SUITABILITY OF FINITE STATE AUTOMATA TO MODEL STRING CONSTRAINTS IN PROBABILISTIC SYMBOLIC EXECUTION

by

Andrew Harris

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2019

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Andrew Harris

Thesis Title: Suitability of Finite State Automata to Model String Constraints in Probabilistic Symbolic Execution

Date of Final Oral Examination: 5th May 2017

The following individuals read and discussed the thesis submitted by student Andrew Harris, and they evaluated their presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Elena Sherman, Ph.D.	Chair, Supervisory Committee
Jim Buffenbarger, Ph.D.	Member, Supervisory Committee
Bogdan Dit, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Elena Sherman, Ph.D., Chair, Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGMENTS

The author wishes to express gratitude to the National Science Foundation who's grant supported a portion of this work.

ABSTRACT

Probabilistic Symbolic Execution (PSE) extends Symbolic Execution (SE), a path-sensitive static program analysis technique, by calculating the probabilities with which program paths are executed. PSE relies on the ability of the underlying symbolic models to accurately represent the execution paths of the program as the collection of input values following these paths. While researchers established PSE for numerical data types, PSE for complex data types such as strings is a novel area of research.

For string data types SE tools commonly utilize finite state automata to represent a symbolic string model. Thus, PSE inherits from SE automata-based symbolic string models to calculate the probabilities of string-based constraints describing program paths. However, to our knowledge, there is lack of research on suitability of automata-based symbolic string models in the context of PSE.

This thesis proposes four automata-based symbolic string models for PSE and analyzes their suitability using two criteria: accuracy and performance. We compare the probability computed by the model to the actual probability and the amount of time took to compute it. Our results show that each model varies in their accuracy, however none is able to consistently compute actual value. In addition, our evaluation did reveal that this amount of inaccuracy depends upon the characteristics of a software program. From these findings we suggest guidance when selecting an automaton model for PSE based on the performance and accuracy requirements and the characteristics of the program under analysis. Additionally, we suggest future areas of research to the accuracy and performance deficiencies observed in our evaluation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DFA – Deterministic Finite State Automaton

FSA – Finite State Automaton

MC – Model Count

NFA – Nondeterministic Finite State Automaton

PC – Path Condition

PSE – Probabilistic Symbolic Execution

SAT – Satisfiable

SCG – String Constraint Graph

SCSF – String Constraint Solver Framework

SE – Symbolic Execution

UNSAT – Unsatisfiable

WFSA – Weighted-Transition Finite State Automaton

LIST OF SYMBOLS

$\mathcal{A}$	Finite State Automata
$\mathcal{L}(\mathcal{A})$	Language of Finite State Automaton
$\mathcal{M}$	Automata-based Symbolic String Model
$\mathcal{S}$	Solution Set
$\mathcal{W}$	Weighted-transition Finite State Automaton

CHAPTER 1

INTRODUCTION

1.1 Quantitative String Analysis

Quantitative string analysis is a type of software engineering technique which quantifies the behaviour of a software program using the number of possible values a string variable can have along different execution paths in the program. Quantitative string analyses have many different applications in the software engineering and development processes including security vulnerability detection [17, 37–41], bug and error detection [7], and test case generation [17]. The ability to analyze the behaviour of programs which quantitative string analysis allows is important to common string heavy programs such as web applications. Quantitative string analysis research has followed a few primary directions in recent years including quantitative information flow analysis [8, 30, 35] and probabilistic symbolic execution [1, 27]. Despite these different approaches to quantitative string analysis, the different analysis techniques have a shared fundamental requirement: a constraint solver capable of quantifying the number of execution paths followed for a set of string values. Such a string constraint solver must rely on an underlying symbolic string model to count the number of string values which can occur along execution paths.

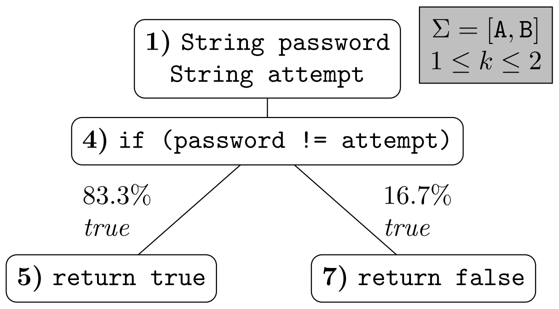
Quantitative string analysis has emerged as a research area in recent years due to the the ability to provide more information than the more common reaching analysis.

A reaching analysis for string variables is used to determine if different sections of code in a program can execute for a given set of string values. Quantitative string analyses are used to count the number of executions which reach different portions of a program rather than the binary *true* or *false* result of the reaching analysis. The more robust quantitative data allows additional software engineering applications such as detection of program errors which can not be detected with a reaching analysis.

1.2 Example

```
1 public boolean checkPassword (
2         String password ,
3         String attempt) {
4     if (password != attempt) {
5         return true;
6     }
7     return false;
8 }
```

(a) Password checking function



(b) `checkPassword` execution tree

Figure 1.1: Password Matching Example

Figure 1.1 demonstrates the usefulness of a quantitative string analysis when compared to a reaching analysis. The Java function in Figure 1.1a is used to check a password from user input (`attempt`) against an already known password (`password`). The function contains an error in line 4 where `password` and `attempt` are compared for inequality instead of equality as indicated from the subsequent returns of `true` and `false` on lines 5 and 7 respectively. Figure 1.1b is the execution tree for this function which illustrates the code paths that can occur during execution of this function. For this example, the string alphabet (Σ) has been restricted to only the letters `A` and `B` and the initial string length (k) is either 1 or 2. As a result of these restrictions, the possible initial string values for both the `password` and `attempt` variables are the following 6 values: `A`, `B`, `AA`, `AB`, `BA`, and `BB`. The differences between the quantitative and reaching analyses is shown when evaluating line 4 of the program and processing the predicate: `password != attempt`. A reaching analysis shows that at least one of the six values for `password` is not equal to one of the six values for `attempt` as shown by the *true* on the branch from node 4 to node 5 in the execution tree. The reaching analysis also shows the same for the branch from node 4 to node 7 with another *true* label. Since both branches reach lines 5 and 7 of the program, no error is detected since that is the expected behavior. The additional information provided in a quantitative analysis is able to expose the error in the function. When the set of six string values which represent `attempt` are tested for equality with the set representing `password`, a total of 36 possible combinations of string values are evaluated for inequality. This results in 83.3% of these combinations evaluating as not equal and there for following the branch to line 5 of the function. The other 16.7% of combinations evaluate as equal and follow the branch to line 7. The behavior of the function as shown by accepting 83.3% of password attempts for the restricted string values is not the expected behavior of the function. Through this example, it is clear that the additional information provided by a quantitative analysis allows for additional useful applications when compared to reaching analyses.

