

Language Fuzzing with Name Binding

Greg Newman

16206371

Supervisor: Bernd Fischer

November 2015

Abstract

The language fuzzing with name binding project generates syntactically valid test programs that exercise the name binding semantics of a language processor. We introduce generation algorithm and a tool, NameFuzz, for the test suite generation. It achieves this by parsing in a ANTLR grammar representing the context free grammar of the language, along with the language's name binding rules in the NaBL meta-language. The test sentences are intended to be either accepted (positive test cases) or rejected (failing test cases) by the language processor. The intention is to promote confidence in the language processor as far as semantic correctness is concerned. The generated test suite is syntactically correct, but the limitations of not taking type checking into account or having a method to evaluate expressions lead to a large number of test sentences that are semantically incorrect. To a degree these limitations are overcome by the combinatorial nature of the generation algorithm that ensures that each possible type correct sentence is generated as well.

Contents

1	Introduction	5
2	Background	7
2.1	Context Free Grammars	8
2.2	Language Fuzzing	9
2.3	ANTLR	12
2.4	Name binding	13
2.5	NaBL	16
3	System Requirements	19
3.1	Product Functions	19
3.2	Specific Requirements	19
3.2.1	Test Suite Generation	19
3.2.2	Test Execution	20
3.3	Non-Functional Requirements	21
4	Test Suite Generation Algorithm	21
4.1	Criteria	22
4.2	NaBL Rules	22
4.3	ANTLR Grammar	23
4.4	Test Suite Generation	28
5	System Architecture	35
5.1	Overall System Architecture	35
5.2	Metadata Extraction	36
5.3	Test Suite Generation	37
5.4	Language Processor Testing	37
5.5	External Tool Integration	38
5.6	Class Structure	38
5.7	Class Diagram	41
5.7.1	Test Generation	42
5.7.2	Language Processor Testing	44
5.8	Deviation from Original Design	44
6	Implementation Details	45
6.0.1	Generating the Test Suite	45
6.0.2	Testing the Language Processor	45

7	Project Evaluation	46
7.1	Testing	46
7.2	Test Suite Verification	46
8	Related Work	48
9	Conclusion	49
10	Appendix A	52
11	Appendix B	55
12	Appendix C	56
13	Glossary	60
13.1	Definitions, Acronyms and Abbreviations	60

1 Introduction

The focus of the project is to extend the process of language fuzzing to incorporate name binding semantics. This enables the testing of language processor correctness with regards to name binding rules. A test suite generation algorithm will be presented as well as a fuzzing tool called NameFuzz that implements the described algorithm.

Language fuzzing is a well developed technique that uses context-free grammars (CFGs) to generate structured test inputs for language processors, such as generating programs for compilers. These techniques help promote confidence in the processor. Previously, language fuzzing techniques have primarily focused on syntactic correctness, however this is not enough as it overlooks the need for semantic correctness. While other work has been done in this area (see section 8 Related Work), none yet focus on the name binding semantics of a language. The primary goal of the project is to implement a language fuzzer that takes a CFG and name binding rules as input in order to generate a test suite that can then test the language processor and provide information about potential flaws as output. We shall illustrate this using an example of two test cases derived from the Java language [6].

```
public class A {
    public static void main(String[] args) {
        x = 3;
    }
}

public class B {
    public static void main(String[] args) {
        int x;
        x = 3;
    }
}
```

Although both of the derived programs are syntactically correct, program **A** will not be accepted by the Java compiler. This is because it is trying to use the variable **x** before it is declared, which is against the name binding

rules of Java which declare that a variable must be declared before it can be referenced [6]. Program B on the other hand is both syntactically and semantically correct as it obeys the rules laid out by the grammar as well as the name binding and type rules of the Java language.

In order to test the language processor for correctness a set of positive and negative test programs should be generated, this means that the compiler should accept these programs, for the positive test cases, or reject them, for the negative test cases. While it is possible to generate syntactically incorrect test cases for the language processor, the focus here shall primarily be on name binding rules thus each test case targets a single name binding rule, that it exercises correctly in the case of positive test cases or intentionally exercises the rule incorrectly. This is to ensure that while testing the language processor it is a simpler process to determine the source of the error that resulted in the incorrect handling of the test program, as opposed to a test program that exercises many rules and not knowing which is the culprit.

There are some limitations with this approach. The most significant of which is type correctness. Not all language processors are concerned with types, such as those for untyped languages. But in the case of typed languages type correctness plays an important part of the semantic correctness. For this project type correctness will not be incorporated, there are two main reasons for this: firstly type correctness is out of the scope for the current project as the primary focus is on name binding semantics, and secondly while the NaBL does support types [14], the language is not currently sufficient to convey all the necessary meta-data to generate type correct test inputs. For example, consider the following Java program:

```
public class C {
    public static void main(String[] args) {
        int x = 1;
        if (x) {
            x = 2;
        }
    }
}
```

Program C is both syntactically correct with respect to the grammar, and

semantically correct, but it is still an incorrect Java program; the condition for the `if` statement must evaluate to a boolean value.

The goal of the project is to create a tool that automates the generation of a test suite that uses name binding semantics to test the correctness of a language processor. The implementation of the approach described above is done in a tool called NameFuzz. For the implementation the user needs to be able to specify the CFG and name binding in some formal manner. The CFG for a language will be parsed in as an ANTLR [2] grammar, and the name binding rules in the NaBL [14] meta-language. The NaBL name binding rules are then used to drive the generation process, with the information provided by the ANTLR grammar is then used to ensure that all test programs are syntactically correct. Section 2, goes into more detail about how the ANTLR grammars and NaBL meta-language are used and Section 5 and 6 provide a detailed outline about the system design and implementation.

The rest of the document is as follows. Section 2 provides an overview of the background of the project and the definitions used in the implementation. Section 3 provides the system and non-functional requirements that the NameFuzz project aimed to fulfill. Section 4 gives an overview and explanation of the test suite generation algorithm used in the implementation. Section 5 provides the system design of the NameFuzz tool. Section 6 describes the implementation of the system. Section 7 provides an evaluation of the project. Section 8 gives an overview of work related to the project. Section 9 provides a conclusion with the results.

2 Background

This section provides the necessary background knowledge in order to better understand the process by which the test programs are generated, and the implementation details of the project itself. In order for the project to generate the test programs it must first be able to generate syntactically correct program sentences, parse a CFG in ANTLR format and parse the name binding rules in the NaBL meta-language. These concepts and tools are explained in detail below.

2.1 Context Free Grammars

The definition we will be using for a context free grammar (CFG) is described in Kozen [15].

A CFG is a four-tuple $G = (N, \Sigma, P, S)$ where:

1. N is a finite set, the non terminal symbols.
2. Σ is a finite set, the terminal symbols disjoint from N .
3. P is a finite set of $N \times (N \cup \Sigma)^*$, the productions.
4. $S \in N$, the start symbol.

The CFG G can describe a language L by generating each string of that language in the following manner [18]

1. Write down the start variable.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until complete.

A sequence of substitutions required to generate a string s from language L is called the *derivation* of the string. Thus s is *derived* from G .

A string s where $s \in (N \cup \Sigma)^*$, and derivable from the start symbol, S , is said to be in sentential form, and if s is of a sentential form which consists only of terminal symbols then s is said to be a sentence. The set of all derivable sentences from the start symbol S of the CFG is called the generated language by G of the CFG and is written $L(G)$. The sentence is ‘accepted’ by G if it is of correct sentential form.

2.2 Language Fuzzing

Manually writing test suites for a language processor is a tedious and time consuming task for any developer. Generating these test suites from the grammar itself saves the developer the burden of having to write the test suite manually. Language fuzzing is a technique often used to generate test data in order to promote confidence in a language processor. It achieves this by generating input test data from a CFG. Running the language processor over the test data then provides indications of where there may be faults within the language processor.

Generating sentences of correct sentential form is a simple task; starting at the given start symbol S randomly apply productions until all non-terminals have been replaced by terminals leaving a sentence. This simple approach has two major flaws, firstly cyclic grammars may take a long time to terminate resulting in extremely long sentences. This method is not sufficient in that the probability of coverage of all aspects of the language is small.

Purdom [17] first developed a systematic method for generating structurally correct test data for a parser. The fuzzer generates tests that are structurally correct from a CFG in an complete exhaustive approach.

```
While productions not covered create a new test case {
  Put the start symbol on the stack
  Repeat until the stack is empty {
    Pop a symbol off the stack
    If the symbol is a terminal {
      print it
    } Else {
      Push the RHS of a production satisfying either:

      - A production that has not yet been applied.
      - A production that introduces a non-terminal that
        is not on the stack for which not all productions
        that rewrite it have been used.
      - A production which will lead to the shortest
        derivation.
    }
  }
}
```

The main problem with this approach is that the number of generated sentences will be small, and each sentence will exercise as many rules as possible, should a problem be uncovered by a sentence it will be problematic to determine which aspect of the sentence resulted in the problem. For example consider the example grammar G below:

R1: $S \rightarrow AB$
R2: $A \rightarrow aC$
R3: $B \rightarrow bC$
R4: $C \rightarrow \epsilon$
R5: $C \rightarrow cC$
R6: $C \rightarrow D$
R7: $D \rightarrow d$

Using the method developed by Purdom, the result would be a single test case that exhaustively covers all the possible rules in the grammar:

R1: AB
R2: aCB
R3: $aCbC$
R4: abC
R5: $abcC$
R6: $abcD$
R7: $abcd$

The resulting test suite: $\{abcd\}$

Much work has been done in providing methods to efficiently generate test data that provide better coverage. Lämmel and Schulte [16] specified a context dependent rule coverage criteria, and developed an algorithm that provides combinatorial coverage from the input grammar. Lämmel and Schulte define the term occurrence which, following from the definition of a CFG, states that for a production rule $p = A \rightarrow bCd$, where $A, C \in N$ and $b, d \in (N \cup \Sigma)^*$ it is said that production p is an occurrence of C , the occurrence also contains the index within the production, so that a single production can be multiple occurrences of the same non-terminal. A test set W will then have context dependent rule coverage if W covers every production under all occurrences.

