# University of Alberta

## Test Case Generation using Symbolic Grammars and Quasi-Random Sequences

by

Alejandro Felix

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering


Spring 2011
Edmonton, Alberta

## Examining Committee

- James Miller, Electrical & Computer Engineering
- Bruce F. Cockburn, Electrical & Computer Engineering
- H. James Hoover, Computing Science

TO

MY

DEAREST

MOTHER

AND

GRANDMOTHER

# Abstract

This work presents a new test case generation methodology, which has a high degree of automation (cost reduction); while providing increased “power” in terms of defect detection (benefits increase). Our solution is a variation of model-based testing, which takes advantage of symbolic grammars (a context-free grammar where terminals are replaced by regular expressions that represent their solution space) and quasi-random sequences to generate test cases.

Previous test case generation techniques are enhanced with adaptive random testing to maximize input space coverage; and selective and directed sentence generation techniques to optimize sentence generation.

Our solution was tested by generating 200 firewall policies containing up to 20 000 rules from a generic firewall grammar. Our results show how our system generates test cases with superior coverage of the input space, increasing the probability of defect detection while reducing considerably the needed number the test cases compared with other previously used approaches.

# Acknowledgments

I'd like to especially thank the University of Alberta – CONACYT agreement who's funding and support made this work possible.

I'll also like to thank my supervisor, Dr. James Miller, for his patience, guidance and assistance during my master studies and research.

Last but not least, a special recognition to my friends. Thank you guys!

# Table of Contents

# I. List of Tables

# II. List of Figures

# 1 Introduction

Fuzzing is a software testing technique used to find bugs through the external or exposed interfaces of systems complementing traditional testing with randomness, protocol knowledge and attack heuristics.

In recent time, fuzzing has become a widely accepted testing approach due to its ease of automation and obtained results (compared with traditional software testing, fuzz testing improves the obtained results). Fuzzing is used by the software industry to improve the quality of their software, by vulnerability analysts to detect software and by third parties (such as hackers) to find and exploit software.

For testing applications that require highly-structured inputs, traditional testing tools and fuzzing tools can be very limited. Examples of this kind of system can be found in any system that uses compilers or interpreters to obtain its parameters values from an external input. These systems process the information in stages such as lexical, pre-processing and syntactical. Each stage usually depends on the successful termination of the previous stage. Due to its nature, most of the testing techniques generated inputs can rarely surpass the first stages.

Systems that require highly structured inputs usually have defined language grammars that control the input syntax. These grammars define the valid input space for all the possible inputs of the system under test. As known, the input surface of any system tends to infinite, so heuristics should be used in order to constraint it and make testing feasible.

Previous work employs black box and white box testing approaches. Black box consists in feeding random inputs to the system under test so they can reveal flaws in the system under test. As mentioned, black box testing approaches are very limited due to the highly-structured inputs where any malformation prevents information from reaching deeper stages. White box fuzzing approaches combine black box testing with dynamic test generation. Starting with a well-formed input, it executes the test and keeps track of the conditionals used. Then those conditionals are negated so different control paths are tested. For large systems, this approach is unfeasible as the possible control paths and constraints can grow exponentially.

To solve these limitations, some black box fuzzing approaches use grammars to reduce the solution space of the generated well-formed inputs and test heuristics. One of those approaches is grammar-based whitebox fuzzing, which is an enhancement of whitebox fuzzing with grammar-based specifications of valid inputs and high-level symbolic constraints expressed in terms of symbolic grammar tokens. This approach guarantees that all of its generated outputs will be parsable system inputs. Nonetheless, several limitations were not solved by this approach as the grammar and the grammar token constraints only reduce the testing input space in terms of generating parsable system inputs.

Parsable system inputs present many of the problems non-parsable inputs present. For example, the input space is reduce, but the total permutations is still of exponential order even using the token constraints and symbolic grammars. Another detected problem is that parsable system inputs have no control over the tokenizing selection algorithm used in fully defined testing inputs. Both limitations make the algorithm practically unusable for most testing strategies that require precise control over the fully defined testing inputs.

To address the detected limitations this work presents an extension to grammar-based whitebox fuzzing where controlling algorithms are added to reduce even more the input space. With these expansions the system will not only explore deeper program paths and will avoid the generation of non-parsable system inputs, but will also increase the solution space covered by each generated test.

# 2 Related Work

Much progress has been accomplished regarding grammar-based testing. Starting from the early 70s, this testing strategy has evolved from its initial formalization to its implementation in testing frameworks.

This section is organized as follows: first an introduction discusses model-based testing; then we review of previous blackbox testing techniques; following this, whitebox approaches will be analyzed to introduce whitebox fuzzing and its implementations; finally we review the latest technologies employed will that define today's state-of-the-art for grammar-based approaches will be presented.

## 2.1 Model-based Testing

| | | |
|---|---|---|
| Model | Subject | **Environment** |
| | | System Under Test (SUT) |
| | Redundancy | Shared test & dev model |
| | | **Separate test model** |
| | Characteristics | Deterministic/**Non-Det.** |
| | | Timed /**Untimed** |
| | | **Discrete**/Hybrid/Continuous |
| | Paradigm | Pre-Post |
| | | Transition-Based |
| | | History-Based |
| | | **Functional** |
| | | Operational |
| Test Generation | Test Selection Criteria | **Structural Model Coverage** |
| | | Data Coverage |
| | | Requirements Coverage |
| | | Test Case Specifications |
| | | Random & Stochastic |
| | | Fault - Based |
| | Technology | **Manual** |
| | | **Random Generation** |
| | | Graph Search |
| | | Model-checking |
| | | **Symbolic execution** |
| | | Theorem proving |
| Test Execution | On/Offline | **Online/Offline** |

**Table 2.1 Taxonomy proposed by Utting and Legeard for model-based testing.**

Model-based testing is *the automatic derivation of concrete test cases from abstract formal models* (1). Utting and Legeard propose a model with seven dimensions (properties) that give a full description of any type of system that uses a model for generating its outputs. The taxonomy is shown in Table 2.1.

According to this taxonomy, grammar-based testing is a type of model-based testing whose dimensions are shown in Table 2.2.

| Dimension | Characteristic | Description |
|---|---|---|
| Subject | Environment | The model of the environment is used to restrict the possible inputs of the model. The model is a context-free grammar that gives a full protocol's description of the input space of the system under test |
| Redundancy | Separate test model | Redundancy is guaranteed as the test tool is built separately from the SUT and goes through its own development lifecycle. |
| Characteristics | Non-Deterministic / Deterministic | Both characteristics are used depending on the approach chosen.. |
| | Untimed | All approaches act over the system under test's input/output so real-time execution is not considered. |
| | Discrete | The testing system depends on the occurrence of asynchronous discrete events over time |
| Paradigm | Functional | The SUT is described as a collection of mathematical functions, in this case the context-free grammar. |
| Test Selection Criteria | Structural Model Coverage | The structure of the model (grammar) is exploited to generate the test cases. |
| Technology | Random Generation | Several systems use constrained random generation |
| | Model-checking | With dynamic test generation, model checking is used for creating new testing inputs for the SUT. |
| | Symbolic execution | White fuzzing approaches use symbolic execution as their main engine for constraining the input space. |
| On/Offline | Online/Offline | Until dynamic test generation was introduced, all systems were executed offline. |

**Table 2.2 Dimensions for model-based testing proposed by Utting and Legeard**

Grammar-based testing focuses on using the model (grammar) to produce test inputs to verify the conformance of the system under test to

the model. Context-free Grammars can produce syntactically correct predicates, but fail to produce semantically correct predicates – a significant limitation that has been studied, as the following sections demonstrate.

## 2.2 Random Testing

The first attempts to employ grammar-based testing date back to the 1970's. Grammars found in language definitions are recipes to generate strings that will be considered to be well-formed and part of the language accepted by the compiler. To satisfy this constraint the following two conditions need to be satisfied:

- All well-formed programs can be written down following the model.
- Only well-formed programs can be written down following the model.

BNF notation satisfies the first one but fails to satisfy the second one. This is comprehensible as the second condition can only be satisfied with context-sensitive grammars.

To satisfy the second condition, Hanford (2) introduces the concept of *dynamic grammars,* which is an extended grammar that considers syntax declaration correctness. Hanford proposes that an approximation to a context sensitive grammar can be achieved by adding rules that produce predicates of a declarative programming language.

Purdom (3) focussed on the problem of sentence generation, presenting an algorithm that produces a set of short sentences from a context-free grammar *such that each production of the grammar is used at least once*. Purdom overlooks semantical correctness but focuses on how to rapidly generate a short test set of sentences that obey the syntax of a context-free grammar.

Both attempts concentrated on the automated generation of test cases, but no further control was proposed. This led to problems controlling the input space which wasn't reduced and includes all the sentences defined by the grammar (that are part of the language definition). Random generation was employed to solve this limitation. Random testing consists of randomly selecting inputs from the input space. This approach is not effective as the input space is too big and the chances of hitting "error crystals" (4) is very small. So a more reliable approach is needed to constrain the produced sentences.

## 2.3 Exhaustive Generation

Exhaustive generation from the grammar was tried by Duncan and Hutchison (5); they proposed the usage of *attributed context-free grammar* as the basic mechanism for grammar production heuristics (constraints over the solution space). Their first step in the algorithm is to change the CFG grammar to an attributed CFG grammar with attributes that will be solved by the system. Those attributes will give control over the production choice of rules, attributes and productions. Attributes act as guides that tell the parser which rules employ and how to employ them.

Take for example the following set of rules:

```
A = B{1} C{0,1} D{m,n} ;

B= “b”;
C= “c”;
D= “d”;
```

In this example each terminal symbol is enclosed between quotes and non-terminal is followed by user-defined attributes annotated between curly brackets. These attributes control the non-terminal boundary values (lower and upper); if m=1 and n=4 an example of the language defined by this grammar would be “bcd” and “bcddd”. The system proposes a set of attributes for controlling the use of certain rules. This gives the system certain context awareness in order to generate semantically correct sentences:

```
Asd = [? terminal = “else”] “else” stmt;

stmt = “stmt” ;
```

In this rule, “[? ... ]” indicates that the variable terminal (set during the system execution) will be evaluated. If this block is evaluated as true, then the rule will be solved during execution time.

This system leaves all the choices to the tester on how to combine the attributes, so the effectiveness of the tool is proportional to the ability of the tester to refine its results and, as no dynamic strategy is suggested, the inspection of the results is done manually. Another limitation of this approach is that each grammar has to be modified to comply with the system syntax; this implies further human interaction with the system.

## 2.4 Constrained Exhaustive Enumeration

Even when the grammar has been restrained the input space is usually too large to attempt exhaustive (test case) generation; to address this

problem, the concept of "constrained exhaustive enumeration" was introduced, which consists of adding constraints to the grammar to reduce the solution space.

Maurer (6) proposes building a grammar in a bottom-up fashion, keeping the solution space as controlled as possible and defining a different test grammar for each different test case. Attributes are added to each grammar to give limited semantic control over the generated outputs.

The probability for selecting a particular alternative of a rule with different choices is annotated before choice. For example, the rule "vowel" can be extended with probabilities, giving the terminal "u" a 33% chance of being chosen compared to the terminal "a" that only has 7% probability.

```
extended_vowel = %15{vowel}

vowel = 1:"a" | 2:"e" | 3:"i" | 4:"o" | 5:"u"
```

Action routines are also another enhancement proposed by Maurer. Action routines allow the grammar to keep track of values previously selected. These values can be used later when a previously selected value is wished to be used.

```
extended_vowel = %{vowel.letter} %{letter}

letter: variable;

vowel = "a" | "e" | "i" | "o" | "u" ;
```

The main drawback of this approach is the buttom-up grammar generation that prevents scalability as the grammars become complex. Another limitation is that when the grammar is defined the system works *generating all possibilities for some productions while allowing other choices to be made at random*.

McKeeman (7) employs the concept of differential testing to address the *oracle problem* employing stochastic grammars. Stochastic grammars are grammars where each of the rules is associated with a probability. Whenever a nonterminal is to be expanded, a random number is generated and compared with the fixed rule probabilities to direct the expansion choice. The author proposes 7 levels of quality assurance for testing a C compiler:

1. Sequence of ASCII characters
2. Sequence of words, separators and white spaces
3. Syntactically correct C program
4. Type-correct C program

5. Statically-conforming C program
6. Dynamically-conforming C program
7. Model-conforming C Program

Stochastic grammars satisfy only levels 1 and 2. For levels 3 – 5, adding more specific rules and probabilities is suggested to achieve the semantic correctness required (7). For levels 6 and 7, a post-generation analysis with a different tool that selects only well-formed inputs is proposed. As only few test cases reach level 7, it is suggested to employ exhaustive generation to generate enough test cases (to raise the probability for having well-formed inputs).

Lämmel (8) presents *controllable combinatorial coverage* where parameters are set to control the grammar such as:

- Depth - limit the number of rules employed to produce the testing sentences.
- Recursion - limit the nested selection of recursive rules.
- Balance - limit the depth variation for argument terms when depth and recursion are not enough to satisfy the desire depth over recursive rules.
- Dependence – defines the syntactic (context-free) or semantic (context-sensitive) options for controlling combinations of arguments when forming new terms.
- Construction – conditions and computations that semantically constrain test-data generation into test cases.

These approaches focus their efforts on describing valid inputs and generating test cases that satisfy the grammar but fail to solve the exhaustive generation of sentences once the grammar has been constrained to a certain depth.

## 2.5 Symbolic Execution

Symbolic execution is a program analysis technique that, instead of supplying inputs to a program, supplies symbols representing arbitrary values that allows the exploration of program executions paths. Symbolic execution is useful in other forms of program analysis such as test input generation (9), program optimization (10), and program debugging (11).

Clarke (9) suggests that the system should be represented as a control flow graph. Then a random input is selected and its control path is traced. The relationships that affect the program flow are determined as a set of constraints in terms of the program's input variables. When a path is symbolically executed, expressions denoting the evolution of the variables are generated. These values are collected and used to constrain the next

set of input test values; this is repeated until all the control paths have been exercised.

For example take the following Java code (its control graph is shown in Figure 2.1.):

```
1    public int myFunction (int var1, int var2){
2
3    var1 = var1 + 1;
4
5    if (var1 > var2)
6        var1=10 - var2;
7    else
8        var1 = var1 / var2;
9
10   return var1;
11   }
12
```

Given random initial values (for example var1 = 10 and var 2 = 0), the symbolic execution records the values of the variables in lines 3, 5, 6 and 10. The final result (line 10) would be expressed as:

```
var1 = 10 - var1 - 1
```

Giving the following constraint for the "if" statement found in line 5:

```
var1 +1 > var2
```

Whenever a conditional transfer of control is executed, one or more constraints representing the branch form of the chosen conditional statement are generated. Each constraint should be sent to an inequality solver to check if it can be solved given previously generated constraints. If the constraint can't be satisfied, it would indicate that the path is unfeasible.

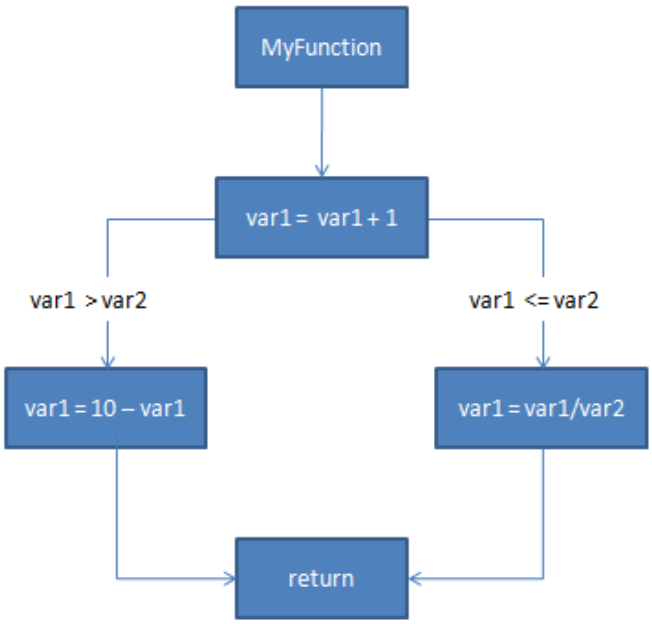


**Figure 2.1 Control graph for example code.**

## 2.6 Concolic Execution

EXE (12), DART (13) and CUTE (14) are examples of automated approaches to systematic testing based on dynamic test generation. All these systems combine constrained random-testing with automatic symbolic and concrete execution (called concolic execution); but are not selective as test inputs are generated randomly and iteratively refined through symbolic execution. This led to uncertainty and forces the program to generate very large set of inputs, so "error crystals" can be found in the input space.

Godefroid and Larlund developed DART: Dynamic Automatic Random Testing (13) one of the first symbolic execution systems to introduce dynamic test generation to fully automate the process of testing. DART uses the following techniques:

- Automated extraction of the interface of a program with its external environment using static source-code techniques.
- Automatic generation of a test driver for this interface that performs random testing.
- Dynamic analysis of the test results and automatic generation of new test inputs to systematically manoeuvre the flow path in an attempt to cover all feasible paths.

Automated extraction refers to the control of the program interfaces with its environment. It detects external variables and external functions (i.e. libraries) and the arguments of a user-specified initial function employed for starting the test execution. DART distinguishes three kinds of C functions:

- Program functions that are defined and used in the program.
- External functions that are controlled by the environment so they are part of the external interface of the program.
- Library functions that are functions not defined in the program but are used by the program.

This step enhances the testing system with external entities awareness, so external calls are treated as black boxes which cannot be instrumented or analyzed. DART is capable of evaluating these external functions in order to create more precise constraints when needed.

Automatic generation is responsible for creating dynamic C code (a main function) that initializes the interface variables at random and executes the test. The initialization of variables takes as arguments a memory location.

Dynamic analysis of test results and automatic generation of new test inputs refers to a constraint solver that takes the results of the test (as memory locations and values) and defines concrete values for each parameter in the external interface forcing new control paths to be executed. Only one value is changed at a time starting from the last recorded one (last value in the execution path).

Some limitations of this approach were related to the complexity of dynamic variables, pointers and data structures. In fact this system fails to correctly generate constraints for these data types and suggests that pure random testing should be employed. Due to the nature of path explosion, the tool is only built to aid in the unit testing phases.

DART sets the basis for further work in symbolic execution testing frameworks. For example Sean et al. (14) develop CUTE: "A Concolic Unit Testing Engine for C" which focuses on the problem of providing methods to extract and solve constraints generated by a program where dynamic data structures are employed. They introduce the idea of a logical input map that represents all inputs as a collection of symbolic variables. From this map, the system builds constraints by performing a symbolical execution of the code under test.

CUTE extends DART by adding better pointer manipulation, separating pointer constraints from integer constraints and keeping them simple. Another enhancement is that the system takes into account code preconditions and sanity checks extending unit testing limitations and adding scalability.

Cadar, et al. introduce EXE (12). EXE keeps track of the execution path constraints in the same fashion as previously discussed. Its main difference is that it employs those constraints and a predefined set of "validations" to check for input values that can cause an error.

For each recorded constraint, EXE's constraint solver verifies if the symbolic expression has a value that satisfies the constraints that can cause a null or out-of-bounds memory reference or a division or modulo by zero error. If true, it reports that the evaluated condition occurs, generates a test case, and terminates; if false, it reports that the evaluated condition does not occur and continues execution. If EXE has a set of constraints on those expressions and the constraint engine can solve them, then EXE detects if any concrete value exists on that path that causes the error. If no concrete value exists that causes the error then the branch is considered to be safe.

For example for the following code:

```
1    public int myFunction (int var1, int var2){
2
3    var1 = var1 + 1;
```

```
4
5    if (var1 > var2)
6        var1=10 - var2;
7    else
8        var1 = var1 / var2;
9
10   return var1;
11   }
12
```

From our previous example, the symbolic constraint for the else execution path found was:

```
var1 + 1 <= var2
```

When EXE's constraint solver reaches line 8, it will look for values that satisfy the constraint and can produce one of the mentioned known errors. For this case the concrete values for the next test could be `var1 = -10` (or any negative number) and var2 = 0. Despite the enhancements over previous systems, execution path explosion and dynamic data structures still limit the scalability of the system.

## 2.7 WhiteBox Fuzzing

Godefroid, Levin and Molnar (15) propose a testing framework that uses a *generational search algorithm* and a code-coverage maximizing heuristic. The initial input is chosen at random from a pool of well-formed inputs and is symbolically executed by the program. The approach consists of recording a concrete execution of the program under test based upon supplying a well-formed input. Subsequently, it symbolically evaluates the recorded trace and gathers constraints on the input variables capturing their interaction with the program. For producing new test cases, the authors propose an algorithm called "generational search" which tries to expand all constraints found in the path constraint; for this they associate with each constraint a score that represents the incremental branch coverage. This approach uses exhaustive generation as all constraints are expanded, but prunes the unsolvable constraints and returns a concrete value, whenever symbolic execution is not possible.

The algorithm is implemented in a symbolic execution framework called SAGE which stands for "Scalable, Automated, Guided Execution". The framework has two main features that separate it from previous frameworks—the generational heuristic algorithm (generational search) and a trace-based x86 binary symbolic execution engine.

The generational heuristic algorithm works as follows:

- Starting with an initial input seed and initial path constraint, it will attempt to expand all of the given constraints (opposed

to the usual approaches where the first one (breadth-first) or only the last one (depth-first) are expanded).

- Then, to prevent these child sub-searches from redundantly exploring overlapping parts of the search space, a parameter is used to prune each existing sub-search.
- The result is a set of test cases called "generation" that will be the input seeds for the next symbolic execution.

The expansion of generational constraints is prioritized by using a heuristic that attempts to maximize block coverage. It computes the incremental block coverage obtained by comparing the actual run to all previous generated runs. The actual run is saved and classified in a list according to its score, with the highest scores placed at the head of the list.

The constraint generation differs from previous symbolic executions implementations in two main ways. First, it adopts a machine-code based approach instead of common source-based implementation because:

- Source-based instrumentation must be adapted to support each language, compiler or build tool adding upfront cost.
- Compilation and post processing tools may introduce differences between the source code and the actual machine code.
- Much third-party source code is not available and JIT–compilers are difficult to test with source-based implementations.

Second, instead of an online instrumentation, the framework implements an offline trace-based constraint generation. In online generation, constraints are generated during program execution by statically injecting fixed or dynamic binary code. This approach results in non-repeatable scenarios where, if the constraint solver fails, the environment is unlikely to be reproducible making it hard to debug. Another encountered problem is that some memory allocations are protected by the operating system, making it very hard to replace at runtime. Offline trace-based constraint generation is chosen as it is completely deterministic because it works on the execution trace that captures the result of all nondeterministic events found during the recorded run.

Path explosion and imperfect symbolic execution are still unsolved limitations. For example, path explosion is not solved for large applications, so performing dynamic test generation compositionally (16) is suggested as a work around. This consists in testing functions separately (unit testing), encoding test results as function summaries, then treating them in a deterministic way with the recorded values corresponding to their input preconditions and output post conditions.

Dynamic data structures analysis (pointer manipulations, arithmetic operations, etc.) constraint solving, calls to the operating-system and library functions that are very difficult to solve symbolically remain as unsolved limitations where concrete values are employed in a classic random fashion.

Another approach was suggested by Ganesh, Leek and Martin (17). Their system uses *dynamic taint tracing* to locate regions of well-formed input files that influence the behaviour of the system under test. Once it has detected those regions, it fuzzes them to produce new test inputs. These approaches change only small parts of the well-formed files so the change is usually syntactically correct, allowing it to pass the parsing phases and reaching "deeper" control paths within the system.

This algorithm was implemented in BuzzFuzz (17), which uses directed whitebox fuzzing. Whitebox fuzzing is designed to produce well-formed test inputs that exercise "deep code" in the semantic core of the program under test. It is based on the following four techniques:

- Taint Tracing - the execution is instrumented to record taint information that represents each input value and the input bytes that influence each value that the program computes.
- Attack Point Selection - the system identifies specific vulnerable points (i.e. library and system calls) and allows the user to specify any arbitrary number of attacking points.
- Directed Fuzzing - for each identified vulnerable point, the system computes the set of input bytes that affect the values at that attack point. This technique is enhanced with fixed "extreme" values (very long or null strings, etc.) to stress the system.
- Directed Testing - dynamic execution over new generated values.

The main benefits of this proposal are:

- Preservation of syntactic structure - directed fuzzing targets input bytes that can be changed without violating the legal syntactic structure of original well-formed inputs.
- Targeted values - fuzzed input bytes are designed to have a high concentration of inputs that can reveal errors that may exist in the detected vulnerable points.
- Coordinated changes - directed fuzzing can identify and alter multiple values of the input space that must change together to reveal errors.

The main difference between traditional symbolic execution and this approach is the symbolic information with which both techniques work. Traditional symbolic execution records a logical expression for each variable

that defines all possible concrete values these variables can take for exercising the execution path; the new approach only maintains the set of input bytes that influence the program variables through the execution path.

## 2.8 Symbolic Grammars

Symbolic grammars are context-free grammars where terminal symbols are substituted with regular expressions that represent the entire solution space for that (precise set) of terminal symbols. They combine the advantages of selective enumerative test generation and directed symbolic test generation.

Model-based exhaustive enumeration is guaranteed to provide valid inputs to the program; however it is likely to produce redundant tests cases, which will exercise exactly the same execution path making them identical from the programs perspective. Test generation based on symbolic execution is directed, systematically exercising new paths; even though, symbolic techniques are expensive and limited by the capacity of the symbolic engine. Performance is also a limitation, symbolic exploration is sensitively slower than exhaustive enumeration; this may lessen as symbolic execution produce no redundant test cases (18).

Symbolic grammars enable a sensitive decrease in the number of strings to be enumerated. For each enumerated symbolic string, the symbolic constants give control over the strategy to generate concrete values (deterministically or non-deterministically) to maximize the path coverage of the program.

Majumdar and Gang Xu (18) propose to *combine the selectiveness of specification-guided* test generation with the directedness of concolic generation through the use of symbolic grammars that give more constraints over the input space that enable deeper control path exploration. Their work proposes to exhaustively pre-generate all feasible inputs up to a certain size from the symbolic grammar and then performing dynamic test generation over those pre-generated inputs.

Take for example the following code:

```
1    public void main (char[] argv){
2        public int myFunction () {
3
4            var1= (int) argv[1];
5            var2= (int) argv[2];
6
7            var1 = var1 + 1;
8
9        if (var1 > var2)
10           var1 = 10 - var2;
11       else
12           var1 = var1 / var2;
13
```

```
14        return var1;
15        }
16    }
```

The grammar that describes the rules for this program would be:

```
expr = “myFunction ” number number ;

number = <number>;
```

The values for the rule “number” are not specified; instead a symbolic constant is specified. This can be understood as the parsing trees shown in Figure 2.2 where the `number` branch in G is replaced in G’ by a symbolic constant whose description will be later given in V.

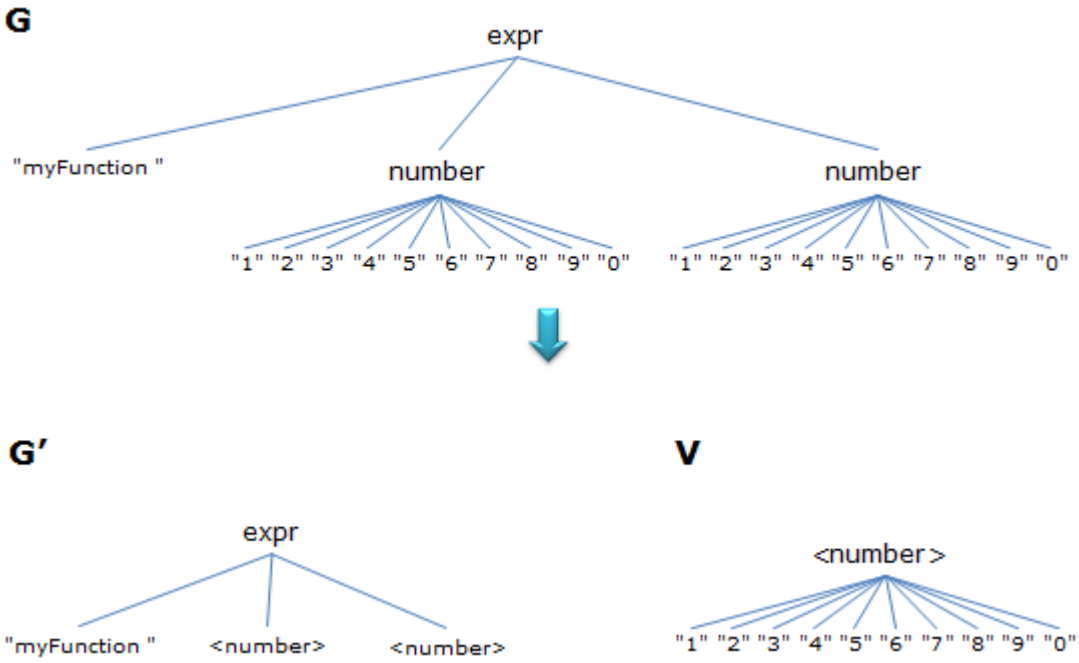



**Figure 2.2 Parse trees for G, G' and V**

The test generation algorithm instantiates symbolic constants with concrete values at runtime in a directed fashion which consists in treating symbolic constants as unconstrained symbolic values which will be solved by concolic execution. The reduction in the number of possible strings in the language enables exhaustive enumeration to scale and provides selectivity. In the other hand, directedness is given by concolic execution as the symbolic constants enables exploration of non-redundant strings.

The system works as follows:

- It converts the concrete grammar to a symbolic grammar by replacing unbound set of concrete constants with symbolic constants.
- It exhaustively enumerates all symbolic sentences from the symbolic grammar up to a certain size.
- For each symbolic sentence, directed testing is achieved by concolic execution where each symbolic constant is considered to be an unconstrained input to be solved.

As this work is based on concolic execution and model-based testing, it inherits their limitations, but the usage of both approaches lessens the effects and helps with scalability.