1.3 Symbolic String Models

While the example in Figure 1.1 clearly demonstrates the usefulness of a quantitative string analysis, the reliability of the analysis depends upon the accuracy of the number of execution paths computed from the set of possible string values.

Because of an inability to store the set of all possible values for non-trivial string alphabets and string lengths, quantitative string analyses must make use of some form of symbolic string model to represent this set of values a string variable can have along different code paths in a program. Because string values in a program are often altered by string operations, the quantitative string analysis must also emulate these operations for the symbolic string model. For example, the CONCATENATE operation joins a second string to a first string so that the characters from the first string precede the characters from the second in the new string resulting from the operation, e.g. $\textsc{Concatenate}(\texttt{A}, \texttt{B}) = \texttt{AB}$. Similar to simulating string operations, string predicates must also be emulated for the symbolic string model in an analysis. For example, the EQUALS predicate checks if a first string value is equal to a second string value and returns *true* or *false* depending on if the strings are the save value, e.g. $\textsc{Equals}(\texttt{AB}, \texttt{BA}) = \textit{false}$. Due to these requirements for a symbolic string model, only a few symbolic string models have proven to be robust enough for use in string analysis research. One proven symbolic string model is the Finite State Automaton (FSA) which was chosen to perform this research analysis on string constraints.

1.4 Thesis Statement

This thesis analyzes four string constraint solvers which utilize finite state automata to model symbolic string values in the context of probabilistic symbolic execution. In particular this thesis addresses the following research statements:

- *Accuracy*:
  - How often do automata-based solvers produce invalid analyses?

  - How does an automata-based string constraint solver solver compare in its calculation of branch probability to the actual branch probability? How do automata-based solvers compare in this calculation to one another?
  - What characteristics of a string constraint effect accuracy of an automata-based solver? How large is this effect?

- *Performance*:
  - What is the relative performance of one automata-based solver compared to others?
  - How does the performance of different automata-based solvers compare for constraint solving specifically?
  - How does the performance of different automata-based solvers compare for model counting specifically?
  - How do the characteristics of a string constraint effect the performance of an automata-based solver?

- *Accuracy vs. Performance trade-offs*:
  - What is the relationship between accuracy and performance for an automata-based solver?
  - What automata-based solver produces the best combination of accuracy and performance?
  - How is the relationship between accuracy and performance affected by specific characteristics of a string constraint?

1.5 Contributions

The main contributions of this thesis are as follows:

- Identification of Collapse Problem (Sections 2.2.3, 2.3.2, 3.1.3, 3.2.3, 3.3.4, and 3.4.6)
- Four Automata-based Symbolic String Models (Sections 3.1, 3.2, 3.3, and 3.4)
- String Constraint Solver Framework (Chapter 4)
- String Constraint Model Counting Oracle (Sections 4.4.3 and 5.3.1)
- Evaluation of Suitability of Automata-based Symbolic String Models (Chapters 5 and 6)

CHAPTER 2

BACKGROUND

This chapter provides the background information relevant to this thesis. Section 2.1 explains Symbolic Execution (SE). Section 2.2 describes an enhancement of SE, Probabilistic Symbolic Execution (PSE). Section 2.3 explores the modeling of string constraints using symbolic string models. Finally, Section 2.4 details the modeling of string constraints using Finite State Automata (FSA) specifically.

2.1 Symbolic Execution

Symbolic Execution (SE) [24] is a path-sensitive static program analysis technique that determines what program inputs result in execution of different code paths. SE is an important software analysis technique with many useful applications such as detecting security vulnerabilities [26], detecting program errors [9], and aiding in generating test cases [22]. While SE was initially used with simple integer variable types, more recent research extends SE to analyze complex types such as floating point numbers [5], sets [10, 25], and strings [34].

2.1.1 Description

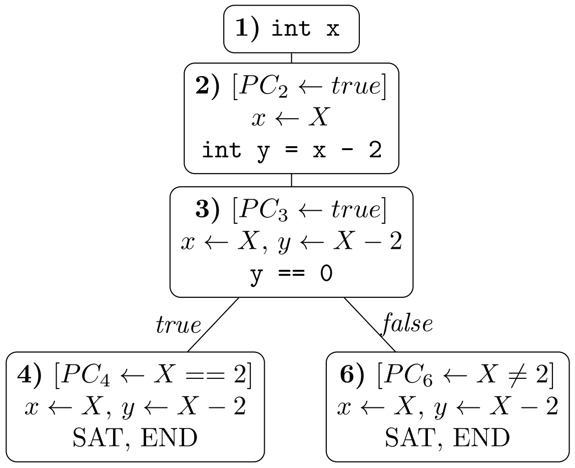
When SE interprets a program, instead of executing the program on a set of concrete inputs values, SE uses a symbolic value each time the program requires a new input value. This means that SE maps each input variable x in the program domain to a corresponding symbolic value X in the symbolic codomain maintained by SE. Initially symbolic values represent all concrete values in the domain of that data type. For example an integer input variable x is mapped to the symbolic value X and has a corresponding solution set S_X containing all possible integers, i.e. $m(x) = X$ and $S_X = \mathbb{Z}$. SE also maintains a symbolic program state, which is a mapping of a program variable to a corresponding symbolic expression. A symbolic expression is a finite combination of symbolic values and domain operations specific to the domain of the symbolic values. In SE, a symbolic expression is produced from a domain operation performed on one or more previous symbolic expressions and is usually assigned to a new program variable. For example, when processing the statement $y = x - 2$ where x is an input variable, SE adds the variable y to the set of known variables with the symbolic expression $X - 2$ where X is a previous symbolic value where $m(x) = X$.

When SE encounters a conditional statement or branch condition, SE must follow either the *true* or *false* branch to continue the analysis. In order to explore each branch, SE generates a *branch constraint*, a symbolic predicate that restricts the set of concrete program inputs that can follow the branch path. The *true* branch constraint is generated by substituting the program variables in the branch condition for their corresponding symbolic expressions from the symbolic program state. The *false* branch constraint is generated from negating that branch condition, i.e., negating the *true* branch constraint.