The result is the ability to generate test inputs that were structurally valid or invalid, as well as being minimal in testing a single syntactic aspect, or occurrence, at a time. This makes it easier for the person doing the testing to identify what caused the error. This is achieved by testing all alternates of non-terminals in every place they occur within all the production rules, and then follow the shortest production to terminal symbols. These alternates are then tested per individual test case ensuring that the test cases remain small, and problems easy to locate.

Again using grammar G as an example, starting at R1 the method would test all A productions followed by all B productions in R1, resulting in the following test cases: aCB and AbC .

The *shortest path* is then followed to make these test cases as simple as possible: ab and ab , which are then added to the test suite to become the set $\{ab\}$. The same is then done for all occurrences of production C in R2 and R3, resulting in aCB and AbC , once again taking the shortest path leads to the test cases: acb and abc . The test suite is now the following set $\{ab, acb, abc\}$.

Finally repeating until completion we are left with test suite set $\{ab, acb, abc, accb, abcc, abcd\}$. Where the test cases are much smaller and simpler to understand should an error need to be located.

These, however, did not take input constraints nor the semantics of the language into consideration. The problem with this is that; while providing good coverage of the input structure whilst testing the language processor, they fail to take into account other semantic rules that are vital for more complex language processors such as compilers. Often resulting in test data that is structurally valid but semantically invalid, or fails to provide coverage of the semantic aspects of the language. The tool developed for this project will make use of an augmented version the combinatorial coverage algorithm developed by Lämmel and Schulte in order to generate the syntactic parts of the test suite. The augmentations to the method will incorporate the name binding semantics.

2.3 ANTLR

The implementation requires a format for the input grammar, as well as a parser for the input grammar. The ANTLR (ANother Tool for Language Recognition) [2] tool is selected for this purpose, as it provides a language to specify a grammar as well as being able to generate a parser for the language. Once the ANTLR grammar is parsed, ANTLR provides a variety of tools to analyze the grammar. Consider the following example of the grammar G , the production names have been renamed in accordance with the naming conventions of ANTLR; production rule S has been renamed to $prod_S$ etc.

```
grammar G;

prod_S : prod_A prod_B ;
prod_A : 'a' prod_C ;
prod_B : 'b' prod_C ;
prod_C : ('c' prod_C)?
        | prod_D
        ;
prod_D : 'd' ;

WS : [ \t \r \n]+ -> skip ;
```

Productions are lowercase words, terminals are enclosed within “ ’ ”, and lexical terms are words in all uppercase followed by a comma. Below is an example of an input sequence and its corresponding parse tree.

Input: *abcd*

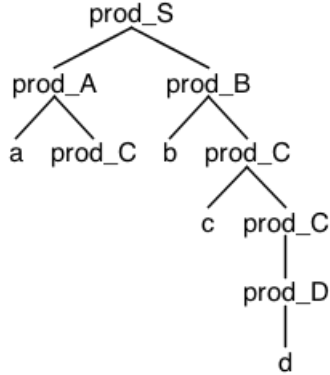


Figure 1: The corresponding parse tree for the ArrayInit ANTLR grammar.

While this is simple enough to illustrate how ANTLR generates a parse tree from an input, it does not entirely reflect how to derive sentences from the grammar, only to construct parse trees from an input sentence for the grammar. In order to derive sentences an ANTLR grammar is used to represent the ANTLR language to generate a parse tree of the input grammar itself. The resulting parse tree for G is displayed in Figure 1. Refer to Section 4 on how these parse trees are used in the generation of the test suite.

2.4 Name binding

Name binding is the process where a unique identifier is allocated for a variable, function, method, class etc. It is primarily concerned with the relation and resolution between definitions and references of these identifiers, as well as the scope rules that govern these relations. Language processors need to be able to make use of the information about the identifier at the reference however languages do not always use the same set of rules regarding the definitions, references nor scopes. There are however general patterns that can be taken advantage of [14] when it comes to generalising these rules between languages. The project will focus on a subset of these rules. This subset will demonstrate the core name binding properties shared across most languages while ignoring some of the more esoteric rules. The following is a description of the supported name binding rules, with examples using the java subset.

Definitions and References

Definitions bind names, also called identifiers, to entities. In the example below, an integer entity is declared and assigned the identifier **x**, this is known as the *declaration*, and this site is known as a definition site. Often declarations are required to be unique, however this is not always the case.

```
public class A {  
    public static void main(String[] args) {  
        int x;  
    }  
}
```

References use the declared name to access the information about the defined entity. In the example below the entity **x** is referred and assigned a value of 1. This is called a reference site. A reference site needs to obey the scope rules of a language or it will be invalid.

```
public class A {  
    public static void main(String[] args) {  
        int x;  
        x = 1;  
    }  
}
```

Scopes

Scopes restrict the range of the definition sites. In program A below the variable **x** is declared in the scope of the **main** function, it is then referenced in an inner scope created by the **if** statement, enclosed in the braces **{}**. This is acceptable as the Java language allows variables declared in outer scopes to be nested within inner scopes. In program B on the other hand the variable **x** is declared in the scope of the **if** statement, and an attempt to reference the variable is made in the outer scope. This is not allowed and will result in an error.

```
public class A {  
    public static void main(String[] args) {  
        int x;  
    }  
}
```

```

        if (true) {
            x = 1;
        }
    }
}

public class B {
    public static void main(String[] args) {
        if (true) {
            int x;
        }
        x = 1;
    }
}

```

Namespaces

A namespace is an abstract container created to hold a logical grouping of unique identifiers that separates them from equivalent identifiers within a separate grouping. The example program below shows how complete namespaces can be used distinguish between variables with the same identifiers. Although the program has three separate variables all with the same identifier it is a valid program because each variable is declared within a unique name space. Without using the namespace to determine the variable it would other be ambiguous to which **x** the developer is referring.

```

public class C {
    public class A {
        public static int x = 0;
    }

    public class B {
        public static int x = 1;
    }
}

```

```
public static void main(String[] args) {  
    int x = C.x;  
    x = B.x;  
}  
}
```

Import visibility modifiers

Imports introduces the definitions in a scope into another where these definitions would not typically be visible. In the example below the variable `y` is would not be visible to the program `C` if not for the `import` statement.

```
public class A {  
    public static int y = 0;  
}
```

```
import A;  
  
public class C {  
    public static void main(String[] args) {  
        int x = A.y;  
    }  
}
```

2.5 NaBL

NaBL is a meta-language with linguistic attributes for declaring for defining name binding and scope rules [7]. The language was originally written for the Spoofax Language Workbench, to be used in conjunction with SDF for software language engineering. The principles behind the language are adapted

here, so that the language fuzzer can extract the necessary meta-data from the name binding rules and incorporate them into the test generation process.

Each binding rule is the name of a production rule in the grammar, followed by the name binding rules of that production rule. There are 4 different name binding rules in NaBL that are focused on in this project. These are the `defines`, `refers`, `scopes` and `imports` rules. Each of these rules has attributes that are discussed below:

- `defines` indicates that the production rule is a declaration. These can be `implicitly` or `explicitly` defined, and either be unique or non-unique.
- `refers` indicates that the production rule is a reference.
- `scopes` Specifies the scope ranges of the rule. These can be explicitly named with a `namespace`, or the key word `subsequent` can be used for the subsequent scope.
- `imports` identifies the importing of variables from one namespace into another. For example, `imports Var from Program`.

Each sentence derived from the CFG will consist of at least one name binding rule, but in most cases more. For this reason the generation process must prioritize which of the rules that it is testing. We shall call a sentence that uses only the name binding rules absolutely necessary to include the prioritized name binding rule in the derivation, as *minimal closure*.

Below is an example of name binding rules specified in NaBL for the NanoPascal Language ANTLR grammar (see Appendix A).

module NanoPascal

imports

namespaces

Var

Program

binding rules

program(x) :
explicitly defines unique Program x
scopes Block, Var

varDecl(x) :
explicitly defines unique Var x
in subsequent scope

variable(x) :
refers to Var x

block(Statement) :
scopes Var

The **namespaces** block defines all the namespaces within a language, in this case they are Var (variables), and Program (the NanoPascal programs.) The **binding rules** block is where all the naming binding and scoping rules are defined. Each rule is a production, the namespace that the production is associated with, and the namespaces that it scopes. The NaBL grammar traditionally allows certain keywords such as unique to be omitted, in which case the declaration is assumed to be unique, but in this case where the language is used to aid language fuzzing the grammar has been modified to make it more explicit. Meaning that keywords can no longer be omitted.