The effectiveness of the constraint solver engine is limited; i.e. if the constraints are beyond the capabilities of the constraint solver, random testing is employed to produce concrete values. The symbolic grammar is as effective as the user defines it; for example, a more descriptive grammar that captures the semantic properties or if deeper executions paths are needed to be exercised, a more descriptive grammar needs to be employed. Finally, the exhaustive generation of test inputs prevents this approach from being used in larger systems.

For our approach, a context-free grammar that gave a complete description of the protocol and valid program inputs was employed. The grammar was a lexical description of valid inputs of the technological base of the system under examination; normally, a programming language, a (communications) protocol (such as HTTP response messages), a container format (such as pdf file) or a policy definition (such as firewall rules).

## 2.9 Grammar-based WhiteBox Fuzzing

Kiezun, Godefroid and Levin (19) introduce grammar-based whitebox fuzzing which is an enhancement to whitebox fuzzing that employs a grammar-based specification of valid inputs. Grammar-based whitebox fuzzing is a dynamic test generation algorithm where symbolic execution generates grammar-based constraints whose intersection with the model grammar is verified using a grammar-based constraint solver. The algorithm has two main components:

- High-level symbolic constraints - expressed in terms of symbolic grammar tokens returned by the lexer, instead of the traditional symbolic bytes read as input.
- A custom constraint solver that solves constraints on symbolic grammar tokens. The solver looks for the intersection between the constraints and a given context-free symbolic grammar.

The grammar represents valid inputs for the program under test so all solutions generated by the constraint solver correspond to valid inputs.

Grammar-based whitebox fuzzing extends traditional whitebox fuzzing algorithms as follows:

- It requires a symbolic grammar G that describes only valid program inputs
- It associates a symbolic variable with each token returned from the tokenizing function.
- It uses the given grammar to require that the new input not only satisfies the execution path constraint, but that it is also is part of the language defined by the grammar

The constraint solver computes language intersection of both the given symbolic grammar (G) and the constraints recorded from the symbolic execution (R) as follows:

- Convert G to a Push-Down Automata (PDA);
- Convert R to a Finite-State Automata (FSA);
- Compute a PDA with the intersection between the PDA (G) and the FSA (R),
- Check the emptiness of the resulting PDA, if empty terminate; else
- Generate any string in that language.

The algorithm implemented (in SAGE) is a simpler version of this one. It computes a grammar representing the intersection of both languages G and R . It takes advantage that any regular language R always constrains only the first n tokens returned by the tokenization function, and that it does not go through an explicit PDA transformation.

Consider the following symbolic grammar:

```
expr = expr op  expr  | <number> ;
op = “+” | “–” | “*” ;
```

With a constraint grammar - taken from recorded runs:

```
T0 = <number>
T1 = “+”
T2 = <number>
T3 = “–”
```

Then the constraints are solved as follows:

1. First the grammar initial production (expr) is duplicated and renamed as expr’.

2. Starting with the duplicated initial production expr’ of the grammar, the algorithm removes the second production as it can't satisfy the constraint "<number>".
3. Then the algorithm examines the next constraint "+" and it expands the non-terminal op in the production, expr = <number> op expr, with the production, op = “+”, as it is the only production that satisfies the constraint from the production op = “+” | “-” | “*”.
4. This is repeated until all the constraints are solved or until a constraint can't be satisfied.

The values expr’ takes for each constraint as the algorithm solves them are shown:

```
expr’ = <number> op expr | <number>;
T0: expr’ = <number> op expr;
T1: expr’ = <number> “+” expr;
T2: expr’ = <number> “+” <number> op expr;
T3: expr’ = <number> “+” <number> “–” expr;
```

Finally, the symbolic sentence is executed by SAGE and the symbolic constants are substituted for concrete values by the constraint solver at runtime.

Some of the limitations of this work are the need for an accurate grammar that correctly describes the input space. Path explosion is lessened by the constraint solver which prunes non-feasible paths; however, the input space remains too open and exhausting random generation makes scalability infeasible.

# 3 Problem Overview

Fuzz testing is a method for discovering faults in software by providing unexpected inputs and monitoring for exceptions. Fuzz testing is divided into different phases: target identification, identify inputs, generate fuzz data, execute fuzzed data, monitor for exceptions and determine exploitability (20). This research focuses on fuzz data generation. For this, previous model-based fuzzing attempts are analyzed and a new approach for model-based fuzz data generation is proposed.

This section is divided as follows: first an analysis of fuzz testing and a definition of the problem to solve will be given. After that a revision of the different technologies employed in building our solution will be presented, Finally, the design of the new approach will be discussed.

## 3.1 Fuzz Testing

Software companies spend a great deal of its time and money in the testing of their software (21), so ways of optimizing this phase have been of interest for researches in the past years. One of the first steps to optimizing this task was to automate it as manual approaches proved to be inefficient. Fuzz testing was a technique that took immediate advantage of this automation. Fuzz testing has emerged as one of the most promising techniques for automated testing (22).

Sutton (20) divides fuzz testing into five basic phases: target identification, inputs identification, fuzz data generation, fuzzed data execution, exceptions monitoring, and exploitability determination. One of the most difficult and expensive parts of these phases is the actual generation of test data. Test data generation was first done by hand, and has evolved towards its complete automation (22).

Different approaches exist for automatic data generation (23).

- Random - generates test cases with a complete quasi-random approach. Its implementation is inexpensive and can quickly generate thousands of test cases. Its main disadvantage is that many of the generated cases are non-meaningful and we can't control which part of the systems are exercised.
- Adaptive - this approach is based on an architecture that implements feedback through an adaptive test generator. The technique provides values for input parameters and produces data values which calculate an indication of test effectiveness. The adaptive test generator uses these values

and previous test data values to produce new test data which attempts to increase test effectiveness.

- Syntax-based - this approach processes data which is expressed in a grammar or notation (i.e., BNF, a message protocol, etc.) and generates test cases from it. This approach is one of the most used for fuzz testing as it can generate large amounts of well-formed data in an automated fashion.
- Path-oriented - the main goal of the approach is to execute a full-path coverage of the application under test. This approach has been implemented mainly by symbolic execution which is very effective, but expensive in terms of domain knowledge and implementation.
- Specification-oriented methods - test generation is based on the specification of the system. The main drawback of these approachs are that specification is not collected in a uniform fashion preventing a general method to escalate.

From all these different approaches syntax-based generation testing was selected as it is one of the most widely used in the industry (21) and the significant advances it has gone through in recent years make it a very promising field of research.

## 3.2 Grammar-based Fuzzing

One of the main advantages of using syntax-based test generation is that test data structure must be documented precisely; this helps to optimize the phases of software maintenance and debugging. Another advantage is that it can generate large quantities of data in an automated fashion. Recent research has focused on optimizing the model's usage (i.e. using all productions in a grammar) and generating test data that focuses on maximizing code coverage or complies with statistical distributions. All of these advantages help the tester to add complexity to the test data (23).

Two main disadvantages of this technique are that some classes of data are impossible to generate (i.e. GUI testing); and that writing the syntax rules for complex sets of test data can be very complex. These limitations are the reasons why this approach has normally been applied for testing complier projects.

One of the most explored model-based fuzz testing strategies is grammar-based fuzzing. The first approaches were bottom-up strategies that created a specific grammar that was able to generate a specific set of values (6). Later this approach was modified for top-down strategies where grammars describing the complete input space are used to generate test data (5).

As the input space of top-down strategies is too large, strategies have been suggested to reduce that space. For example attributed grammars were proposed by Maurer (6) to give context-awareness to the context-free grammar. Another approach was studied by McKeen (7) who employed stochastic grammars to associate probabilities to its rules.

One of the most recent approaches was suggested by Majmudar (18) who introduces the concept of symbolic grammars. Symbolic grammars are context-free grammars where terminal symbols are substituted with regular expressions that represent the solution space for that precise set of terminal symbols. The reduction in the number of possible strings in the language enables exhaustive enumeration to scale and provides selectivity. However, the input space is still too large and relies basically in the symbolic execution engine to constrain the sentences generated with the symbolic grammar. A solution was proposed by Kiezun et al. (19) where high-order symbolic constants are used in a constraint symbolic grammar to reduce the input space and execute a directed generation of sentences.

## 3.3 Limitations

Three main limitations have been identified regarding grammar-based testing:

- Employment - grammar-based test generation can be employed in a wide variety of systems that receive a structured input as any system that receives a structured input can be expressed with a grammar created in a bottom up fashion (6). But it is encouraged for systems that have highly structured inputs (such as programming languages, protocols, format containers or policy definitions) that can be defined with a context-free grammar (or can be transformed to one) (19). For systems or tests that do not have structured inputs, grammar-based testing is not recommended.
- Context-sensitive inputs – due to their nature, context-free grammars (CFG) are not context sensitive, so for this kind of testing which is based on the employment of CFG. In practice the set of valid inputs of a system is bounded by an approximated grammar which is a simplified representation of valid inputs. Approximated grammars are subsets of the grammar that describe the entire solution space, but with added rules to approximate context-sensitive behaviours needed to create parsable inputs. (19)
- Domain knowledge – grammar-based testing requires a limited amount of domain knowledge: the formal grammar,

the criteria to convert formal grammar into a symbolic grammar; and, if needed, the constraint grammar definition. Formal grammars are sometimes readily available;, i.e. for our initial test the http-cookie formal grammar defined in RFC 2965 was found in a matter of minutes.

## 3.4 Grammar-based Test Data Generators

Up to now the advantages and limitations of grammar-based data generation for grammar-based fuzz testing has been discussed. It is important to establish the generic model of any grammar-based test data generator, is shown in Figure 3.1.

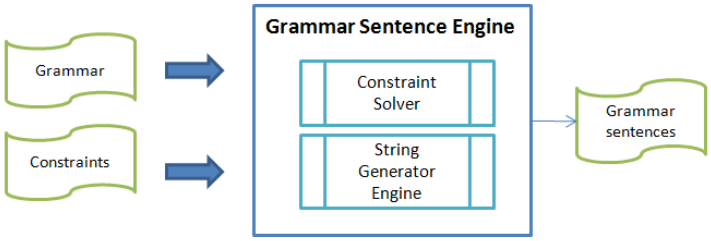



**Figure 3.1 Generic model for grammar-based data generator**

Where:

- Grammar - is a set of rules that represents the structure of the input solution space.
- Constraints – fuzzing heuristic (22) which sets boundary values for the input solution space.
- Grammar sentence engine – in charge of solving the constraints and the grammar to produce grammatical sentences that comply with both.
- Constraint solver – solves the input constraints, in the context of the grammar, so that the string generator engine can use them to generate valid sentences.
- String generator engine – generates grammatical sentences using the computed constraints from the constraint solver.
- Grammar sentences – grammar-based generated sentences which comply with both the input grammar and other constraints (these are the actual test cases).

Limitations on each entity can be found in Table 3.1.

| | Limitations | Previous Solutions |
|---|---|---|
| **Grammar** | Lack of expressiveness (CFG modification for system employment) | • Attributed grammars (6) (8)<br>• Stochastic grammars |

| | | |
|---|---|---|
| | | (7)<br>• Symbolic grammars (18) |
| **Constraints** | Effective representation of inputs<br>(The format in which the inputs are fed to the system). | • Byte code level (17)<br>• High-level symbolic level (19) |
| **Constraint Solver** | Effective constraint solving<br>(How the system solves its constraints) | • Symbolic solver (18)<br>• Language intersection (19) |
| **String Generation Engine** | Lack of grammar rules usage control<br>(Technique used to control rules during sentence generation) | • Attributed grammars (6) (8)<br>• Stochastic grammars (7)<br>• Uniform distribution (24)<br>• Shortest string (3) |
| | Lack of sentence length control<br>(Technique used to control sentence length) | • Fixed length (8) (24) |
| | Unconstrained generation of sentences<br>(Technique used to generate sentences from the grammar) | • Random approaches (18)<br>• Selective approaches (8) |
| **Grammar Sentences** | Impossibility to choose which sentences to generate<br>(Technique used to select which sentences to generate) | • Enumeration (18)<br>• Directed generation (19)<br>• Selective post-phase (8) |
| This table is a summary of sections 3.1.1 and 3.1.2 | | |

**Table 3.1 Limitations for grammar-based data generator components**

## 3.5 Systems Comparison

In this section the different attempts to generate test cases for each approach will be discussed. As a simple application it is consider a simplified version of SimpleCalc that is employed by several authors (18) that explore this technology). The following approaches were chosen as they establish the state-of-the-art at the moment and are the most more related to our work:

1. Random Generation of sentences from a concrete grammar.
2. Symbolic Test Generation introduced by Majumdar et al. (18) uses symbolic grammars and exhaustive generation.

3. Symbolic Constrained Test Generation introduced by Kiezun et al. (19) uses symbolic grammars and symbolic constraints to bound the input domain.

The BNF-grammar for our simplified version of Simple Calc (18) is the following:

```
expression =  singleExpresion | operationExpression |
     numbers ;
singleExpresion = "(" expression ")" | "-" expression ;
operationExpression =  expression operator expression ;
operator = "+" | "-" | "*" | "/" | "%" | "^" | "v" ;
numbers = number number;
number = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
     | "9" ;
```

The program takes an arithmetic expression, parentheses for precedence, various numerical and some bitwise operators. Numerical operators are directly applied, and precedence is handled by parsing. Let's suppose that the implementation contains some simple and common bugs: division or modulus by zero.

For our tests, we will take a maximum sentence length for simplicity. Note that numbers are considered a single terminal symbol for our comparison. Length is measured in terminal symbols found in the generated concrete sentence, for example, for the concrete sentence "30" "+" "50" we have three terminal symbols thus, a length of three.

The approach will be considered to solve the problem when a division or module by zero test cases is generated. The difference of these approaches with our proposed solution will be used to explore its strengths and benefits over previous work.

### 3.5.1 Random Generation

Random generation approaches takes a grammar and generates random sentences from it. Usually exhaustive generation of all possible inputs is infeasible as the input space is too large.

Let's take Table 3.2, which gives the total number of different sentences for a given length that can be produced by the SimpleCalc grammar.

| Length | Valid unique strings | Percentage of tests |
|---|---|---|
| 3 | 70 200 | 0.288% |
| 4 | 140 300 | 0.289% |
| 5 | 49 420 300 | 0.368% |

**Table 3.2 Total number of different sentences for a given length for the SimpleCalc grammar**

It is clear that as the string length keeps increasing, the number of valid unique strings grows exponentially. Thus enumeration is not a good option for test case generation in this case.

The probability of producing a grammatical sentence of length five that has the exact values needed for exposing the desired pattern (‘/00’ or ‘%00’) for lengths three to five are extremely low, so hitting a bug with this technique is highly improbable. The problem is that random testing is neither directed nor selective.

### 3.5.2 Symbolic Test Generation

The first step proposed for this method is to convert a context-free grammar (CFG) into its equivalent symbolic grammar. This is accomplished by replacing the CFG’s terminal symbols (in our grammar, the number is considered to be a terminal symbol) with regular expressions (symbolic constants) that represent the entire solution space for that precise set of terminal elements. In our example, two rules were substituted (operation and numbers) with symbolic constants that represent the values defined by the substituted rules:

```
expression =  singleExpresion | operationExpression |
     <NUMBERS> ;
singleExpresion = "(" expression ")" | "–" expression ;
operationExpression =  expression <OPERATOR> expression ;
```

With the symbolic grammar, the method undertakes an exhaustive generation of symbolic sentences up to length five (producing symbolic sentences of length less than five is not of interest in our example). With this new approach, the enumeration is simple and manageable (only length-five symbolic sentences are shown):

"-" "-" "-" "-" <NUMBERS>
"-" "-" "(" <NUMBERS> ")"
"-" "-" <NUMBERS> <OPERATOR> <NUMBERS>
"-" <NUMBERS> <OPERATOR> "-" <NUMBERS>
"(" "(" <NUMBERS> ")" ")"
"("<NUMBERS> <OPERATOR> <NUMBERS> ")"
"(" <NUMBERS> ")" <OPERATOR> <NUMBERS>
<NUMBERS> <OPERATOR> "-" "-" <NUMBERS>
<NUMBERS> <OPERATOR> "("<NUMBERS> ")"
<NUMBERS> <OPERATOR> <NUMBERS> <OPERATOR> <NUMBERS>

The approach proposes that exhaustive generation should be made with these symbolic sentences, selecting test cases that are of interest or that could exercise interesting execution paths. Common criteria for

selection would suggest selecting symbolic sentences which are very simple, equivalent or that show common patterns in them. For example applying exhaustive generation over "-" <NUMBERS> <OPERATOR>"-" <NUMBERS> which is the simplest form of the <NUMBERS> <OPERATOR><NUMBERS> pattern and is present in seven of the ten generated symbolic sentences, makes it the "obvious" testing target. The approach uses concolic execution (14) for test cases selection, but as test generation phase of this system is the only element discussed, it is not applicable. To overcome this limitation, the approach suggests constrained exhaustive enumeration of concrete sentences from symbolic sentences. In our example, this approach would select the symbolic sentence "-" <NUMBERS> <OPERATOR>"-" <NUMBERS> with a simple enumeration approach:

```
"-" "00" "+" "-" "00"
"-" "01" "+" "-" "00"
"-" "02" "+" "-" "00"
"-" "03" "+" "-" "00"
...
"-" "99" "+" "-" "00"
"-" "00" "-" "-" "00"
...
```

For this enumeration, each possible concrete value the symbolic sentence can take was generated, so it is clear that the search is directed but it is not very selective once the symbolic sentences have been generated.

It should be noted that even though this approach makes enumeration manageable (up to certain lengths), the number of symbolic sentences can grow exponentially for complex grammars. Another limitation is that there is no proposed strategy to control the total number of symbolic constants in symbolic sentences. For example, generating sentences that are of no interest for our testing strategy (in our example symbolic sentences containing less than five concrete constants are likely to be produced).

### 3.5.3 Constrained Symbolic Test Generation

This approach employs symbolic grammars in a similar way as symbolic test generation, but introduces a symbolic constraint grammar which restricts the input space. The symbolic constraint grammar consists of single symbolic non-terminal element rules. Each of these rules represents a symbolic constant constraint that the generated symbolic sentence should satisfy. To satisfy these constraints, the system computes the intersection between the symbolic constraint grammar and the symbolic grammar exploiting the fact that, by construction, any regular language always constrains only the first n-symbolic constants of the symbolic grammar, where n is the total number of rules in the symbolic constraint grammar.

The algorithm guarantees that productions that violate the constraints during computation will be pruned from the search, making it a directed search.

Using this strategy, once the symbolic sentence is generated, concrete values are instantiated to comply with concolic execution. As only test generation is being used for this approach, the instantiation of concrete values for our tests has to be implemented separately. This method instantiates symbolic values with pseudo-random values, making it very ineffective when specific inputs are required to expose a system flaw.

To illustrate this, the previously defined symbolic grammar and the following symbolic constraint grammar are employed:

```
T0 = <NUMBERS> ;
T1 = <OPERATOR> ;
```

The symbolic constraint grammar is defined by the user looking to prune from the search any non-interesting paths. For the symbolic constrained grammar, any input that begins with the symbolic constants <NUMBERS> and <OPERATOR> was found promising. So only the following strings are generated after the languages intersection is computed:

<NUMBERS> <OPERATOR> "-" "-" <NUMBERS>
<NUMBERS> <OPERATOR> "(" <NUMBERS>")"
<NUMBERS> <OPERATOR> <NUMBERS> <OPERATOR> <NUMBERS>
<NUMBERS> <OPERATOR> "-" <NUMBERS>
<NUMBERS> <OPERATOR> <NUMBERS>

It should be mentioned that not only sentences of length five are generated, but all strings that comply with the constraint up to length five are produced. This enhances the previous approach as it directs the testing generation only with test cases that are "of interest" for our testing strategy. It can be seen that any of these symbolic sentences are likely to expose the failure in the same amount of tries (except the third one).

The main limitation of this approach is that it doesn't present a good strategy for instantiating concrete sentences from the symbolic sentences. This is a major drawback; thus, after generating a desired symbolic grammar, the algorithm resorts to pure random generation.

Another limitation is that for each test case, a symbolic sentence is generated. This makes it difficult to produce test cases that share the same structure defined by the symbolic sentence. For our example, five symbolic sentences are generated.

Finally, when a symbolic constant is solved its instantiation is not controlled, thus losing any relation between the elements that compose the regular expression that represents the solution space for the symbolic constant. This affects directly the results obtained when replacing the symbolic constants for concrete values as they raise the probability of producing syntactical incorrect test cases when a relation between its concrete values exists.

### 3.5.4 Our Approach

The strongest features of the two discussed systems were combined in our approach. Meanwhile for the limitations that both approaches present we suggest new alternatives to lessen their effects and produce a novel approach for grammar-based generation.

The main functionality of our system consists of three sequential phases: **constraint solving**, **symbolic sentences generation** and **concrete sentence production**. These phases use a symbolic grammar, symbolic values grammar and a symbolic constraint expression.

The symbolic grammar has the same characteristics as previously discussed. The symbolic value grammar is where symbolic constants are defined with concrete constants; it is used to instantiate symbolic constants. Finally the symbolic constraint expression is a regular expression expressed in terms of symbolic constants and concrete values that replace the symbolic constraint grammar previously discussed where its left hand side should be the initial symbol of the symbolic grammar. These three components are shown:

**Symbolic Grammar**

```
expression =  singleExpresion | operationExpression |
     <NUMBERS> ;
singleExpresion = "(" expression ")" | "-" expression ;
operationExpression =  expression <OPERATOR> expression ;
```

**Symbolic Values Grammar**

```
NUMBERS = number number;
OPERATOR = "*" | "/" | "%" | "+" | "-" | "^" | "v" ;
number = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
     | "9" | "0" ;
```

**Symbolic Constraint Expression**

```
expression = <NUMBERS> <OPERATOR> ;
```

For the **constraint solving** phase, our system computes the language intersection defined by the symbolic grammar and the symbolic constraint expression to produce a symbolic constrained grammar. For our example, the constraint expression that is equivalent to the symbolic constraint

grammar employed in the last example and the symbolic grammar for the SimpleCalc was used as follows:

**Symbolic Constrained Grammar**

```
expression = <NUMBERS> <OPERATOR> expression' ;
expression' =  singleExpresion | operationExpression |
     <NUMBERS> ;
singleExpresion = "(" expression' ")" | "-" expression' ;
operationExpression =  expression' <OPERATOR> expression' ;
```

It can be appreciated that the rule expression' is now part of the constrained symbolic grammar and that the rule expression has been redefined with the computed intersection of the languages. With these changes, it is guaranteed that the constrained symbolic grammar can only generate sentences that comply with the language intersection.

After this phase, **symbolic sentences generation** starts in which the system computes grammar-based, lexically accurate, unique sentences from the symbolic constrained grammar that comply with the symbolic constraint expression. Using our example these would be the results of this phase:

```
<NUMBERS> <OPERATOR> "-" "-" <NUMBERS>
<NUMBERS> <OPERATOR> "(" <NUMBERS>")"
<NUMBERS> <OPERATOR> <NUMBERS> <OPERATOR> <NUMBERS>
<NUMBERS> <OPERATOR> "-" <NUMBERS>
<NUMBERS> <OPERATOR> <NUMBERS>
```

Constrained symbolic test generation is not aware of previous generated sentences, so it is likely to generate the same symbolic sentences several times. Our system overcomes this limitation by producing only unique symbolic sentences, allowing the system to focus its efforts on the **concrete generation phase**.

The symbolic sentences represent different grammatically valid structures that are used to produce concrete sentences during the **concrete generation phase**. For this purpose, our system uses quasi-random techniques replacing traditional pseudo-random techniques for the instantiation of symbolic constants. This phase produces the same number of concrete sentences for each symbolic sentence, giving the tester control over the relation of symbolic constants elements and the number of concrete sentences that will be produced for each symbolic sentence. For example Table 3.3 enlists 10 concrete sentences for two symbolic sentences.

| <NUMBERS> <OPERATOR> <NUMBERS> | <NUMBERS> <OPERATOR> <NUMBERS> <OPERATOR> <NUMBERS> |
|---|---|
| 71-62 | 04^05v19 |
| 17*08 | 45-50%36 |

| | |
|---|---|
| 09/90 | 00*30+90 |
| 44v35 | 18/85^21 |
| 03%03 | 72v68-76 |
| 58+49 | 01%13*59 |
| 75^66 | 47+54/04 |
| 20/11 | 20%00^45 |
| 40v80 | 74+27-00 |
| 95%26 | 33^81*18 |

**Table 3.3 Concrete sentences for two symbolic sentences**

It can be seen that values are evenly distributed through the input domain. Take for example the concrete values for the symbolic constant <OPERATOR>, where in 10 elements each element only appears twice only after all the other elements have been generated.

This is the main idea of our system. The design and description of the main features will be presented in the next section.

# 4 The System

In previous sections the current state-of-the-art was analyzed and the current limitations were discussed. Then we gave an overview of our system. Section 4 presents a detailed discussion of the system which is based on the following concepts:

- The solution shall accept any context-free grammar (CFG).
- The solution shall employ a symbolic grammar derived from the CFG.
- The solution shall be flexible enough to accept any given grammar with minimal changes
- The solution shall employ symbolic constraints.
- The solution shall constrain its input space with human-readable restrictions.
- The solution shall employ a constraint solver that computes the intersection between the symbolic constraints and the symbolic grammar.
- The solution shall use all selectable grammar productions in a uniform fashion to produce grammar sentences.
- The solution shall provide control over the volume of terminals in the sentence.
- The solution shall implement a selective strategy for instantiating symbolic sentences.
- The solution shall generate directed concrete test cases that share the same symbolic sentence.
- The solution shall give control over symbolic constant instantiation to maintain the relation between the elements that compose concrete values.

## 4.1 Definitions

Through section 3 we have been faithful to the definitions and terms used by the authors of the approaches analyzed. However as they are not consistent through the literature, hence, at this point, these terms will be defined more formally here and these definitions are used throughout the remainder of this document.

- Let $\Pi$ be a finite alphabet.
- A terminal is a regular expression over $\Pi$.
- A **context-free grammar** (a set of recursive productions used to generate patterns of strings) is defined as a 4-tuple $G = (B, \Sigma, R, S)$ where:
  - B is a finite set of non-terminals.

- ∑ is a finite set of terminals.
- R is a finite set of production rules of the form $B \to (B \cup \Sigma)^+$.
- S ∈ B is the start symbol.

- The language L(G) ⊆ Π* of G is defined as usual (25):
  - You use a grammar to describe a language by generating each string of that language in the following manner.
    1. Write down the start symbol. It is the non-terminal on the left-hand side of the top rule, unless specified otherwise.
    2. Find a non-terminal that is written down and a rule that starts with that non-terminal. Replace the written down non-terminal with the right-hand side of that rule.
    3. Repeat step 2 until no non-terminals remain.
  - All strings generated in this way constitute the language of the grammar.
- The language $L_h(G) \subseteq \Pi^*$ of G is defined as all strings containing h terminals of Π.
- Let $\Phi = \{TE_1, \dots, TE_i, \dots, TE_m\}$, where Φ is referred to as a **concrete sentence (CS)** of length m and $TE_i \in \Sigma$ is referred to as a **concrete constant**.
- Let $A = \{\alpha_1, \dots, \alpha_i, \dots, \alpha_k\}$ be k symbolic constants not in Π where $\alpha_i \epsilon A$ is referred to as a **symbolic constant**.
- A **symbolic grammar (G')** for G is defined as a 4-tuple $G' = (F, \Upsilon, R', S)$ where:
  - F ⊂ B where B is the finite set of non-terminals of G.
  - $\Upsilon = \Sigma \cup A$ where Σ is the set of terminals of G.
  - R' is a finite set of production rules of the form $B \to (B \cup \Upsilon)^+$
  - S is the start symbol of G
- The language of G' is defined as $L_d(G') \subset (\Pi \cup A)^*$.
- Let $\Psi = \{SE_1, \dots, SE_i, \dots, SE_n\}$ where Ψ is referred to as a **symbolic sentence (SS)** of length n and $SE_i \in \Upsilon$.
- A **symbolic constraint expression (E)** of G' is defined as a symbolic sentence.
- A **symbolic constraint grammar (C')** of E is defined as a 3-tuple $C' = (N, \Upsilon, R')$ where:
  - N is a finite set of non-terminals.
  - Υ is the finite set of terminals of G'.
  - R' is a finite set of production rules of the form $N \to \Upsilon'$.
- A **symbolic constrained grammar (C)** is defined as $C = G' \cap C'$

- A **symbolic value grammar (V)** for G' is defined as 3-tuple $V = (W, \Lambda, P)$
  - W is a finite set of non-terminals where A $\in$ W and A is the set of symbolic constants in G'.
  - $\Lambda$ is a finite set of terminals.
  - P is a finite set of production rules of the form $W \rightarrow (W \cup \Lambda)^*$.
- Let Ç be a finite set of production rules of the form $W \rightarrow \Lambda^+$.
- Let $\Xi = \{pCE_1, \dots, pCE_i, \dots, pCE_z\}$ where $\Xi$ is referred to as a **pre-concrete sentence (pCS)** of length z and $pCE_i \in Ç$.

## 4.2 Technologies Employed

The purpose of the system is to generate lexically-accurate examples of a given symbolic grammar that complies with a set of additional user-defined constraints. The user-defined constraints place limitations on the initial symbolic grammar to provide an effective specialisation. With these examples, a set of effective test cases provided that the symbolic grammar and the constraints provide an adequate representation of the input space can be generated. The effectiveness of the test cases is enhanced by a user-defined symbolic definition grammar (symbolic value grammar) that will define specialized structures for different test cases that share the same symbolic sentences.

Our work is mainly based on the following techniques

- Symbolic grammars which were discussed in section 2.8.
- Symbolic constraints which are used for constraining the input space of a symbolic grammar with a symbolic constraint grammar, allowing the test case generation to be directed (19).
- Adaptive random testing, proposed by Tappenden and Miller (26), which is a strategy whose objective is to increase the effectiveness of random testing by attempting to maximize the coverage of the input space.
- Selective and directed concrete sentence generation techniques that enhance symbolic sentence and symbolic constants instantiation. These techniques are novel in our approach; as in previous grammar-based systems, concrete sentence generation was computed during concolic execution.
- Genetic algorithms for enhancing some punctual deterministic heuristics used on the symbolic sentence generator.