Once a branch constraint is generated, it is conjoined with constraints of all previously taken branches to yield a single combined constraint called the *path condition* (PC). If no previous branches exist, the PC is *true.* A PC is a conjunction of

```java
public void testInt(int x) {
    int y = x - 2;
    if (y == 0) {
        println("true");
    } else {
        println("false");
    }
}
```

(a) `testInt` method


1) `int x`
2) $[PC_2 \leftarrow true]$ $x \leftarrow X$ `int y = x - 2`
3) $[PC_3 \leftarrow true]$ $x \leftarrow X,\ y \leftarrow X - 2$ `y == 0`
true
false
4) $[PC_4 \leftarrow X == 2]$ $x \leftarrow X,\ y \leftarrow X - 2$ SAT, END
6) $[PC_6 \leftarrow X \neq 2]$ $x \leftarrow X,\ y \leftarrow X - 2$ SAT, END

(b) `testInt` SE execution tree

Figure 2.1: The `testInt` Java method demonstrating SE.

predicates which restrict S_X by a function PC_n such that $PC_n(X) \rightarrow S_{nX}$, where S_{nX} is the set of all concrete values for input variable x which reach program point n. For example, if the PC at a program point n is $X == 2$, the function $PC_n(X)$ would produce the set $\{2\}$ since only the integer 2 satisfies the predicate $X == 2$. From this definition it is clear that a PC exists for all statements in a program analyzed using SE starting with the initial PC with a simple *true* predicate.

A typical SE analysis passes the PC computed for a branch to a constraint solver to determine if the PC of the branch is satisfiable (SAT) or unsatisfiable (UNSAT). The solver produces a symbolic model from the constraint and uses this model to determine satisfiability. If the solver determines a branch is SAT, SE continues interpreting statements along that code path. If the solver determines a branch is UNSAT, it is reported but not followed for any further analysis. In this way SE systematically analyzes all feasible paths of execution in a program.

2.1.2 Example

A simple example of SE is shown in Figure 2.1 for the Java method `testInt` in Figure 2.1a. The SE tree in Figure 2.1b contains four nodes labeled 2, 3, 4, and 6 which correspond to the those line numbers in `testInt`. The PC and variables for each node in the tree correspond to values before that line of code and after the previous line. The first step of SE is initializing symbolic variables for each input variable. The only input variable of the method is the integer parameter x which is initialized to the unrestricted symbolic variable X. The PC is also initialized at this time as $PC_1 \leftarrow true$. Both the initial PC and the initial symbolic variable x are displayed in node 2 as PC_2 and x_2 respectively. Next, the assignment statement on line 2 is interpreted and the new variable y is initialized to the symbolic value $X - 2$ from $x - 2$ where x is replaced by its symbolic variable X. This addition of the y variable is reflected in node 3 of the SE tree representing the state before interpreting line 3.

Line 3 of `testInt` contains the first conditional statement of the method and requires the generation of both *true* and *false* branch constraints. To generate these constraints, the condition must be transformed into its symbolic equivalent by substituting for the variables in the conditional statement: $y == 0 \rightarrow X - 2 == 0 \rightarrow X == 2$. The *true* branch becomes $X == 2$. Next, SE follows the *true* branch and must generate the new PC for this code path by conjoining the *true* branch constraint, $X == 2$, with PC_3 which yields $PC_4 \leftarrow true \wedge X == 2$ which can be simplified to $PC_4 \leftarrow X == 2$. SE then determines that PC_4 is satisfiable since $2 \in S_X$, thus satisfying the $PC \leftarrow X == 2$. This satisfied branch is reported by SE as indicated in the SE tree for node 4 by the "SAT" token. Since the true branch leads to line 4, SE next interprets this line which consists of a side-effect free statement, i.e., it does not alter the symbolic program state. After line 4, there are no more statements for SE to interpret along this code path within the `testInt` method. When SE encounters the end of a code path, it reports the end, as is shown in node 4 of the tree by the "END" token, and backtracks to explore other paths.

The *false* branch constraint for the condition on line 3 is generated by negating the true branch constraint: $\neg(X == 2) \rightarrow X \neq 2$. SE must then generate the PC for this *false* branch path by conjoining the branch constraint, $X \neq 2$, and PC_3 which produces $PC_6 \leftarrow true \wedge X \neq 2$ or $PC_6 \leftarrow X \neq 2$. SE then determines that PC_6 is satisfiable since all but one integer in S_X satisfies the condition (2 being the single exception). SE reports that this branch constraint is satisfied as shown in node 6 of the tree by the "SAT" token. Following the false branch SE advances to line 6 along the current code path. Since the statement on line 6 is side-effect free, SE attempts to advance to the next statement. Because there are no more statements along this path, SE produces the "END" token. At this point SE has exhaustively explored all possible code paths within the *testInt* method and has found that both branches of the condition in line 3 are SAT. This information reported by SE can be used for many applications such as in a reachability analysis which can conclude that lines 4 and 6 are reachable due to the *true* and *false* branches being SAT.

2.1.3 Limitations

The largest drawback of SE is poor scalability due to the path sensitive nature of the technique requiring the traversal of too many paths. The number of paths grows exponentially as the number of conditional statements in the program increases. Additionally, the presence of loops in the program increases the performance cost since new branch constraints are generated each time the loop condition is checked. To mitigate this problem, SE exposes only a finite number of loop iterations which limits the performance cost at the expense of analysis completeness. Trade-offs of this type between performance and accuracy are commonly used to adjust the performance of SE based analyses, allowing otherwise impractical analyses to be completed.

SE is also limited by the quality of information it reports, where satisfiability of a code path provides only a coarse 1 or 0 approximation of the probability that the path will be executed. This lack of quantitative information limits the types of analyses which can be conducted using the SE technique. This limitation can be seen in the `testInt` example in Figure 2.1 where the condition on line 3 produced a *true* branch with one satisfying input, 2, while the *false* branch is satisfied by all the remaining inputs. Clearly the *false* branch is overwhelmingly more probable to execute than the *true* branch, but SE simply reports both as SAT.

Both the performance and the quantitative information limitation are addressed by the probabilistic symbolic execution technique.

2.2 Probabilistic Symbolic Execution

Probabilistic Symbolic Execution (PSE) [16] is an enhancement of SE which provides a quantitative analysis of execution for a software program in the form of conditional branch probabilities rather than branch satisfiability (SAT or UNSAT) reported by SE. This enhancement allows PSE to prioritize code paths more likely to be executed while differing less probable paths to analyze when time permits. Though PSE is an emerging static analysis technique, it is already being used to aid program understanding and error detection [16], to compute software reliability [4, 13], and to provide quantitative information flow analysis for software security [29]. While most of this early research has been restricted to analyses of integer variables, there is increasing interest into using PSE to analyze string variables in programs [1].