This promoted readability and prevents subtle bugs when generating test suites.

3 System Requirements

3.1 Product Functions

The function of the tool is to use a CFG and name binding rules to generate a suite of small test cases that aim to systematically exercise individual name binding rules that promote confidence in a language processor's correctness. The tool will need to generate a test suite the both positive and negative test cases.

Positive test programs are those that implement a name binding rule correctly, the language processor should accept these as correct. While negative test cases intentionally incorrectly implement a name binding rule. The language processor should not accept these and should throw an error.

The target audience of the NameFuzz tool language processor developers that wish to test their language processor for name binding correctness.

3.2 Specific Requirements

3.2.1 Test Suite Generation

The system shall, given a CFG and name binding rules, generate sentences that are suitable as test cases.

3.2.1.1 The system shall accept a file containing the CFG defined in the ANTLR metasyntax.

3.2.1.2 The system shall accept a file containing name binding rules in the metalanguage NaBL (Name Binding Language).

3.2.1.3 The sentences shall be structurally correct as determined by the CFG.

Rationale: This allows the focus of the project to be on exercising the name binding rules and not the syntactic form.

3.2.1.4 Each test case shall be minimal in that it primarily exercises a single given name binding rule as specified in NaBL under minimal closure.

Rationale: This is so that each test case is specific to a single rule, allowing the user to more easily identify which rule was incorrectly processed by the language processor. And whether or not the error message was correct.

3.2.1.5 The system shall generate positive test data that aim to correctly exercise a single name binding rule.

3.2.1.6 The system shall generate negative test data that aim to incorrectly exercise a single name binding rule or name constraint for each test case.

3.2.1.7 The system shall store the test suite in a local directory as valid program files recognised by the language processor.

3.2.2 Test Execution

The system shall be able to execute a language processor over a given test suite that is constructed according to 3.3.1.

3.2.2.1 The system shall determine which of the test cases are parsed/failed correctly/incorrectly by means of analysing the state of the language processor as well as printed messages.

Rationale: This is done to provide meaningful feedback to the user.

3.2.2.2 The system shall convey these results to the user, by stating which of the tests passed and which failed.

3.2.2.3 Each test case, whether passed or failed, shall be accompanied by a message detailing the name binding rule of the test case, and there error if there is one.

Rationale: The system needs to determine whether or not the language processor failed the test case for the correct reason.

3.3 Non-Functional Requirements

3.3.1 The system shall be able to generate test cases for block-structured languages.

Rationale: This is to ensure support for a wide variety of commonly used languages.

3.3.2 The system shall be developed such that additional name binding features can be incorporated later on.

Rationale: There are variations of name binding rules that will not be supported in this version. Consideration for later implementation in later versions will increase the testing ability of the tool.

4 Test Suite Generation Algorithm

This section provides an overview and explanation of the test suite generation algorithm including; what criteria the test suite must fulfill, the information extracted from the NaBL and ANTLR files, the data structures used, and the generation algorithm itself.

Generation of the test suite is focused on primarily exercising a single name binding rule per test case by minimising the number of name binding rules present to just those that are absolutely required for the test case to be valid. This shall be termed as having the test case under *minimal closure*, and the rule that is being exercised shall be referred to as the *target rule* or just the *target*.

The generation of the test suite can be broken down into three separate components. The parsing of the NaBL name binding rules to extract the relevant information, parsing the ANTLR grammar and using the parse tree to build a graph and finally using the information from the previous two parts to generate the test suite. Isolating the components into these stages makes the process of the test suite generation easier to understand as well as easier to implement and debug.

4.1 Criteria

A test suite shall be successfully generated if it fulfills the following criteria.

1. Each test case shall exercise a single target rule under minimal closure of the other rules.
2. Name binding rules should be implemented in the correct order. Declarations must precede usage.
3. To ensure that it is easier for a developer to locate potential errors, test sentences will be generated to test those rules that require the least number of accompanying name binding rules. The process of the test suite generation will first create a set of tests prioritizing definition rules, then prioritizing reference rules, then scope rules and finally import rules.

4.2 NaBL Rules

The NaBL rules contain all the semantic information about the language. Firstly the NaBL rules for the accompanying grammar are parsed and all the necessary meta information about the rules is extracted. Consider the NaBL rules provided for the small NanoPascal grammar provided in Appendix A. Each binding rule associates a production with a namespace, and gives the necessary details about the type of rule and its characteristics. For example, the binding rule `varDecl` associates the production `varDecl` in the grammar to the namespace `Var`. The `defines unique` clause of the rule tells us that `varDecl` is a production associated with declarations, and that each declaration is unique. The `scopes` clause also informs us which scope the declaration will take place in.

The `variable` binding rule on the other hand has the clause `refers` which indicates that the `variable` production is associated with references to the namespace `Var`. Once these two are parsed we know how the namespace `Var` is declared, referred to and the conditions under which these two must happen.

The parsing of the NaBL rules continues in this fashion for all the namespaces and binding rules, storing the information in a data structure called

the NaBL index. The NaBLIndex is a lookup table for information about namespaces, productions and scopes. Below is an example of an entry in the NaBLIndex for namespaces:

```
Key: Var
declProduction: varDecl
refProduction: variable
```

And another for more information about the productions associated with a namespace:

```
Key: varDecl
ruleType: declaration
unique: true
scopes: subsequent
```

```
Key: variable
ruleType: reference
```

Entries such as these provide a quick and efficient way to look up the required meta data during the test suite generation process. The NaBLIndex is also capable of returning a list of all the productions associated with name binding rules. During the test generation process, which will become apparent later, the list of productions associated with name binding rules is used to ensure that name binding rules that are not required to exercise the target rule are not included in the test sentence. Referring to the example above, assume that the current target rule is `varDecl` which is a declaration rule. A suitable test case would not then include the `variable` name binding rule as it is not required to exercise the target. However if `variable` is the target rule, and we know that it is a reference rule then the associated declaration rule `varDecl` would have to be incorporated into the test case for the target rule to be valid, and satisfy declaration and usage criteria.

4.3 ANTLR Grammar

Parsing the ANTLR Grammar

The ANTLR grammar contains all the syntactic information about the language required in order to construct test cases of correct sentential form. Parsing the grammar results in a parse tree from which a graph representation of the language is constructed. Each *production node*, visualised with a single border, in the graph represents a production rule in the grammar. Each of these nodes contains an ordered list of terminals and non-terminals, representing the right hand side (RHS) of the production rule, required to complete the rule. *Alternative nodes* or *alternatives*, visualised with dual borders, are used to represent possible alternatives for a production rule. Edges on the graph represent rules that need to be completed in order for the generated sentence to be in the correct sentential form, thus when traversing a tree only one alternative may be selected at a time.

Constructing Sentences

The graph can also be used to construct sentences, here is an example of how this works however it is important to note that this example is a single derivation to assist explanation, the actual generation algorithm is explained later in this section; select the *current node* to be the given start node and print out each terminal as is encountered in the current node. If a non-terminal (other production rule) is encountered, recursively enter that production node, making it the new current node, and repeat the process until the algorithm returns to the start node and all terminal symbols have been printed out. At that point the remaining terminals are printed out as the current node is moved back up the graph to the start node by recursion. If there is more than one edge from a alternative node, only one can be selected at a time in order to ensure that the sentence remains in correct sentential form. If the current production node has a form of EBNF notation such as `?`, `*`, `+`, then the node can be used, repeated or ignored as dictated by the annotation.

The lexical rules, those that are in capital letters in the grammar, are resolved by generating words from them that obey the regular expression that defines them. Consider the rule `ID : [a-zA-Z][a-zA-Z0-9]*`. This rule defines identifiers that can begin with either a lower case or uppercase letter, followed by any combination of letters or digits between 0 and 9 under star closure. This rule would be resolved by reducing it to a random word derived from the regular expression, for example *a*, *ab*, *ab1*, *Ab*, *Ab1*, *AbbBBb2* are