Our approach takes all the advantages of these technologies and adds novel enhancements to lessen their limitations. The combination of all these techniques and enhancements produces a novel approach for model-based

test data generation. These techniques will be discussed in the following sections.

### 4.2.1 Symbolic Constraint

Our approach takes the initial idea of a symbolic constraint grammar defined by Kiezun et al (19). In this approach, the input space of a symbolic grammar is constrained with a symbolic constraint grammar, allowing the test case generation to be directed and comply with the symbolic execution properties.

The original implementation employs dynamic testing where selected concrete sentences are used as symbolic constraint seeds for subsequent iterations. For this, the system must transform a symbolic sentence into its symbolic constraint grammar equivalent in order to be able to use it. It was found through our experiments, that for test data generation, this approach becomes impractical as there is no formal heuristic suggested for this transformation, making it difficult for the tester to correctly define the symbolic constraint grammar fed to the system. For example, consider the following symbolic sentence, from the SimpleCalc grammar used in the previous examples:

```
<NUMBERS> <OPERATOR> "(" <NUMBERS> ")"
```

To convert it into its symbolic constraint grammar, a rule for each terminal symbol must be added:

```
T0 = <NUMBERS> ;
T1 = <OPERATOR> ;
T2 = "(" ;
T3 = <NUMBERS> ;
T4 = ")" ;
```

This approach is not formally defined and other approaches might exist. Because of this the grammar must be substituted with a regular expression expressed in terms of symbolic constants and concrete values which are part of the symbolic grammar. Our approach uses the same symbolic sentence generated by the system, but preceded it with the starting symbol of the symbolic grammar:

```
expression = <NUMBERS> <OPERATOR> "(" <NUMBERS> ")"
```

With this enhancement, first, the starting symbol of the symbolic grammar (which originally had to be provided separately) is explicitly set; and second, the constraint definition is simplified by using the same symbolic sentence generated by the system (with the previously mentioned modification). This last modification allows the user to only provide the symbolic constraint expression, suppressing the explicit definition of the

symbolic constraint grammar. Please note, the system still uses the symbolic constraint grammar internally.

A symbolic constraint expression is produced from a well-formed input (usually a well-formed concrete sentence that is part of the language described by the system input grammar). A common way of producing an initial seed is using inputs generated randomly from the symbolic grammar; another approach could be to select a seed from previous attempts (a pool of known input seeds) that are already known to be likely to exercise sections of the program that are of interest (19).

### 4.2.2 Adaptive Random Testing

Adaptive random testing seeks to maximize the effectiveness of traditional random testing by spreading the test cases evenly across the input domain. It employs the random generation of test cases with a selection criterion used to evaluate the best available candidate (27). The basis for adaptive random testing methodologies has its origins in the observation that errors often occur within failure regions within the input domain (28).

Quasi-random sequences are mathematical sequences whose low-discrepancy properties allow them to provide a sequence with a uniform distribution of values. The low-discrepancy property ensures that the constructed sequences are "evenly-spaced"; that is, minimise the discrepancy from the quantitative definition of "evenly-spaced" (26).

Quasi-random sequences have several limitations, most seriously their deterministic nature. Each time a sequence of the same length is calculated a new sequence is generated with the same numbers in the same order, so the sequences are clearly not random. Several attempts have been proposed to overcome this limitation, such as scrambling the sequence each time it needs to be reused (28). However this solution is not currently available for real-world testing applications as an adequate scrambling method that retains the low-discrepancy nature of the quasi-random sequence is not computationally feasible. For addressing this problem, each time the sequence finishes and must be repeated, it is proposed that a different starting point in the sequence should be chosen.

For our implementation several options exist; the quasi-random sequence proposed by Sobol (29) is used. This sequence is widely used for financial simulations and has been proposed for use in software testing by Chi & Jones (30). Another alternative is the heuristic based approach of Tappenden and Miller (26) which has many characteristics which suggest that it may be superior to the Sobol Sequence. However, solutions exist for generating the Sobol sequence of linear algorithmic complexity; whereas, Tappenden and Miller is of quadratic complexity. Hence, a Sobol generator

is being utilised as it provides a more assured mechanism for producing a highly-scalable test generator.

With this strategy, the generic model of Grammar-based Test Data Generators is enhanced by replacing pseudo-random evaluations of symbolic constants by selections from quasi-random sequences.

### 4.2.3 Selective and Directed Concrete Sentence Generation

This enhancement to symbolic grammars has not been examined as symbolic grammars are commonly only considered as a sub-component of concolic execution. Different alternatives are employed to give control of concrete sentence generation and symbolic constant instantiation.

Selective and directed concrete sentence generation is proposed via two approaches for controlling how concrete sentences are instantiated from symbolic sentences.

### 4.2.4 Concrete Sentence Generation Control

As discussed, another limitation is that for each concrete sentence a symbolic sentence is generated. The original algorithm transforms each symbolic sentence into a concrete sentence, thus the probability of having a concrete sentence that shares the same symbolic sentence in a set of generated strings decreases dramatically. To address this problem, using a concrete sentence generation control mechanism, which adds control over the symbolic constants is proposed. This approach consists in generating a user-defined fixed set of different concrete sentences for each symbolic sentence opposed to generating a single concrete sentence for each symbolic sentence as previous work suggested (19).

### 4.2.5 Symbolic Constants Instantiation Control

In previous work, when a symbolic constant is solved any relation between its sub-components is discarded. This affects directly the results obtained when replacing the symbolic constants for concrete values as this increases the probability of producing syntactically incorrect test cases.

In our approach, symbolic constants instantiation control which adds explicit relations between symbolic constants sub-components which must be instantiated communally is proposed (by low discrepancy sequences) to increase the probability of producing syntactically correct concrete sentences. To accomplish this, the concept of "**entities"** is introduced, which are groups of symbolic constants that are related and hence are

treated as a single compound value (for example IP addresses or port numbers).

### 4.2.6 Removing Deterministic Componentry

During the examination and testing of the symbolic sentence generation, several deterministic algorithms that controlled the rule usage for producing symbolic sentences were found. These algorithms could be enhanced with an evolutionary computation approach to produce more random (better distributed) symbolic sentences.

For this enhancement, these algorithms (*choose(l)* and $g_i(n)$ in McKenzie's algorithm (24) ) were transformed into an evolutionary computation introducing a mutation computation (31) to provide a stochastic component to these algorithms The algorithms work together to select the next rule that will guarantee to maintain the uniform distribution implemented by McKenzie. With the introduced mutation computation add a user defined probability that will allow the algorithm to choose an arbitrary rule. This enhancement prevents rule selection becoming too similar to each other; thus making the rule selection better distributed.

## 4.3 System Overview

| **Symbolic Grammar (G')** | | **Symbolic Value Grammar (V)** | |
|---|---|---|---|
| **El.** | **Description** | **El.** | **Description** |
| F | expression, singleExpresion, operationExpression | W | OPERATOR, NUMBER |
| Y | <OPERATOR>, <NUMBER>, "(", ")", "-" | Λ | *, "/", "+", "-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "0" |
| R' | • expression = singleExpresion \| operationExpression \| <NUMBER> ;<br>• singleExpresion = "(" expression ")" \| "-" expression ;<br>• operationExpression = expression <OPERATOR> expression ; | P | • OPERATOR = "*" \| "/" \| "+" \| "-" ;<br>• NUMBER = "0" \| "1" \| "2" \| "3" \| "4" \| "5" \| "6" \| "7" \| "8" \| "9" \| "0" ; |
| S | expression | | |

**Figure 4.1 Members description for elements of G' and V**

For an overview of the system phases and how they interact, the following example is given (this section will be followed by a section with a detailed example).

A simplified version of the previously defined G', V and E (section 3.1.4.4) are used (Figure 4.1 shows a description of members conforming G' and V):

**Symbolic Grammar (G')**

```
expression =  singleExpresion | operationExpression |
     <NUMBER> ;
singleExpresion = "(" expression ")" | "-" expression ;
operationExpression =  expression <OPERATOR> expression ;
```

**Symbolic Value Grammar (V)**

```
OPERATOR = "*" | "/" | "+" | "-" ;
NUMBER = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
     | "9" | "0" ;
```

**Symbolic Constraint Expression (E)**

```
expression = <NUMBER> <OPERATOR> ;
```

Some examples of sentences that are part of $L_d(G')$ would be the following:

```
"(" "(" "(" "-" <NUMBER> <OPERATOR> <NUMBER> ")" ")" ")"
<NUMBER> <OPERATOR> "-" <NUMBER> <OPERATOR> <NUMBER>
"(" "(" <NUMBER> ")" <OPERATOR> "-" <NUMBER> ")"
"-" <NUMBER>
"-" <NUMBER> <OPERATOR> "(" <NUMBER> ")"
```

The first phase of the system is the **constraint solving phase** where the intersection between G' and E is computed, the result would therefore be a *symbolic constrained grammar* (C). To produce C, the system makes changes in S (the initial rule of G') to ensure that:

- All sentence produced by C comply with E, and
- $L_C(C) \subset L_d(G')$

For this example, the resulting C would be:

**Symbolic Constrained Grammar (C)**

```
expression = <NUMBER> <OPERATOR> expression' ;
expression' =  singleExpresion | operationExpression |
     <NUMBER> ;
singleExpresion = "(" expression' ")" | "-" expression' ;
operationExpression =  expression' <OPERATOR> expression' ;
```

The next phase of the system is the **symbolic sentences generation phase** where symbolic sentences are generated from C. In the following examples, it is clear that the sentences produced by this phase comply with E and are a subset of G':

```
<NUMBER> <OPERATOR> "-" <NUMBER>
<NUMBER> <OPERATOR> "(" <NUMBER> ")"
<NUMBER> <OPERATOR> "(" <NUMBER> ")" <OPERATOR> <NUMBER>
```

```
<NUMBER> <OPERATOR> <NUMBER>
<NUMBER> <OPERATOR> <NUMBER> <OPERATOR> "(" "-" <NUMBER> ")"
```

The final phase is the **concrete sentences generation phase** where the produced symbolic sentences are instantiated with concrete values. This phase introduces adaptive testing strategies for rules and terminal symbols selection from V. In Figure 4.2, the instantiated concrete sentences are shown for each of the previously produced symbolic sentences.

```
<NUMBER><OPERATOR><NUMBER>
          1 * 2
          9 + 8
          7 / 1
          4 - 9
          5 * 3

<NUMBER><OPERATOR>"("<NUMBER>")
      "<OPERATOR><NUMBER>
          1 / ( 4 ) + 3
          3 - ( 3 ) - 7
          9 / ( 8 ) / 2
          0 + ( 7 ) * 9
          2 * ( 0 ) + 1

<NUMBER> <OPERATOR> "("
      <NUMBER> ")"
        5 - ( 0 )
        4 * ( 9 )
        9 + ( 3 )
        1 / ( 7 )
        2 - ( 5 )

<NUMBER> <OPERATOR> "-" <NUMBER>
            4 - - 6
            5 + - 5
            1 * - 1
            0 + - 9
            2 / - 0

<NUMBER> <OPERATOR> <NUMBER> <OPERATOR> "(" "-" <NUMBER> ")"
                    1 + 2 * ( - 2)
                    2 - 3 / ( - 8)
                    7 * 8 + ( - 3)
                    8 / 1 - ( - 7)
                    5 + 0 / ( - 5)
```

**Figure 4.2 Symbolic sentences and their corresponding instantiated concrete sentences.**

## 4.4 Working Example

In this section, a large example illustrates how the employed technologies work together. Let's suppose a database engine must be tested with a set of test cases derived from a simplified version of the BNF Grammar for ISO/IEC 9075:1999 (32) for "SELECT" statements:

**BNF Grammar (G)**

```
Select = "SELECT " column_name (", " column_name )* "FROM "
     table_name "WHERE " BCond ;
BCond = BCond "OR " BTerm | BTerm ;
BTerm = BTerm "AND " BFactor | BFactor ;
BFactor = "NOT " BCond | id "IS NULL " | ATerm | STerm ;
ATerm = value aop value ;
Value = id | number ;
STerm = id "LIKE " value | id sop value | id sop id ;
aop = "=" | "<" | ">" | "<=" | ">=" | "!=" ;
sop = "=" | "!=" ;
column_name = "CustomerID" | "CompanyName" | "ContactName" |
     "ContactTitle" | "Address" | "City" | "Region" |
     "PostalCode" | "Country" | "Phone" | "Fax" ;
table_name = "Customers" ;
id = column_name ;
```

```
number = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
     | "9" ;
value = (letter)+ ;
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
     | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
     "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" |
     "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
     "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" |
     "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;
```

Examples of the sentences part of the grammar's language are also given:

```
SELECT CustomerID FROM Customers WHERE kGHEn = trnQWE OR NOT
     PostalCode = YJIkkl ;
SELECT CompanyName FROM Customers WHERE Fax LIKE  HDllKK AND
     Country != Region ;
SELECT ContactTitle FROM Customers WHERE Phone IS NULL OR
     BFactor OR City = a ;
SELECT City FROM Customers WHERE Jaw >= jKuuIgfDeWA ;
SELECT Region FROM Customers WHERE ContactTitle IS NULL AND
     Address LIKE  mNnN ;
```

### 4.4.1 Prerequisites

The prerequisites will be the generation of G' and V from G. As it was explained in Section 4.2.4, a mechanism which groups symbolic and concrete constants that are related will be introduced. A single compound value is a group of symbolic and concrete constants whose individual generated values should be related to the previously generated values of all members if the group. This is especially useful when the group of constants represent concepts such like IP addresses, mail addresses, ages, etc. Compound values are represented in G' as symbolic constants (one symbolic constant in G' for each compound value). For each symbolic constant, a group of symbolic and concrete constants is defined in V.

G' and V will be generated to control the instantiation of the `STerm` symbolic constants. Any symbolic constant defined in G' will be instantiated as a set of grouped values defined in V. Thus it should be taken into account in this feature to decide how those symbolic constants should be instantiated

For this example, let us take the following production rules from G:

```
BFactor = ATerm | STerm ;
STerm = column_name sop column_name ;
column_name = "CustomerID" | "CompanyName" | "ContactName" |
     "ContactTitle" | "Address" | "City" | "Region" |
     "PostalCode" | "Country" | "Phone" | "Fax" ;
sop = "=" | "!=" ;
```

A one word approach would be to declare `column_name` and `sop` as symbolic constants in G' and leaving their concrete definitions to appear in V:

**Example G'**

```
BFactor = ATerm | STerm ;
STerm = <column_name> <sop> <column_name> ;
```

**Example V**

```
column_name = "CustomerID" | "CompanyName" | "ContactName" |
     "ContactTitle" | "Address" | "City" | "Region" |
     "PostalCode" | "Country" | "Phone" | "Fax" ;
sop = "=" | "!=" ;
```

This way of defining G' and V will be solved by our solution as three single separate values (`<column_name> <sop> <column_name>`), each one with its own low discrepancy sequence but with no relationship to the other selected values in the other sequences. Clearly, this is not desirable if the terms have a relationship. An alternative would be to solve all three terms together. Let us assume three non-terminals in the `STerm` rule to be solved as a single value. For this G' and V will be defined:

**Example G'**

```
BFactor = ATerm | <STerm> ;
```

**Example V (expands the symbols introduced in G')**

```
STerm = column_name sop column_name ;
column_name = "CustomerID" | "CompanyName" | "ContactName" |
     "ContactTitle" | "Address" | "City" | "Region" |
     "PostalCode" | "Country" | "Phone" | "Fax" ;
sop = "=" | "!=" ;
```

This way of defining G' and V will be solved as a single value, with a single low discrepancy sequence; that is, a relationship is maintained between the terms during the selection of values for that entire (STerm) symbolic constant definition. These different approaches provide the tester with the flexibility to control how symbolic values will be instantiated.

Back to our example, we define the following G':

```
Symbolic Grammar ( (G') for a grammar (G) )
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " BCond ;
BCond = BCond "OR " BTerm | BTerm ;
BTerm = BTerm "AND " BFactor | BFactor ;
BFactor = "NOT " BCond | <ID> "IS NULL " | ATerm | STerm ;
ATerm = Value <AOP> Value ;
Value = <ID> | <NUM> ;
STerm = <ID> "LIKE " <VALUE> | <ID> <SOP> <VALUE> | <ID>
     <SOP> <ID> ;
```

G’ gives a complete description of the protocol for any SQL engine. As established by Majumdar (18) for creating G’, SQL context-free grammar G was taken and substituted h-elements of ϒ, where its regular expression has more than a single concrete value, with h-elements of A ( “Northwind” SQL example database for MS SQL Server (33) will be employed for this purposes). For this example the following symbolic value grammar (V) is defined:

**Symbolic Value Grammar (V)**

```
AOP = "=" | "<" | ">" | "<=" | ">=" | "!=" ;
SOP = "=" | "!=" ;
COLUMNS_NAMES = COLUMN_NAME ”, ” COLUMN_NAME ;
COLUMN_NAME = "CustomerID" | "CompanyName" | "ContactName" |
     "ContactTitle" | "Address" | "City" | "Region" |
     "PostalCode" | "Country" | "Phone" | "Fax" ;
TABLE_NAME = "Customers";
VALUE = (letter)+;
ID = COLUMN_NAME ;
NUM = number ;
number = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
     | "9" ;
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
     | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
     "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" |
     "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
     "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" |
     "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;
```

Suppose a script which fills the table "Customers" (which comes as default in the Northwind (33) installation) with no NULL data. Let’s assume a set of test cases that will test the correctness of an implementation of this script is needed. A set of "SELECT" statements that will query for NULL values will be generated; if any test query returns any non-empty result, an error has been found. To test for this condition, we will generate a set of queries where at least one of any set of random columns has a NULL value. Test cases will be generated with a manageable length, so we chose to generate symbolic sentences with a maximum length of 15 elements – this is an arbitrary selection at this stage. For this, the following symbolic constraint expression (E) is defined:

```
"SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME> "WHERE " <ID>
     "IS NULL " "AND "
```

E establishes the terminals of G’ that should be satisfied while generating symbolic sentences from G’; therefore it only constrains the first terminals, allowing the grammar to generate sentences to a maximum of length 15 as long as they satisfy these constraints.

With G', V and E defined, a description of the different phases implemented in our system can be started. For this the system will execute its three phases in a sequential order.

### 4.4.2 Constraint Solving Phase

The **constraint solving phase**'s first step consists in computing the intersection between the G' and E.

This is performed internally by generating the constraint grammar from the user defined E, replacing each production in the start symbol of G' until the last value in the constraint is satisfied. The symbolic constraint grammar (C') computed would be:

```
T0 = "SELECT " ;
T1 = <COLUMNS_NAMES> ;
T2 = "FROM " ;
T3 = <TABLE_NAME> ;
T4 = "WHERE " ;
T5 = <ID> ;
T6 = "IS NULL " ;
T7 = "AND " ;
```

With C' generated, the intersection with G' is computed as follows:

```
1   input: symbolicGrammar ∈ G', symbolicConstraintGrammar ∈ C'
2   output: constrained symbolic grammar C
3
4   // Make a copy of G'
5   C <- G'
6
7   // Clone the first rule of G' and add it to C
8   r ∈ R'  <- G'.FirstRule
9
10  // Give the rule a name not in G'
11  r.leftHand ∉ F
12
13  // Rename all instances G'.FirstRule in C with r.LeftHand
14  FOR EACH rule ∈ R' IN G'
15   FOR EACH element IN rule
16     IF element = G'.FirstRule THEN
17      element <- r.LeftHand  ;
18     END IF
19    END
20   END
21  END
22
23  // Add r to C
24  C.add(r) ;
25
26  FOR EACH production IN C'
27   thisElement = r(element++);
28
29   DO
30    DO
31     // Gets the next expansion
32
```

```
33      expansion <- thisElement(expansion++);
34
35      // If no more expansions available then unsolvable.
36      IF NO MORE expansions THEN
37       return NULL;
38
39     // Compares the first element in the expansion
40     WHILE expansion(0) ≠ constraintProduction THEN
41
42     // Substitutes element in r with its expansion
43     r(element) <- expansion ;
44     // Redefines the working element with the expansion encountered
45     thisElement <- G.FirstProduction(thisElement);
46
47    UNTIL thisElement ∈ ϒ
48
49   END
50
     return C ;
```

The algorithm undertakes the following steps:

- Lines 8 – 11: The start symbol of G' is copied and renamed as r.
- Lines 14 to 21: All the references to the start symbol in G' are redirected to r. This step is needed to guarantee that the grammar will keep its capability of producing the exact same language L(G') taking into account the restrictions that will be added later by the algorithm.
- Line 24: Add r to C (which holds G' (line 5)). The algorithm continues and it encounters the constraint T0 in C' and (line 27) satisfies it with the first element found in the rule "Select" (no expansion is needed as this element is a terminal value, line 46).
- This is repeated for constraints T1 to T4 (loops through lines 29 - 46) and their corresponding terminal values in the start symbol (S) of G'.
- Then the algorithm expands the first found non-terminal "BCond" (line 32) from G' and creates a new definition of S (line 44) for each found possible expansion that can comply with T5:

```
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " BCond "OR " BTerm ;
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " BTerm ;
```

Two choices are available. The algorithm will attempt to solve both trying to satisfy T5 (line 32), both can eventually solve it. However, as it was established 15 terminal elements as a limit the first choice will eventually be pruned from the search; therefore only the second expansion ("Bterm") will be explicitly explored:

```
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " BTerm "AND " BFactor ;
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " BFactor ;
```

The only expansion that can satisfy the constraint is the first one (BTerm "AND " BFactor). The second one ("BFactor") fails when the algorithm attempts to satisfy the constraint T7, after satisfying T6, as no more elements remain in the start symbol. Expanding the first alternative:

```
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " BTerm "AND " BFactor "AND " BFactor ;
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " BFactor " "AND " BFactor ;
```

For the first expression, a same state to the one found when expanding ("Bterm") is encountered. So for simplicity, only "BFactor" will be expanded, as again, to satisfy the constraint:

```
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " "NOT " BCond " "AND " BFactor ;
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " <ID> "IS NULL " "AND " BFactor ;
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " ATerm " "AND " BFactor ;
Select = "SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME>
     "WHERE " STerm " "AND " BFactor ;
```

From the four choices, the only expansion that cannot solve the constraint ("IS NULL") is the first one "NOT" as it is a terminal element that cannot be expanded. Therefore, it is pruned from the search. The algorithm will attempt to solve the remaining sentences, expanding the non-terminal elements (ATerm and STerm through lines 30 - 39). The expansion for these three sentences would be:

```
Select = "SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE
     " <ID> "IS NULL " "AND " BFactor ;
Select = "SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE
     " <ID> <AOP> Value " "AND " BFactor ;
Select = "SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE
     " <ID> "LIKE " <STRING>" "AND " BFactor ;
Select = "SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE
     " <ID> <SOP> <STRING> " "AND " BFactor ;
Select = "SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE
     " <ID> <SOP> <ID> " "AND " BFactor ;
```

The next step in the algorithm is to satisfy T6 "IS NULL " (line 27). The only available choice that satisfies this constraint is the first one, so the

algorithm abandons the other searches upon discovering that they are unsatisfiable-able. Finally, the algorithm will attempt to satisfy the T7 "AND ", which is satisfied by the remaining sentence:

```
Select = "SELECT” <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE
     " <ID> "IS NULL " "AND " BFactor ;
```

As all the constraints are satisfied the new symbolic constrained grammar C will look like this:

**Constrained Symbolic Grammar**

```
Select = "SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE
     " <ID> "IS NULL " "AND " BFactor ;
Select’ = "SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME>
     "WHERE " BCond ;
BCond = BCond "OR " BTerm | BTerm ;
BTerm = BTerm "AND " BFactor | BFactor ;
BFactor = "NOT " BCond | <ID> "IS NULL " | ATerm | STerm ;
ATerm = Value <AOP> Value ;
Value = <ID> | <NUM> ;
STerm = <ID> "LIKE " <STRING> | <ID> <SOP> <STRING> | <ID>
     <SOP> <ID> ;
```

Taking a closer look to the grammar, the nonterminal Select’ (created in line 11) is never used by any rule making it an unreachable rule. Because of this, a final step that removes any non-reachable rules is taken to optimize symbolic string generation. The result of this phase is:

**Constrained Symbolic Grammar**

```
Select = "SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE
     " <ID> "IS NULL " "AND " BFactor ;
BCond = BCond "OR " BTerm | BTerm ;
BTerm = BTerm "AND " BFactor | BFactor ;
BFactor = "NOT " BCond | <ID> "IS NULL " | ATerm | STerm ;
ATerm = Value <AOP> Value ;
Value = <ID> | <NUM> ;
STerm = <ID> "LIKE " <STRING> | <ID> <SOP> <STRING> | <ID>
     <SOP> <ID> ;
```

C is now a grammar that complies with both G’ and E, and is ready to be used in the next phase.

### 4.4.3 Symbolic Sentences Generation Phase

In the **symbolic sentences generation phase,** an implementation of the approach suggested by McKenzie (24) is utilised. This approach proposes a uniform distribution for production rules that maximizes production rule coverage (making it imperative to use all production rules)

and guarantees that all strings from the given grammar are equally likely to be generated. As our approach objective is to employ any CFG this approach is our best option as all production rules are treated without bias.

In this approach, the production rules (R) defined for any context-free grammar (G) are used to generate symbolic sentences (SS) up to a specified length (d) in a uniform fashion. Our solution enhances previous approaches (19) by preventing the re-generation of previously generated symbolic sentences.

For this example, as E consists of 10 terminal elements, five or less additional terminal elements are required in our test cases (remember, 15 elements was set as a maximum size constraint).

The `GenerateFronNonTerminal` algorithm starts as follows:

```
1   input: nonterminal ∈ F, d (length of SS)
2   output: string ∈ SS
3
4   Let q ∈ Υ+
5   Let r'⊆ R' be of the form nonterminal → q;
6   Let λ be an example of r' and be referred to as an expansion of
    nonterminal.
7
8   // Chooses an expansion from all selectable expansions.
9   result ∈ r' <- Choose(nonterminal, d);
10  // Expands the nonterminal with the chosen expansion
11  string <- Expand(result , q(0), d);
12
13  return string;
```

- It chooses an expansion of the element received (line 9). The `Choose` routine will be discussed later.
- It expands the first element `q(0)` in the selected production rule `result` (line 11).

For our example, the start symbol ("select") will be used as a starting point and an arbitrary length, between the length of the start symbol (11) and the previously defined maximum length (15), for which 12 is chosen.

The routine receives "select" and the length 12 and chooses between all of the possible expansions of "select". The start symbol for C has only one expansion, so this will be selected (line 9) and the start symbol will be expanded by the function Expand (line 11).

The algorithm for Expand works as follows:

```
1   input: rule ∈ R', element(i) ∈ q, d (length of SS)
2   output: result ⊆ SS
3
4   s1 ⊆ SS <- "";
```

```
5   s1 ⊆ SS <- "";
6
7   IF element ∉ F THEN
8       s1 <- element;
9
10      // If more elements continue expanding
11      IF more elements IN rule to the right of this element THEN
12        nextElement ∈ q  <- element(i+1) ;
13        // Expand recursively
14        s2 <- Expand(rule, q, d - 1);
15      END
16
17      // Concatenate and return results
18      return s1 + s2 ;
19  ELSE
20      // Call function GenerateFronNonTerminal
21      s1 <- GenerateFromNonterminal (element, d + 1)
22
23      // If more elements continue expanding
24      IF more elements IN rule to the right of this element THEN
25        nextElement <- element(i++);
26        // Expand recursively
27        s2 <- Expand(rule, nextElement, d);
28      END
29
30      // Concatenate and return results
31      return s1 + s2 ;
32
33  END
```

- If the `element` is a `terminal` (line 7), then it stores the value of the `element` in `s1` (line 8); if more elements are to the right (line 11), it expands them recursively and stores the computed values in `s2` (line 14).
- If the `element` is a `nonterminal` (line 19), then it calls `generateFromNonTerminal` and stores the value of the element in `s1` (line 21); if more elements are to the right (line 24), it expands them recursively and stores the computed values in `s2` (line 27).
- Finally, it concatenates the computed values (lines 18 and 31)

Continuing with our example, the expansion selected is shown:

```
"SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE " <ID>
      "IS NULL " "AND " BFactor
```

The element selected is "SELECT " (line 1) and as it is a symbolic constant (terminal value for C) its evaluation is true for line 7. Now the algorithm looks for the next element (to the right of this one) and as this evaluation is true (line 11), it takes this element and tries to expand it recursively in line 14. For this call on Expand, the selected expansion is now <COLUMNS_NAMES> which is a symbolic constant and a terminal for C; thus as for the previous case, it will go through lines 7 and 11, with the difference that on line 14 it will chose the next element "FROM ".

The algorithm will continue parsing the rule and adding results recursively to `s1` and `s2` until a symbolic sentence is generated as final result.

```
"SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME> "WHERE " <ID>
     "IS NULL " "AND " <COLUMN_NAME> "IS NULL "
```

The `Choose` routine which was employed in the `GenerateFromNonTerminal` and `Expand` algorithms will be explored.

The following is the algorithm of this routine which holds our enhancement which introduces evolutionary computation principles to the selection of production rules.

```
1   input: nonterminal ∈ F, d (length of SS)
2   output: element ∈ q
3
4   Let q be an expansion of nonterminal;
5   Let UC be a count variable
6   // Fetches list containing the expansions and its use count for
7   // this nonterminal and d.
8   [q, UC] <- usageList(nonterminal, d);
9
10  // Calculates a random number between 0 - 1 to use for operation
    selection.
11  decider <- Random(0, 1);
12
13  // Decides which operation to execute.
14  IF decider < stochastic percentage  THEN
15        element <- uniform(q);
16  ELSE
17        // Select a random element from the list.
18        element <- random(q);
19  END
20
21  // Updates list with the new selected expansion.
22  usageList.Update(element, d);
23
24  return element;
25
```

- It uses `usageList` (line 8) a list which is indexed by a non-terminal (left-hand side of an expression) and the length of a symbolic sentence which is derivable from the non-terminal. The list returns a set of pairs $[q \in \Upsilon^+, UC]$, where UC is the total number of times that q has been selected by this algorithm.
- It decides if it should continue with the original flow or apply a random selection mechanism (this is a novel element in our approach). For this, it compares a pseudo-random value (line 8) with a stochastic percentage (user-defined variable). The exact value of this parameter is relatively unimportant and the system default value produces acceptable results in all known situations. If the random value is less than the percentage, it selects the alternative operation; otherwise it continues. If increased

stochastic behaviour is required the stochastic percentage could instead follow a cooling schedule to allow an increased selection of pseudo-random values at the start of the cycle which then decreases as the cooling schedule is applied.