2.2.1 Description

PSE updates the symbolic program state in exactly the same way as traditional SE, but handles reporting conditional branches differently. In SE the constraint solver reports a branch constraint as either SAT or UNSAT, which is in fact a coarse approximation of the probability of executing a branch where UNSAT and SAT correspond to probabilities 0 and 1 respectively. PSE aims to provide a more detailed understanding of program execution through the use of program path probabilities. These path probabilities allow for quantitative analyses of a program rather than satisfiability analyses.

When PSE processes a conditional branch, it determines which of two branch paths to follow by selecting the branch with the larger execution probability. Each execution probability for a branch is the ratio of the number of concrete execution paths which satisfy the branch condition to the number of concrete execution paths which reach the branch condition. This probability calculation is only possible due to the bijective relationship between the set of all execution paths and the set of input value combinations which is the k-fold Cartesian product produced from the sets of concrete values for k input variables. Utilizing this relationship, the solver can calculate the number of execution paths reaching a program point as the *model count* of the *solution set* for the program point's PC.

Definition A *solution set*, denoted as $\mathcal{S}$, is the k-tuple set of solution values for a symbolic path constraint or PC. Alternatively, a *solution set* $\mathcal{S}_n$ is the k-fold Cartesian product of the sets of values for k input variables which can reach at a program point n, i.e. $\mathcal{S}_n = S_{nX1} \times \cdots \times S_{nXk}$ where $S_{nXi} \forall i : 1 \leq i \leq k$.

Definition *Model count*, denoted as MC, is the total number of solutions for a symbolic constraint or PC. Alternatively, *model count* is the cardinality of a solution set, i.e. $MC(PC_n) = |\mathcal{S}_n|$ at a program point n.

The branch probability calculation P_b is shown in Formula 2.1 as the ratio of the MC of the PC after the branch $MC(PC_b)$ to the MC of the PC before the conditional statement is interpreted $MC(PC_c)$.

$$P_b(PC_b, PC_c) = \frac{MC(PC_b)}{MC(PC_c)} \tag{2.1}$$

In addition to branch probability, the overall probability that a program point is reached can also be used by PSE. This overall probability, called global execution probability and denoted as P_g, can be used to prioritize all the remaining unexplored paths for analysis. It is the ratio of the number of concrete execution paths reaching a program point to the total number of concrete execution paths in the program. Formula 2.2 shows this probability calculation as the ratio of the MC of the PC at program point n ($MC(PC_n)$) to the MC of the initial program inputs ($MC(PC_{init})$).

$$P_g(PC_n, PC_{init}) = \frac{MC(PC_n)}{MC(PC_{init})} \tag{2.2}$$

2.2.2 Example

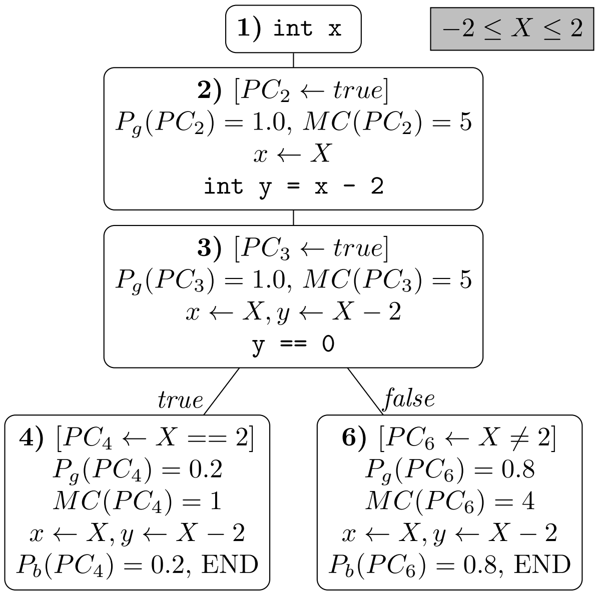

1) int x
$-2 \leq X \leq 2$
2) $[PC_2 \leftarrow true]$
$P_g(PC_2) = 1.0$, $MC(PC_2) = 5$
$x \leftarrow X$
int y = x - 2
3) $[PC_3 \leftarrow true]$
$P_g(PC_3) = 1.0$, $MC(PC_3) = 5$
$x \leftarrow X, y \leftarrow X - 2$
y == 0
true
false
4) $[PC_4 \leftarrow X == 2]$
$P_g(PC_4) = 0.2$
$MC(PC_4) = 1$
$x \leftarrow X, y \leftarrow X - 2$
$P_b(PC_4) = 0.2$, END
6) $[PC_6 \leftarrow X \neq 2]$
$P_g(PC_6) = 0.8$
$MC(PC_6) = 4$
$x \leftarrow X, y \leftarrow X - 2$
$P_b(PC_6) = 0.8$, END

Figure 2.2: `testInt` PSE execution tree

To demonstrate PSE, consider the `testInt` Java method from Figure 2.1a which produces the PSE tree in Figure 2.2. For this PSE example, the input variable domain is restricted to $\{i \in \mathbb{Z} | -2 \leq i \leq 2\}$ in order to better demonstrate PSE. This is shown as the grey area left of the PSE tree. This restriction results in $S_X = \{-2, -1, 0, 1, 2\}$ for symbolic variable X initialized from input variable x. PSE then interprets the `testInt` method identical to SE until line 3 where the first conditional statement is encountered. The program variables before the execution of line 3 have symbolic values $x = X$ and $y = X - 2$ just as in SE. PSE interprets the condition statements exactly as SE by substituting variables for the corresponding symbolic values. This produces $PC_4 \leftarrow X == 2$ for the *true* branch and $PC_6 \leftarrow X \neq 2$ for the *false* branch.

Instead of determining satisfiability for each branch, PSE calculates the execution probability of each branch using Formula 2.1. The *true* branch probability is calculated using Formula 2.1 as $P_b(PC_4, PC_3)$ where the solution sets for PC_4 and PC_3 are $\mathcal{S}_4 = \{2\}$ and $\mathcal{S}_3 = \{-2, -1, 0, 1, 2\}$ respectively. The resulting *true* branch probability is 0.2. The *false* branch probability is calculated in the same way again using Formula 2.1 as $P_b(PC_6, PC_3)$ where the solution sets for PC_6 and PC_3, which are $\mathcal{S}_6 = \{-2, -1, 0, 1\}$ and $\mathcal{S}_3 = \{-2, -1, 0, 1, 2\}$ respectively. The resulting *false* branch probability is 0.8.

With both branch probabilities calculated, PSE chooses to explore the *false* branch since it is more likely to be executed than the *true* branch. Upon reaching the end of the path at line 6, PSE reports an "END" token. PSE next backtracks to the single remaining path to explore, following the *true* branch from the branch condition on line 3 to line 4. PSE reports with another "END" token after line 4 and completes the analysis since no unexplored paths remain. This analysis determines that line 6 is four times more likely to be executed than line 4 under the restricted input domain.