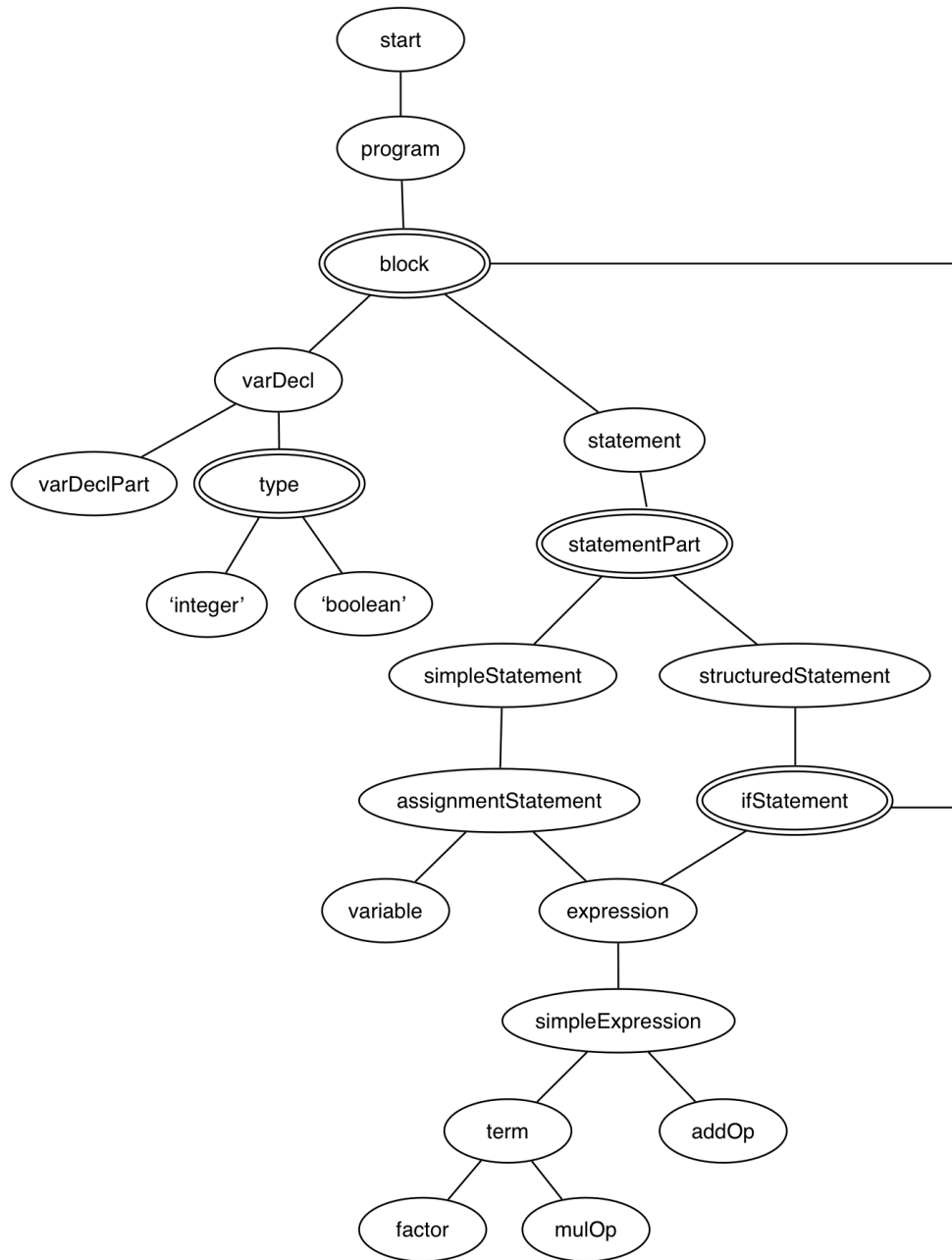


Figure 2: Graph representation of the NanoPascal grammar

all valid identifiers. This is simply done through a function that generates identifiers as required.

Consider the following sentence derived from the graph above. Set **start** as the current node, iterating over the RHS the production rule **program** (not to be confused with the terminal '**program**') is encountered and set to the current node. Iterating over the RHS printing out the terminals and resolved lexical rules results in:

```
program Ab ;
```

The **block** non-terminal is encountered and set as the current node. **block** has the EBNF notation ***** which means the rule can be exercised 0 to many times. Choosing to exercise the rule once results in **block** becoming the new current node. **block** has two alternatives, **varDecl** and **statement**, **varDecl** is chosen and set as the current node. Printing out the terminals and resolved lexical rules in the RHS:

```
program Ab ; var a
```

The next symbol **varDeclPart** is also a non-terminal with the EBNF notation *****. Choosing not to exercise the rule results in **varDecl** remaining as the current node. Continuing to print out the terminal symbols in the RHS:

```
program Ab ; var a :
```

The **type** alternative node is encountered next and set as the current node. The RHS has two alternatives, '**boolean**' and '**integer**'. Although these would result in two **type** production nodes with different RHSs they are referred to by the RHS in the graph to assist in distinguishing between the two. Selecting '**integer**' and processing the RHS results in:

```
program Ab ; var a : integer
```

Now that the RHS of **type** is complete, the parent **varDecl** is set to the current node, and the remaining symbols in its RHS are processed:

```
program Ab ; var a : integer ;
```

As each node's RHS is completely processed, the parent once again becomes the current node and the remainder of its terminals are printed out. Continuing in this fashion for the remainder of the graph results in the following sentence which is of the correct sentential form:

```
program Ab ; var a : integer ; .
```

Context Dependent Rule Coverage

Adapting the combinatoric approach developed by Lämmel and Schulte [16] for the graph makes achieving context dependent rule coverage simple. All one needs to do is ensure that every combination of alternatives in the graph is selected at least once. An easy way to achieve this is to remember the path taken through the graph to generate the sentence. Each time a sentence is completed back propagate through the graph. If an alternative node is encountered check to see whether or not there are unexplored alternatives, if so set the alternative production rule as the current node, mark it as explored and complete the sentence. If all alternatives have been explored then mark all of its children as unexplored such that each can be explored again from a different point in the graph. Once a sentence is complete print out each of the terminals on the stack. Any cycles in the graph must be followed once to ensure that there is coverage of nested rules as well. Consider the previous example for constructing a sentence:

```
program Ab ; var a : integer ; .
```

Back propagating through the graph until we arrive at the **type** production node and have the choice between the alternatives '**boolean**' and '**integer**'. Since '**integer**' has already been used, it would have been marked as explored, thus '**boolean**' would be set to the current node and marked as explored. Continuing the construction would result in:

```
program Ab ; var a : boolean ; .
```

The algorithm would then back propagate back to the choice of alternatives before the **type** production, and mark both alternatives as unexplored as all available alternatives have been used. From there it would continue to back

propagate all the way back to the choice of alternatives before the **block** production node, where the next of alternatives would be the **statement** production, from which each combination of alternatives would then be generated. Continuing in this way results in a set of sentences that represent each possible combination of production rules and achieves context dependent rule coverage.

Pretty Printing

Generalizing pretty printing for each ANTLR grammar is no simple task as there is simply not enough semantic information to ensure a complete logical structure during pretty printed. To ensure that the sentences are still easily readable and do not consist of a single straight line, the following rules are used:

- Each time the RHS of a production node is completely processed, start a new line and decrease the indentation amount during the back propagation.
- Each time a new current node is selected, start a new line, and increase the indentation amount.
- If the new current node is a repetition resulting from a from of EBNF notation, start a new line but do not increase the indentation amount.

This allows a simple structure in the generated sentences. The above example pretty printed:

```
program Ab ;
    var a :
        boolean
    ;
.
```

4.4 Test Suite Generation

The test suite generation for the name binding rules is a *target* driven process, where each target is a name binding rule and that rule will be the priority of

the generation process. The ideal test sentence would consist of no other rules other than the target, however this is impractical as it would result in test sentences that are not of correct sentential form. What can happen though is to make sure that the test sentence is under minimal closure, meaning that other than the target only name binding rules that are absolutely necessary for the sentence's correctness will be added as well. Below is a high level representation of the algorithm.

```

fundef generate(s, targets):
    if empty(targets):
        complete_closure(s)

    for symbol in RHS:
        if symbol is terminal:
            print symbol

    else if symbol has ENBF notation:
        for e in get_repeats(symbol):
            generate(s, targets)

    else if symbol == target.peek():
        target.pop()
        generate(symbol, targets)

    else if symbol is production rule node:
        generate(symbol, targets)

    else if symbol is alternative node:
        for alt in get_relevant(symbol):
            mark_explored(alt)
            generate(alt, clone(targets))
        unmark_all(symbol)

```

This algorithm is a simple modification of the algorithm for context dependent rule coverage in the section above. The main difference is that the process is now driven by the target with a goal of completing the rule under

minimal closure. The caveat here is that the criteria of context dependent rule coverage shall not be met with regards to all occurrences that are associated with other name binding rules. Context dependent rule coverage is secondary to the criteria of the rules being under minimal closure.

The algorithm takes two inputs, the start production node and a stack that represents the order of targets that need to be visited for the sentence to be of the correct form. The targets in the target queue contain all the production rules, and criteria under which the name binding rule should be executed, for instance whether or not the rule is a declaration, reference, whether or not the declaration of the rule might be unique or not, the scope information about the rule and whether or not the rule has been exercised positively or negatively.

The three functions `complete_closure`, `get_relevant` and `get_repeats` require further explanation. `complete_closure` is called once all the targets in the stack have been met. This method completes the generate process in a similar manner but ignores all production rules that are not absolutely necessary to the completion in correct sentential form. `get_relevant` is a query to the NaBLIndex data structure. This method returns all the production rules from the alternative node that will lead to the target production rule. Consider the alternative node `block` in Figure 2 and with the current target `varDecl`, `get_relevant(block)` will return both `varDecl` and `statement` as relevant production nodes because both lead to the `varDecl` rule being exercised. However if `variable` was the current target then only `statement` would be returned as `varDecl` does not lead to `variable` being exercised. The `get_repeats` method repeats the production a number of times in order to achieve coverage of the EBNF notation, for example if the rule can be repeat none to many times, it will create three separate test sentences; one where the rule is not used at all, one where the rule is used once and one where the rule is used twice.