- The routine `uniform` selects an element based upon:

  Let $\overline{UC} = \frac{1}{\#q} \sum_q UC_q$ , where # is the cardinality of q

  $$\exists q \cdot \overline{UC} > UC_q \wedge \min_q \left\| UC_q - \overline{UC} \right\|_2$$

- The stochastic branch (line 16) selects an element at random from the list (line 18).
- The list is updated with the selected expansion (line 22) and the element selected is returned (line 24).

Back to our example, a set of five symbolic sentences generated by our system in these circumstances is shown:

**Symbolic Sentences**

```
"SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE " <ID>
     "IS NULL " "AND " <COLUMN_NAME> "IS NULL "
"SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE " <ID>
     "IS NULL " "AND " <NUM> <AOP> <COLUMN_NAME>
"SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE " <ID>
     "IS NULL " "AND " "NOT "  <COLUMN_NAME>  "IS NULL "
"SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE " <ID>
     "IS NULL " "AND " "NOT " "NOT " <COLUMN_NAME> <AOP>
     <NUM>
"SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE " <ID>
     "IS NULL " "AND " "NOT " "NOT " "NOT " <COLUMN_NAME>
     "IS NULL "
```

If more symbolic sentences are required then the routines are executed iteratively as many times as defined (by the user). For future iterations, new length values are generated to be passed to the `GenerateFromNonTerminal` routine.

### 4.4.4 Concrete Sentence Production Phase

Finally, the **concrete sentence production** phase is initiated where concrete sentences (CS) are generated from the symbolic sentences (SS) employing the symbolic value grammar (V).

The phase starts with the following algorithm:

```
1   input: grammar ∈ V, sentences ∈ SS, nr (total number of concrete
    sentences to be generated)
2   output: testCases ∈ CS
3
4   // Parse each element in the symbolic sentence
```

```
5   FOR EACH element ∈ ϒ IN SS
6          // If it is a symbolic constant
7          IF element ∈ Λ THEN
8                 results.adds(SolveSymbolicConstant(V, element, nr))
9          ELSE IF element ∈ Σ THEN
10                results.adds(element);
11         END
12  END
13
14  // Convert results to a readable output
15  testCases <- ConstructConcreteSentence(results)
16
17  return testCases
18
```

Let's take the first symbolic sentence of our example:

```
"SELECT " <COLUMN_NAME> "FROM " <TABLE_NAME> "WHERE " <ID>
      "IS NULL " "AND " <COLUMN_NAME> "IS NULL "
```

Let's assume nr has a value of 5. In the first iteration, the algorithm parses the concrete value `"SELECT "` and adds its value to the results variable (line 10). In the second iteration, when the parser finds the symbolic constant `<COLUMNS_NAMES>`, it now calls the routine `SolveSymbolicConstant` (line 8), explained below, which will compute the concrete value for this symbolic constant. To continue the explanation, let us suppose that the returned value is the symbolic constant conformed of the concrete constants `"City" "," "Region"` , so this value is added to the results. This is repeated until the entire symbolic sentence is parsed.

```
    SolveSymbolicConstant

1   input: grammar ∈ V, nonterminal ∈ W, nr (total number of
2          concrete sentences to be generated)
3   output: results ∈ Λ
4
5   Let β ⊆ P and have the form W → Λ+
6   Let η = nonterminal ∈ Λ → β+
7
8   Let entity ∈ β+ be the right hand of η
9   Let sequence be a low discrepancy sequence • ∀entity •(∃ sequence • entity
    → sequence)
10
11  Let T be the set of permutations of an entity
12  Let τ ∈ T • ∀τ •(∃ seq ∈ sequence • τ → seq))
13
14
15  // Compute the entity for this nonterminal
16  entity <- GenerateEntity(grammar, nonterminal)
17
18  // Solve the entity
19  FOR EACH position IN entity
20         // Fetch next item in the low discrepancy sequence
21         concreteValues ∈ τ <- values(entity.sequence)
22         // Add values to results.
23         results.add(concreteValues)
24  END
25
```

```
26  // size of results = nr
27  return results
28
```

Consider the following definition of discrepancy. Given a sequence of numbers which belong to the right-open interval [0,1), and can be scaled if necessary. Consider the s-dimensional right-open unit cube $\mathrm{I}^s$ $[0,1)^s$; $s \geq 1$. For N points $\{x_1, x_2, ....., x_N\} \in \mathrm{I}^s$; and a subinterval $J \in \mathrm{I}^s$; if A(J) counts the number of points $x_i \in J$ and V(J) is the volume of J , discrepancy D(J,N) is defined as:

$$\mathrm{D(J,N)} = \left| \frac{A(J)}{N} - v(J) \right|$$

The discrepancy is the difference between the proportion of points in J compared to the full unit cube $\mathrm{I}^s$ and the volume of the 'box' J compared to $\mathrm{I}^s$.

The worst-case discrepancy, i.e. the worst-case distribution of a set/sequence of points$\{ x_1, x_2, ....., x_N \} \in \mathrm{I}^s$ is called the *star-discrepancy* and is defined as:

$$\mathrm{D^*(N)} = \max_{\mathrm{J}} \mathrm{D(J,N)}$$

The goal of a *low-discrepancy* sequence is to minimize this star discrepancy. This definition is utilized by the algorithm.

For our example, `<COLUMNS_NAMES>` is the first symbolic constant to be solved. First, the entity for COLUMNS_NAMES is generated; as the production rule COLUMN_NAME satisfies the entity definition (line 8) then the entity is:

```
<COLUMNS_NAMES> = COLUMN_NAME ", " COLUMN_NAME;
```

The entity now is solved. The algorithm fetches the values for this position in the entity using its associated low discrepancy sequence (line 21). As this is repeated 5 times, 5 values are added to results (line 23):

```
"City"          ", "   "Address"
"ContactTitle"  ", "   "Phone"
"Country"       ", "   "Region"
"ContactName"   ", "   "CostumerID"
"Address"       ", "   "CompanyName"
```

It can be appreciated that the symbolic constants have been substituted with their corresponding definitions in V, but the values chosen cover more input space than pure pseudo-random approaches. This is more easily appreciated if 20 concrete sentences are generated in the same fashion:

```
SELECT City, Address FROM Customers WHERE Phone IS NULL AND PostalCode IS NULL
SELECT ContactTitle, Phone FROM Customers WHERE Region IS NULL AND Fax IS NULL
SELECT Country, Region FROM Customers WHERE CustomerID IS NULL AND City IS NULL
SELECT ContactName, CustomerID FROM Customers WHERE CompanyName IS NULL AND
ContactTitle IS NULL
SELECT Address, CompanyName FROM Customers WHERE PostalCode IS NULL AND Country
IS NULL
SELECT Phone, PostalCode FROM Customers WHERE Fax IS NULL AND ContactName IS
NULL
SELECT Region, Fax FROM Customers WHERE City IS NULL AND Address IS NULL
SELECT CustomerID, City FROM Customers WHERE ContactTitle IS NULL AND Phone IS
NULL
SELECT CompanyName, ContactTitle FROM Customers WHERE Country IS NULL AND Region
IS NULL
SELECT PostalCode, Country FROM Customers WHERE ContactName IS NULL AND
CustomerID IS NULL
SELECT Fax, ContactName FROM Customers WHERE Address IS NULL AND CompanyName IS
NULL
SELECT Country, Fax FROM Customers WHERE City IS NULL AND ContactTitle IS NULL
SELECT ContactName, City FROM Customers WHERE ContactTitle IS NULL AND Country
IS NULL
SELECT Address, ContactTitle FROM Customers WHERE Country IS NULL AND
ContactName IS NULL
SELECT Phone, Country FROM Customers WHERE ContactName IS NULL AND Address IS
NULL
SELECT Region, ContactName FROM Customers WHERE Address IS NULL AND Phone IS
NULL
SELECT CustomerID, Address FROM Customers WHERE Phone IS NULL AND Region IS NULL
SELECT CompanyName, Phone FROM Customers WHERE Region IS NULL AND CustomerID IS
NULL
SELECT PostalCode, Region FROM Customers WHERE CustomerID IS NULL AND
CompanyName IS NULL
SELECT Fax, CustomerID FROM Customers WHERE CompanyName IS NULL AND PostalCode
IS NULL
```

### 4.4.5 Pseudo-Random vs. Quasi-Random

A comparison between our approach using low discrepancy sequences and pure pseudo-random approaches will be made (as all previous approaches have utilised). Taking our example and comparing it with our system working with a symbolic grammar, symbolic values grammar and symbolic constraint expression similar to the previously used. The change in the symbolic grammar can be found in the rule "Select" where the `<COLUMNS_NAMES>` symbolic constant has been replaced with two symbolic (and one concrete) constants `<COLUMN_NAME> ", " <COLUMN_NAME>` (found in the example's Symbolic Value Grammar):

**Symbolic Grammar**

```
Select = "SELECT " <COLUMN_NAME> ", " <COLUMN_NAME> "FROM "
     <TABLE_NAME> "WHERE " BCond ;
```

**Symbolic Constraint Expression**

```
"SELECT " <COLUMN_NAME> "," <COLUMN_NAME> "FROM "
     <TABLE_NAME> "WHERE " <ID> "IS NULL " "AND "
```

As discussed, the manner in which these values are defined affects directly how they are solved by our solution. In the previous example, these constants were solved as a single entity for this definition they will be solved as three independent entities.

```
SELECT City, Country FROM Customers WHERE Address IS NULL AND Fax IS NULL
SELECT ContactTitle, ContactName FROM Customers WHERE Phone IS NULL AND City IS
NULL
SELECT Country, Address FROM Customers WHERE Region IS NULL AND ContactTitle IS
NULL
SELECT ContactName, Phone FROM Customers WHERE CustomerID IS NULL AND Country IS
NULL
SELECT Address, Region FROM Customers WHERE CompanyName IS NULL AND ContactName
IS NULL
SELECT Phone, CustomerID FROM Customers WHERE PostalCode IS NULL AND Address IS
NULL
SELECT Region, CompanyName FROM Customers WHERE Fax IS NULL AND Phone IS NULL
SELECT CustomerID, PostalCode FROM Customers WHERE City IS NULL AND Region IS
NULL
SELECT CompanyName, Fax FROM Customers WHERE ContactTitle IS NULL AND CustomerID
IS NULL
SELECT PostalCode, City FROM Customers WHERE Country IS NULL AND CompanyName IS
NULL
SELECT Fax, PostalCode FROM Customers WHERE ContactName IS NULL AND PostalCode
IS NULL
SELECT ContactName, Country FROM Customers WHERE Fax IS NULL AND Fax IS NULL
SELECT Address, ContactName FROM Customers WHERE City IS NULL AND City IS NULL
SELECT Phone, Address FROM Customers WHERE ContactTitle IS NULL AND ContactTitle
IS NULL
SELECT Region, Phone FROM Customers WHERE Country IS NULL AND Country IS NULL
SELECT CustomerID, Region FROM Customers WHERE ContactName IS NULL AND
ContactName IS NULL
SELECT CompanyName, CustomerID FROM Customers WHERE Address IS NULL AND Address
IS NULL
SELECT PostalCode, CompanyName FROM Customers WHERE Phone IS NULL AND Phone IS
NULL
SELECT Fax, PostalCode FROM Customers WHERE Region IS NULL AND Region IS NULL
SELECT City, Fax FROM Customers WHERE CustomerID IS NULL AND CustomerID IS NULL
```

The difference may not be obvious given that both answers are valid. For making an informal comparison, the Cartesian product of the set of values of <COLUMN_NAME> (defined in the symbolic values grammar) will be employed. Ordered pairs will be represented in a table, where the first set (first occurrence from left to right) will be the rows and the second set (second occurrence from left to right) will be the columns.

For this comparison the first 60 tuples were generated (half the input space) and then we draw them in the table previously defined. The results

for three approaches: pseudo-random, low discrepancy single values and low discrepancy entity value will be analyzed.

Figure 4.3 shows the distribution for the pseudo-random approach (using the grammar defined in this section) where the symbolic constants `<COLUMN_NAME> ", " <COLUMN_NAME>` were solved using the pseudorandom approach used by all other, systems of this type:

| RANDOM | Address | City | CompanyName | ContactName | ContactTitle | Country | CustomerID | Fax | Phone | PostalCode | Region | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Address | • | | • | • | | | ● | | | | | 5 |
| City | | | • | • | | | | | • | | • | 4 |
| CompanyName | | | • | • | | • | | | | • | • | 5 |
| ContactName | | | | | | | | | • | | ● | 4 |
| ContactTitle | | | | ● | • | | • | | • | | | 5 |
| Country | | | | • | • | • | | | • | | • | 5 |
| CustomerID | • | | | | | ● | | | | | | 3 |
| Fax | | | | | | | • | • | ● | • | • | 6 |
| Phone | | • | ● | • | • | | | | • | | | 8 |
| PostalCode | • | | | | | • | • | | ● | • | | 6 |
| Region | • | | ● | • | • | | • | • | • | | • | 9 |
| | 4 | 1 | 9 | 8 | 4 | 5 | 6 | 2 | 10 | 3 | 8 | |

**Figure 4.3 Distribution for the pseudo-random approach.**

The numbers on the edges of the table show that the distribution is randomly biased through the columns where the value "City" appears with 1 occurrence while "CompanyName" and "Phone" appear with 10 concurrencies. The large circles represent coordinates with more than one count, so it is clear that for example in the ordered pair <"Phone", "CompanyName"> many test cases hit the same pairs making only 49 out of 60 test cases unique. The figure also shows that there are large areas that are poorly tested like the area defined by <"City","Address">: <"Fax","CompanyName"> which represents 20% of the total input space.

Figure 4.4 shows the distribution for the low discrepancy single approach (using the grammar defined in this section) where the symbolic constants `<COLUMN_NAME> ", " <COLUMN_NAME>` were solved using our low discrepancy proposal solving each symbolic constant separately:

| Single | Address | City | CompanyName | ContactName | ContactTitle | Country | CustomerID | Fax | Phone | PostalCode | Region | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Address | | | | • | ● | | | | • | | • | 5 |
| City | | | ● | | • | • | | • | | | | 5 |
| CompanyName | | | | | | | • | • | ● | ● | | 6 |
| ContactName | • | • | | | | • | | | • | | • | 5 |
| ContactTitle | | • | | • | | • | | | | ● | | 5 |
| Country | • | | | • | • | | | ● | | | | 5 |
| CustomerID | ● | | ● | | | | | | | • | • | 6 |
| Fax | | ● | | | • | | ● | | | • | | 6 |
| Phone | • | | | | | ● | • | | | | • | 5 |
| PostalCode | | • | • | | | | | ● | | | ● | 6 |
| Region | | | • | ● | | | ● | | • | | | 6 |
| | 5 | 5 | 6 | 5 | 5 | 5 | 6 | 6 | 5 | 6 | 6 | |

**Figure 4.4 Distribution for the low discrepancy single approach.**

The numbers on the edges show that the distribution now is more evenly distributed (all values on the edges are now between five and six). This is achieved by the low discrepancy sequences used to make the selections. As in the previous table, the large circles represent ordered pairs which occur more than once; an improvement in how the repeated values are distributed 45 out of 60 test cases is unique with a maximum of two repeated cases for any combination can be appreciated. The graph also shows that there are not "large areas" that are poorly tested as was experienced in the random approach; this improves the overall coverage of the input space with less test cases covering more input space.

Figure 4.5 shows the distribution for the low-discrepancy entity approach (using the grammar defined in the working example) where the symbolic constant `<COLUMNS_NAMES> is` solved using the low discrepancy sequences and the symbolic constant is considered as a single entity value.

| ENTITY | Address | City | CompanyName | ContactName | ContactTitle | Country | CustomerID | Fax | Phone | PostalCode | Region | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Address | | • | | | | • | • | | • | • | | 5 |
| City | | • | • | | | | | • | • | | | 4 |
| CompanyName | | | • | | | • | • | • | • | | | 5 |
| ContactName | • | | • | • | • | | | • | | | • | 6 |
| ContactTitle | • | • | | • | | • | | | | • | • | 6 |
| Country | | | | | | | • | | • | • | | 3 |
| CustomerID | • | • | | | | • | • | | • | • | | 6 |
| Fax | • | | • | • | • | | | • | | | • | 6 |
| Phone | | • | • | • | | | • | • | • | | | 6 |
| PostalCode | • | | | • | • | • | | | | • | • | 6 |
| Region | • | • | • | • | • | | | • | | | • | 7 |
| | 6 | 6 | 6 | 6 | 4 | 5 | 5 | 6 | 6 | 5 | 5 | |

**Figure 4.5 Distribution for the low discrepancy entity approach.**

The numbers on the edges show that the distribution remains even due to the low discrepancy sequence used when solving the entity. As before, the large circles represent coordinates which occur more than once; it can be seen that there are no bold circles indicating that all the test cases produced are unique. The table also shows that there are no large areas that are poorly tested, optimizing the previous example by covering more input space with the same generated test cases.

It is now clear that our approach is significantly better than pure pseudo-random enumeration usually employed when exhaustive generation is not feasible. It has also been demonstrated that our approach provides a better coverage of the input space when values are related between them and solved as a single entity, which is novel to this kind of test case generation.

## 4.4.6 Conclusions

A walkthrough of our solution was given starting with its prerequisites and through its three phases giving a detailed insight of the computations occurring at each phase. To finalize, a summary of the solution is presented:

- The main functionality of our system consists of three sequential phases: **constraint solving**, **symbolic sentence generation** and **concrete sentence production**. This phases use a:

- S*ymbolic grammar* - abstract representation of a context-free grammar (that represents the concrete input syntax of the system under test) where some original terminals are replaced with symbolic constants (regular expressions that represent the entire solution space for the replaced terminals). Each rule in the symbolic grammar represents a set of CFG rules, where each symbolic constant will be later instantiated with concrete constants (terminals) during execution.
- *Symbolic values grammar* - a grammar where symbolic constants are defined with concrete constants (terminals); it is used to instantiate symbolic constants during execution.
- *Symbolic constraint expression* - a regular expression expressed in terms of symbolic constants and concrete constants that the generated symbolic sentences should satisfy.

- To initiate a description of the system under test (SUT), input is constructed in the form of a **symbolic constraint expression,** which is a description of the testing objectives. Our system already has a description of the SUT's language or protocol in the form of a *symbolic grammar* and a *symbolic values grammar*.
- The *symbolic constraint expression* is supplied to the system.
- The system passes *symbolic constraint expression* to the **constraint solving phase,** which restricts the *symbolic grammar* by forcing it to also conform to the supplied *symbolic constraint expression*.
- The output of the **constraint solving phase** is passed to the **symbolic sentences generation phase** (part enumeration part constraint solving). Several specific terms match each more generic term; therefore we get enumeration which is controlled by the objective *uniform distribution*.
- The output of **symbolic sentences generation phase** is passed to the **concrete sentence production phase** (part enumeration part constraint solving). The more generic output of the **symbolic sentences generation phase** is forced to bind to *concrete values* as these are the final testing values. Because, in general, several specific terms match each more generic term we get enumeration. The enumeration is controlled by the objective l*ow-discrepancy sequence* when applicable or uncontrolled (random generation) when the objective is inapplicable.

This section has established how our enhancements (like evolutionary algorithms) and novel approaches (like low discrepancy sequences) mix together to form a new and promising testing tool for generating test cases based on symbolic grammars.

# 5 Future Work

As we developed and tested our system, several details came to our attention as interesting research fields which go beyond this thesis.

## 5.1 Specifying System Constants

In section 4.4.1, a starting sentence length was needed to control the symbolic sentence generation phase. Our criterion for selecting it was arbitrary as we didn't employ any formal method to compute a starting length.

Similarly in section 4.4.3, we specified that the user must provide the system with the total number of desired symbolic sentences. We have found that the total number of symbolic sentences is not fixed; neither can it be determined taking into account the size of the system, neither the nature of the software nor the objectives of the testing case. When possible, we suggest that an enumeration of all symbolic sentences from the symbolic grammar should be made and to use these computed values as a starting number that can be reduced in future iterations depending on the testing results. For small and simple grammars enumeration is possible due to their nature, but even with this advantage, enumeration is not possible when using complex grammars, for example grammars that contain recursive rules.

Concrete sentences are discussed on section 4.4.4 and this raises the question of how many concrete sentences should be instantiated from each symbolic sentence.

Research should be done in this field in order to remove human interaction and to aid the tester to set these parameters with a formal method.

## 5.2 Constraint Structure

Constraints were discussed in section 4.4.2 where their importance was established as a mechanism for limiting the size of the solution space. The question raises about which and how many symbolic and concrete constants should be included in the initial constraint.

A well-formed input was used as an initial value (in section 4.4.1) to set the rule elements and the length employing an explorative testing approach. This kind of approach is not formal and different constraints could have given similar results. It can be argued that the selected approach is not optimal. For example, the used constraint was:

```
"SELECT " <COLUMNS_NAMES> "FROM " <TABLE_NAME> "WHERE " <ID>
     "IS NULL " "AND "
```

This was derived from the test case specification, but in practice test case specifications are rarely detailed enough to derive the test case to this extent. This leaves the problem of defining the constraint structure entirely to the tester; clearly, there is a need to guide the tester in this regard.

## 5.3 Constraint Solving

The constraint solving literature (17) (19) (34) (35) omit details about constraint solving heuristics; these details include the order in which constraints should be solved. Usually linear constraint solving (first term followed by second term followed by third term) is assumed and found in different implementations.

In the symbolic sentences generation phase, we utilise an implementation of the approach suggested by McKenzie (19) (34). This approach solves constraints from left to right in consecutive order, which is common practice while working with grammar-based approaches. However, these implementations may fail when left-recursion is present, thus forcing a grammar simplification before entering the constraint solving phase.

Research on alternatives to linear constraint solving is a promising field as left-recursion is present in many non-context-free grammars. With these enhancements the system might be extended from using only BNF representation for CFG widening its industrial application.

# 6 Case Study

In this section, a comparison between our solution and previous tools that employ grammar-based approaches in a real-world situation will be presented. For this, an overview of the system to test will be given, and then a description of our evaluation criteria, and finally the generated empirical results will be discussed.

## 6.1 Firewalls

A system or group of systems that enforces an access control policy between two networks is called a firewall. It implements a network access policy by forcing connections to pass through the firewall, where they can be examined and evaluated.

Nowadays corporations face a variety of information system attacks against their local area networks (LANs) and wide area networks (WANs). Many of these attacks come from the Internet in three basic groups:

- People which find a technological challenge in attacking a corporation's network;
- People which find a promising field for high-tech vandalism in attacking a corporation's network; and
- People associated with a competitor with an agenda who see the corporation’s network as a strategic target

A firewall is then a fundamental asset of network security. It allows the corporation to focus its security efforts where the corporation’s information system connects to the Internet. Some of tasks that can be performed by a firewall are:

- The control and prevention of attacks from untrusted network services.
- Monitoring of traffic passing through the system.
- Audit the corporation’s network
- Alerts to the corporation’s staff of anomalies within the network.
- Logging features and statistic’s collection (which can be used to create a network profile).
- Providing the ability to control access to internal systems.

A cornerstone of a firewall’s functionality is its capability for implementing and enforcing a network access policy. As a firewall provides access control to users and services, a network access policy can be enforced.

Firewalls can only protect against attacks going through the firewall. For example, there are many organizations investing in expensive firewalls and neglecting the numerous other backdoors into their network. A firewall must be part of a consistent overall organizational network security architecture.

## 6.2 Firewall Policies

The main function of a firewall is to centralize access control, so it allows the implementation of a security policy to ensure security within an organization. The centralized access control divides the local network into two separate parts: the trusted and the untrusted. This approximation allows the firewall to effectively manage the resources within the trusted network and safeguard them from the untrusted network.

A firewall policy consists of a set of conditions represented by a list of rules. The workflow is simple; whenever there is an incoming packet (regardless its origin), it is analyzed and the firewall decides if it should be let through or blocked based on the firewall's policy.

Firewalls suffer from the same quality problems as other software products; even though trusted vendors are generally trustworthy and vulnerabilities tend to be patched relatively quickly. The main concern regarding firewall security testing is that firewalls must be configured with a rule set that implements an appropriate security policy for the actions that the organization wants the firewall to perform. Therefore the firewall policies are always costume made thus, the likelihood of a security vulnerability arising from misconfiguration is much greater than from a bug in the software itself (36).

Even small companies have firewalls with several hundred rules. Furthermore, firewall policies and, by definition, rule sets change with time. Changes are often implemented by adding or changing rules, resulting in more complexity which increases the probability of errors.

As the main vulnerability found in a firewall is its policy, the importance of producing test cases based on firewall policies is an obvious starting point for firewall testing.

### 6.2.1 Firewall Rules

The list of rules that comprise a firewall policy have different structures depending on the implementation of the firewall. Even though all firewall rules, regardless the vendor, include at least the following common fields:

- Source address - the address (i.e. IP or domain) from where the package origins.

- Source port - the port number from where the package origins in the defined source address.
- Destination address - the address (i.e. IP or domain) that the package is intended to reach.
- Destination port - the port that the package is intended to reach on the defined destination address.
- Protocol - the protocol in which the message is encoded.
- Action - generally speaking the action will be to deny or allow the transit of a package from the source to the destination.

| | Src. Addr. | Src. Port | Dst. Addr. | Dst. Port | Protocol | Action |
|---|---|---|---|---|---|---|
| R1: | 198.32.24.87 | 50 | localhost.com | 30 | ICMP | Allow |
| R2: | 198.32.25.* | 80 | localhost.com | 85–90 | IPv6 | Allow |
| R3: | localhost.com | 80 | * | 80 | * | Allow |
| R4: | * | * | * | * | * | Deny |

**Table 6.1 Generic firewall policy**

The generic firewall policy shown in Table 6.1 consists of 4 simple rules:

- Rule one allows all incoming packages to port 30 from a specific source address and port with protocol ICMP.
- Rule two allows all incoming packages to port 80 from all addresses within the range 198.32.25.0 - 198.32.25.255 on ports 85 to 90 with protocol IPv6.
- Rule three allows all outgoing packages from port 80 to any destination address on port 80.
- Rule four denies all incoming packages that don’t match any of the above.

Policy rules order is very important as the packet filtering process is performed by sequentially matching the packets information with the policy rules beginning with the first one and continuing sequentially until they reach the final rule (which usually is of the type of Rule four – denying anything that doesn’t matches). This doesn't happen in some firewalls where filtering rules are disjoint meaning that the ordering of the rules is no considered.

Firewalls that work in a sequential matching fashion are the most common approach, so it is common to have policy rules that are related. In this case, the ordering of rules becomes a priority and a security issue. If the related rules are not carefully ordered, some rules will result in a firewall policy anomaly.

A firewall policy anomaly is defined as the existence of two or more rules that filter the same set or a subset that another rule filters. Firewall policy anomalies generally fall into four categories (37):

- Shadowing anomaly - a rule is said to be shadowed when a previous rule matches all the packets that match this rule, such that the shadowed rule won't be reached.
- Correlation anomaly - two rules are correlated if they have different filtering actions and those actions overlap a subset of each rule's filtered packets.
- Generalization anomaly -a rule is a generalization of a preceding rule if one rule filters all of the packets that the other rule filters.
- Redundancy anomaly - a rule is said to be redundant if there is another rule that filters the exact same packages as another one.

As firewall policies keep growing in size (usually hundreds of rules), the importance of tests that are based on firewall policies is essential.

### 6.2.2 Firewall Testing

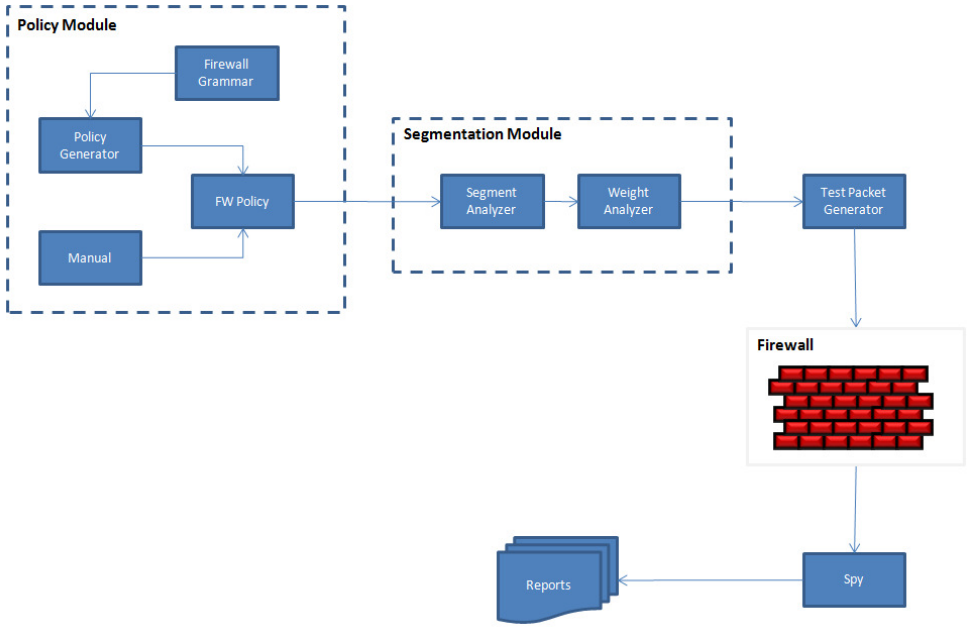



**Figure 6.1 Firewall testing framework proposed by Al-Shaer etal.**

Al-Shaer et al. (38) have proposed one of the most complete frameworks for firewall testing which is illustrated in Figure 6.1. It consists in the following logical divisions:

- Policy Module - the system described accepts a file that has firewall policies that comply with a given grammar. To produce this file two options are given, the first one is to use a policy generator, the second one is to produce manually the file and feed it to the system.
- Segmentation Module - policy rules are separated into segments that will cover all the rules in the testing policy. A segment is a subset of the complete address space. These segments are formed by dividing the total rules in the policy into groups that have similar traits. For a detailed discussion of segments; see Al-Shaer et al. (38).
- Test Packet Generator - responsible for generating the actual packages for testing (producing as many packages as needed to cover all the segments with a bias set by the segment importance).
- Spy - a standalone sniffer that receives the original package sent from the generator and the packets that come from the firewall. Both are received as bitmaps, when the test is over the bitmaps are compared and a final bitmap is sent to the framework for analysis.
- Reports - The results from the framework are analyzed and several reports can be made including: policy complexity, package generation and segmentation effectiveness.