2.2.3 Limitations

While PSE uses probabilities for different code paths to address the performance cost of SE, it does incur a new performance cost. These additional computations are required for the MC used to compute branch probabilities. The performance cost of the MC calculation depends on the algorithm used to calculate the MC for each branch. The MC calculation algorithm performance cost is specific to each underlying model used to represent the symbolic values. In this way, the choice of symbolic model affects the performance of a PSE analysis. Just as in SE, PSE can sacrifice accuracy for better performance by over-approximating solution sets. However, unlike regular SE which identifies branches as SAT or UNSAT, PSE determines the likelihood of taking that branch. Such calculations are more sensitive to a loss in accuracy.

In addition to performance issues, PSE also suffers from accuracy problems which can affect the validity of the analysis. A PSE analysis is invalid if it calculates that a path PC_x has a higher probability than another path PC_y, i.e., $P_{PSE}(PC_x) > P_{PSE}(PC_y)$, while the opposite is actually true, i.e. $P_{act}(PC_x) < P_{act}(PC_y)$. We have used the term *collapse* to describe the problem which results in this type of invalid analysis. A *collapse* occurs when two or more distinct possible values in the solution set for a variable constraint before an operation or predicate are "collapsed" into a single possible value in the solution set of the variable constraint as a result of the operation or predicate. This collapse results from the combination of two factors: non-injective variable operations and the assignment of operation results to new symbolic values.

An injective function is a function which never maps distinct elements from its domain to the same element of its codomain. In the context of a program, a non-injective function maps two or more combinations of parameters to the same concrete value from the operation. However, a non-injective operation alone is not sufficient to produce an invalid analysis. For example, integer variables are able to use non-injective operations such as division but still do not suffer from the collapse problem because integer variables can be expressed as functions of symbolic variables. These functional relationships between integer variables define n-dimensional convex polytopes which contain the solution set integers within.

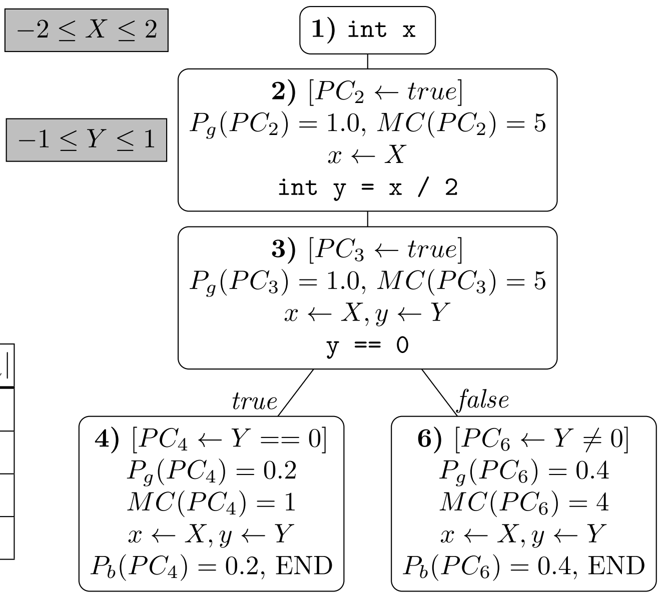
The second factor causing collapse problems is the assignment of operation results to new symbolic variables. That is the creation of a new symbolic variable as the result of an operation with no explicit relation to the symbolic value arguments of the operation. For example, the symbolic value Y is assigned to the integer variable y from the integer division operation where the symbolic value X restricted by the

```
1 public void testDiv(int x) {
2     int y = x \ 2;
3     if (y == 0) {
4         println("true");
5     } else {
6         println("false");
7     }
8 }
```

(a) `testDiv` method

PC_n	$\mathcal{S}_n$	$\|\mathcal{S}_n\|$	V_n	$\|V_n\|$
PC_2	$\{-2,-1,0,1,2\}$	5	$\{-2,-1,0,1,2\}$	5
PC_3	$\{-2,-1,0,1,2\}$	5	$\{-2,-1,0,1,2\}$	5
PC_4	$\{0\}$	1	$\{0,0,0\}$	3
PC_6	$\{-1,1\}$	2	$\{-1,1\}$	2

(b) Actual Solution Set vs. Approximated Solution Set


$-2 \le X \le 2$
$-1 \le Y \le 1$
1) int x
2) $[PC_2 \leftarrow true]$
$P_g(PC_2) = 1.0$, $MC(PC_2) = 5$
$x \leftarrow X$
int y = x / 2
3) $[PC_3 \leftarrow true]$
$P_g(PC_3) = 1.0$, $MC(PC_3) = 5$
$x \leftarrow X, y \leftarrow Y$
y == 0
true
false
4) $[PC_4 \leftarrow Y == 0]$
$P_g(PC_4) = 0.2$
$MC(PC_4) = 1$
$x \leftarrow X, y \leftarrow Y$
$P_b(PC_4) = 0.2$, END
6) $[PC_6 \leftarrow Y \neq 0]$
$P_g(PC_6) = 0.4$
$MC(PC_6) = 4$
$x \leftarrow X, y \leftarrow Y$
$P_b(PC_6) = 0.4$, END

(c) `testDiv` PSE execution tree

Figure 2.3: The `testDiv` Java method demonstrating the division problem in PSE

inequality $-2 \le X \le 2$ is divided by the integer 2, i.e., $y \leftarrow Y : y = \frac{X}{2}$. This assignment requires the set of values represented by Y to contain each of the values represented by X divided by 2, resulting in a restriction of Y by the inequality $-1 \le Y \le 1$. Assignments in this manner usually occurs due to a complex variable domain and requires advanced models to accurately model the symbolic values for the domain. This results in a need to produce a new symbolic value from variable operations. The combination of non-injective variable operations and producing new symbolic values results in a symbolic value which cannot have a one-to-one relation to the operation inputs and similarly cannot have a one-to-one relation to the execution paths of the program.