Cloning the target stack for each alternative is so that the target gets exercised at least once in each possible relevant alternative. Should not all the targets be exercised by some combinations of the alternatives then that sentence is simply ignored as it does not result in `complete_closure` being called.

The name binding rule coverage depends of how the target stack is populated for that rule. Each rule type requires a slightly different process, each of which is described in detail below.

Positive Test Generation

Declarations

Declaration rules are the simplest to test. The target production node that defines the rule need simply be pushed onto the target stack, and during generation the rule is exercised in each possible location in the grammar. Declaration rules that are unique are simply exercised once, and the production rule is exercised again but with a different identifier. For those that are not unique the process is the same approach however the identifiers are now equal. A simple example from the NanoPascal grammar.

```
program A ;  
    var a : integer ;  
    var b : integer ;  
.
```

And if the declaration rule for **Var** was not unique:

```
program A ;  
    var a : integer ;  
    var a : integer ;  
.
```

References

Reference rules are slightly more complex. The declaration production rule needs to be exercised before the reference production rule, and the references need to take place in each of the possible scope. Thus the conditions that must be satisfied is that first the declaration rule must be exercised, the generation algorithm already ensures that this happens at each possible location, so it must be at the top of the stack. Next, the production rules associated with the scope rule must follow. This ensure that after the declaration rule is exercised the generation algorithm enters into one of the relevant scopes before exercising the reference rule. And lastly the reference rule that needs to be exercised. Below is an example of an exercised reference rule.

```

program A ;
    var a : integer ;
    if (true) then
        a := 0 ;
.

```

Scopes

Scopes are defined simultaneously as the references, as the only method with which to test the scoping mechanisms of the language processor is to make use of declarations and references. Since the set of test sentences generated from the references clause consists of each possible combination of declarations and references, it is safe to assume that these simultaneously test the scope mechanisms.

Imports

Imports require that a namespace be generated before hand in order to import. All the identifiers are stored in a symbol table. The import clause must then be at the top of the target stack, and instead of resolving to a random identifier, the namespace from the symbol table is used. After this the target stack is set up much the same as with the reference generation, however all instances of ID are now taken from the symbol table, not randomly generated.

```

public class A {
    public static int y = 0;
}

import A;

public class B {
    public static void main(String[] args) {
        int x = A.y;
    }
}

```

Negative Test Generation

Declarations

Negative test sentences for the declaration rules is as simple as the positive sentences. The unique rule is simply negated.


```

program A ;
    var a : integer ;
    var a : integer ;
.

```

References

The generation of the negative test sentences for the reference rule is again very similar to that of the positive scope rules. The main differences are that a set of target stacks are created that do not specify declaration targets, this tests that variables are defined before use. The scopes for the references are replaced with the set difference of all the scope production rules are the scope production rule for the reference. This ensures that the reference is tested at each possible location, where it should not be.

```

program A ;
    if (true) then
        a := 0 ;
.

```

```

program A ;
    if (true) then
        var a : integer ;
        a := 0 ;
.

```

Scopes

As explained above the scope mechanisms are tested simultaneously with the references.

Imports

Imports cannot be tested in a negative fashion because it is not possible to distinguish incorrect imports from undeclared references and name errors. Consider the following Java example, class B and C demonstrate the only two ways in which to test the import negatively, but in both cases the error in the name, or an undeclared variable. It does not test the actual namespace semantics of the import.

```

public class A {
    public static int y = 0;
}

```

```
import A;

public class B {
    public static void main(String[] args) {
        int x = A.b;
    }
}

import Z;

public class C {
    public static void main(String[] args) {
        int x = A.y;
    }
}
```

5 System Architecture

This section contains a high-level overview of the design of NameFuzz tool that fulfills the requirements of Section 3. It is important to note that there are deviations from the original design, these deviations are mentioned during this chapter but the rationale of these decisions are provided at the end of this chapter.

5.1 Overall System Architecture

The system can be divided into 3 main subsystems; Metadata Extraction, Test Suite Generation and finally Language Processor Testing.

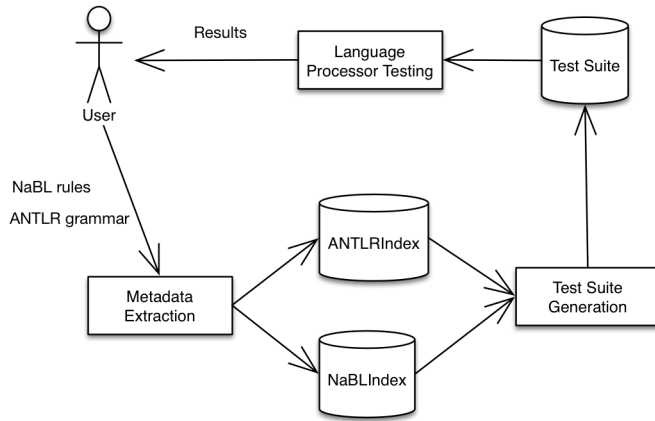


Figure 3: System Overview

The Metadata Extraction sub-system is responsible of extracting the CFG and name binding rules from files provided by the user, this information is then stored in the appropriate data structures.

The Test Suite Generation sub-system is responsible for using the metadata acquired through the Metadata Extraction to generate a test suite of both

positive and negative test cases. The test suite is then saved to disk so that it may be used again later. The Language Processor Testing sub-system uses the generate test suite in order to test the language processor, the results of the testing are then displayed to the user.

5.2 Metadata Extraction

The Metadata Extraction sub-system is responsible of extracting the CFG and name binding rules from files provided by the user to then be used by the Test Suite Generation sub-system to generate a test suite of both positive and negative test cases. The metadata is provided in two separate files, one contains the ANTLR grammar (.g4) and the other contains the NaBL rules (.nab).

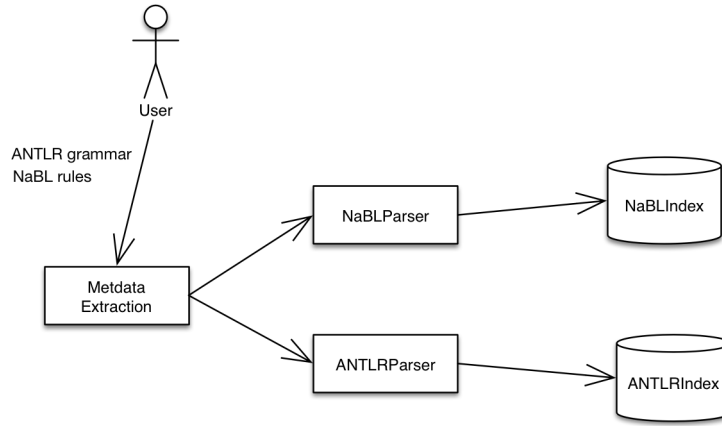


Figure 4: Metadata Extraction System

The ANTLR grammar contains the CFG necessary to generate syntactically correct test cases. The grammar is parsed in and all relevant metadata is extracted and stored in the ANTLRIndex. The ANTLRIndex is a data structure that is able to store and retrieve relevant information about the grammar for the generation process.

The NaBL rules contain all the semantic information required for the gen-

eration process. The rules are parsed and the extracted metadata is stored in the NaBLIndex. The NaBLIndex is a datastructure that allows for the storage and retrieval of the the metadata extracted about the NaBL rules. The NaBLIndex is not to be confused with the tool of the same name in the Spoofox Workbench [7]. In order to be compatible with ANTLR and NameFuzz this NaBLIndex is created to fulfill the same requirements but to interface with these different tools.

5.3 Test Suite Generation

The Test Suite Generation sub-system uses the metadata extracted by the Metadata Extraction subsystem in order to generate the test suite to be used by the Language Processor Testing subsystem.

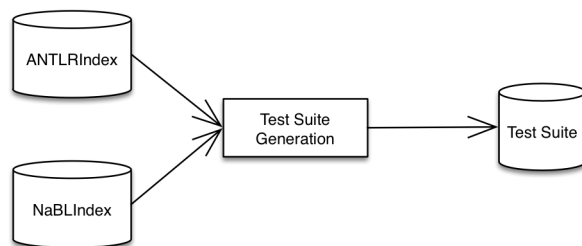


Figure 5: Test Suite Generation System

This sub-system retrieves information about the syntactic structure of the test sentences from the ANTLRIndex and retrieves the information about the semantics of the language from the NaBLIndex. As each test sentence is generated it is then saved to the current working directory.

5.4 Language Processor Testing

The Language Processor Testing sub-system uses the generated test suite to test the language processor.

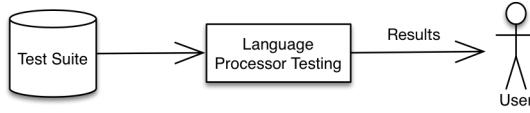


Figure 6: Language Processor Testing System

During this process the language processor is tested with each test case and the results of the test case are printed out for the user as each test completes. For test cases that passed a simple “PASSED” message is shown indicating that the test case was successful. For test cases that were not successful a message detailing the test case number, the line of the error as well as the expected result are shown to the user.