From this model, a simplified firewall testing framework should consist of the following two phases:

1. Generating policies with different configurations that include different rule complexity, interaction, etc.; and
2. The generation of packets to test the implementation of the system under test (SUT) using the previously generated policies.

Generating firewall policies in an automated fashion presents the following challenges:

- The policy generator should be applicable to different firewall configurations and specifications unique of each vendor.
- The policy generator should be able to cover the rule configurations completely (field values, rule complexities, rule relations, etc.).

Firewall testing falls into two approaches:

- A static one where the firewall policy is analyzed and modified to optimize it.
- A dynamic one where the firewall is subjected to user designed traffic and monitored for anomalies.

Even though policies are the starting points for any testing regarding firewalls, little has been written regarding policy generation for firewall testing and sometimes it is omitted (38), (39) or just suggested (37). For static approaches, the most common method is to use existing firewall policies or modifications of them (37), (40), (41); another common technique is to analyze network traffic and build the firewall policy from this analysis (42). For dynamic approaches, the main concern is to develop frameworks that automate the testing task and generate packages to test the firewall (43), (44).

All approaches use model-based approaches for generating policies. The model is usually a previously generated policy (40), a predefined rule generator that changes fields in the rule (45) or a grammar that describes the vendor's features and configurations (44).

One of the most powerful and up-to-date policy generators is ClassBench (45); which is a tool for testing and evaluating firewalls. The tool focuses in generating rules based on user-defined probabilities for each field that composes the rules. The main limitation of this tool is that it does not consider rule complexity and field interactions or features. Another limitation is that it is not flexible; it cannot use user-defined models of policy grammars and does not guarantee an efficient coverage of the testing space.

The policy generator shown in Figure 6.1 takes as a starting point the limitations of ClassBench and proposes the use of the following strategies:

- A policy grammar – an attributed grammar with defined attributes for each rule that controls the syntactic accuracy during generation.
- A BNF Graph – a directed graph corresponding to the finite state automata of the grammar with a unique initial and final state. Each node in the graph represents a nonterminal in the attributed grammar and is associated with a user-defined probability for generation purposes.
- Generation Process – the framework uses a complete traversal of the BNF graph to generate a rule; repeating the process for each generated rule. For generating a rule, the initial state is visited and each node, using its grammar attributes and probabilities, produces a terminal element. When the final state is reached, the rule has been generated and the process can be repeated to generate further rules.

These approaches have been found to present several different limitations which are listed in Table 6.2.

| Limitation | Description |
|---|---|
| No formal steps to transform a | • There is no suggested or formal |

| | |
|---|---|
| firewall rule grammar in BNF notation into an attributed firewall grammar. | procedure to add attributes to a firewall rule grammar in BNF notation. This can lead to errors and adds unintended complexity to the grammar production.<br>• Lack of flexibility to adapt to different firewall rule definition. |
| Pseudo-random rule generation | • Excludes exhaustive generation where applicable and relies only in probabilities to maintain an even distribution over nodes selected in the graph. |
| Uncertain rule complexity | • Rule complexity is managed probabilistically, excluding exhaustive generation where possible. |
| Unconstrained input space. | • The generation process doesn't control how final tokens are selected. |
| Unaware of previous results. | • A count is added to each node in the BNF graph during execution time, so the node is only aware of previous attempts on itself. |

**Table 6.2 Policy grammar, BNF Graph and Generation Process approaches limitations.**

Firewall testing is highly complex. For this reason, this research will focus on policy generation strategies, where grammars and sentence generation techniques can be employed and evaluated without the need of external systems.

### 6.2.3 Our Approach

For the experiments this research area was selected as, traditionally, firewall testing frameworks usually use for automated generation of their firewall policies model-based generation approaches. Grammars have been recently employed with a good impact in firewall testing frameworks, so this will allow us to make fair comparisons of the efficiency of our solution as previous methods are available for comparison.

Our solution proposes the following approaches that will lessen the limitations discussed for the Al-Shaer et al (44) policy generator as shown in Table 6.3.

| Limitation | Description |
|---|---|
| No formal steps to transform a firewall rule grammar in BNF notation into an attributed firewall grammar. | • Our solution defines a simple approach to transform usual firewall rule grammar in BNF notation to its equivalent symbolic grammar. |

| | |
|---|---|
| | • Any firewall rule grammar can be easily transformed with minimal changes. |
| Pseudo-random rule generation | • We include exhaustive generation where applicable and low discrepancy generation where not. |
| Uncertain rule complexity | • Rule complexity is managed by grammar sentence length and the **symbolic sentences generation**'s engine. |
| Unconstrained input space. | • We include exhaustive generation where applicable and low discrepancy generation where not. |
| Unaware of previous results. | • All previous generated outputs are considered in the generation of test cases. |

**Table 6.3 Lessened limitations by our solution.**

For evaluating this solution's accuracy, a set of empirical experiments was designed and it will be discussed in the following sections.

## 6.3 Case Study

As discussed, firewall policy generation has not been subject of exhaustive study; thus leaving a field full of possibilities for research. Firewall policies consist of policy rules that, independent of their manufacturer, share a basic syntax. This trait makes its selection an obvious one as it gives enough flexibility to experiment with the impact of our solution and it is rigid enough to avoid the unrestricted growth of other kind of grammars that can give an infinite set of solutions (e.g. recursive grammars). Therefore a simplified firewall rule grammar that will allow us to have a proper insight of how practical test policies can be generated with different systems and the benefits that our solution provides will be used.

### 6.3.1 Previous Firewall Testing Approaches

Hoffman et. al. (46) have recently published that there are few case studies in grammar-based test generation. Because of this research, topics where firewall policies are needed for testing will be explored. The main source for policies are mainly divided in two,

- Model-based generation (47) (48) and;
- Firewall policies used in real world situations (university firewalls, corporation firewalls, etc.) (49)

For firewall testing there exist frameworks (44) (50) which include several features that range from policy generation, packet generation and result evaluation. These two approaches altogether are widely used in the mentioned frameworks and in other research topics where firewall policies are needed for testing (51). These frameworks consider a firewall policy generation but don't go in detail of how they generate their policies, giving

only a brief description of their methodologies which coincide with the previously discussed.

As a result of this, a testing oracle or an external system to which a comparison could be made in the level of detail that was managed in the experiments, was not available. The most similar approach is the one proposed by Al-Shaer (44). Details on their work were requested but as their research were sponsored by Cisco, their information was not available for disclosure.

### 6.3.2 Methodology

An analysis and a comparison of four different test case generation methods were chosen as our experiments:

- Random generation – This experiment consisted of random generation of sentences from a BNF-Grammar where the only restrictions are the defined grammar rules. For this experiment, a simple sentence generation engine which solves the grammar in a pseudo-random fashion was developed.
- HAMPI system - the HAMPI string solver (19) (34) is used to solve the symbolic grammar derived from the BNF-Grammar used in the "random" experiment. It is believed that HAMPI can be viewed as the state-of-the-art for this type of problem and that our system can be built upon this previous work.
- Our system – This experiment is divided into two parts: one will apply only adaptive testing techniques for symbolic constant instantiation, and the second one will apply the enhancements over symbolic sentence generation.
    - Single approach - Our solution will be executed without any grouping of symbolic constants. With this only the enhancements that concern symbolic constant instantiation will be employed.
    - Entity approach - Our solution will be executed taking advantage of our enhancements over concrete sentence generation which group all the rules that represent a concept that is needed to be solved as a single value.

Effectively, random generation acts as the control situation – it is widely understood and has well-known properties. HAMPI (19) (50) is considered to represent state-of-the-art in test case generation of this type. HAMPI is a general solution and is utilised because no specify firewall policy-oriented solution is available.

#### *6.3.2.1 Grammars*

The system was tested with several grammars ranging from 4 to 200 production rules (e.g. http-cookies, SQL, JavaScript, ANSI C, etc.). We believe that although our system is capable of solving large context-free grammars, the analysis of the results becomes increasingly complex as the magnitude of the grammar increases. In our experiments, the Extended Cisco IOS access list grammar which can be viewed as an attempt to define vendor-natural firewall grammar will be employed. This grammar is in fact a subset of the grammar used by Al-Shaer (44), which incorporates a significant number of vendor-specific terms.

The grammar proposed in (44) corresponds to an attributed grammar which is presented in a notation that can be understood by the grammar parser introduced in their work; therefore it contains special characters and attributes that are not part of a BNF notation. Our system can only parse BNF grammars thus the attributed grammar should be converted into its equivalent BNF representation.

Let's take for example the following definition of the "action" production rule:

```
action\FieldID(0) := "permit"\V(1) | "deny"\V(0)
```

This rule contains annotated attributes at the right side of the terminal and non-terminal elements. For our purposes these attributes are removed leaving it exclusively with BNF valid operators:

```
action = "permit" | "deny"
```

This grammar presents other set of attributes which give information to the parser about how production rules should be selected. For removing this kind of attributes a method that keeps most of their properties when transformed into BNF notation was selected. The method chosen was adding redundancy to the rules that were favoured by the original attributed grammar; for example :

| **Rule** | Probability |
|---|---|
| PROTOCOL_NUMBER = PROTOCOL_NUMBER_L0 \| PROTOCOL_NUMBER_L2 ; | |
| PROTOCOL_NUMBER_L0 = "1" \| "2" \| "4" \| "6" \| "17" \| "41" \| "58" ; | 50% |
| PROTOCOL_NUMBER_L2 = [0-255]; | 50% |
| **Rule with redundancy** | |
| PROTOCOL_NUMBER = PROTOCOL_NUMBER_L0 \| PROTOCOL_NUMBER_L2 ; | |
| PROTOCOL_NUMBER_L0 = "1" \| "2" \| "4" \| "6" \| "17" \| "41" \| "58" ; | 75% |
| PROTOCOL_NUMBER_L2 = PROTOCOL_NUMBER_L0 \| [0-255]; | 25% |

**Table 6.4 Example redundancy added to the grammar.**

In this case, it is of interest to favour the group of numbers defined by the rule "PROTOCOL_NUMBER_L0" so this rule is added in "PROTOCOL_NUMBER_L2" to raise the probability of a terminal element that belongs to "PROTOCOL_NUMBER_L0" to be selected.

Another change to consider is the set of rules which define specific-vendor rules terms (for this case extended Cisco IOS access list). These rules define flags, connections and logging. Take for example the initial rule:

```
S := "access-list" Policy-Number action proto SrcAddr
     [SrcPort] DestAddr [DestPort] [ACK] [FIN] [PSH]
     [RST] [SYN] [URG] [Prec] [Tos] [Established]
     [Logging] [Fragments] [icmpquals] [igmpquals]
```

This after removing Cisco IOS specific fields is simplified to:

```
S = proto SrcAddr [SrcPort] DestAddr [DestPort] action
```

With this modification our grammar is modified into a simplified version of the original which gives generic firewall rules that contain the commonly used fields for the most used products (Check Point Software's Firewall-1, CyberGuard's CyberGuard, Microsoft's Windows Firewall, NetGuard's Guardian, Milkyway's SecurIT, etc. (52) ).

Finally, a subset of the rules was renamed to give a more descriptive annotation; and rules which are related as they are part of the same concept were grouped (i.e. Source address).

```
SIMPLIFIED: S = proto SrcAddr [SrcPort] DestAddr
     [DestPort] action
RENAMED: RULE = PROTOCOL_NUMBER SOURCE DESTINATION
     ACTION ;
```

After the discussed changes were applied to the original attributed grammar, the resulting equivalent BNF grammar (**G**) used for the "random" experiment is presented:

```
RULE = PROTOCOL_NUMBER SOURCE DESTINATION ACTION ;
PROTOCOL_NUMBER = PROTOCOL_NUMBER_L0 | PROTOCOL_NUMBER_L1 |
     PROTOCOL_NUMBER_L2 ;
PROTOCOL_NUMBER_L0 = "1" | "2" | "4" | "6" | "17" | "41" |
     "58" ;
PROTOCOL_NUMBER_L1 = PROTOCOL_NUMBER_L0 | "9" | "15" | "37" |
     "43" | "44" | "59" | "60";
PROTOCOL_NUMBER_L2 = PROTOCOL_NUMBER_L0 | PROTOCOL_NUMBER_L1 |
     [0-255];
SOURCE = IP_ADDRESS PORT ;
DESTINATION = IP_ADDRESS PORT ;
PORT = PORT_NONE | PORT_SINGLE | PORT_LT | PORT_GT | PORT_NE |
```

```
     PORT_RANGE ;
PORT_NONE = "";
PORT_SINGLE = DECIDE_PORT ;
PORT_LT = LT DECIDE_PORT ;
PORT_GT = GT DECIDE_PORT ;
PORT_NE = NE DECIDE_PORT ;
PORT_RANGE = DECIDE_PORT SLASH DECIDE_PORT ;
DECIDE_PORT = PORTS_Commons | PORTS_Not_Commons ;
PORTS_Not_Commons = PORTS_Others | PORTS_Assigned | [0-65535];
PORTS_Commons = "1" | "7" | "9" | "11" | "13" | "17" | "18"
     |"19" | "20" | "21" | "22" | "23" | "25" | "37" | "39" |
     "42" | "43" | "50" | "53" | "67" | "68" | "69" | "70" |
     "79" | "80" | "88" | "95" | "101" | "102" | "105" | "107"
     | "109" | "110" | "111" | "113" | "115" | "117" | "119" |
     "123" | "137" | "138" | "139" | "143" | "161" | "162" |
     "163" | "164" | "177" | "178" | "179" | "191" | "194" |
     "199" ;
PORTS_Others = "201" | "202" | "204" | "206" | "209" | "210" |
     "213" | "220" | "369" | "370" | "372" | "443" | "444" |
     "487" | "610" | "611" | "612" ;
PORTS_Assigned = "1524" | "1525" | "1645" | "1646" | "1812" |
     "1813" | "2401" | "2430" | "2431" | "2432" | "2433" |
     "3306" | "5002" | "5308" | "7000" ;
IP_ADDRESS = IP_ADDRESS_COMPLETE | IP_WILD ;
IP_WILD = IP_OTHER | IP_ADDRESS_SET ;
IP_ADDRESS_SET = IP_ADDRESS_SET_1000 | IP_ADDRESS_SET_0100 |
     IP_ADDRESS_SET_0010 | IP_ADDRESS_SET_0001;
IP_ADDRESS_COMPLETE = IP_NUMBER DOT IP_NUMBER DOT IP_NUMBER DOT
     IP_NUMBER ;
IP_ADDRESS_SET_1000 = IP_NUMBER DOT IP_DECIDE DOT IP_DECIDE DOT
     IP_DECIDE ;
IP_ADDRESS_SET_0100 = IP_DECIDE DOT IP_NUMBER DOT IP_DECIDE DOT
     IP_DECIDE ;
IP_ADDRESS_SET_0010 = IP_DECIDE DOT IP_DECIDE DOT IP_NUMBER DOT
     IP_DECIDE ;
IP_ADDRESS_SET_0001 = IP_DECIDE DOT IP_DECIDE DOT IP_DECIDE DOT
     IP_NUMBER ;
IP_DECIDE = " * " | IP_NUMBER ;
IP_OTHER = "127 . 000 . 000 . 001" ;
IP_NUMBER = [0-255];
ACTION = "permit" | "deny" ;
LT = "<" ;
GT = ">" ;
NE = "!=" ;
DOT = " . ";
SLASH = " - ";
```

Two rule modifications that were in the original grammar and were kept in our simplification should be mentioned (it was decided to keep them based on their purpose, which was to mimic real life firewall rules):

- Protocols and ports numbers have been biased to favour most commonly used selections. The grammar refers to user-defined values which are defined based on (53) for protocols and (54) for port numbers.
- IP addresses are distinguished between local host, specific addresses (containing no wildcards) and addresses that contain one or more wild cards.

The next step was to define the symbolic grammar **G'**. For defining it, the same criteria in section 4.3.1 suggested by Majumdar (18) was applied. This criterion gave the derived the following symbolic grammar (**GS'**):

```
RULE = <PROTOCOL_NUMBER> SOURCE DESTINATION <ACTION>  ;
SOURCE = IP_ADDRESS PORT ;
DESTINATION = IP_ADDRESS PORT ;
PORT = PORT_NONE | PORT_SINGLE | PORT_LT | PORT_GT | PORT_NE |
     PORT_RANGE ;
PORT_NONE = "";
PORT_SINGLE = <DECIDE_PORT> ;
PORT_LT = LT <DECIDE_PORT> ;
PORT_GT = GT <DECIDE_PORT> ;
PORT_NE = NE <DECIDE_PORT> ;
PORT_RANGE = <DECIDE_PORT> SLASH <DECIDE_PORT> ;
IP_ADDRESS = IP_ADDRESS_COMPLETE | IP_WILD ;
IP_WILD = IP_OTHER | IP_ADDRESS_SET ;
IP_ADDRESS_COMPLETE   =   <IP_NUMBER>   DOT   <IP_NUMBER>   DOT
     <IP_NUMBER> DOT <IP_NUMBER> ;
IP_ADDRESS_SET  =  IP_ADDRESS_SET_1000  |  IP_ADDRESS_SET_0100  |
     IP_ADDRESS_SET_0010 | IP_ADDRESS_SET_0001;
IP_ADDRESS_SET_1000   =   <IP_NUMBER>   DOT   <IP_DECIDE>   DOT
     <IP_DECIDE> DOT <IP_DECIDE> ;
IP_ADDRESS_SET_0100   =   <IP_DECIDE>   DOT   <IP_NUMBER>   DOT
     <IP_DECIDE> DOT <IP_DECIDE> ;
IP_ADDRESS_SET_0010   =   <IP_DECIDE>   DOT   <IP_DECIDE>   DOT
     <IP_NUMBER> DOT <IP_DECIDE> ;
IP_ADDRESS_SET_0001   =   <IP_DECIDE>   DOT   <IP_DECIDE>   DOT
     <IP_DECIDE> DOT <IP_NUMBER> ;
IP_OTHER = "127 . 000 . 000 . 001" ;
LT = "<" ;
GT = ">" ;
NE = "!=" ;
DOT = " . ";
SLASH = " - ";
```

These symbolic constants are then defined into its correspondent symbolic values grammar (**VS**):

```
PROTOCOL_NUMBER  =  PROTOCOL_NUMBER_L0  |  PROTOCOL_NUMBER_L1  |
     PROTOCOL_NUMBER_L2  ;
PROTOCOL_NUMBER_L0 = "1" | "2" | "4" | "6" | "17" | "41" |
     "58" ;
PROTOCOL_NUMBER_L1 = PROTOCOL_NUMBER_L0 | "9" | "15" | "37" |
     "43" | "44" | "59" | "60" ;
PROTOCOL_NUMBER_L2 = PROTOCOL_NUMBER_L0 | PROTOCOL_NUMBER_L1 |
     [0-255];
DECIDE_PORT = <PORTS_Commons> | <PORTS_Not_Commons> ;
PORTS_Not_Commons = PORTS_Others | PORTS_Assigned | [0-65535];
PORTS_Commons = "1" | "7" | "9" | "11" | "13" | "17" | "18" |
     "19" | "20" | "21" | "22" | "23" | "25" | "37" | "39" |
     "42" | "43" | "50" | "53" | "67" | "68" | "69" | "70" |
     "79" | "80" | "88" | "95" | "101" | "102" | "105" | "107" |
     "109" | "110" | "111" | "113" | "115" | "117" | "119" |
     "123" | "137" | "138" | "139" | "143" | "161" | "162" |
     "163" | "164" | "177" | "178" | "179" | "191" | "194" |
     "199" ;
PORTS_Others = "201" | "202" | "204" | "206" | "209" | "210" |
     "213" | "220" | "369" | "370" | "372" | "443" | "444" |
     "487" | "610" | "611" | "612" ;
```

```
PORTS_Assigned = "1524" | "1525" | "1645" | "1646" | "1812" |
     "1813" | "2401" | "2430" | "2431" | "2432" | "2433" |
     "3306" | "5002" | "5308" | "7000" ;
IP_DECIDE = WILD_CARD | IP_NUMBER ;
IP_NUMBER = [0-255];
ACTION = "permit" | "deny" ;
WILD_CARD = " * " ;
```

Once **GS** and **VS** have been defined, a new symbolic grammar (**GE'**) and a new symbolic values grammar (**VE**) will be derived from G, which will be used for the "entity" experiment. There is not a single way for defining entities, therefore they will consider each of the fields in a firewall rule (protocol, source address, action, etc.) these criteria will generate the following **GE'**:

```
RULE = <PROTOCOL_NUMBER> <SOURCE> <DESTINATION> <ACTION>  ;
```

And the following **VE**:

```
PROTOCOL_NUMBER = PROTOCOL_NUMBER_L0 | PROTOCOL_NUMBER_L1 |
     PROTOCOL_NUMBER_L2  ;
PROTOCOL_NUMBER_L0 = "1" | "2" | "4" | "6" | "17" | "41" |
     "58" ;
PROTOCOL_NUMBER_L1 = PROTOCOL_NUMBER_L0 | "9" | "15" | "37" |
     "43" | "44" | "59" | "60" ;
PROTOCOL_NUMBER_L2 = PROTOCOL_NUMBER_L0 | PROTOCOL_NUMBER_L1 |
     [0-255] ;
SOURCE = IP_ADDRESS PORT ;
DESTINATION = IP_ADDRESS PORT ;
PORT = PORT_NONE | PORT_SINGLE | PORT_LT | PORT_GT | PORT_NE |
     PORT_RANGE ;
PORT_NONE = "";
PORT_SINGLE = DECIDE_PORT ;
PORT_LT = LT DECIDE_PORT ;
PORT_GT = GT DECIDE_PORT ;
PORT_NE = NE DECIDE_PORT ;
PORT_RANGE = DECIDE_PORT SLASH DECIDE_PORT ;
DECIDE_PORT = PORTS_Commons | PORTS_Not_Commons ;
PORTS_Not_Commons = PORTS_Others | PORTS_Assigned | [0-65535];
PORTS_Commons = "1" | "7" | "9" | "11" | "13" | "17" | "18" |
     "19" | "20" | "21" | "22" | "23" | "25" | "37" | "39" |
     "42" | "43" | "50" | "53" | "67" | "68" | "69" | "70" |
     "79" | "80" | "88" | "95" | "101" | "102" | "105" | "107"
     | "109" | "110" | "111" | "113" | "115" | "117" | "119" |
     "123" | "137" | "138" | "139" | "143" | "161" | "162" |
     "163" | "164" | "177" | "178" | "179" | "191" | "194" |
     "199" ;
PORTS_Others = "201" | "202" | "204" | "206" | "209" | "210" |
     "213" | "220" | "369" | "370" | "372" | "443" | "444" |
     "487" | "610" | "611" | "612" ;
PORTS_Assigned = "1524" | "1525" | "1645" | "1646" | "1812" |
     "1813" | "2401" | "2430" | "2431" | "2432" | "2433" |
     "3306" | "5002" | "5308" | "7000" ;
IP_ADDRESS = IP_ADDRESS_COMPLETE | IP_WILD ;
IP_WILD = IP_OTHER | IP_ADDRESS_SET ;
IP_ADDRESS_SET = IP_ADDRESS_SET_1000 | IP_ADDRESS_SET_0100 |
     IP_ADDRESS_SET_0010 | IP_ADDRESS_SET_0001;
IP_ADDRESS_COMPLETE = IP_NUMBER DOT IP_NUMBER DOT IP_NUMBER DOT
     IP_NUMBER ;
```

```
IP_ADDRESS_SET_1000 = IP_NUMBER DOT IP_DECIDE DOT IP_DECIDE DOT
     IP_DECIDE ;
IP_ADDRESS_SET_0100 = IP_DECIDE DOT IP_NUMBER DOT IP_DECIDE DOT
     IP_DECIDE ;
IP_ADDRESS_SET_0010 = IP_DECIDE DOT IP_DECIDE DOT IP_NUMBER DOT
     IP_DECIDE ;
IP_ADDRESS_SET_0001 = IP_DECIDE DOT IP_DECIDE DOT IP_DECIDE DOT
     IP_NUMBER ;
IP_DECIDE = " * " | IP_NUMBER ;
IP_OTHER = "127 . 000 . 000 . 001" ;
IP_NUMBER = [0-255];
ACTION = "permit" | "deny" ;
LT = "<" ;
GT = ">" ;
NE = "!=" ;
DOT = " . ";
SLASH = " - ";
```

The derived grammars (one context-free and two symbolic) and the two symbolic values grammars will be the ones use from this point on for our experiments. Finally, few changes as possible were made from one derivation to the other so, after analysis, the comparison between the obtained results of each experiment can be done without complex interpretations.

#### *6.3.2.2 Constraints*

The "random" experiment employs a BNF grammar, thus it doesn't have the need for any constraints.

For the "HAMPI" and "single" experiments, an enumeration of all the possible symbolic sentences derived from **G'** and **GS'** was made; this is possible due to symbolic grammars that greatly restraining the solution space. With this approach 6084 symbolic sentences were generated, including the following examples:

```
<PROTOCOL_NUMBER> <IP_NUMBER> "." <IP_DECIDE> "."
     <IP_DECIDE> "." <IP_DECIDE> <PORTS_Not_Commons> "127 .
     000 . 000 . 001" <PORTS_Not_Commons> <ACTION> ;
<PROTOCOL_NUMBER> <IP_NUMBER> "." <IP_NUMBER> "."
     <IP_NUMBER> "." <IP_NUMBER> <OPERAND>
     <PORTS_Not_Commons> <IP_NUMBER> "." <IP_NUMBER> "."
     <IP_NUMBER> "." <IP_NUMBER> <PORTS_Not_Commons>
     <ACTION> ;
<PROTOCOL_NUMBER> <IP_NUMBER> "." <IP_NUMBER> "."
     <IP_NUMBER> "." <IP_NUMBER> <PORTS_Commons> <IP_DECIDE>
     "." <IP_DECIDE> "." <IP_DECIDE> "." <IP_NUMBER>
     <ACTION> ;
<PROTOCOL_NUMBER> <IP_NUMBER> "." <IP_DECIDE> "."
     <IP_DECIDE> "." <IP_DECIDE> <OPERAND> <PORTS_Commons>
     "127 . 000 . 000 . 001" <PORTS_Commons> "-"
     <PORTS_Commons> <ACTION> ;
```

This enumeration will be used as the constraints for these experiments as they guarantee that all possible combinations from the grammar are instantiated during the concrete sentence-generation phase. The decision to enumerate all the solution space for **GS'** was taken to generate all the possible values, so the results obtained vary as little as possible. Therefore the variation is concentrated in the **concrete sentence generation phase.** This phase is where the "competing" approaches differ.

For the entity experiment, no constraints were generated as the grammar consists of the following single rule:

```
RULE = <PROTOCOL_NUMBER> <SOURCE> <DESTINATION> <ACTION>;
```

#### *6.3.2.3 Other Parameters*

Some other parameters were also defined for this experiment. They were set to large values so they did not have an impact in the final results. The values selected were:

- Sentence length - .set to 30 as the maximum possible used fields (concrete values) is 24 (this will be explained later).
- Grammar depth - set to 30 as the maximum possible number of nested rules is 13.

It should be mention that these values are arbitrary and any one that exceeds the maximum possible value will have the same effects.

### 6.3.3 Evaluation Criteria

As discussed in section 6.3.1, there exists little work on firewall policy generation. Therefore an accepted testing oracle was not available to compare against or to act as a gold standard. An alternative is presented in Al Shaer's (42) framework, which describes a set of evaluation criteria including:

- Field coverage - which evaluates whether all the optional fields have been employed.
- Space coverage - which evaluates the coverage of individual fields (how diverse are the selected values for a specific field).

It was found that these evaluation criteria were too limited for our purposes, since they do not contain sufficient detail on how space coverage is achieved. This problem was overcome by making use of the grammars that are employed by the systems under analysis. Taking advantage of the nature of model-based testing, which bounds its input and output space with its model, the grammar can be employed and create from it a set of criteria which will function as an oracle for our evaluations. In this selected approach, an additional measure will be considered:

- Concrete constants generation probability - the likelihood that a concrete constant is selected by its set of production rules.

Quasi-random sequences gives another criterion to consider which is the solution space covered by the different implementations considered, which is proportional to the number of unique values generated by each evaluated system; therefore we will consider the following measure:

- Unique values generation – the uniqueness of a result compared against previously generated results.

This set of evaluation concepts was selected since it will provide insight into how each system reflects the properties (distribution, bias, etc.) defined in the grammar through its production rules.

In the following sections, the specific details for each field that will be evaluate will be presented.

#### *6.3.3.1 Rules Distribution*

Rules distribution will be measured by comparing each produced rule against previously generated rules and classifying them between unique and not unique (previously generated).

This count is one of the most important criteria that will be analyzed as it will give a clear exemplification about our proposed solution, which is based in quasi-random sequences, enhances diversity over pseudo-random based systems.

#### *6.3.3.2 Rule Length*

Rule length is defined as the total number of fields within a single rule. Table 6.5 presents three firewall rules having different rule lengths.

| Rules | Rule length |
|---|---|
| `233 11.*.048.213 127.000.000.001 accept` | 16 |
| `58 2.89.127.147 80 212.254.105.206 deny` | 17 |
| `17 *.182.096.044 114.*.44.194 194 – 34408 deny` | 22 |

**Table 6.5 Firewall rules with different length.**

It is observed that the minimum length is 16 if no optional values are selected, while the maximum length is 22 if both source and destination IP addresses include a range (where a range consists of 3 fields).