The example in Figure 2.3 demonstrates a collapse problem for integer variables. Since integer variables do not usually produce collapse problems, we assign the result of the division operation on line 2 to a new symbolic value, fulfilling the previously discussed requirements for the collapse problem to occur. The Java method `testDiv` in Figure 2.3a is a slight modification of the `testInt` method from Figure 2.1a where a division operation is used on line 2 instead of a subtraction operation. A table is presented in Figure 2.3b displaying for each line n in the program: the solution set denoted $\mathcal{S}_n$, the size of the solution set denoted $|\mathcal{S}_n|$, the bag (multiset) of actual values which would appear in the program denoted V_n, and the number of values in the bag denoted $|V_n|$. The PSE execution tree is presented in Figure 2.3c. As in the previous example, the integer variable domain for the input values is restricted to the set of integers values $\{-2, -1, 0, 1, 2\}$. The results of the assignment of y on line 2 are shown in the tree where $y \leftarrow Y$ instead of assigning y in relation to X as was done in `testInt` in Figure 2.2. The bag of actual values and the solution sets are in agreement for both PC_2 and PC_3 due to both being *true*, this also results in a MC of 5 representing all the execution paths in the program.

The collapsed value problem emerges when the *true* and *false* branch probabilities are calculated from the branch constraints. The *true* branch is followed when the input (y) value is 0 which occurs for three different x input values $\{-1, 0, 1\}$ and therefore three execution paths. However, PSE calculates the $MC(PC_4)$ from $\mathcal{S}_4$ which is $\{0\}$ since 0 is the only value which satisfies the PC_4. This is a clear example of the collapse of multiple execution paths represented by the x inputs -1, 0, and 1 into the single value 0 in the solution set for PC_4. The subsequent probability calculations are affected by this difference in MC where the branch probability of 0.2 is obtained using Formula 2.1. This is in contrast to the bag of actual values in Table 2.3b which shows 3 out of 5 execution paths execute line 4 for an actual probability of 0.6. The *false* branch is modeled correctly for PSE where $P_b(PC_6/PC_3) = 0.4$ corresponds to the 2 of 5 execution paths execute line 6. This difference in the probability of the *true* branch without a difference in the probability of the *false* branch causes PSE to determine that the *false* branch is more likely to be executed since $P_b(PC_6/PC_3) > P_b(PC_4/PC_3)$. The value table clearly shows that with 3 of 5 execution paths, the *true* branch is actually more likely than the *false* resulting in an invalid analysis reported by PSE.

This collapse problem is an example of a limitation of PSE due to the dependence on the MC obtained for PCs. Since the MC is calculated from the solution set of a constraint, the accuracy of PSE depends upon the accuracy of the symbolically modeled constraint. PSE analyses of integer variables avoids the collapse of paths by calculating a MC from the volume of the convex polytope which is defined by the constraint inequalities in a k-dimensional vector space [16] where k is the number of input variables to the program. Other complex data types are not able to easily utilize a vector space to model constraints and must calculate MCs from the solution set of the a constraint, which results in the collapse of paths. This becomes a common problem when analyzing other variable data types such as strings where a nearly all operations are not injective. It is clear that the choice of underlying symbolic model is key to minimizing or eliminating the impact of this collapsing paths problem.

2.3 Modeling String Constraints

While many methods exist for modeling symbolic string variables, only a few of these types of representation have been found appropriate for string analysis research. These representations include most significantly bit-vectors [3, 23], axiom based models [42], and finite state automata (FSA) [1, 7, 18, 19, 34, 36–41]. These different string models have been used to detect SQL injection vulnerabilities [17, 37–41], to detect string variable errors in programs [7], and to generate test cases for string variable inputs [17]. This thesis focuses on the use FSA to model symbolic strings.

2.3.1 Symbolic String Models

Strings form a fundamental datatype in general purpose programming languages. A string is a finite sequence of symbols or characters that are chosen from a non-empty set of symbols called an *alphabet*, denoted Σ. The length of a string, k, can be any non-negative integer with the special case of the 0 length string called the *empty string* ε. Strings are represented in many different ways depending on the programming language such as a null-terminated array in C, an explicit length array in Java, or a singly linked list in Haskell. Additionally, often different string-like data types can exist in the same programming language. For example, the `String` and `StringBuilder` classes in Java are different data types but both represent the same abstract string. This is similar to abstract integers implemented as both 32 and 64-bit concrete integer data types in a programming language. Because of the common abstraction for string-like data types, each can be modeled by the same symbolic string model just as different n-bit integers are modeled by the same integer inequalities. In addition to modeling each string-like data type through a single symbolic string model, the operations and predicates of the the string-like data type must be modeled as well.

Formal language theory defines a set of simple operations for strings under which the language is closed, e.g. CONCATENATE. Unlike integer arithmetic which is often included in the programming language itself, string functions and operations are usually implemented as library functions. For example, the Java `StringBuilder` class includes the `append` method which is an implementation of the abstract string operation CONCATENATE. While it is possible to analyze these functions as part of the detailed analysis, such an approach would be inefficient. Instead, string functions are interpreted as single operations for the abstract string data type allowing algorithms to model distinct operations for the chosen symbolic string model. While this approach is sufficient for most common string operations, the complexity of others, e.g. FORMAT has rendered this approach impractical for those string functions. Additionally, string operations which return data types other than strings are difficult to analyze. These operations are mixed constraint type operations where the analysis must utilize symbolic values for the other data types being returned from the operation. For example, the Java `String` class contains the method `length` which returns an integer. In PSE, a symbolic integer model would be needed to model the behavior of the result of the `length` operation and an algorithm would be needed to simulate the correct possible length values of the symbolic string model.

String predicates are handled similarly to mixed constraint type string operations since a function is applied to a string data type and returns a boolean data type. Unlike string operations which occur in a particular branch of execution in a program, most predicates occur as branching conditions. For example the Java `String` class contains the `equals` method which is an implementation of the abstract string predicate EQUALS and is usually used in `if` statements to determine if the code in the statement is executed. In PSE, this is represented in both branches of the predicate by transforming using appropriate algorithms such that the symbolic string model following the branch condition models the string values satisfying the branch condition. For example, the symbolic string variable y represents the string variable `y` in the Java `if` condition `y.contains("B")` where $\Sigma = \{\mathtt{A}, \mathtt{B}\}$, $k = 1$, and $y = \{\varepsilon, \mathtt{A}, \mathtt{B}\}$ before the branch condition. After this condition, $y_t = \{B\}$ follows the *true* branch and represents the string values where $\textsc{Contains}(y, \texttt{B}) = true$. $y_f = \{\varepsilon, A\}$ follows the *false* branch and represents the string values where $\textsc{Contains}(y, \texttt{B}) = false$. Just as some string operations impractical to emulate for symbolic string models due to the operation complexity and/or the mixed constraint data types, some string predicates can occur with sufficient complexity to render an analysis of the predicate impractical.

Due to these requirements for symbolic string models, only a limited number of model types have been found to be sufficiently robust to represent the alphabet and length requirements while also being flexible enough models to simulate operations and predicates.