5.5 External Tool Integration

The ANTLR tool [2] is used to generate the two parsers for the Metadata extraction subsystem. One parser for the ANTLR grammar, and the other for the NaBL rules.

5.6 Class Structure

The class structure is represented in the diagrams below. A line with no annotations is used to represent a dependency between classes. A black diamond is used to represent a part-of relationship, showing what attributes the object will contain. A white diamond is used to represent a has-a relationship, showing a link between objects and classes within the diagrams. Finally the white arrow represents is-a relationships, demonstrating inheritance between classes.

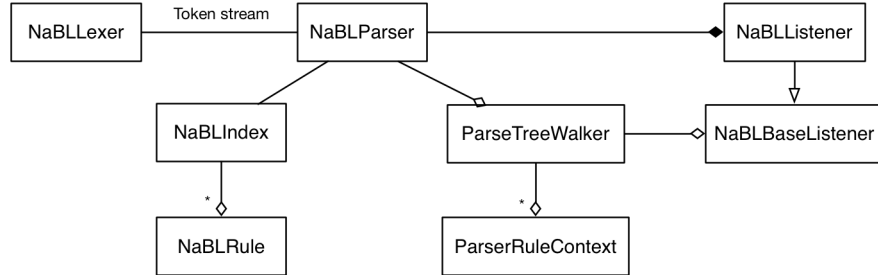


Figure 7: NaBL parser class relations

The `NaBLLexer` tokenises the provided grammar (Requirement 3.2.1.2). The `NaBLParser` reads in the token stream and creates a parse tree, stored in the `ParserRuleContext`. The `NaBLBaseListener` inherits from the `NaBLListener` class which is used by the `ParseTreeWalker` to walk the parse tree to populate the `NaBLIndex`. The information about each name binding rule is stored in `NaBLRule` objects.

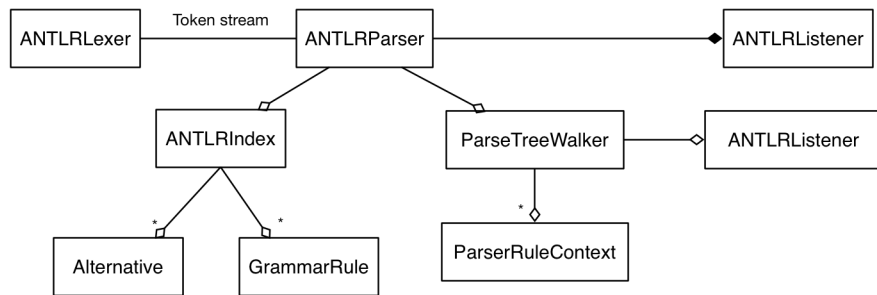


Figure 8: ANTLRParser class relationships

The relations here are much the same as above, the `ANTLRLexer` tokenises the provided grammar (Requirement 3.2.1.1). The `ANTLRParser` reads in the token stream and creates a parse tree, stored in the `ParserRuleContext`.

The `ANTLRBaseListener` inherits from the `ANTLRListener` class which is used by the `ParseTreeWalker` to walk the parse tree to populate the `ANTLRIndex`. The information is then used to construct a graph representation of the language with the `GrammarRule` and `Alternative` objects.

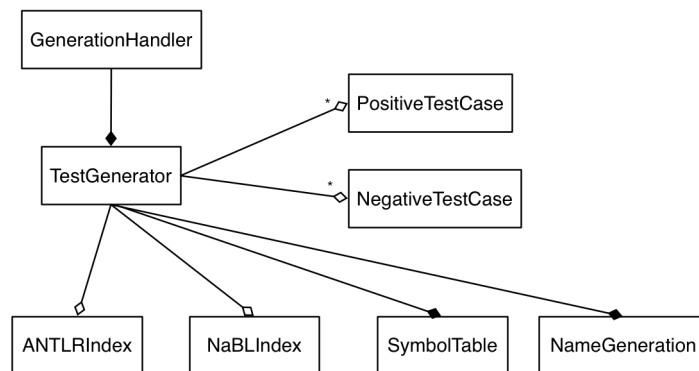


Figure 9: GenerationHandler class relationships

The `GenerationHandler` uses the `ANTLRIndex` and `NaBLIndex` provided by `ANTLRParser` and `NaBLParser` respectively. It then relies on the `SymbolTable` and `NameGeneration` classes to assist in generating the test suite which is then stored in the `PositiveTestCase` and `NegativeTestCase` objects.

The `TestHandler` class controls the testing of the language processor using the test suite of positive test cases and negative test cases stored in the `PositiveTestCase` and `NegativeTestCase` respectively. The `LanguageProcessorHandler` class abstracts the implementation details away from the `TestHandler` class.

The `FuzzHandler` class oversees the whole process. It accepts commands from the user through the command line to control the different subsystems, as well as pass in the necessary parameters to the relevant subsystem.

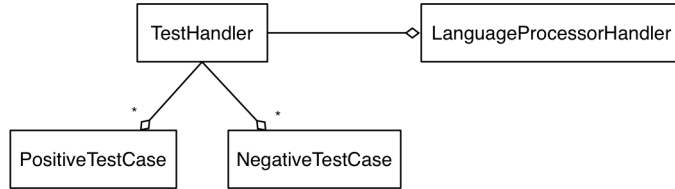


Figure 10: TestHandler class relationships

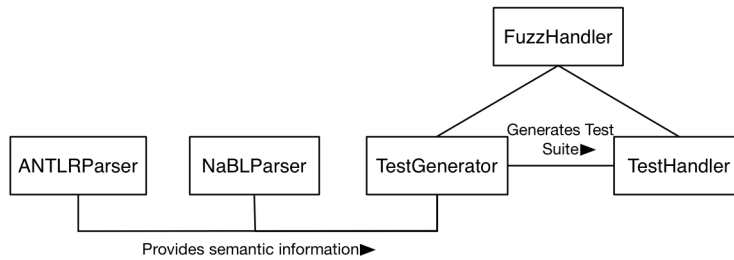


Figure 11: FuzzHandler class relationships

5.7 Class Diagram

The main classes in the system are NaBLParser, NaBLIndex, ANTLRParser, ANTLRIndex, GenerationHandler, TestGeneration, GenerationHandler, TestHandler, FuzzHandler.

All the main methods as well as the main fields are shown in Figures 12 and 13, as well as all relationships between the classes. The FuzzHandler class is responsible for the user interaction to the system. Through this class the user determines the files from which to generate the test suite as well as test the language processor with the test suite.

5.7.1 Test Generation

The FuzzHandler class is the control class with which the user interacts through the command line. Once the user initiates the test suite generation process the `genTestSuite` method is called, which initializes the GenerationHandler class. The GenerationHandler class is responsible for the test suite generation. The `parseNaBL` and `parseANTLR` methods are used to parse the NaBL rules and ANTLR grammar, respectively, and retrieve the NaBLIndex and ANTLRIndex. The TestGenerator class then begins the generation process. Each generated test case is then either as a PostiveTestCase or NegativeTestCase object. The PostiveTestCase class only stores a Java String representation of the object as well as the rule being exercised, while the NegativeTestCase class stores a Java String representation of the test case, the rule being exercised as well as the line number where the test is exercised. This is so that the system can determine whether or not the language processor identified the correct error.