The concrete constants generation probability is determined by the rule "PORT" which is responsible for specifying the possible expansions from where the systems can make their selection:

```
PORT = PORT_NONE | PORT_SINGLE | PORT_LT | PORT_GT |
     PORT_NE | PORT_RANGE;
```

From the possible selections from which PORT can chose, PORT_LT, PORT_GT, PORT_NE will generate only lengths of two, while PORT_NONE, PORT_SINGLE and PORT_RANGE produce lengths of zero, one and three respectively; therefore a greater concentration of values for length two is expected (with 50% of the chances to be selected).

A firewall rule must have two port values (source and destination) and each port value will favour lengths of magnitude two, therefore the rule length will concentrate its results in any permutation containing these values.

| Length | Distribution |
|---|---|
| 16 | 2.78% |
| 17 | 5.56% |
| 18 | 19.44% |
| 19 | 22.22% |
| 20 | 30.56% |
| 21 | 16.67% |
| 22 | 2.78% |

**Table 6.6 Rule length expected distribution**

Table 6.6 shows how the probability for generating extreme cases (lengths 16 and 22) are the same (1 out of 36 total permutations) as they can only be generated when both port definitions select the exact same expansion (PORT_NONE and PORT_RANGE). In contrast, rules with length 20 are the most favoured as 11 out of the 36 total permutations produce rules with this magnitude.

### *6.3.3.3 Protocol Numbers*

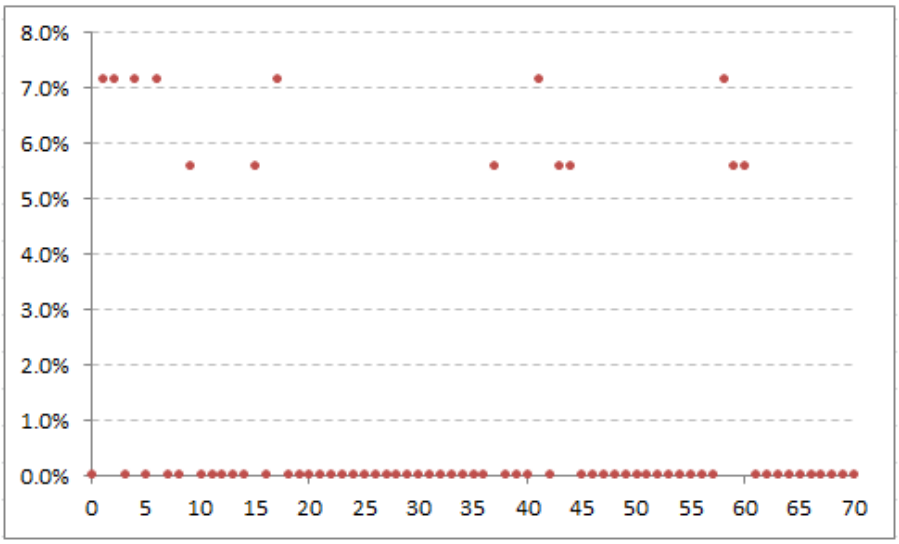
Protocol number rules are biased to favour the most commonly used protocols (54), and thus it is expected that the analysis will reflect this bias. For asserting these expectations the production rules, which control concrete constants generation probability, will be employed in an example. First the production rules are shown:

```
PROTOCOL_NUMBER = PROTOCOL_NUMBER_L0 | PROTOCOL_NUMBER_L1 |
     PROTOCOL_NUMBER_L2 ;
PROTOCOL_NUMBER_L0 = "1" | "2" | "4" | "6" | "17" | "41" | "58" ;
PROTOCOL_NUMBER_L1 = PROTOCOL_NUMBER_L0 | "9" | "15" | "37" | "43" |
     "44" | "59" | "60" ;
PROTOCOL_NUMBER_L2 = PROTOCOL_NUMBER_L0 | PROTOCOL_NUMBER_L1 |
     NUMBER_0_255 ;
NUMBER_0_255 = "0" | "1" | "2" | "3" | . . . | "253" | "254" |
     "255" ;
```

As an example, the probability of generating the protocol number 255 beginning with the PROTOCOL_NUMBER rule can be calculated. The first rule to be visited is PROTOCOL_NUMBER; this rule has three possible selections and each selection has a probability of 33.33%. The next rule to visit is PROTOCOL_NUMBER_L2 which has three possible selections, each one having an associated probability of 33.33%. Finally the option NUMBER_0_255 in PROTOCOL_NUMBER_L2 stands for a range of numbers which can be evenly generated, therefore it can be considered as a production rule with a total number of available selections equal to the possible values specified in the range (256 for this case) associating a probability of 0.392% to each one of them. The probability to produce the number 255 with this rule path is the product of the three visited rules which gives us 0.0434%.

It should be noted that this calculation is more complex for rules such as `PROTOCOL_NUMBER_L0` that can be reached with different "rule paths"; and whose values can be generated from other rules (i.e. "1" can be produced from `PROTOCOL_NUMBER_L0` and `[0-255]`).

**A) Range from 0 to 70**



B) Range from 71 to 255

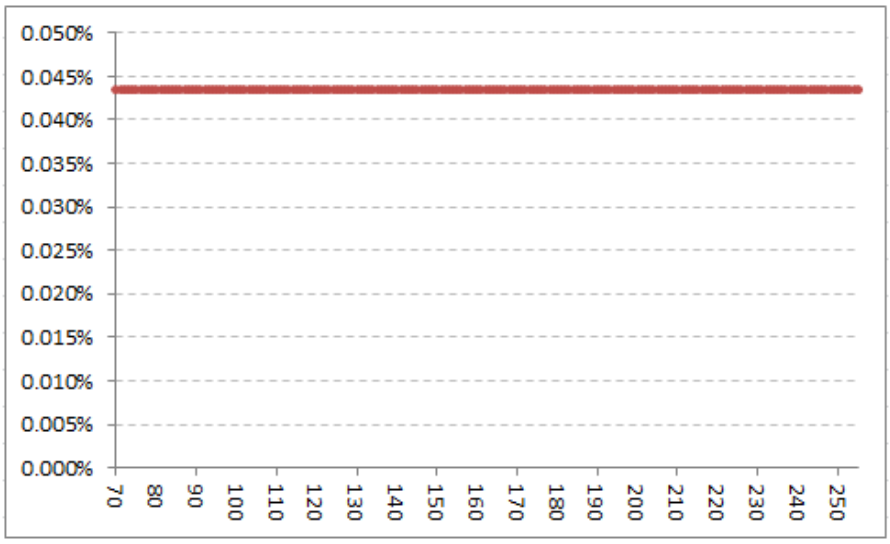



**Figure 6.2 A-B Expected distribution for protocol numbers.**

Using this methodology, the expected results for the protocol numbers can be calculated and are summarized in Figure 6.2 (the distribution should be the same for all of the experiments as the same definition is true for all of them).

#### *6.3.3.4 IP Address Structure*

IP address structure probability was calculated employing the same methodology previously discussed; however, using the following set of rules:

```
IP_ADDRESS = IP_ADDRESS_COMPLETE | IP_WILD ;
IP_WILD = IP_OTHER | IP_ADDRESS_SET ;
IP_ADDRESS_SET = IP_ADDRESS_SET_1000 | IP_ADDRESS_SET_0100 |
     IP_ADDRESS_SET_0010 | IP_ADDRESS_SET_0001;
IP_ADDRESS_COMPLETE = IP_NUMBER DOT IP_NUMBER DOT IP_NUMBER
     DOT IP_NUMBER ;
IP_ADDRESS_SET_1000 = IP_NUMBER DOT IP_DECIDE DOT IP_DECIDE
     DOT IP_DECIDE ;
IP_ADDRESS_SET_0100 = IP_DECIDE DOT IP_NUMBER DOT IP_DECIDE
     DOT IP_DECIDE ;
IP_ADDRESS_SET_0010 = IP_DECIDE DOT IP_DECIDE DOT IP_NUMBER
     DOT IP_DECIDE ;
IP_ADDRESS_SET_0001 = IP_DECIDE DOT IP_DECIDE DOT IP_DECIDE
     DOT IP_NUMBER ;
IP_DECIDE = " * " | IP_NUMBER ;
IP_OTHER = "127 . 000 . 000 . 001" ;
```

- IP_ADDRESS is the starting rule for this set of production rules, thus it holds the 100% of the solution space.

- From it two paths can be followed IP_ADDRESS_COMPLETE and IP_WILD with an associated probability of 50% for each one of them.
- If IP_ADDRESS_COMPLETE is chosen it can be induced that 50% of the total produced rules will contain no wild cards as this rule can only generate this kind of values.
- For IP_WILD two selections are possible IP_OTHER and IP_ADDRESS_SET, each one with an associated probability of 50% relative to this rule and a 25% relative to the total solution space.
- If IP_OTHER is chosen then it can be induced that this 25% of the total solution space will only generate IP addresses that contain no wild cards as this rule can only generate the IP value for localhost (127.0.0.1).
- For the IP_ADDRESS_SET rule, which holds the remaining 25% of the solution space, four options are available with an associated probability of 6.25%. Each one of these four options can generate 8 different combinations, thus associating a probability of 0.781% of the total solution space to each one of them.

The complete results for the solution space can be found in Table 6.7 and its resulting distribution is shown in Table 6.8. From this table, it is observed that rules with no wild cards have a 78.13% of being selected, rules with one and two wild cards with 9.38% each; the distribution value is further reduced to 3.12% for rules having three wild cards.

| IP_ADDRESS_COMPLETE | |
|---|---|
| 50% | #.#.#.# |

| IP_OTHER | |
|---|---|
| 25% | #.#.#.# |

| IP_ADDRESS_SET_1000 | |
|---|---|
| 0.781% | #.#.#.# |
| 0.781% | #.#.#.* |
| 0.781% | #.#.*.# |
| 0.781% | #.*.#.# |
| 0.781% | #.#.*.* |
| 0.781% | #.*.#.* |
| 0.781% | #.*.*.# |
| 0.781% | #.*.*.* |

| IP_ADDRESS_SET_0100 | |
|---|---|
| 0.781% | #.#.#.# |
| 0.781% | #.#.#.* |
| 0.781% | #.#.*.# |
| 0.781% | *.#.#.# |
| 0.781% | #.#.*.* |
| 0.781% | *.#.#.* |
| 0.781% | *.#.*.# |
| 0.781% | *.#.*.* |

| IP_ADDRESS_SET_0010 | |
|---|---|
| 0.781% | #.#.#.# |
| 0.781% | #.*.#.# |
| 0.781% | *.#.#.# |
| 0.781% | #.#.#.* |
| 0.781% | *.*.#.# |
| 0.781% | #.*.#.* |
| 0.781% | *.#.#.* |
| 0.781% | *.*.#.* |

| IP_ADDRESS_SET_0001 | |
|---|---|
| 0.781% | #.#.#.# |
| 0.781% | #.#.*.# |
| 0.781% | #.*.#.# |
| 0.781% | *.#.#.# |
| 0.781% | #.*.*.# |
| 0.781% | *.#.*.# |
| 0.781% | *.*.#.# |
| 0.781% | *.*.*.# |

**Table 6.7 Each production rule is shown with their possible generated IP address values and their associated probability (# stands for any no-wild card value).**

| Wild Cards | Pattern | Distribution |
|---|---|---|
| 0 | #.#.#.# | 78.125% |

| | | |
|---|---|---|
| 1 | #.#.#.* | 2.344% |
| | #.#.*.# | 2.344% |
| | #.*.#.# | 2.344% |
| | *.#.#.# | 2.344% |
| 2 | #.#.*.* | 1.563% |
| | #.*.#.* | 1.563% |
| | #.*.*.# | 1.563% |
| | *.#.#.* | 1.563% |
| | *.#.*.# | 1.563% |
| | *.*.#.# | 1.563% |
| 3 | #.*.*.* | 0.781% |
| | *.#.*.* | 0.781% |
| | *.*.#.* | 0.781% |
| | *.*.*.# | 0.781% |
| 4 | *.*.*.* | 0.000% |

**Table 6.8 IP structure expected distribution (# stands for any no-wild card value).**

### *6.3.3.5 IP Address Size*

IP address size is measured as the number of bits needed to represent each of the numbers comprising an IP address. Each IP number needs a certain quantity of bits to be generated that range between one and eight. As each IP address is composed of four IP numbers, the maximum complexity will be 32 and the minimum 4.

In the previous section, it has been discussed that the set of rules which produce IP addresses, generate both no-wild card and wild card values. For this criterion only values with no-wild cards will be considered, so the 78.13% (calculated in section 6.3.3.3) of the solution space that contains this type of value will conform the solution space for this criterion.

There are three main paths for generating an IP address with no wild cards:

- From `IP_ADDRESS`;
- From `IP_OTHER`; and
- From `IP_ADDRESS_SET_1000`, `IP_ADDRESS_SET_0100`, `IP_ADDRESS_SET_0010` and `IP_ADDRESS_SET_0001` where all the instances of `IP_DECIDE` select the `IP_NUMBER` path.

The associated probabilities for `IP_ADDRESS`, `IP_OTHER`, appear in Table 6.7. `IP_ADDRESS_SET_1000`, `IP_ADDRESS_SET_0100`, `IP_ADDRESS_SET_0010` and `IP_ADDRESS_SET_0001` can be found in Table 6.7. As we are only considering a subset of the total solution space, the solution space for this criterion must be calculated. For this calculation, the distribution associated

for each rule will be divided between the total solution space that contains no wild card values. That is, for IP_ADDRESS_COMPLETE, its distribution (50%) is divided between the total distribution for no wild cards values (78.13%) giving a value of 64.0%. This method is applied for calculating the solution space distribution for each of the rule paths considered. Table 6.9 shows the rule paths, its solution space distribution, the solution space distribution for this criterion and the possible sizes these paths can generate.

| Rule Path | Solution Space percentage | Solution space percentage for this criterion | Possible size |
|---|---|---|---|
| IP_ADDRESS_COMPLETE | 50.00% | 64.0% | 4-32 |
| IP_OTHER | 25.00% | 32.0% | 10 |
| IP_ADDRESS_SET | 3.13% | 4.0% | 4-32 |

**Table 6.9 Solution and criterion solution space percentage for each rule.**

In this table, it is clear that the IP_OTHER distribution will have a significant impact on the expected distribution as 32% of the criterion values will produce IP addresses of size 10. For calculating the complete expected distribution, the numbers of bits that are needed to represent each IP address are enumerated:

| IP Address | Bits | Value |
|---|---|---|
| 10.10.10.10 | 4 + 4 + 4 + 4 | 16 |
| 10.100.100.255 | 4 + 7 + 7 + 8 | 26 |
| 100.100.100.100 | 7 + 7 + 7 + 7 | 28 |
| 255.255.255.255 | 8 + 8 + 8 + 8 | 32 |

**Table 6.10 IP Address size calculation.**

For example, the only permutation of IP numbers that will result in an address of size four would be the case when the four IP numbers require only a single bit for their representation (1+1+1+1); the same is true for the size 32, where each of the four values require 8 bits. The probability of each number is calculated (with the same methodology employed in the previous section) and the distribution is shown in Table 6.11, where it can be appreciated that a large concentration of values is present in value 10 as was shown in Table 6.9.

| Bits | Distribution |
|---|---|
| 4 | 0.017% |
| 5 | 0.066% |
| 6 | 0.166% |
| 7 | 0.332% |
| 8 | 0.580% |
| 9 | 0.928% |

| Bits | Distribution |
|---|---|
| 12 | 2.669% |
| 13 | 3.382% |
| 14 | 4.078% |
| 15 | 4.708% |
| 16 | 5.222% |
| 17 | 5.570% |

| Bits | Distribution |
|---|---|
| 20 | 5.222% |
| 21 | 4.708% |
| 22 | 4.078% |
| 23 | 3.382% |
| 24 | 2.669% |
| 25 | 1.989% |

| Bits | Distribution |
|---|---|
| 28 | 0.580% |
| 29 | 0.332% |
| 30 | 0.166% |
| 31 | 0.066% |
| 32 | 0.017% |

| | | | | | |
|---|---|---|---|---|---|
| 10 | 33.492% | 18 | 5.703% | 26 | 1.392% |
| 11 | 1.989% | 19 | 5.570% | 27 | 0.928% |

**Table 6.11 IP Address size expected distribution.**

#### *6.3.3.6 IP Values Distribution*

The distribution of IP addresses can be measured by comparing each produced rule's IP address against previously generated rules' IP address and classifying them as either unique or previously generated. This will provide insight into how the different solutions impact the generation of IP addresses and will help to evaluate the coverage of the solution space that each solution produces.

The total space covered for each experiment will also be measured. It will be calculated by computing the ratio between the unique produced IP values and the total possible IP values ($2^{32}$). This evaluation will give us an insight of how the quasi-random sequences implemented in our solution optimizes previous work.

#### *6.3.3.7 Ports usage*

Similar to the rules for protocol numbers, port production rules are also biased to favour the most commonly used ports, and again, it is expected that the analysis reveal this bias (53). Ports can be divided into three groups:

- PORTS_Commons - This first group include the most commonly used ports. This group includes ports such as: Echo (port 7), Active Users (port 11), file transfer control (port 21), Telnet (port 23), host name server (port 42), World Wide Web (port 80), simple file transfer protocol (port 115), and the internet message access protocol (port 143)
- PORTS_Others – This is a less frequently used group compared to PORTS_Commons. In this group, ports such as AppleTalk routing (port 201), The Quick Mail Transfer Protocol (port 209), Protocol v3 (port 220), and Simple Asynchronous File Transfer (port 487) are found.
- PORTS_Assigned - - These are the ports not controlled by the IANA (Internet Assigned Numbers Authority) and can be used by non-administrative users or non-administrative user processes.

These groups are defined in our grammars by the following rules:

```
DECIDE_PORT = PORTS_Commons | PORTS_Not_Commons ;
PORTS_Not_Commons = PORTS_Others | PORTS_Assigned | [0-65535];
PORTS_Commons = "1" | "7" | "9" | "11" | "13" | "17" | "18" | "19" |
     "20" | "21" | "22" | "23" | "25" | "37" | "39" | "42" | "43" |
```

```
      "50" | "53" | "67" | "68" | "69" | "70" | "79" | "80" | "88" |
      "95" | "101" | "102" | "105" | "107" | "109" | "110" | "111" |
      "113" | "115" | "117" | "119" | "123" | "137" | "138" | "139" |
      "143" | "161" | "162" | "163" | "164" | "177" | "178" | "179" |
      "191" | "194" | "199" ;
PORTS_Others = "201" | "202" | "204" | "206" | "209" | "210" | "213"
     | "220" | "369" | "370" | "372" | "443" | "444" | "487" | "610"
     | "611" | "612" ;
PORTS_Assigned = "1524" | "1525" | "1645" | "1646" | "1812" | "1813"
     | "2401" | "2430" | "2431" | "2432" | "2433" | "3306" | "5002"
     | "5308" | "7000" ;
```

Employing the previously discussed methodology in section 6.3.3.2, Table 6.12 can be derived for the expected results' distribution.

| RULE | Distribution |
|---|---|
| PORTS_Commons | 50.00% |
| PORTS_Others | 16.67% |
| PORTS_Assigned | 16.67% |
| [0-65535] | 16.67% |

**Table 6.12 Port usage expected distribution values**

From a simple inspection of the rules, the maximum port value which is biased by a rule is port "7000" (BBS service), and all the subsequent ports (greater than 7000) will have the same probability as they are all derived from the `[0-65535]` rule. It is noted that that other groups are discussed in (54), however, are omitted from this discussion as they refer to specific services like UNIX, Kerberos, Linux, BSD, etc.

### *6.3.3.8 Port operators distribution*

| Operator | Description |
|---|---|
| > # | Less than |
| < # | More than |
| != # | Different |
| # - # | Range |

**Table 6.13 Common port operators considered.**

Our grammars consider the common port operators listed in Table 6.13. The rules controlling the concrete constants generation probability are as follows:

```
PORT = PORT_NONE | PORT_SINGLE | PORT_LT | PORT_GT |
     PORT_NE | PORT_RANGE ;
PORT_NONE = "";
PORT_SINGLE = DECIDE_PORT ;
PORT_LT = LT DECIDE_PORT ;
PORT_GT = GT DECIDE_PORT ;
PORT_NE = NE DECIDE_PORT ;
PORT_RANGE = DECIDE_PORT SLASH DECIDE_PORT ;
```

Using the previously discussed methodology in section 6.3.3.2, the same probability is shared by all the possible paths of rule `PORT`; therefore a uniform distribution can be expected (with a 16.66% probability for each rule).

#### *6.3.3.9 Action Values Distribution*

Action values distribution will provide an insight of how rule selection varies depending on the technology used for concrete constant generation. The action rule is straightforward with only two terminal values that have no variation with the different generated grammars:

```
ACTION = "permit" | "deny" ;
```

It is simple enough to appreciate that it is a binary selection where each of the options should be expected to have a 50% of the choice.

## 6.4 Empirical Results

Empirical results that cover firewall rules will be discussed in the next following sections. Two analyses are provided:

- Policies - how policies were generated; and
- Rules distribution - how the generated rules cover the solution space.

Finally, discussion over individual firewall rules will be presented in the last section.

### 6.4.1 Policies

For the evaluation, 200 different polices with policy sizes ranging from 100 – 24000 rules were generated as suggested in AL-Shaer's experiment (44). In their work, there is no suggestion on how to determine the total rules per policy. The first option was to produce them in a deterministic fashion with uniform increments as follows:

$$totalRules = \sum_{n=0}^{totalPolicies-1} (minRules + n\,increment)$$

$$increment = \frac{maxRules - \min Rules}{totalPolicies}.$$

Here minRules and maxRules are the minimum and maximum desired number of rules (100 – 24000). However, it is believed that firewalls having

the same number of rules are highly improbable in practice, as each user builds its own firewall policy depending on their particular needs and objectives. In an attempt to mimic real world firewall policies, a non-deterministic component was added to our policy generator:.

$$policy_i = policy_{i-1} + increment * (1 + randomBetween(-0.01,0.01))$$

$$policy_0 = minRules$$

The function randomBetween specifies a random number between -0.01 and 0.01. This function allows the generated policy rules to vary within a 1% range from the deterministic value. Figure 6.3 illustrates the number of rules per policy for each experiment:

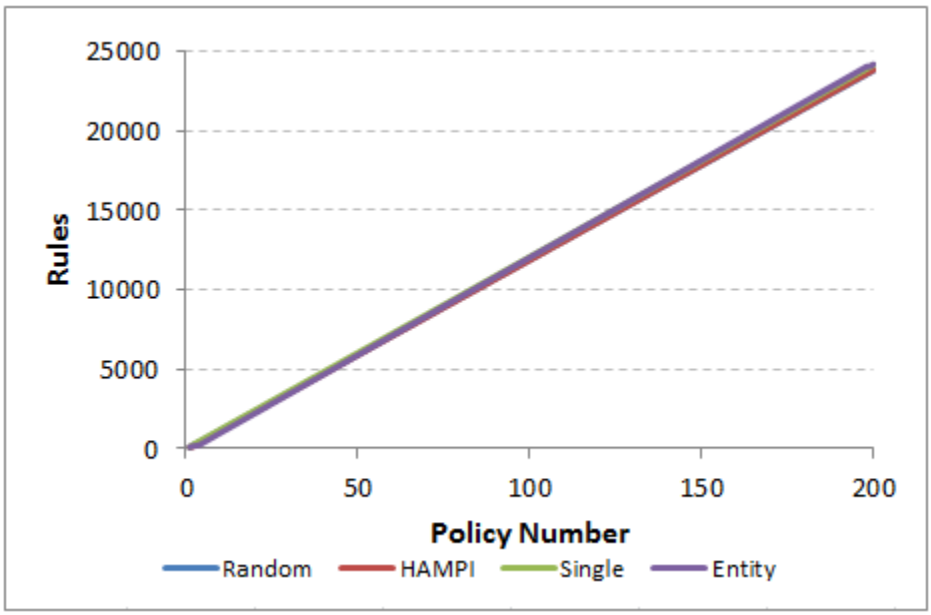



**Figure 6.3 Rules per policy empirical distribution**

For our analysis, the percentage (relative) error will be calculated as follows:

$$increment = \left|\frac{\upsilon - \upsilon_{approx}}{\upsilon}\right| \times 100.$$

Here $\upsilon$ is the expected value and $\upsilon_{approx}$ is the empirical value, this methodology will be used for all the calculations of percentage error.

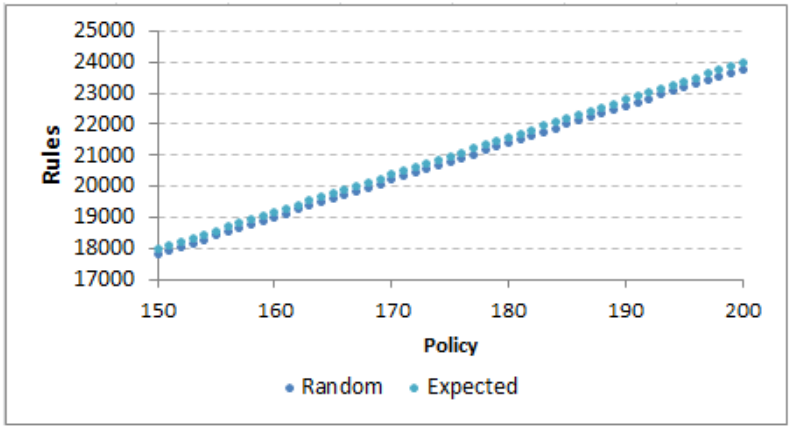
Back to this criterion, the percentage error between the deterministic algorithm and each one of our experiments was calculated; the results are

shown in Table 6.14. A comparison between the expected distribution and the results for each of the experiments is presented in Figure 6.4.

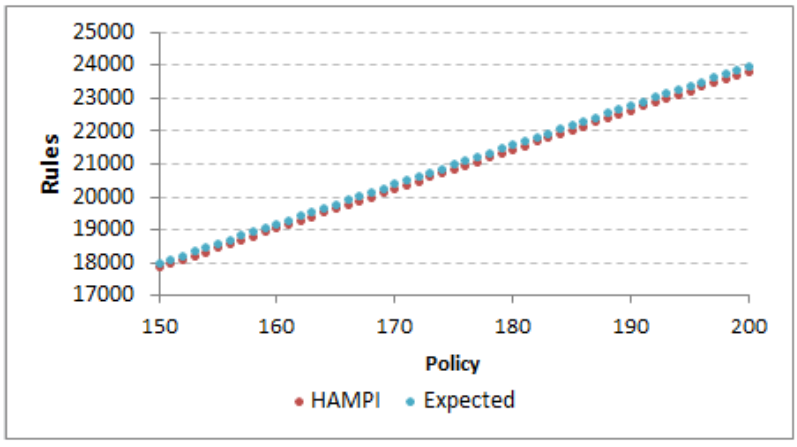
| Total Rules | | |
|---|---|---|
| **Experiment** | **Rules** | **%.E.** |
| Random | 2,388,100 | 0.83% |
| HAMPI | 2,386,518 | 0.89% |
| Single | 2,427,137 | 0.79% |
| Entity | 2,418,275 | 0.43% |
| *Totals* | 9,620,030 | 0.12% |
| | | |
| Ideal | 2,408,000 | 0.00% |
| Ideal Total | 9,632,000 | 0.00% |

**Table 6.14 Total rules per experiment and their percentage error.**

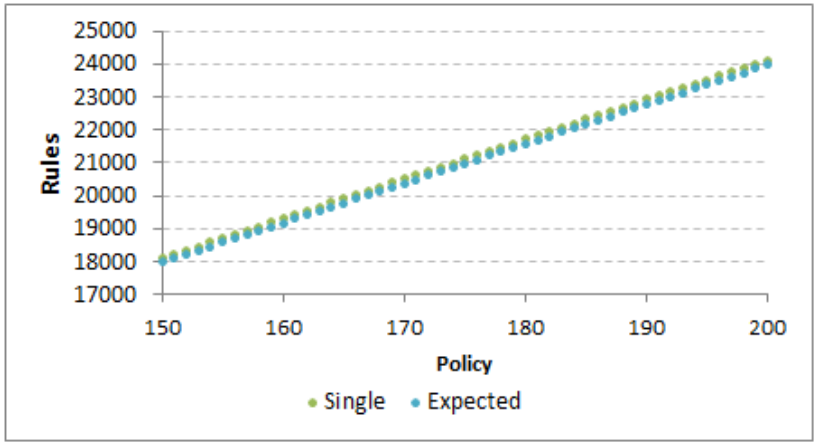
A




B



C




D

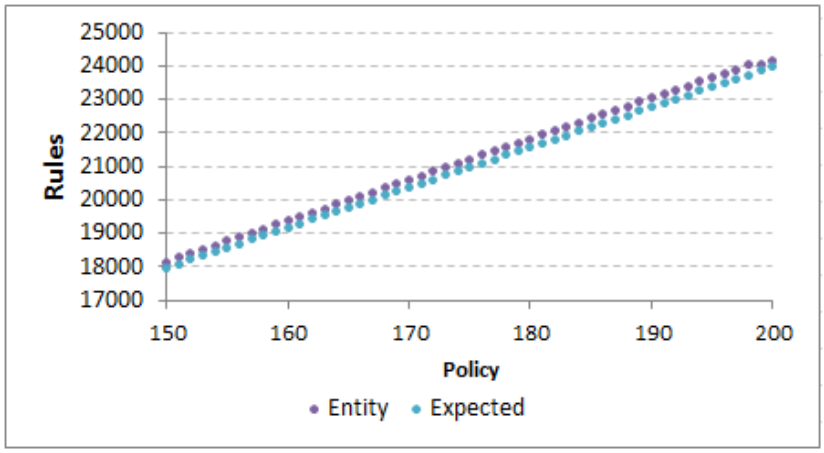



**Figure 6.4 A-D Detail for policy empirical distribution for policies in the range from 150 to 200.**

It is clear that the percentage error for any of the experiments is always below 1%; this is an acceptable result for our purposes here.