2.3.2 Limitations

While symbolic string models are currently used in many areas of string analysis research, some limitations still exist for all such models. One such limitation we have identified in our explorations of symbolic string models is a collapse problem which occurs as a result of some sequences of string operations and/or predicates. As shown in Section 2.2.3, non-injective operations can result in the "collapse" of multiple values in the constraint solution set (execution paths) into a single value in the resulting solution set. This is a common problem for string variables since many common string operations such as SUBSTRING and REPLACE are non-injective operations.

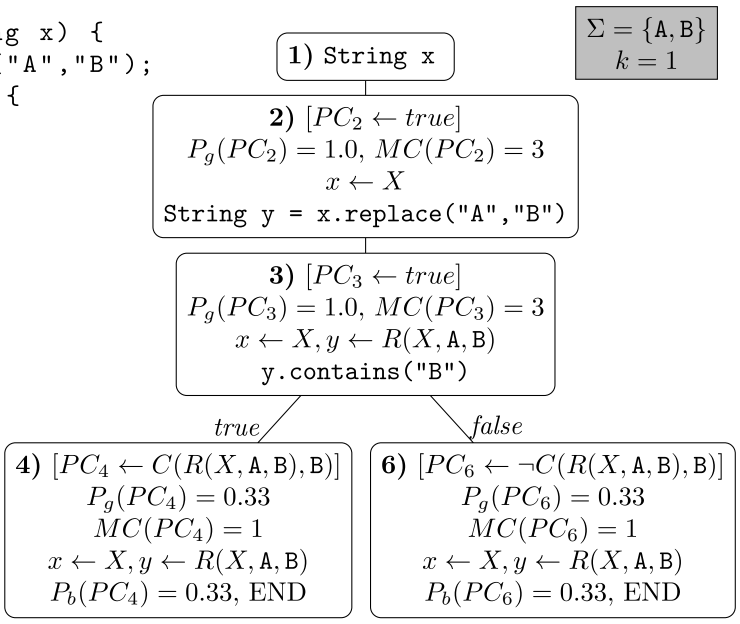
Figure 2.4 illustrates an example of the collapse problem for string variables. PSE of the Java method `testStr` in Figure 2.4a produces the corresponding tree in Figure 2.4c. Table 2.4b presents a table that displays for each line n in the program: the solution set denoted $\mathcal{S}_n$, the size of the solution set denoted $|\mathcal{S}_n|$, the bag (multiset)

```
public void testStr(String x) {
    String y = x.replace("A","B");
    if (y.contains("B")) {
        println("true");
    } else {
        println("false");
    }
}
```

(a) `testStr` method

PC_n	$\mathcal{S}_n$	$\lvert\mathcal{S}_n\rvert$	V_n	$\lvert V_n\rvert$
PC_2	$\{\varepsilon, \mathtt{A}, \mathtt{B}\}$	3	$\{\varepsilon, \mathtt{A}, \mathtt{B}\}$	3
PC_3	$\{\varepsilon, \mathtt{A}, \mathtt{B}\}$	3	$\{\varepsilon, \mathtt{A}, \mathtt{B}\}$	3
PC_4	$\{\mathtt{B}\}$	1	$\{\mathtt{B}, \mathtt{B}\}$	2
PC_6	$\{\varepsilon\}$	1	$\{\varepsilon\}$	1

(b) Actual Solution Set vs. Approximated Solution Set


$\Sigma = \{\mathtt{A}, \mathtt{B}\}$
$k = 1$
1) String x
2) $[PC_2 \leftarrow true]$
$P_g(PC_2) = 1.0$, $MC(PC_2) = 3$
$x \leftarrow X$
String y = x.replace("A","B")
3) $[PC_3 \leftarrow true]$
$P_g(PC_3) = 1.0$, $MC(PC_3) = 3$
$x \leftarrow X, y \leftarrow R(X, \mathtt{A}, \mathtt{B})$
y.contains("B")
true
false
4) $[PC_4 \leftarrow C(R(X, \mathtt{A}, \mathtt{B}), \mathtt{B})]$
$P_g(PC_4) = 0.33$
$MC(PC_4) = 1$
$x \leftarrow X, y \leftarrow R(X, \mathtt{A}, \mathtt{B})$
$P_b(PC_4) = 0.33$, END
6) $[PC_6 \leftarrow \neg C(R(X, \mathtt{A}, \mathtt{B}), \mathtt{B})]$
$P_g(PC_6) = 0.33$
$MC(PC_6) = 1$
$x \leftarrow X, y \leftarrow R(X, \mathtt{A}, \mathtt{B})$
$P_b(PC_6) = 0.33$, END

(c) `testStr` PSE execution tree

Figure 2.4: The `testStr` Java method demonstrating PSE with string variables.

of actual values which would appear in the program denoted V_n, and the number of values in the bag denoted $|V_n|$. The input strings for `testStr` is restricted to the alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}\}$ and limited to an initial length $k = 1$ in order to simplify the example. As a result of this restriction, the input string variable x can only have three possible values: ε, `A`, or `B`. These initial input values assigned to the input variable x as the symbolic string value X as shown in node 2 of the PSE tree. The bag of actual values and the solution sets are in agreement for both PC_2 and PC_3 due to both being *true*, this also results in a MC of 5 representing all the execution paths in the program.

Just as in `testDiv` from Figure 2.3, the collapse problem emerges when calculating probabilities for the *true* and *false* branches of the conditional statement. The *true* branch will be followed when the y string contains the `B` symbol which occurs for two different x input strings $\{\texttt{A},\texttt{B}\}$ and therefore two distinct execution paths. However, PSE calculates the $MC(PC_4)$ from $\mathcal{S}_4$ which is $\{\texttt{B}\}$ since `B` is the only string value of y which satisfies PC_4. It is clear that the two execution paths represented by the x values `A` and `B` have collapsed into the single solution set value `B`. The subsequent branch probability calculation are affected by the difference in MC and produce a probability of 0.33 using Formula 2.1. The actual execution probability can be calculated from the actual bag of string values shown in the table for PC_4 where 2 out of 3 execution paths execute line 4 for an actual probability of 0.66. The *false* branch of the condition is modeled correctly with a branch probability of 0.33 representing 1 out of 3 execution paths executing line 6. Due to the collapse problem, PSE will report that line 4 is equally likely to execute as line 6 when line 4 is actually twice as likely to execute as line 6. This example clearly demonstrates how easily an invalid string analysis can occur with all commonly known symbolic string models.