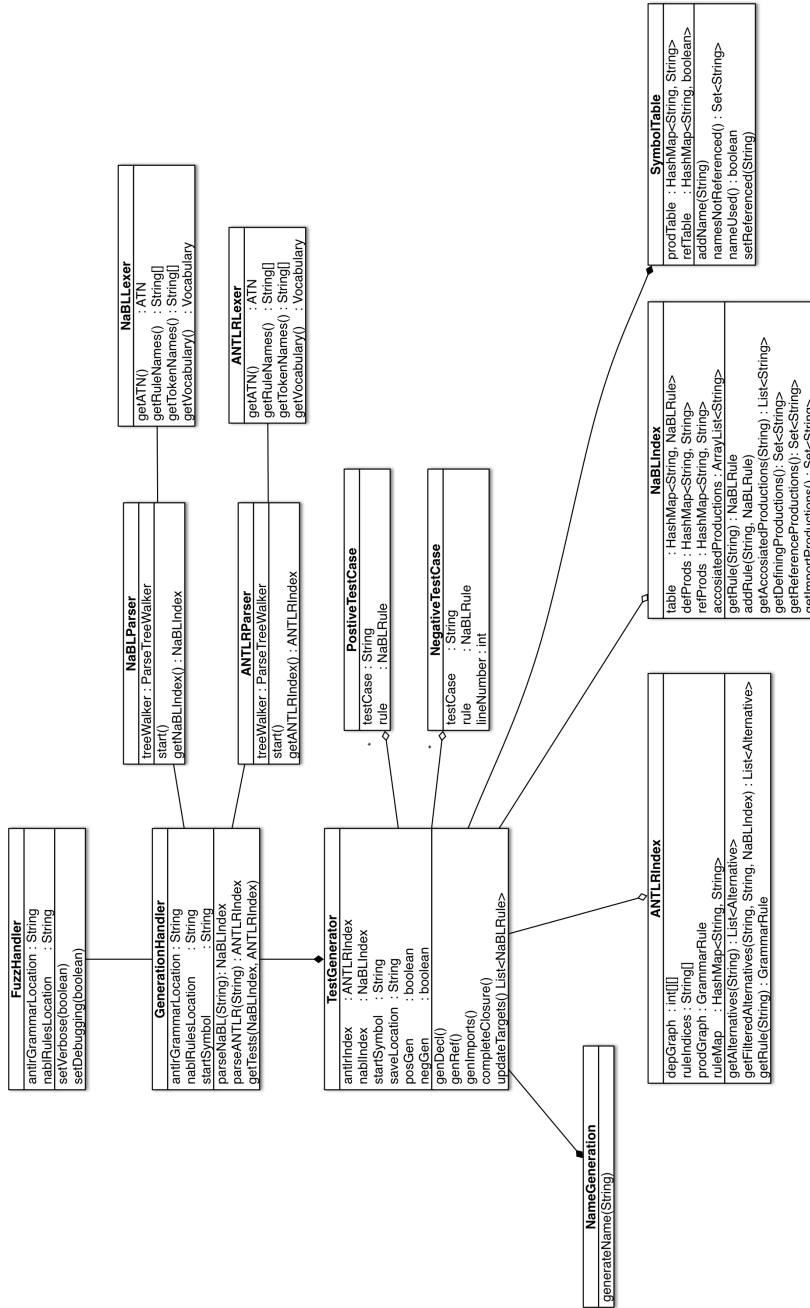


Figure 12: GenerationHandler class diagram

5.7.2 Language Processor Testing

The FuzzHandler is also the class through which the user interacts to control the testing of the language processor. The actual testing is done in the TestHandler class. This is a simple class with only three methods `testPositiveCases`, `testNegativeCases` and `testAllCases`. The user also uses the FuzzHandler class to interact with the testing of the language processor.

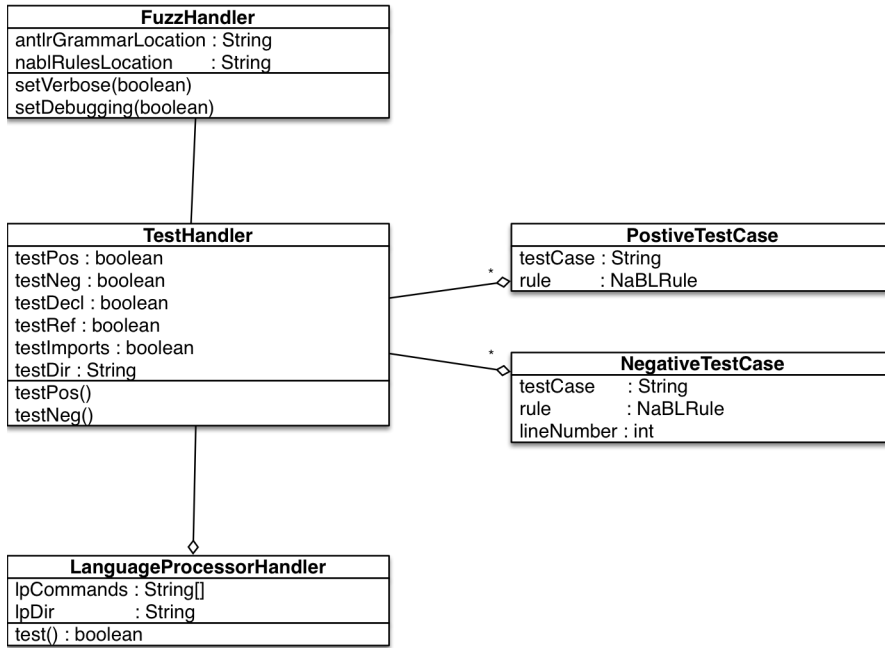


Figure 13: TestHandler Class Diagram

5.8 Deviation from Original Design

There are two significant changes from the original design. Firstly the Syntax Definition Formalism (SDF) and JSGLR [9] tools, that were intended to be used for the grammar and parsing of the grammar, is exchanged for the ANTLR tool. This is mainly due to the difficulty in finding the relevant documentation for JSGLR. ANTLR is both well documented and has a strong community support. Secondly the NaBLIndex [14] with the Java backend

that is part of the Spoofox workbench is no longer used, because it is not compatible with ANTLR. For this reason an ANTLR compatible version is created for the project.

All other changes from the original design are to assist the transitions from the SDF approach to the ANTLR approach. The high level sections of the system remain the same.

6 Implementation Details

The project has been implemented as the NameFuzz tool in the Java 8 programming language [4]. The tool is a .jar [5] file that can be used from the command line or integrated with other projects by simply importing the file. This section demonstrates the use of the tool through the command line.

6.0.1 Generating the Test Suite

The generation of the test suite is done automatically by the NameFuzz tool. The user needs to provide the grammar in the ANTLR format and the NaBL rules in the NaBL format as well as the **gen** parameter. There are several additional commands available to the user to customize the test suite generation. These are listed in the NameFuzz manual in Appendix B.

```
java -jar NameFuzz.jar -gen java.g4 namerules.nab
```

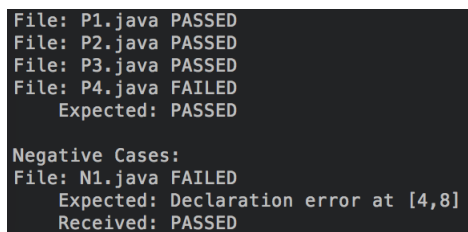
The generated files are then saved to the current directory by default, there are two reasons for this. Firstly due to the potential number of potential number of files being very large the size of the test suite can easily exceed the standard memory limitations of the Java Virtual Machine. And secondly should the want to run multiple tests, they need not regenerate the test suite each time.

6.0.2 Testing the Language Processor

The testing of the language processor is also an automated task. The user is required to provide the command required to run the language processor to be tested as well as the **test** parameter. The tool will then proceed to read

the test cases as well as test the language processor with each one.

```
java -jar NameFuzz.jar -test 'javac'
```



```
File: P1.java PASSED
File: P2.java PASSED
File: P3.java PASSED
File: P4.java FAILED
    Expected: PASSED

Negative Cases:
File: N1.java FAILED
    Expected: Declaration error at [4,8]
    Received: PASSED
```

Figure 14: Example output

7 Project Evaluation

The evaluation of the project is broken down into two phases. The testing of individual functions and verifying the generated test suite.

7.1 Testing

Most of the testing of the individual functions of the project were carried out using unit tests. Other, more difficult to test, function's output was compared to worked out simple examples such as the function that generates the graph representation of the grammar. A debugging mode was also added to the tool so that the flow of the program could be verified with greater ease.

7.2 Test Suite Verification

Verifying that the generated test suites are correct is more complicated, as a large number of test sentences are generated and to some extent each requires individual attention to give insight into the correctness of the generation algorithm. There are two separate aspects of the test suite to verify; the syntactic correctness and the semantic correctness.

In order to verify that the generated test suite is syntactically correct a small

subset of a language, such as NanoPascal, were used to generate a test suite. The ANTLR tool was then used to generate a parser for the same grammar. While parsing the generated test suite 100% of the sentences were successfully parsed by the generated parser. The full Pascal and C# grammars [1] were used to generate a test suite and following the same procedure showed that all generated sentences were accepted by the parser without error.

The semantic correctness of the test suite was verified by generating a test suite from the Pascal and C# grammars with the accompanying name binding rules in Appendices A and C, and verified that the corresponding production compilers [8][3] accepted all the positive test cases and rejected all the negative test cases for the violating the prioritized name binding rule in each of the test sentences. The main weakness of this project's approach to the name binding semantics comes to light here as only a small number of the sentences in the test suite do this successfully. There are two main reasons for this:

Firstly not taking type checking into account results in each combination of possible types being generated in separate sentences. For example consider the following test sentences generated from the NanoPascal language:

```
program A ;  
    var a : integer ;  
    a := 0 ;  
.  
  
program B ;  
    var a : integer ;  
    a := true ;  
.  
  
program C ;  
    var a : boolean ;  
    a := true ;  
.  
  
program D ;  
    var a : boolean ;
```

```
a := 0 ;  
.
```

Only test sentences A and C are type correct the rest are not. This results in an excessive number of test sentences generated where only a few are type correct.

Secondly, related to type checking but more complex is the lack of constraint solver for taking the evaluation of expressions. This results in the expressions containing both incompatible types as well as evaluating to the incorrect type in a location where the semantics of a language forbid it. For example if the expression that is the condition of an if statement where to evaluate to a integer and not a boolean value. Similarly to the first problem is this results in many incorrect test sentences being generated and only a few correct sentences.

Due to the combinatorial nature of the generation algorithm all possible combinations of cases are checked which results in all possible type correct test sentences being generated, but many more type incorrect sentences being generated too. The solution in the implementation is to ignore all test cases that are rejected by the language processor on account of a type error, while this acts as a filter for the sentences it does not fix the time and memory inefficiencies of generating all the incorrect test cases. A permanent clean solution for these two problems would be to incorporate type checking into the process and to extend the NaBL grammar in such a way that constraint checking becomes possible.

In conclusion we can see that while the NameFuzz tool is effective in generating a test suite to test for semantic correctness the is still a long way to go to limit the number of incorrect test sentences in the suite.

