TTime '");' @@ (TTime)

Exit ::= ExitTime ExitDate @@ (dtimeStd ExitDate ExitTime)
Exit ::= ExitDate ExitTime @@ (dtimeStd ExitDate ExitTime)

ExitTime ::= ExitAmPm ExitTimeInput @@ (time24Std ExitAmPm ExitTimeInput)
ExitTime ::= ExitTimeInput ExitAmPm @@ (time24Std ExitAmPm ExitTimeInput)
ExitDate ::= 'driver.findElement(By.id("ExitDate")).clear();
              driver.findElement(By.id("ExitDate")).sendKeys("' TDate '");' @@ (TDate)

ExitAmPm ::= ExitAm @@ (ExitAm)
ExitAmPm ::= ExitPm @@ (ExitPm)
ExitAm ::= 'driver.findElement(By.name("ExitTimeAMPM")).click();' @@ (am)
ExitPm ::= 'driver.findElement(By.cssSelector
           ("input[name='ExitTimeAMPM'][value='PM']")).click();' @@ (pm)
ExitTimeInput ::= 'driver.findElement(By.id("ExitTime")).clear();
                   driver.findElement(By.id("ExitTime")).sendKeys("' TTime '");' @@ (TTime)
```

Figure 5.5: Parts of syntax and valuation functions of parking lot calculator

Operations in valuation functions were defined in the domain class to calculate time, data and cost information for derived variables and eventually a cost can be calculated,

which was the oracle. We located failing test cases by comparing two values, one was the oracle of the generated JUnit test case; another was web testing results, which were the parking costs calculated from the subject system by executing the operation defined in the JUnit test case. Our approach simplified the process to get expected results from JUnit test cases instead of executing all test cases manually or automatically independently.

Table 5.4 shows testing results obtained in Song's work including failing test cases ratio and failing test case number by running five different groups of test cases.

Table 5.4: Report of web testing results on the parking lot calculator

	Failing Ratio	Case Number	Failing Cases Number
1	9%	100	9
2	14.0%	200	28
3	10.7%	300	32
4	11.3%	400	45
5	11.2%	500	56

Section 6 Conclusions and future work

We presented an automatic semantic-based oracle generation algorithm based on the denotational semantic approach. The approach realizes the mapping between the test case generation and associated oracle generation. Furthermore, we presented strategies to construct a semantic tree, which represents the semantic meaning of a test case. Our framework ensures that every generated oracle correctly represents the meaning of a test case as long as a correct denotational semantics rule associated with the test case grammar is given.

We have presented an automatic oracle generation framework based on our algorithms. The framework takes context-free grammar and semantic rules as input, produces test cases along with an associated oracle. Experimental results demonstrate the effectiveness of our oracle generation.

In the future, we will continue to enhance our framework in the following aspect:

- The optimization of the semantic tree:

  We noticed the possibility of simplifying the semantic tree structure from the experiments of our approach. The current framework binds each underived variable of test generation with a corresponding semantic node, where a semantic subtree will be extended when this variable is derived. However, there is an exception: when the associated valuation function is an identify

function, where the output is simply the same as the input. Recall that because the production rule in arithmetic expression example (E ::= F @@ (F)), the subtree will contain a regular semantic node E. Because this node does not influence the calculation for the sematic value of E, where λF.(F) is used, the semantic tree can be optimized to be compact one by omitting this semantic node.

- More use cases in practical applications:

We collected the experimental results from an open source license scanning tool, a grading system for Java programs handling arithmetic calculations, and a web test framework for an online application in this study. The results illustrate the effectiveness of our oracle generation approach and eventually an ability of fault detection. However, the experimental subjects are limited to small or middle scale systems. Because the complexity of input specifications and evaluation varies from applications, we plan to apply our approach to larger scale and more complicated applications.

Reference

1. K. Hanford: Automatic generation of test cases. *IBM Systems Journal* 9(4), pages 242-257, 1970.
2. H.F. Guo and Z. Qiu: Automatic Grammar-based Test Generation. In *The 25th IFIP International Conference on Testing Software and Systems*, pages 17-32, 2013.
3. H.F. Guo, H. Siy, and Z. Qiu: Locating fault-inducing pattern from structural inputs. In *the 29th Symposium on Applied Computing, Software Engineering Track*, pages 1100-1107, 2014.