## 6.4.2 Rules Distribution

The results for the four experiments are illustrated in Table 6.15; where we can find annotated the number of non-unique firewall rules. From simple inspection, it can be seen that the experiment which presents the greater number of not unique cases is the "random" experiment with 14286 for the total solution space generated. While the "HAMPI" experiment shows 474 not-unique cases. These two results contrast with the results obtained from the analysis of the "single" and "entity" experiments where the count drops to 5 and 1 respectively.

The reason for these results was discussed in section 4.2.2 where it was established how quasi-random sequences maximize the coverage of the solution space. This is asserted in these results where the experiments where pseudo-random algorithms are employed for test case generation

produce a greater number of not-unique results than the two experiments which employ quasi-random sequences ("single" and "entity").

| Generated | Random | HAMPI | Single | Entity |
|---|---|---|---|---|
| 100000 | 153 | 85 | 0 | 0 |
| 200000 | 386 | 111 | 0 | 0 |
| 300000 | 661 | 156 | 2 | 0 |
| 400000 | 1004 | 185 | 3 | 0 |
| 500000 | 1392 | 188 | 3 | 0 |
| 600000 | 1836 | 190 | 3 | 0 |
| 700000 | 2306 | 191 | 4 | 0 |
| 800000 | 2791 | 192 | 4 | 0 |
| 900000 | 3369 | 194 | 4 | 0 |
| 1000000 | 3955 | 194 | 4 | 0 |
| 1100000 | 4590 | 196 | 4 | 0 |
| 1200000 | 5261 | 227 | 5 | 1 |
| 1300000 | 5922 | 261 | 5 | 1 |
| 1400000 | 6651 | 261 | 5 | 1 |
| 1500000 | 7426 | 261 | 5 | 1 |
| 1600000 | 8224 | 261 | 5 | 1 |
| 1700000 | 9010 | 339 | 5 | 1 |
| 1800000 | 9793 | 341 | 5 | 1 |
| 1900000 | 10666 | 410 | 5 | 1 |
| 2000000 | 11513 | 449 | 5 | 1 |
| 2100000 | 12376 | 449 | 5 | 1 |
| 2200000 | 13311 | 474 | 5 | 1 |
| 2300000 | 14286 | 474 | 5 | 1 |

**Table 6.15 Each experiment is annotated with its accumulated count of not-unique rules for each interval of 100000 firewall rules.**

The differences between the random and HAMPI experiments respond to the use of constraints by the "HAMPI" experiment as the constraints enforce diversity in the rule structure. As discussed in section 6.3.2.2, an enumeration of the possible symbolic sentences was made, using this enumeration guarantees that all the possible structures are used and this favours diversity as is demonstrated in the results. For explaining the differences between the "single" and "entity" approaches, we should focus on the **concrete sentence generation phase** where the systems employ different approaches for symbolic constants instantiation. The "single" experiment separately instantiates symbolic constants with a quasi-random sequence. In contrast, the "entity" experiment instantiates entities grouped symbolic constants (entities) with a quasi-random sequence. The solution

space for an entity is greater than the solution space of a single symbolic constant (as the solution space for an entity is the Cartesian product of the grouped symbolic constants) thus, the possibilities of generating not-unique values decreases dramatically when employing entities, as asserted by the empirical results.

The evidence demonstrates the superiority of the implemented solution over previous approaches.

### 6.4.3 Rule Length

In Table 6.16, it can be appreciated that the "HAMPI" and "single" experiments show expected behaviour in their results as the rule length depends solely in the **symbolic sentences generation phase**, thus there should be no substantial difference as they both employ the same symbolic grammar and symbolic values grammar for their generation. These trends start from near a frequency of zero for rules with length 16 and rise to a frequency near 30% for length of 20. The frequency reduces to 29% for a length 21 and finally drops to a value of 10% for length 22. The behaviour is a direct result of how the system creates sentences from the symbolic grammar using McKenzie's algorithm (24) which is set by the "sentence length" parameter (discussed in section 6.3.2.3) to favour the generation firewall rules with this rule length.

| Rule Length | Expected | Random | HAMPI | Single | Entity |
|---|---|---|---|---|---|
| **16** | 2.778% | 2.738% | 0.465% | 0.651% | 1.955% |
| **17** | 5.556% | 5.544% | 1.919% | 2.388% | 7.369% |
| **18** | 19.444% | 19.311% | 9.012% | 9.553% | 16.026% |
| **19** | 22.222% | 22.297% | 19.250% | 19.142% | 18.772% |
| **20** | 30.556% | 30.524% | 29.214% | 30.179% | 38.612% |
| **21** | 16.667% | 16.762% | 30.347% | 28.967% | 15.102% |
| **22** | 2.778% | 2.825% | 9.794% | 9.119% | 2.165% |

**Table 6.16 Rule length empirical distribution**

The "entity" experiment behaviour responds directly to the enhancements of our solution over the symbolic sentences generation phase which instantiates grouped symbolic constants (entities) instead of instantiating them separately. Impact from the **symbolic sentences generation phase** is null as there are no constraints and the symbolic grammar consists on a single production rule. This fact leaves the full control of how the entities are solved to the **concrete sentences generation phase**. This phase solves the rules as follows:

1. A rule is chosen.

2. Its possible expansions are associated with a unique quasi-random sequence.
3. The engine selects the expansion based on the associated quasi-random sequence.

This guarantees that all the productions have the same probability, as the quasi-random sequences enforces that the entire solution space is covered before repeating values. This behaviour explains the distribution which matches well with the one expected.

In order to provide a more formal comparison between these results, the percentage error is calculated in Table 6.17.

For each experiment each value is shown with their respective percentage error, in the last rows of the table the average, median and accumulated percentage error for each experiment is annotated.

| Bits | Random | HAMPI | Single | Entity |
|---|---|---|---|---|
| 16 | 1.45% | 83.26% | 76.55% | 29.62% |
| 17 | 0.21% | 65.46% | 57.01% | 32.64% |
| 18 | 0.69% | 53.65% | 50.87% | 17.58% |
| 19 | 0.34% | 13.37% | 13.86% | 15.53% |
| 20 | 0.10% | 4.39% | 1.23% | 26.37% |
| 21 | 0.57% | 82.08% | 73.80% | 9.39% |
| 22 | 1.71% | 252.57% | 228.28% | 22.06% |
| **Average** | 0.72% | 79.26% | 71.66% | 21.88% |
| **Median** | 0.57% | 65.46% | 57.01% | 22.06% |
| **Accumulated** | 5.06% | 554.79% | 501.60% | 153.18% |

**Table 6.17 Rule Length percentage error.**

The random experiment shows almost complete agreement with the expected distribution as evidenced by the negligible error values ranging between 0.1% and 1.71%. Entity follows with the second lowest percentage error having a maximum of 32.64%. The "HAMPI" and "single" approaches have values that have a median of 65.46% and 57.01% respectively, which when compared to the median of "random" (0.57%) and "entity" (22.06%) makes them the less acceptable cases.

This behaviour is due to the nature of the systems where the following elements affect the results:

1. Grammar depth - The "random" approach becomes less accurate when the number of rules needed to reach a concrete value increases (this will be corroborated in further criteria i.e. address size). To make the selection of rule length the production rule PORT must be visited; thus to

reach it only one derivation must be made starting from the starting symbol. The number of needed rules substitutions is the minimum favouring an almost null percentage error.

2. The error shown is related directly to how symbolic sentences are produced from McKenzie's algorithm. It has been discussed that this algorithm favours the production of grammar sentences with a specified sentence length which can be appreciated by the obtained results for "HAMPI" and "single" experiments. Our system is based on this algorithm, therefore the bias is still present if no correction algorithm is implemented in the original system. Our solution corrects it employing entities which are solved using their associated quasi-random sequences; this approach minimizes the impact of McKenzie's algorithm over the "entity" results giving an optimization of 72.34% over the "HAMPI" accumulated percentage error.

### 6.4.4 Symbolic Sentences Representation

In section 4.2.4, it was discussed that our solution proposes a concrete sentence generation control mechanism to control the symbolic sentences instantiation mechanism. This mechanism was proposed in order to solve the limitations encountered in previous approaches where the total number of concrete sentences instantiated from a symbolic sentence is not controlled. To measure the impact of our enhancements, we traced the number of concrete sentences instantiated from each symbolic sentence; these results are shown in Table 6.18 with their averages and standard deviations for each rule length.

| | | Random | HAMPI | Single | Entity |
|---|---|---|---|---|---|
| **10** | **Average** | 1816.028 | 322.444 | 329.000 | 1839.000 |
| | **St. Dev.** | 659.934 | 138.214 | 0.000 | 0.000 |
| **11** | **Average** | 919.368 | 331.188 | 339.000 | 931.000 |
| | **St. Dev.** | 339.970 | 137.572 | 0.000 | 0.000 |
| **12** | **Average** | 800.623 | 377.564 | 385.000 | 811.000 |
| | **St. Dev.** | 360.481 | 162.585 | 0.000 | 0.000 |
| **13** | **Average** | 462.221 | 397.472 | 404.359 | 468.000 |
| | **St. Dev.** | 169.071 | 170.895 | 0.480 | 0.000 |
| **14** | **Average** | 389.394 | 376.815 | 383.000 | 394.000 |
| | **St. Dev.** | 181.164 | 168.450 | 0.000 | 0.000 |
| **15** | **Average** | 231.645 | 413.402 | 420.000 | 235.000 |
| | **St. Dev.** | 85.142 | 169.790 | 0.000 | 0.000 |
| **16** | **Average** | 117.132 | 402.948 | 410.000 | 118.207 |
| | **St. Dev.** | 44.849 | 157.628 | 0.000 | 0.405 |

**Table 6.18 Symbolic sentences representation average and standard deviation by rule length.**

From the obtained results, it can be observed that the "random" experiment presents standard deviations that range from 44.849 (for length 16) to 659.934 (for length 10). This shows how random generation does not favour any specific symbolic structure resulting in uncertainty in their final representation (it should be mentioned that the "random" experiment has no symbolic sentence generation phase, therefore its equivalent symbolic sentences had to be computed from the generated concrete sentences).

The HAMPI experiment shows standard deviations which range from 137.572 to 170.895, these deviations illustrate that the results from the "random" experiment have been significantly improved. This improvement is a direct consequence of constraint usage which forces all symbolic structures to be used; but the standard deviation still shows the uneven coverage of these structures.

For "single" and "entity" experiments, the standard deviations reach values of 0 in all lengths but in two. This is expected as the concrete sentence generation control mechanism implemented guarantees that symbolic sentences are covered evenly:

1. Symbolic sentences are received from the **symbolic sentences generation phase** by the **concrete sentences generation phase**.
2. Pre-concrete sentences are derived from the symbolic sentences.
3. The experiments' total rules are calculated.
4. For each symbolic sentence, the number of rules to be instantiated are calculated dividing the total rules between the total derived pre-concrete sentences.

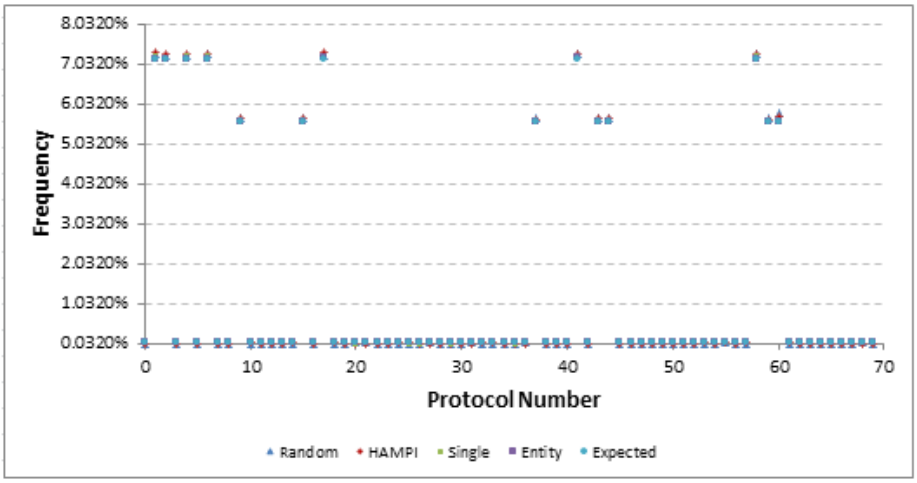
Step 4 of the methodology uses a deterministic algorithm; therefore each of the symbolic sentences is covered with the same rules, resulting in a standard deviation of zero. Standard deviations that are not zero (length 13 for "single" and 16 for "entity") are explained by the fact that step 4 can return non-integer values which have to be adjusted to deliver the exact quantity of total rules. The average between these two experiments vary (despite their standard deviations) due to pre-concrete sentences derivation which is based on symbolic constants for the "single" experiment and entities for the "entity" experiment (entities group symbolic constants producing fewer combinations that ungrouped symbolic constants).

## 6.4.5 Individual Rule Fields Analysis

Evaluation of the behaviour of individual firewall fields is required to fully analyze the effects of our proposed approach; a discussion for evaluation criteria which analyze individual firewall rule fields is presented in this section.

### *6.4.5.1 Protocol Numbers*

**A) Range from 0 to 70**




**A) Range from 70 to 255**

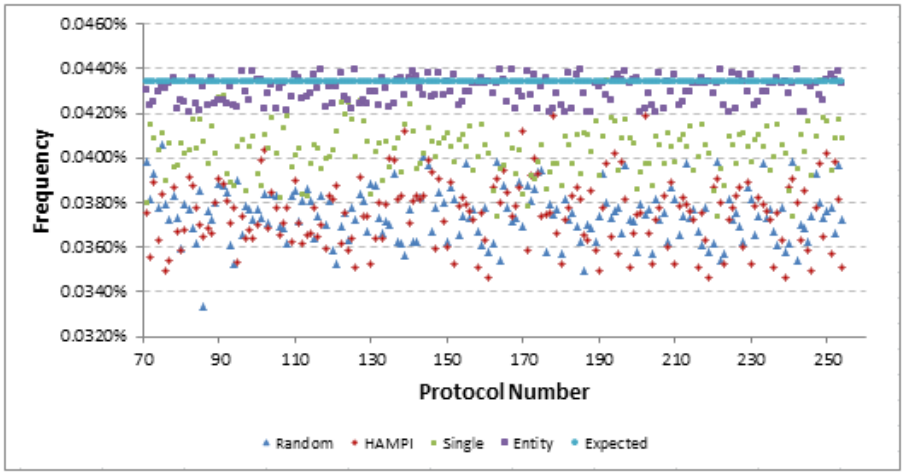



**Figure 6.5 A-B Protocol numbers empirical distribution**

| Experiment | Rule |
|---|---|
| Random | `RULE = PROTOCOL_NUMBER SOURCE DESTINATION ACTION` |
| HAMPI and Simple | `RULE = <PROTOCOL_NUMBER> SOURCE DESTINATION <ACTION>` |
| Entity | `RULE = <PROTOCOL_NUMBER> <SOURCE>` |

| | |
|---|---|
| | <DESTINATION> <ACTION> |

**Table 6.19 Changes made in the different grammars.**

| | Random | HAMPI | Single | Entity |
|---|---|---|---|---|
| **Average** | 12.178% | 12.875% | 10.428% | 1.249% |
| **Median** | 12.012% | 11.534% | 10.156% | 1.022% |
| **Mode** | 9.793% | 11.534% | 15.257% | 0.471% |
| **Standard Deviation** | 4.231% | 4.163% | 4.449% | 0.917% |
| **MAX** | 24.457% | 21.514% | 15.406% | 3.160% |
| **MIN** | 0.025% | 0.027% | 0.544% | 0.031% |
| **Accumulated** | 3117.585% | 3296.078% | 2669.581% | 319.775% |

**Table 6.20 Protocol numbers first-order statistics percentage error for its 256 elements solution space.**

Figure 6.5 presents the empirical results for the four experiments compared to their expected distribution. This behaviour can be attributed to the changes illustrated in Table 6.19 which took place during the derivation of the symbolic grammars. The symbolic constant <PROTOCOL_NUMBER> did not change significantly with the derivations of the symbolic grammars from the context-free grammar used in the "random" experiment and this can be appreciated in the results. Percentage error distribution is calculated to evaluate how they compare to the expected ones, which are given in Table 6.20. The table illustrates that the experiment which shows the least accumulated percentage error from the expected values is the "entity" experiment varying with a value of 319.78%, followed by the "single" experiment with a value of 2669.58%. This result is confirmed by the median and average values where the "entity" experiments raises as the best of the 4 experiments.

For this criterion, the solution space coverage was omitted as the protocol numbers have a solution space of 256 elements is completely covered by each of the experiments' generated values as shown in Figure 6.5.

### *6.4.5.2 IP Address Structure*

| Wild cards | Pattern | Expected | Random | HAMPI | Single | Entity |
|---|---|---|---|---|---|---|
| 0 | #.#.#.# | 78.13% | 77.902% | 42.006% | 41.580% | 77.718% |
| 1 | #.#.#.* | 2.34% | 2.387% | 6.210% | 6.325% | 1.801% |
| | #.#.*.# | 2.34% | 2.389% | 6.220% | 6.271% | 3.448% |
| | #.*.#.# | 2.34% | 2.378% | 6.231% | 6.259% | 1.806% |

| | | | | | | |
|---|---|---|---|---|---|---|
| | *.#.#.# | 2.34% | 2.399% | 6.219% | 6.264% | 4.390% |
| 2 | #.#.*.* | 1.56% | 1.568% | 4.138% | 4.179% | 0.915% |
| | #.*.#.* | 1.56% | 1.564% | 4.181% | 4.172% | 1.283% |
| | #.*.*.# | 1.56% | 1.581% | 4.145% | 4.173% | 0.591% |
| | *.#.#.* | 1.56% | 1.564% | 4.147% | 4.172% | 0.271% |
| | *.#.*.# | 1.56% | 1.575% | 4.134% | 4.147% | 1.390% |
| | *.*.#.# | 1.56% | 1.588% | 4.126% | 4.187% | 1.468% |
| 3 | #.*.*.* | 0.78% | 0.770% | 2.064% | 2.076% | 0.964% |
| | *.#.*.* | 0.78% | 0.768% | 2.041% | 2.053% | 2.523% |
| | *.*.#.* | 0.78% | 0.767% | 2.077% | 2.063% | 0.717% |
| | *.*.*.# | 0.78% | 0.792% | 2.049% | 2.070% | 0.707% |
| 4 | *.*.*.* | 0.00% | 0.000% | 0.000% | 0.000% | 0.000% |

**Table 6.21 IP address structure empirical distribution**

The bias for generating rules with no wild cards can be observed in Table 6.21 where for the case of no wild cards (#.#.#.#), "HAMPI" and "single" perform significantly poorer than "random" and "entity". For the "random" experiment, the behaviour is as expected as biased rules are more likely to be chosen (as discussed in section 6.3.3.4). The behaviour for the "entity" experiment responds directly to the fact that for the concrete sentence generation phase three independent groups are considered (as discussed in section 6.3.3.3). The behaviour for the "HAMPI" and "single" experiments shows the impact of the biased rules, but with less concentration of data in those points. This is due to:

1. No groups are considered as in the "entity" experiment; and
2. Rules are selected during the symbolic sentence generation phase employing McKenzie's algorithm whose selection algorithm favours a uniform selection of production rules regardless of the grammar definition. It can be appreciated that for the "entity" experiment McKenzie's algorithm is not employed for rule selection but rather quasi-random sequences, which results in the less significant accumulated percentage error (shown in Table 6.22).

| | **Random** | **HAMPI** | **Single** | **Entity** |
|---|---|---|---|---|
| **Average** | 1.24% | 147.25% | 150.10% | 1.19% |
| **Median** | 1.20% | 164.68% | 167.84% | 1.07% |
| **Standard Deviation** | 0.9458% | 47.642% | 48.589% | 0.8242% |
| **MAX** | 3.02% | 166.42% | 171.75% | 3.47% |
| **MIN** | 0.00% | 0.00% | 0.00% | 0.00% |
| **Accumulated** | 19.92% | 2355.98% | 2401.53% | 19.10% |

**Table 6.22 IP address structure first-order statistics percentage error.**

An inspection to Table 6.22 shows that the experiments which present the largest accumulated percentage error are "HAMPI" (2355.98%) and "single" (2401.53%), which are non-desirable results, while "random" (19.92%) and "entity" (19.10%) both show expected dispersion. Comparing the three presented values in to Table 6.22 the "entity" experiment has the most acceptable data of the four compared experiments.

IP address structure's solution space coverage was omitted as its solution space consists solely of 16 elements which is completely covered by the experiments' generated values as shown in Table 6.21.

#### *6.4.5.3 IP Address Size*

The results for both source and destination are the same (as the same definition is used for both) therefore we will only discuss the results for the source case. The numbers of bits for each source IP and their distributions are presented in Table 6.23. A special point worth mentioning is the notable concentration of values in the address size ten, which corresponds to the IP value of localhost (see the initial grammar for its definition).

| Bits | Expected | Random | HAMPI | Single | Entity |
|---|---|---|---|---|---|
| **4** | 0.017% | 0.001% | 0.001% | 0.001% | 0.000% |
| **5** | 0.066% | 0.003% | 0.003% | 0.003% | 0.002% |
| **6** | 0.166% | 0.007% | 0.008% | 0.008% | 0.005% |
| **7** | 0.332% | 0.017% | 0.020% | 0.019% | 0.012% |
| **8** | 0.580% | 0.036% | 0.043% | 0.042% | 0.017% |
| **9** | 0.928% | 0.076% | 0.091% | 0.088% | 0.061% |
| **10** | 33.492% | 33.070% | 20.168% | 21.682% | 33.241% |
| **11** | 1.989% | 0.331% | 0.400% | 0.390% | 0.355% |
| **12** | 2.669% | 0.599% | 0.726% | 0.709% | 0.651% |
| **13** | 3.382% | 0.963% | 1.170% | 1.143% | 1.053% |
| **14** | 4.078% | 1.427% | 1.736% | 1.697% | 1.573% |
| **15** | 4.708% | 1.995% | 2.429% | 2.376% | 2.194% |
| **16** | 5.222% | 2.670% | 3.252% | 3.182% | 2.916% |
| **17** | 5.570% | 3.445% | 4.196% | 4.107% | 3.753% |
| **18** | 5.703% | 4.283% | 5.216% | 5.109% | 4.576% |
| **19** | 5.570% | 4.937% | 6.012% | 5.890% | 5.197% |
| **20** | 5.222% | 5.410% | 6.584% | 6.454% | 5.622% |
| **21** | 4.708% | 5.699% | 6.928% | 6.794% | 5.846% |
| **22** | 4.078% | 5.794% | 7.033% | 6.900% | 5.848% |
| **23** | 3.382% | 5.683% | 6.881% | 6.755% | 5.669% |
| **24** | 2.669% | 5.343% | 6.444% | 6.329% | 5.201% |
| **25** | 1.989% | 4.747% | 5.690% | 5.592% | 4.494% |
| **26** | 1.392% | 3.858% | 4.579% | 4.502% | 3.601% |

| | | | | | |
|---|---|---|---|---|---|
| **27** | 0.928% | 3.071% | 3.581% | 3.522% | 2.809% |
| **28** | 0.580% | 2.379% | 2.692% | 2.649% | 2.080% |
| **29** | 0.332% | 1.774% | 1.913% | 1.883% | 1.501% |
| **30** | 0.166% | 1.246% | 1.244% | 1.226% | 0.928% |
| **31** | 0.066% | 0.776% | 0.692% | 0.682% | 0.585% |
| **32** | 0.017% | 0.359% | 0.268% | 0.264% | 0.210% |

**Table 6.23 IP Address size empirical distribution.**

The concentration for size 10 for each experiment can be analyzed as follows:

- The "random" experiment has a major concentration without reaching the expected 33%. This was expected as this experiment fits into the probabilities shown in Table 5.22 due to its pseudo-random behaviour.
- The "HAMPI" and "single" experiments behave as previous analysis have shown; but presenting a concentration of 21% for this value which is not expected. This concentration corresponds to how the **symbolic sentence generation phase** favours sentences with greater length.
- The "entity" experiment has a concentration of 33.241% which shows the impact of the **concrete sentences generation phase** which instantiates IP address as entities instead of a concatenation of isolated symbolic constants (like in the "HAMPI" and "single" experiments).

| | **Random** | **HAMPI** | **Single** | **Entity** |
|---|---|---|---|---|
| **Average** | 219.49% | 208.58% | 205.31% | 164.53% |
| **Median** | 91.80% | 92.82% | 92.54% | 93.44% |
| **SD** | 412.64% | 321.61% | 316.46% | 247.49% |
| **MAX** | 2067.9% | 1517.2% | 1494.7% | 1168.8% |
| **MIN** | 1.3% | 7.9% | 5.8% | 0.7% |
| **Accumulated** | 6365.16% | 6048.89% | 5954.05% | 4771.43% |

**Table 6.24 IP address size percentage error first order statistics for its 28 elements solution space..**

The percentage error for this criterion is presented in Table 6.24; from where it can be seen that the experiment with lowest accumulated error is "entity" with 4771.43% followed by "single" (5954.05%), "HAMPI" (6048.89%) and "random" (6365.16%) in that order. The only measurement where "entity" experiment does not raises as the most desirable result is the median which is the largest of the four, even though the distance between the most desirable median ("random" with 91.80) and its value is of 1.64% which compared to the distance for the best and worst

cases for the average (54.96%) and accumulated percentage error (1593.73%) is acceptable.

As for previous criteria, the IP address size solution space coverage was omitted as its solution space consists of 18 elements which is covered by each of the experiments' generated values as shown in Table 6.23.

#### *6.4.5.4 IP Values Distribution*

| Experiment | Generated Rules | Expected Rules to compute | Computed Rules | Percentage Error |
|---|---|---|---|---|
| **Random** | 2388100 | 1791075 | 1770463 | 1.1642% |
| **HAMPI** | 2386518 | 1789889 | 2000990 | 10.5498% |
| **Single** | 2427137 | 1820353 | 2041792 | 10.8453% |
| **Entity** | 2418275 | 1813706 | 1812737 | 0.0534% |

**Table 6.25 Computed rules for IP values distribution analysis**

| | Random | HAMPI | Single | Entity |
|---|---|---|---|---|
| **100000** | 96.6858% | 89.8705% | 90.8820% | 97.9810% |
| **200000** | 95.9600% | 87.9546% | 89.5082% | 97.5742% |
| **300000** | 95.5879% | 86.6787% | 88.0045% | 97.4573% |
| **400000** | 95.2802% | 85.6105% | 86.7587% | 97.3214% |
| **500000** | 95.0264% | 84.5209% | 85.4649% | 97.1685% |
| **600000** | 94.7717% | 83.5607% | 84.4914% | 97.0972% |
| **700000** | 94.5557% | 82.6068% | 83.5076% | 97.0893% |
| **800000** | 94.3316% | 81.6135% | 82.6303% | 96.7956% |
| **900000** | 94.1122% | 80.6315% | 81.7886% | 96.9356% |
| **1000000** | 93.9088% | 79.8810% | 80.9631% | 96.8841% |
| **1100000** | 93.7046% | 78.9748% | 80.1387% | 96.8300% |
| **1200000** | 93.5296% | 78.3025% | 79.3554% | 96.8411% |
| **1300000** | 93.3415% | 77.6474% | 78.6020% | 96.7850% |
| **1400000** | 93.1786% | 76.9266% | 77.8900% | 96.7973% |
| **1500000** | 93.0119% | 76.4009% | 77.2624% | 96.8169% |
| **1600000** | 92.8407% | 75.7481% | 76.6177% | 96.7447% |
| **1700000** | 92.6883% | 75.2494% | 75.9966% | 96.7458% |

**Table 6.26 Unique rules percentage empirical distribution values**

The solution space covered by each one of the experiments is of high interest for our comparison, as it will determine the impact of pseudo-random and quasi-random technologies for symbolic constant instantiation.

From the total generated rules, we only analyzed the rules which had a different value from localhost, as this IP address will occupy 25% of the solution space accordingly to the grammar definition (discussed in section

6.3.3.4). For each experiment, its generated rules, expected rules (different from localhost), the actual rules computed and the percentage error between expected and computed rules are shown in Table 6.25. This table shows a percentage error for the "HAMPI" and "computed" which can be explained by the low concentration of IP address in length 10 (corresponding to the localhost IP address) as discussed in section 6.3.3.5.

| | **Possible** | **Random** | **HAMPI** | **Single** | **Entity** |
|---|---|---|---|---|---|
| **100000** | 0.00233% | 0.0023% | 0.0021% | 0.0021% | 0.0023% |
| **200000** | 0.00466% | 0.0045% | 0.0041% | 0.0042% | 0.0045% |
| **300000** | 0.00698% | 0.0067% | 0.0061% | 0.0061% | 0.0068% |
| **400000** | 0.00931% | 0.0089% | 0.0080% | 0.0081% | 0.0091% |
| **500000** | 0.01164% | 0.0111% | 0.0098% | 0.0099% | 0.0113% |
| **600000** | 0.01397% | 0.0132% | 0.0117% | 0.0118% | 0.0136% |
| **700000** | 0.01630% | 0.0154% | 0.0135% | 0.0136% | 0.0158% |
| **800000** | 0.01863% | 0.0176% | 0.0152% | 0.0154% | 0.0180% |
| **900000** | 0.02095% | 0.0197% | 0.0169% | 0.0171% | 0.0203% |
| **1000000** | 0.02328% | 0.0219% | 0.0186% | 0.0189% | 0.0226% |
| **1100000** | 0.02561% | 0.0240% | 0.0202% | 0.0205% | 0.0248% |
| **1200000** | 0.02794% | 0.0261% | 0.0219% | 0.0222% | 0.0271% |
| **1300000** | 0.03027% | 0.0283% | 0.0235% | 0.0238% | 0.0293% |
| **1400000** | 0.03260% | 0.0304% | 0.0251% | 0.0254% | 0.0316% |
| **1500000** | 0.03492% | 0.0325% | 0.0267% | 0.0270% | 0.0338% |
| **1600000** | 0.03725% | 0.0346% | 0.0282% | 0.0285% | 0.0360% |
| **1700000** | 0.03958% | 0.0367% | 0.0298% | 0.0301% | 0.0383% |
| **1800000** | 0.04191% | 0.0385% | 0.0311% | 0.0317% | 0.0405% |

**Table 6.27 Percentage solution space covered percentage accumulated values.**

The total number IP source address was classified between unique and not unique IPs; the results are shown in Table 6.26. In this table, it is observed that the "random" and "entity" experiments generate more diverse production rules (91.8886% and 96.7266%) as they have more diversity than the other two experiments (74.1972% and 75.6955%). This is simple to deduce for the "random" experiment, as the solution space contains $2^{32}$ possible values, and the probability of obtaining the same value in several occasions is very small. For the "entity" solution, the result is expected showing the highest rate of unique IP addresses becoming more evident as the count increases. The data also shows that experiments "HAMPI" and "single" yield poor results (74.1972% and 75.6955%). The results differ from each other with a constant value of 1.4983% this is due to the nature of the original engine and our solution: they only differ in how they instantiate symbolic constants during the **concrete sentence phase**

and the simple employment of adaptive random testing gives an immediate enhancement to the HAMPI system. The results for the last row of Table 6.26 shows the results for 1.7 million computed rules (maximum valid number of computed rules for all experiments as shown in Table 6.25), illustrating that the entity experiment is ~5% better than the random experiment which is its closest competitor.