8 Related Work

Other work has been done to incorporate language semantics into language fuzzing. Harm and Lammel [12] investigated the use of both syntactic and semantic aspects of attribute grammars, this introduced a “two dimensional”

coverage concept that used a combination of rule based coverage criterion and domain coverage of the attributes. The algorithm they developed generates structurally sound test suites that achieves the coverage specified and systematically discards all the test data that are not semantically correct. Godefroid et al. [11] developed an algorithm that generates semantically correct test suites using symbolic checking and a custom grammar-based constraint solver. Dewey et al. [10] went on to develop a method using Constrained Logic Programming (CPL) to generate input data that was able to exercise structural and semantic aspects of a language, improving on the stochastic approaches. Kermati et al. [13] used constrained input grammars to generate semantically valid test inputs. Using a modified BNF style to describe the structured part of the input specification, and a variation of attribute grammars to define constraints among the structural parts of the input.

While much other work has been done in the incorporating semantics into the language fuzzing, none yet have focused specifically on name binding semantics. This project outlines an approach to solving this problem.

9 Conclusion

The project successfully used name binding rules in the NaBL language, a CFG in the ANTLR language in order to generate a test suite to test a language processor for correctness.

Evaluating the test suite revealed that a large number of sentences in the test suite are not type correct, however the combinatorial nature of the algorithm ensures that all possible type correct sentences exist within the test suite. This means that the shortcoming of the approach is that an excessive number of test sentences are created in the process and this can be both time and space consuming. The incorrect test sentences are due to the implementation not taking type correctness into account, as well as the limitations of the NaBL language .

References

- [1] Antlr grammars: <https://github.com/antlr/grammars-v4>.
- [2] Antlr: <http://wwwantlr.org/>.
- [3] Csharp : <https://msdn.microsoft.com/en-us/library/>.
- [4] Java 8 : <http://docs.oracle.com/javase/8/docs/api/>.
- [5] Java jar file: <http://docs.oracle.com/javase/8/docs/technotes/guides/jar/>.
- [6] Java language: java.com.
- [7] Nabl name binding language <http://metaborg.org/nabl/>.
- [8] Pascal : <http://www.pascal-programming.info/index.php>.
- [9] Syntax definition formalism (sdf) <http://www.meta-environment.org/meta-environment/sdf>.
- [10] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 725–730, 2014.
- [11] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. pages 206–215. ACM, ACM, New York, NY, 2008.
- [12] J. Harm and R. Lammel. Two-dimensional approximation coverage. *Informatica*, 24(3):355–369, 2000.
- [13] Hossein Keramati and Seyed-Hassasn Mirian-Hosseiniabadi. Generating semantically valid test inputs using constrained input grammars. *Information and Software Technology*, 57:204–216, 2015.
- [14] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

- [15] Dexter C Kozen. *Automata and Computability*. Springer, Ithaca, NY, USA, 1997.
- [16] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *Testing of Communicating Systems*, volume 3964, pages 19–38. Springer Berlin Heidelberg, 2006.
- [17] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.
- [18] Michael Sipser. *Introduction to the Theory of Computation*. Thomson, Boston, Massachusetts, USA, 2006.

10 Appendix A

Nano Pascal ANTLR Grammar

```
grammar NanoPascal;

start : program
;

program : 'program' ID ';' block* '.'
;

block : varDecl
      | statement
;

varDecl : 'var' ID (varDeclPart)* ':' type ';'
;
varDeclPart : ',' ID
;

type : 'boolean'
      | 'integer'
;

statement : 'begin' (statementPart )+ 'end'
;

statementPart : simpleStatement
              | structuredStatement
              | block
;

simpleStatement : assignmentStatement
;
assignmentStatement : variable ':= ' expression ';' ;
```

```

;

structuredStatement : ifStatement
;

ifStatement : 'if' expression 'then' statement 'else' statement
| 'if' expression 'then' statement
;

expression : simpleExpression
;
simpleExpression : term (addOp term)*
;
term : factor (mulOp factor)*
;
factor : variable
| INT
| BOOLEAN
;

sign : '+'
| '-'
;

addOp : '+'
| '-'
;

mulOp : '*'
;

variable : ID
;

INT : [0-9]+ ;

BOOLEAN : 'true' | 'false' ;

```

ID : [a-zA-Z][a-zA-Z0-9]* ;

WS : ' ' ;

Nano Pascal NaBL Rules

module NanoPascal

imports

namespaces

Var

Program

binding rules

program(x) :

explicitly defines unique Program x
scopes block, Var

varDecl(x) :

explicitly defines unique Var x
in subsequent scope

variable(x) :

refers to Var x

block(Statement) :

scopes Var

11 Appendix B

NameFuzz Manual

NameFuzz is a simple tool to use. To generate a test suite from the command line the user enters:

```
java -jar NameFuzz.jar -gen <- optional parameters> <grammar> <nabl rules>
```

And to test the language processor:

```
java -jar NameFuzz.jar -test <- optional parameters> <command>
```

The optional parameters list:

- p Positive tests only
- n Negative tests only
- d Declaration rules only
- r Reference rules only
- s scope rules only
- i imports rules only

These can be coupled together to form a chain of commands. For example `-pdr` will only generate or test positive declaration and reference name binding rules.

12 Appendix C

Below are the name binding rules for the C# programming language.

module names

```
version 0.1
status under development

author Gabriel Konat, TU Delft
author Guido Wachsmuth (g.h.wachsmuth@tudelft.nl), TU Delft
```

imports

```
nbl/-
  include/CSharp
  desugar
  types
```

namespaces

```
Namespace

Class

Function
Field

Variable
```

properties

```
parameter-types of Function: List(Type)
```

binding rules // Namespaces

```
Namespace(x, _) :
  defines non-unique Namespace x
```



```

    scopes Namespace, Class

UsingPart(x) :
    imports Class from Namespace x

UsingPart(u, x) :
    refers to Namespace x in Namespace y
    where u refers to Namespace y

binding rules // Classes

Class(x, _) :
    defines unique Class x of type Type(x)
    scopes Field, Function

Class(x, y, _) :
    defines unique Class x of type Type(x)
    scopes Field, Function
    imports Field, imported Field, Function, imported Function from Class y

Interface(x, _) :
    defines unique Class x of type Type(x)
    scopes Function

Interface(x, y, _) :
    defines unique Class x of type Type(x)
    scopes Function
    imports Function, imported Function from Class y

PartialClass(x, _) :
    defines non-unique Class x of type Type(x)
    scopes Field, Function

PartialClass(x, y, _) :
    defines non-unique Class x of type Type(x)
    scopes Field, Function
    imports Field, imported Field, Function, imported Function from Class y

```

TypePart(c) :
 refers to Namespace c
 otherwise refers to Class c

binding rules // Fields

FieldDef(t, x) :
 defines Field x of type t

FieldAccess(exp, f) :
 refers to Field f in Class t
 where exp has type Type(t)

binding rules // Functions

FunDef(t, x, p*) :
 defines Function x
 of type t
 of parameter-types pt*
 where
 p* has type pt*

FunDef(t, x, p*, _) :
 defines Function x
 of type t
 of parameter-types pt*
 where
 p* has type pt*
 scopes Variable

FunCall(exp, f, args) :
 refers to Function f of parameter-types arg-types in Class e
 where exp has type Type(e)
 where args has type arg-types

FunCall(x, args) :
 refers to Function x of parameter-types arg-types
 where args has type arg-types

```

Param(t, x) :
    defines Variable x of type t

binding rules // Variables

VarDef(t, x, _) :
    defines Variable x of type t in subsequent scope

VarDef(t, x) :
    defines Variable x of type t in subsequent scope

InferredVarDef(x, e):
defines Variable x of type t in subsequent scope
where e has type t

VarRef(x) :
    refers to Variable x otherwise
    refers to Field x

binding rules // Control flow

For(t, x, init, cond, stmt, body) :
    defines Variable x of type t in cond, stmt, body

ForEach(t, x, e, body) :
    defines Variable x of type t in body

Block(stmt*) :
    scopes Variable

```

13 Glossary

13.1 Definitions, Acronyms and Abbreviations

Alphabet A set of recognised characters.

Compiler A language processor that generates executable code.

Context Free Grammar (CFG) A method of specifying the words of formal language.

Declaration The introduction of an entity with a given name ‘id’ into a scope.

Entity Usually refers to variables, constants, functions and classes.

Identifier The ‘id’ or name assigned to an entity.

Import/export visibility modifiers Keyword in a declaration that determines whether or not the binding is visible/ accessible to other namespaces.

Language Fuzzing A set of techniques that use a CFG to generate test cases for a language processor.

Language Processor A program that performs the tasks required to process a language.

Namespace A namespace is an abstract container created to hold a logical grouping of unique identifiers.

Scope The sections in a program where the name bindings of an entity are valid.

Syntax Definition Formalism A metasyntax used to describe CFGs.

Sentence A structurally and syntactically valid program constructed from a CFG.

Test Suite A set of test cases, where each test case aims at testing a different part of the system.

Test Case A sentence that demonstrates a rule that the language processor must either accept or reject.

Visibility Refers to whether or not the binding of an entity is valid in the a given scope.