4. R. Ferguson and B. Korel: The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, vol. 5, no 1. January, pages 63-86, 1996.
5. S.R. Shahamiri, W.M.N.W. Kadir, S. Ibrahim, and S.Z.M. Hashim: An automated framework for software test oracle. *Information and Software Technology (2011)*, 53(7), pages 774-788, July 2011.
6. L.P. Sobotkiewicz: A new tool for grammar-based test case generation. Master Thesis, University of Victoria, 2004.
7. D.L. Bird and C.U. Munoz: Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3), pages 229–245, 1983.

8. C. Burgess: The automated generation of test cases for compilers. *Software Testing, Verification and Reliability*, 4(2), pages 81–99, 1994.

9. C. Burgess and M. Saidi: The automatic generation of test cases for optimizing fortran compilers. *Information and Software Technology*, 38(2), pages 111–119, 1996.

10. E.G. Sirer and B.N. Bershad: Using production grammars in software testing. In *2nd conference on Domain-specific languages*, pages 1-13. ACM Press, 1999.

11. D. Hoffman: Heuristic Test Oracles. *Software Testing & Quality Engineering Magazine*, pages 29-32 1999

12. J.A. Whittaker: What is software testing? And why is it so hard? *IEEE Software 17 (2000)* pages 70-79

13. P.C. Jorgensen: Software Testing: a Craftsman's Approach, second edition. *CRC Press, LLC*, 2002.

14. J.D. Day and J.D. Gannon: A test oracle based on formal specifications. In *Proc. SoftFair, A Second Conf. on Software Development Tools, Techniques, and Alternatives*, pages 126-130, San Francisco, Dec 1985. ACM Press.

15. L. Baresi and M. Young: Test Oracles. *Technical Report CIS-TR-01-02*, August 2001

16. H. Robinson: Using pre-oracled data in model-based testing. *Microsoft*, July 1999.

17. Q. Xie and A.M. Memon: Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16(1), February 2007 Article No. 4

18. P. Le Gall and A. Arnould: Formal specifications and test: Correctness and oracle. *11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop Oslo, Norway, September 19–23, 1995*, pages 342-358, 1996

19. M. Last and M. Freidman: Black-box testing with info-fuzzy networks. In *M. Last, A. Kandel, H. Bunke (Eds.), Artificial Intelligence Methods in Software Testing, World Scientific*, 2004, pages 21-50.

20. M. Last, M. Friendman and A. Kandel: Using data mining for automated software testing. *International Journal of Software Engineering and Knowledge Engineering 14 (2004)*, pages 369-393.

21. S.R. Shahamiri, W.M.N.W. Kadir, S. Ibrahim, and S.Z. Mohd-Hashim: Artificial neural networks as multi-networks automated test oracle. *Automated Software Engineering, in preparation*. September 2012, 19(3), pages 303-334

22. D. A. Schmidt: Denotational Semantics: A methodology for Language Development. *Wm. C. Brown Publishers*. 1988

23. M. Last, M. Friedman, and A. Kandel: The Data Mining Approach to Automated Software Testing. *KDD '03 Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining,* pages 388-396, 2003.

24. S.L.Pfleeger: Software Engineering: Theory and Practice. 2nd Edition. *Prentice-Hall*, 2001.

25. APACHE Version 2.0. *The Apache Software Foundation,* http://www.apache.org/licenses/LICENSE-2.0, cited March 2014.

26. The FOSSology project. http://www.fossology.org/projects/fossology, cited March 2014.

27. P. Godefroid, A. Kiezun, and M. Y. Levin: Grammar-based whitebox fuzzing. *ACM SIGPLAN Notices,* 43(6), pages 206–215, 2008.

28. E. Bagheri, F. Ensan, and Dragan Gasevic: Grammar-based test generation for software product line feature models. In *the Conference of the Centre for Advanced Studies on Collaborative Research. IBM,* 2012.

29. L.X. Zheng and H.M. Chen: A systematic framework for grammar testing. In *IEEE/ACIS Int. Conf. on Computer and Information Science,* 2009.

30. ADAPTIVE PUBLIC LICENSE Version 1.0. *University of Victoria,* http://opensource.org/licenses/APL-1.0, cited March 2014.

31. Y.S. Song and H.F. Guo: Selenium Web Testing via Grammar-based Automation. Master Thesis-equivalent project report, University of Nebraska, 2014.

32. The Selenium project: http://seleniumhq.org/, cited April 2014.

33. P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber: Automatic testing of reactive systems. In *32nd IEEE Real-Time Systems Symposium*, pagers 200-209, 1998.

34. P.R. Henriques, M.J.V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu: Automatic generation of language-based tools using the LISA system. *Software, IEE Proceedings*, 152(2), pages 54-69, April 2005.

35. J.W. de Bakker and E.P. de Vink: Denotational models for programming languages: applications of Banach's fixed point theorem. *Topology and its Applications*, 85(13), pages 36-52, 1998.

36. G. Gupta: Horn logic denotations and their applications. In *The Logic Programming Paradigm*, pages 127-159. Springer, 1999.