Results are provided for the total solution space covered for our sample in Table 6.27 the table illustrates that the experiment that covers the largest solution space using less rules is the "entity" experiment; which covers 0.0405% of the total possible values (232). This adds to the results obtained in Table 6.26 which demonstrates that the "entity" experiment is the best approach of the four discussed.

#### *6.4.5.5 Ports usage*

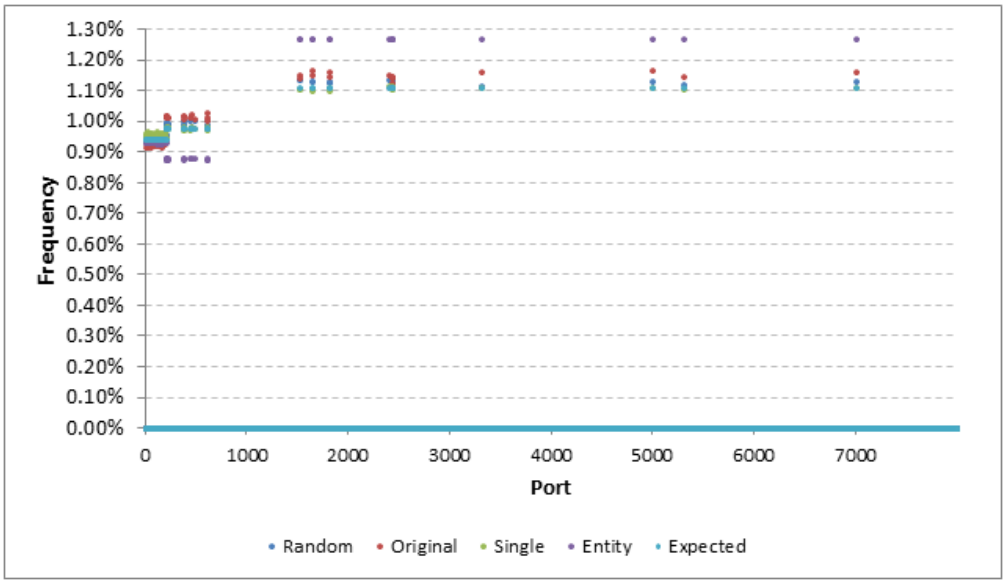



**Figure 6.6 Ports usage empirical distribution for 0-8000 interval**

Figure 6.6 shows the interval for source ports 0-8000 (only the source case is presented since the destination case is almost identical). The interval for ports 8001 to 65535 is omitted given the fact that the values are too large to obtain meaningful in information. For the same reason this analysis we will make use of the percentage error's first order statistics for each experiment which are presented in Table 6.28.

The accumulated error shows such a large values (in the order of millions) as a direct result of the large solution space analyzed (65536 data points); however the average shows that percentage errors were in the order of 35.55%, 38.14%, 30.22% and 19.35% for "random", "HAMPI", "single" and "entity" experiments respectively which is makes this statistic

acceptable. From this accumulated values, the "entity" experiment shows a significant improvement over the other analyzed approaches; this can be attributed to the employment of quasi-random sequences for rule selection during the **concrete sentences generation phase**.

The "entity" experiment bests all experiments in average and median measurements; this behaviour is explained by the quasi-random sequences employed for symbolic constants instantiation selecting between the solution spaces in a controlled fashion. This behaviour is supported by the fact that the smallest standard deviation is present in the "entity" experiment, which indicates that all concrete values have been evenly employed. One of the most significant statistics is the MAX measurement which is ~94% smaller for the "entity" experiment compared to the analyzed algorithms; as previous behaviours, this is explained with the **concrete sentences generation phase** and the use of entities which use quasi-random sequences for symbolic constants instantiation.

Port's solution space coverage was omitted as its solution space consists of 65536 elements which are completely covered by the experiments' generated values as shown in Table 6.28.

| | Random | HAMPI | Single | Entity |
|---|---|---|---|---|
| **Average** | 35.55% | 38.14% | 30.22% | 19.35% |
| **Median** | 31.36% | 32.15% | 23.02% | 11.40% |
| **Mode** | 17.90% | 5.61% | 12.03% | 11.40% |
| **SD** | 33.60% | 34.58% | 30.22% | 14.89% |
| **MAX** | 1147.96% | 1127.11% | 1098.63% | 59.14% |
| **MIN** | 0.0261% | 0.1300% | 0.0013% | 1.3302% |
| **Accumulated** | 2329537.74% | 2499830.26% | 1980208.34% | 1268438.12% |

**Table 6.28 Ports usage percentage error dispersion first order statistics for its 65536.**

### *6.4.5.6 Ports Operators Distribution*

| Operations | Random | HAMPI | Single | Entity |
|---|---|---|---|---|
| **< #** | 16.818% | 15.956% | 15.053% | 16.657% |
| **!= #** | 16.564% | 16.158% | 14.909% | 16.578% |
| **> #** | 16.537% | 14.796% | 15.433% | 16.471% |
| **None** | 16.558% | 6.871% | 7.997% | 16.687% |
| **# - #** | 16.762% | 31.537% | 30.975% | 16.623% |
| **#** | 16.761% | 14.682% | 15.632% | 16.676% |

| | | | | |
|---|---|---|---|---|
| **Average** | 0.682% | 29.740% | 28.617% | 0.367% |
| **Median** | 0.654% | 11.906% | 10.548% | 0.261% |
| **Accumulated** | 4.774% | 208.182% | 200.320% | 2.569% |

**Table 6.29 Ports Operators empirical distribution values and percentage error.**

The frequency for each operator is described in Table 6.29. As expected, the "random" experiment has an almost perfect uniform distribution; this stems from the fact that all the values have an equal probability of being generated. For the "HAMPI" and "single" experiments the symbolic sentences generation phase algorithms demonstrates how McKenzie's algorithm favours larger rules, resulting in a higher concentration of values in the "# - #" (range) value which, as discussed in section 6.3.4.1, derives in sentences with greater length; this explains their large accumulated percentage error values..

Finally, the "entity" experiment illustrates the variation that the **concrete sentences generation phase** introduces within the solution. Given the fact that quasi-random sequences were implemented, a more even selection resulted and yielded a minimal accumulated percentage error. This result was expected since rules are chosen employing an associated quasi-random algorithm

Like in previous criteria, the ports operators' solution space coverage was omitted as its solution space consists of six elements which are covered by each of the experiments' generated values as shown in Table 6.29.

### *6.4.5.7 Action distribution*

Action has only two possible values; its results are shown in Table 6.30 and their relative percentage errors are presented in Table 6.31.

| | **Random** | **HAMPI** | **Single** | **Entity** |
|---|---|---|---|---|
| **Deny** | 50.355% | 49.987% | 49.996% | 50.00002 |
| **Allow** | 49.645% | 50.013% | 50.004% | 49.99998 |

**Table 6.30 Action empirical distribution.**

| | **Random** | **HAMPI** | **Single** | **Entity** |
|---|---|---|---|---|
| **Deny** | 0.7093% | 0.0258% | 0.0086% | 0.00004% |
| **Allow** | 0.7093% | 0.0258% | 0.0086% | 0.00004% |

**Table 6.31 Action distribution percentage values.**

Differences between pseudo-random and quasi-random approaches become evident with this analysis. The two experiments which use pseudo-random algorithms for concrete constants generation ("Random" and "HAMPI") show a percentage error which contrasts with the percentage error near 0% of approaches that employ quasi-random sequences. This behaviour is as expected, as quasi-random sequences select, in an orderly fashion, values from the solution space while pseudo-random select between these values with no order making it more probable to favour any of both values. This behaviour is corroborated by our results where the

"random" experiment shows a percentage error of 0.7093% while the "entity" experiments show the minimal error of 0.00004%.

Action's solution space coverage was omitted as it has a binary solution space therefore it is trivial to cover it within a few generated rules.

## 6.4.6 Discussion

Several conclusions have been drawn for each individual evaluation criterion. A discussion of the experiments utilizing several criterions will be presented; which will provide an overview of the system's performance.

### *6.4.6.1 HAMPI's Pseudo-Random vs. Single's Quasi Random*

The behaviour of both "HAMPI" and "single" experiments seems to follow a similar trend in all criteria. Between the two, the "single" experiment is superior because it has a better distribution of values with less percentage error for all the analysis presented. This can be explored by comparing each of the experiments percentage error medians for each criterion.

| | **Pseudo Random** | **Quasi Random** |
|---|---|---|
| **Rule Length** | 65.46% | 57.01% |
| **Protocol Numbers** | 11.534% | 10.156% |
| **IP Address Structure** | 164.68% | 167.84% |
| **IP Address Size** | 92.82% | 92.54% |
| **Ports Usage** | 32.15% | 23.02% |

**Table 6.32 Pseudo-random and quasi-random percentage error medians.**

This can be attributed to the final steps of **concrete sentence generation**, where the "HAMPI" experiment uses a traditional pseudo-random approach, while the "single" experiment uses quasi-random sequences.

### *6.4.6.2 Computational Cost*

The computational cost of generating test cases based on quasi-random sequences remains inexpensive for all experiments. For the creation of 200 policies, an Intel Dual Core 2.3GHz with 4MB RAM was used and the computational times for each experiment are contained in Table 6.33.

| **Experiment** | **Minutes** |
|---|---|
| Random | 107 |
| HAMPI | 5 |

| | |
|---|---|
| Single | 440 |
| Entity | 36 |

**Table 6.33 Experiment's performance in minutes**

While the performance will vary with the specific platform, the results will remain consistent with the implementation presented here. The "random" generation has neither constraints nor production control algorithms, so it visits all of the rules that are required each time it generates a sentence from the grammar. Thus it ranks as the second slowest of the four. The "HAMPI" experiment has the best performance, requiring 5 minutes to generate the 200 policies. This provides a point of reference between the original system and the enhancements.

The "single" experiment has the slowest generation ranking in . The "entity" experiment ranks second requiring only 36 minutes to generate the 200 policies. These two experiments share the **symbolic sentence generation** phase. The only difference is the number of symbolic sentences that are instantiated during the **concrete sentence generation** phase. In this phase, a quasi-random sequence has to be associated to each element, while for the "single" experiment it associates a sequence to each symbolic constant found in the symbolic sentences. For the "entity" experiment, it associates one for each entity. This has a direct impact when producing large results. Therefore, it can be concluded that, for our implementation, that **concrete sentence generation** is where the system invests most of its resources. Therefore, its performance is proportional to the number of symbolic sentences produced.

## 6.4.7 Limitations

A series of limitations arose during the experimentation process, and the most significant are summarized in this section.

### *6.4.7.1 Grammar restrictions*

The number of rules that a firewall policy must have is not constant. Therefore, firewall grammars define rules structure for a single firewall rule. If a firewall policy is needed then the firewall rule grammar must produce as many sentences as required to produce a firewall policy with a specified number of rules.

This approach forces the tester to specify the number of rules that a certain policy must have in order to reveal system flaws; thus, preventing full automation. This limitation becomes more evident when several firewall policies with different numbers of rules are required., For our experiments we decided to apply Majumdar's suggested approach (18) which consisted of 200 policies with a policy length that ranged from 100 to 20000 rules;

however, the limitation remains a forcing us to implement an arbitrary algorithm that selected between random policy lengths.

#### *6.4.7.2 Semantic Management*

Semantic control has been a limitation for all grammar-based approaches as the employment of context-free grammars is a common practice; therefore, even though firewall rules are relatively simple, these limitations were encountered.

Semantic correctness is paramount as only semantic correct test cases can make it into the system under test as semantically incorrect ones are discarded in the initial phases of parsing preventing them from exercising any actual code paths. Context-free grammars produce syntactically correct sentences but fail to achieve semantic correctness, generating several incorrect test cases. Incorrect test cases consume time and resources, thus preventing their generation gains importance. This limitation confines the use of grammar-based approaches to systems whose input can be modeled with relatively simple grammars.

Semantic verification has been a subject of research since the formalization of grammar-based testing. Several solution attempts have been suggested (i.e. attributed grammars, second phase parsing trees, extended grammars etc.), however, regardless these and other methods have been suggested for semantic control, there is no definitive answer for which approach is better to solve this problem.

Our solution implemented an extended grammar for controlling the semantic correctness of the grammar rules; this simplified the overall analysis. This approach, while correct for our purposes, will probably have to be tailored to fit each scenario.

### 6.4.8 Summary

Two tables are presented here showing the summary of the obtained results focusing on:

- Values distribution – how empirical data is scattered through its corresponding solution space.
- Concrete constants generation probability – which experiments are more representative of the expected results.

#### *6.4.8.1 Values distribution*

Table 6.34 shows the results for the distributions for each of the four experiments and for each evaluation criterion.

The table shows that "random" and "entity" experiments presents 8 out of 8 experiments with a distribution proportional to grammar bias while "HAMPI" and "single" experiments present this behaviour only in 2 of the criterion. These two experiments show concentrations of values proportional to the expected ones but with a larger medians and percentage errors. From this evidence it can be concluded that "random" and "entity" experiments generate results following grammar rules definition.

| | **Random** | **HAMPI** | **Single** | **Entity** |
|---|---|---|---|---|
| **Rule length** | Proportional to grammar definition. | Concentration in lengths 22 and 23. | | Proportional to grammar definition. |
| **Protocol number distribution** | Proportional to grammar bias. | | | |
| **IP complexity** | Proportional to grammar bias. | Concentration in expected values but with different density. | | Proportional to grammar bias. |
| **IP structure** | Proportional to grammar bias. | Concentration in expected values but with different density. | | Proportional to grammar bias. |
| **IP values distribution** | Good ratio between produced rule and solution space covered. | The worst ratio between produced rules and solution space covered. | Poor ratio between produced rule and solution space covered. | Maximum ratio between produced rule and solution space covered. |
| **Ports usage** | Proportional to grammar bias. | | | |
| **Port operators distribution** | Proportional to grammar bias. | Concentration proportional to operators' length. | | Proportional to grammar bias. |
| **Action distribution** | Proportional to grammar bias. | Minor random bias. | Expected 50-50 distribution | |

**Table 6.34 Summary of empirical vs. expected values distribution.**

### *6.4.8.2 Concrete constants generation probability*

Each of the experiments was ranked according to their dispersion and percentage error median. The experiment with the lowest dispersion and lowest percentage error median would rank as number one.

| | Random | HAMPI | Single | Entity |
|---|---|---|---|---|
| Rule length | 1 | 4 | 3 | 2 |
| Protocol number distribution | 4 | 3 | 2 | 1 |
| IP complexity | 4 | 3 | 2 | 1 |
| IP structure | 2 | 3 | 4 | 1 |
| IP values distribution | 2 | 4 | 3 | 1 |
| Ports usage | 4 | 3 | 2 | 1 |
| Port operators distribution | 2 | 4 | 3 | 1 |
| Action distribution | 4 | 3 | 2 | 1 |

**Table 6.35 Experiment concrete constants generation ranking.**

Table 6.35 shows that for seven out of eight criteria, the "entity" experiment was superior to the any of the other approaches. It is followed by the "random" ranking first in only one of the eight categories. Greater differences are in the lower rankings, where "entity" ranks second in one criterion while "random" experiment ranks last in four criteria.

To reiterate, these selected criteria indicate that the best experiment is the "entity" experiment, which utilizes all of the enhancements implemented in our solution.

# 7 Conclusions

The obtained results assert that quasi-random sequences provide improved space coverage over the two analyzed systems that deploy pseudo-random approaches. Another benefit that our solution offers is with respect to grammar design. Pseudo-random approaches results are completely bound by the grammar rules, therefore the resulting concrete sentences are trustful to the grammar definition. The HAMPI string solver focuses on rules usage and sentence length. For achieving this goal, grammar rules no longer have the complete control over sentence production; this adds bias and uncertainty about which test cases are going to be produced given a certain grammar. Our solution takes the best of both approaches, employing sentence length and rules usage from the HAMPI string solver and adding an extra phase which solves the symbolic values grammar employing quasi-random sequences for rule selection, which generates results which are accurate representations of the grammar design.

Symbolic and concrete constants have been shown to be beneficial for test design as symbolic grammars allow complete enumeration for simple grammars and an acceptable degree of enumeration for large and complex grammars, which helps test design heuristics as it allows automated testing tools to focus on sections of the solution space maximizing results. Our solution enhances this benefit by introducing an extra phase, replacing previous pseudo-random techniques for symbolic constants instantiation with a novel approach utilizing quasi-random sequences. This new extra phase (concrete sentences generation phase) first identifies user-defined entities (groups of symbolic constants) and then associates with each entity a quasi-random sequence that will control which values from the solution space will be used for concrete constant selection.

Our solution introduces the concrete sentences generation phase, which enables the tester to define entities and solve them as a single value. This new approach raises the importance of entity design as the obtained results will benefit directly from their implementation. During the analysis of our experiments, it was asserted that entities offer an advantage over previous

approaches which can be attributed to the correct design of entities during symbolic grammar derivation. This approach is flexible enough to fit different testing needs and designs, which adds to the benefits of the proposed solution. The Concrete sentences generation phase implements a concrete sentence generation control mechanism, which evenly instantiates symbolic sentences into concrete sentences guaranteeing that all symbolic sentences are equally represented. This approach results in a better solution space coverage increasing the likeliness of detecting defects.

Despite all the benefits that our solution introduce over pseudo-random approaches and grammar string solvers, there is still more research to be done. First, quasi-random sequences can only be employed where the solution spaces are enumerable. One way of overcoming this problem for our experiments is to divide the solution space into subsets and then apply quasi-random sequences to each subset. However, even when the solution space can be enumerated, it should be used carefully as enumerating large solution spaces can result in a major impact to performance. Some other limitations as grammar restrictions and semantic management remain as a research field for future work which will help to completely automate the testing process.

Finally, these experiments give us the confidence to claim that our solution can be employed to produce vast amounts of test data with little effort and with all the advantages of grammar-based generation and quasi-random testing.

# 8 Works Cited


1. **Utting, M., Pretschner, A. and Legeard, B.** *A Taxonomy of Model-Based Testing.* New Zealand : Department of Computer Science, The University of Waikato, 2006. Tech. Rep, 4.

2. *Automatic Generation of Test Cases.* **Hanford, K.** 1970, IBM Systems Journal, p. 9(4).

3. *A sentence generator for testing parsers.* **Purdom, P.** 1972, BIT Numerical Mathematics, p. 12(3).

4. *NASA Software failure characterization experiments.* **Finelli, G. B.** 1991, Reliability Engineering & System Safety 32(1-2), pp. 155-169.

5. *Using Attributed Grammars to Test Designs and Implementations.* **Duncan, A. G. and Hutchison, J. S.** San Diego, California, United States : International Conference on Software Engineering, 1981. Proceedings of the 5th international conference on Software engineering. pp. 170 - 178. 170 - 178 .

6. *Generating test data with enhanced context-free grammars.* **Maurer, P.** s.l. : IEEE Software, 1990, IEEE Software, p. 7(4).

7. *Differential testing for software.* **McKeeman, W. M.** 1998, Digital Technical Journal, 10(1), pp. 100-107.

8. *Controllable combinatorial coverage in grammar-based testing.* **Lammel, R. and Schulte, W.** New York City, USA : Springer Verlag, 2006. The 18th IFIP International Conference on Testing Communicating Systems (TestCom 2006).

9. *A System to Generate Test Data and Symbolically Execute Programs.* **Clarke, L.** 1976, IEEE Trans Software Eng. 2, pp. 215-222.

10. **Darlington, J.** *A Semantic Approach To Automatic Program Improvement.* Edinburgh : University of Edinburgh, 1972. Ph.D. Th.

11. *Symbolic execution and program testing.* **King, J. C.** 1976, Commun. ACM, pp. 385-394.

12. *EXE: automatically generating inputs of death.* **Cadar, V., et al., et al.** Alexandria, Virginia, USA : ACM, 2006. Proceedings of the 13th ACM conference on Computer and communications security. pp. 322 - 335.

13. *DART: directed automated random testing.* **Godefroid, P., Klarlund, N. and Sen, K.** New York, NY, USA : ACM, 2005. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 213 - 223 .

14. *Cute: a concolic unit testing engine for c.* **Sen, K., Marinov, D. and Agha, G.** Lisbon, Portugal : ACM, 2005. Proceedings of the 10th European software engineering

conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 134-143.

15. **Godefroid, P., Levin, M. and Molnar, D.** *Automated whitebox fuzz testing.* Redmond, Washington, U.S. : Microsoft, 2007. Technical Report MS-TR-2007-58.

16. *Compositional Dynamic Test Generation.* **Godefroid, P.** Nice, France : Annual Symposium on Principles of Programming Languages, 2007. Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 47 - 54.

17. *Tainted-based Direct Whitebox Fuzzing.* **Ganesh, V., Leek, T. and Martin, R.** Vancouver, Canada : ICSE 2009, 2009. Proceedings of the 31st International Conference on Software Engineering. pp. 474 - 484.

18. *Directed test generation using symbolic grammars.* **Majumdar, R. and Xu, R. G.** Atlanta, Georgia, USA : Automated Software Engineering, 2007. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. pp. 134-143.

19. **Kiezun, A., Levin, M. Y. and Godefroid, P.** *Grammar-based WhiteBox Fuzzing.* Seattle, WA : Microsoft, 2007. Microsoft Research Tech Report, MSR-TR-2007-154.

20. What is Fuzzing? [book auth.] M. Sutton, A. Greene and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Upper Sadle River, NJ : Pearson Education, 2007.

21. **Prasanna, M., et al., et al.** A survey on Automatic Test Case Generation. *ACAD Journal.* 2005.

22. Introduction. [book auth.] A. Takanen, J.D. DeMott and C. Miller. *Fuzzing: for Software Security Testing and Quality Assurance.* Norwood, MA. USA : Artech House, 2008.

23. *The Automatic Generation of Test Data.* **Ince, D.C.** 1987, The Computer Journal, pp. 63-69.

24. **McKenzie, B.** *Generating strings at random from a context free grammar.* s.l. : Department Of Computer Science, University of Canterbury, 1997. Technical Report TR-COSC 10/97.

25. **Sipser, M.** Context Free Languages. *Introduction To the Theory of Computation.* Boston, Massachusetts, USA : Thomson Course Technology, 2006, pp. 99-134.

26. *A Novel Evolutionary Approach for Adaptive Random Testing.* **Tappenden, A.F., & Miller, James.** 2009, IEEE Transactions on Reliability, pp. 58 (4) 619-633.

27. *Proportional sampling strategy: A compendium and some insights.* **Chen, T.Y., Tse, T.H. and Yu, Y.T.** 2001, Journal of Systems and Software, Vol. 58, pp. 65–81.

28. *Quasi-random testing.* **Chen, T.Y. and Merkel, R.** New York, NY, USA : ACM, 2005. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. pp. 309-312.

29. *Uniformly distributed sequences with additional uniformity properties.* **Sobol, I.M.** 1967, Journal of Computational Mathematics and Mathematical Physics, p. 6.

30. *Computational investigations of quasirandom sequences in generating test cases for specification-based tests.* **Chi, H. and Jones, E.L.** Monterey, California : Winter Simulation Conference, 2006. Proceedings of the 38th conference on Winter simulation . pp. 975 - 980 .

31. *Using Genetic Algorithm for Unit Testing of Object Oriented Software.* **Gupta, N.K. and Rohil, M.K.** Nagpur, Maharashtra : IEEE International, 2008. erging Trends in Engineering and Technology, 2008. ICETET '08. First International Conference on Emerging Trends in Engineering. pp. 308 - 313 .

32. BNF Grammar for ISO/IEC 9075:1999 - Database Language SQL (SQL-99). [Online] savage.net.au, 07 26, 2004. [Cited: 06 09, 2010.] http://savage.net.au/SQL/sql-99.bnf.html.

33. Northwind and pubs Sample Databases for SQL Server 2000. *Microsoft Download Center.* [Online] Microsoft. [Cited: 06 09, 2010.] http://www.microsoft.com/downloads/details.aspx?familyid=06616212-0356-46a0-8da2-eebc53a68034&displaylang=en.

34. *Hampi: A Solver for String Constraints.* **Kiezun, A., et al., et al.** Chicago USA : ISSTA 2009, 2009. In Proceedings of the International Symposium on Testing and Analysis.

35. *Constraint-Based Automatic Test Data Generation* . **Offutt, A.J. and DeMillo, R. A.** 1991, IEEE Transactions on Software Engineering, pp. 900-910.

36. *Securing cyberspace for the 44th presidency.* Washington, D.C. : Center for Strategic and International Studies (CSIS), 2008.

37. *Modeling and Management of Firewall Policies.* **Al-Shaer, E. and Hamed, H.** 2004. IEEE Transactions on Network and Service Management. Vols. 1-1.

38. *Automated pseudo-live testing of firewall configuration enforcement.* **Al-Shaer, E., El-Atawy, A. and Samak, T.** 3, April 2009, Selected Areas in Communications, IEEE Journal on, Vol. 27, pp. 302-314.

39. *Policy segmentation for intelligent firewall testing.* **El-Atawy, A., et al., et al.** Boston : IEEE Computer Society , 2005. In 13th IEEE International Conference on Network Protocols .

40. *Model-based Firewall Conformance Testing.* **Brucker, A.D., Brügger, L. and Wolff, B.** s.l. : Testcom, 2008. In Testcom/FATES 2008. pp. 103-118.

41. *Verified Firewall Policy Transformations for Test Case Generation.* **Brucker, A.D., et al., et al.** s.l. : IEEE, 2010. 2010 Third International Conference on Software Testing, Verification and Validation.

42. *FIREMAN: a toolkit for FIREwall Modeling and ANalysis.* **Yuan, L. and Chen, H.** 2006. In Proceedings of IEEE Symposium on Security and Privacy . pp. 199--213.

43. *Dynamic rule-ordering optimization for high-speed firewall filtering.* **Hamed, H. and Al-Shaer, E.** New York, NY, USA : ACM, 2006. Proceedings of the 2006 ACM Symposium on Information, computer and communications security. pp. 332 - 342.

44. *Blowtorch: a framework for firewall test automation.* **Yoo, K. and Hoffman, D.** New York, NY, USA : ACM, 2005. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. pp. 96-103.

45. *Classbench: A packet classification benchmark.* **Taylor, D.E. and Turner, J.S.** 3, s.l. : IEE/ACM Trans., 2007, IEE/ACM Trans. Networking, Vol. 15, pp. 499-511.

46. *Two case studies in grammar-based test generation.* **Hoffman , D., et al., et al.** 0164-1212, s.l. : Journal of Systems and Software, Available online 11 August 2010, Vol. In Press.

47. *Systematic Structural Testing of Firewall Policies.* **JeeHyun , H., et al., et al.** s.l. : IEEE, 2008. IEEE Symposium on. pp. 105-114.

48. *Fault Localization for Firewall Policies.* **Jee Hyun, H., et al., et al.** s.l. : IEEE. IEEE International Symposium on Reliable Distributed Systems. pp. 100-106.

49. *Formal Verification of Firewall Policies.* **Liu, A.X.** s.l. : IEEE, 2008. IEEE International Conference on Communications, . pp. 1494-1498.

50. *Integrated Framework for Automated Firewall Testing and Validation.* **En-Nouaary, A. and Akiki, M.** [ed.] Information Technology: New Generations (ITNG). s.l. : 2010 Seventh International Conference on, April 2010. pp. 768-773.

51. *Verified Firewall Policy Transformations for Test Case Generation.* **Brucker, A.D., et al., et al.** [ed.] Verification and Validation (ICST) Software Testing. s.l. : Third International Conference on, 2010.

52. **Vacca , J. R. and Ellis, S. R.** Appendix B - Worldwide Survey of Firewall Products. *Firewalls: Jumpstart for Network and Systems Administrators.* s.l. : Digital Press, 2005.

53. Chapter 2 - Protocols Guide . [book auth.] Javvin, and Inc. Technologies. *Network Protocols Handbook.* s.l. : Javvin Technologies, 2007.

54. **Project, The Computer Technology Documentation.** Commonly Used Network Ports. *The Computer Technology Documentation Project.* [Online] CTDP, February 3, 2001. [Cited: September 26, 2010.] http://www.comptechdoc.org/independent/networking/guide/netports.html.

55. **Meyer, Michael J.** The Project Martingale. [Online] 09 24, 2003. [Cited: 02 01, 2010.] http://martingale.berlios.de/Martingale.html.

56. *Generation of Pairwise Test Sets Using a Genetic Algorithm.* **McCaffrey, J.D.** Seattle, WA, U.S.A. : IEEE International, 2009. Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International. Vol. 1, pp. 626-631.

57. Software Vulnerability Analysis. [book auth.] A. Takanen, J.D. DeMott and C. Miller. *Fuzzing: for Software Security Testing and Quality Assurance.* Norwood, MA. USA : Artech House, 2008.

58. **Whittaker, J. A.** *How to Break Software:A Practical Guide to Testing.* s.l. : Addison-Wesley, 2002.