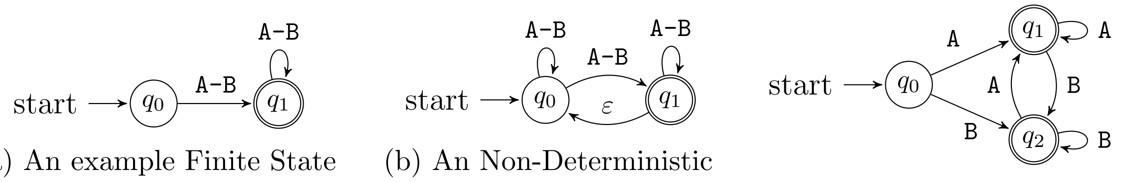
2.4 Modeling String Constraints with Finite State Automata

All currently known symbolic string models suffer from the problem of non-injective path collapse. In order to guarantee a valid PSE analysis for string variables, a symbolic string model is needed which is not susceptible to this problem. Currently, no symbolic string model is known which avoids this problem, thus one must be created. While an entirely novel symbolic string model could be created to avoid this collapse problem while maintaining acceptable performance, it would be much easier to refine a known symbolic string model with proven performance in string analyses. As previously discussed, common model choices for symbolic strings include axiom-based models, bit-vector based models, and finite state automaton based models. Of these choices, the automaton based model was chosen for modification in this work due to the flexible data structure and existing Java libraries which already support using automata-based models in SE.

2.4.1 Finite State Automata

A finite state automaton (FSA) is a mathematical model of computation that operates in response to an external sequence of symbols. A FSA accepts as input a string and either accepts or rejects the string as the result of processing this string. The set of strings accepted by a FSA ($\mathcal{A}$) is called the language of the FSA, denoted as $\mathcal{L}(\mathcal{A})$, and is always a regular language. More formally, a deterministic finite state automaton $\mathcal{A}$ is defined as the quintuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states, $Q \neq \varnothing$
- Σ is a finite set of symbols called the alphabet, $\Sigma \neq \varnothing$
- δ is the transition function, $\delta : Q \times \Sigma \rightarrow Q$
- q_0 is the start state, $q_0 \in Q$
- F is the set of accepting or final states, $F \subseteq Q$



(a) An example Finite State Automaton
(b) An Non-Deterministic FSA
(c) A Non-Minimal FSA

Figure 2.5: Example FSAs

Figure 2.5a is a diagram of a simple FSA that will be used to explain and demonstrate the FSA quintuple. The FSA represented by the diagram accepts a non-empty input string of any combination of `A` and/or `B` symbols. The set of states Q for this FSA contain only two states, q_0 and q_1, i.e. $Q = \{q_0, q_1\}$. The start state of the FSA is the q_0 state and is not an accepting state, i.e. $q_0 \notin F$. The state q_1 is an accepting state, i.e. $q_1 \in F$. The alphabet Σ of this FSA consists of the two symbols `A` and `B` meaning that no other symbols can appear in a string accepted by the automaton. The transition function δ is comprised of the four distinct transitions: $(q_0, \texttt{A}) \rightarrow q_1, (q_0, \texttt{B}) \rightarrow q_1, (q_1, \texttt{A}) \rightarrow q_1, (q_1, \texttt{B}) \rightarrow q_1$. In the diagram, the four distinct transitions are represented by only two directed lines: the line from q_0 to q_1 for the range of symbols from `A` to `B` and the line from q_1 to itself for the same range of symbols. This FSA begins its operation with the start state q_0 as the current state when processing the input string. If the input string is the empty string, the FSA has completed processing the input string and it is rejected since $q_0 \notin F$ in this FSA. If the symbol is not `A` or `B`, the input string is also rejected since there is no transition from state q_0 for any other symbols. When the FSA processes the first symbol and it is either `A` or `B`, the current state is changed to state q_1 as required by the transition function δ. Similarly to state q_0, when a `A` or `B` symbol is processed while state q_1 is the current state, δ requires updating the current state to q_1. Also similarly to state q_0, if a symbol other than `A` or `B` is processed while the current state is q_1, the input string is rejected. When the last symbol in the input string is processed and the string has not already been rejected for invalid symbols, if the current state is an accepting state like q_1, then the input string is accepted, otherwise the input string is rejected.

One characteristic of FSAs important for understanding the use of automata as symbolic string models is determinism where a FSA is either a deterministic (DFA) or non-deterministic (NFA) automaton. For DFAs, the codomain of the transition function δ is a set of states rather than a powerset of states. In other words, every state has only a single transition possible for every input symbol. Additionally, NFAs allow the use of epsilon transitions, a transition from one state to another without processing any input symbols: $(q_{current}, \varepsilon) \rightarrow q_{new}$. The FSA diagramed in Figure 2.5b is a NFA which accepts the same language of strings as the previous FSA in Figure 2.5a which is a DFA. In the NFA diagram, the transition function from the state q_0 has two possible result states when processing either of the symbols `A` or `B`. The NFA also contains an epsilon transition from the state q_1 to the state q_0. One important property of DFAs and NFAs is the relationship between the two where all DFAs ($\mathcal{A}_D$) are also NFAs ($\mathcal{A}_D$) such that $\mathcal{A}_D \subset \mathcal{A}_N$. While NFAs have a different transition function δ than DFAs, every NFA has an equivalent DFA where both automata accept the same language of input strings. Because of this property, every NFA can be converted to an equivalent DFA using a method known as subset construction [31]. Due to these properties and methods, it is often easier to construct a NFA to accept a language of strings and then convert the automata to an equivalent DFA rather than attempting to construct the DFA instead.

A type of automaton which is important for understanding automata as symbolic string models is a minimal automaton. For each regular language, there exists a minimal DFA which accepts the language. This minimal DFA has no equivalent DFA with a smaller number of states. A minimal DFA can be created from any non-minimal DFA by removing unreachable automaton states and merging equivalent automaton states. Figure 2.5c shows a non-minimal DFA which accepts the same language as the minimal DFA in figure 2.5a. There are three well common algorithms used for minimizing DFAs: Hopcroft's algorithm [20], Moore's algorithm [28], and Brzozowski's algorithm [6]. Each of these three algorithms have different best case and average case run times which make the choice of algorithm to use in minimization variable depending upon the context in which it will be used.

2.4.2 Automata as Symbolic String Models

Automata can be used to model symbolic string values such that the language of the automaton is the solution set. There are four important areas of focus when modeling symbolic string constraints with automata: initializing the automata for the input string variable, emulating string operations, processing predicate branch conditions, and model counting the symbolic string automata.

Initializing String Variables

Alphabet:	Set of symbols
Bounding Length:	Integer upper bound
String Type:	Null (Empty), Concrete (Empty String, Literal), Unknown (Simple, Complex)

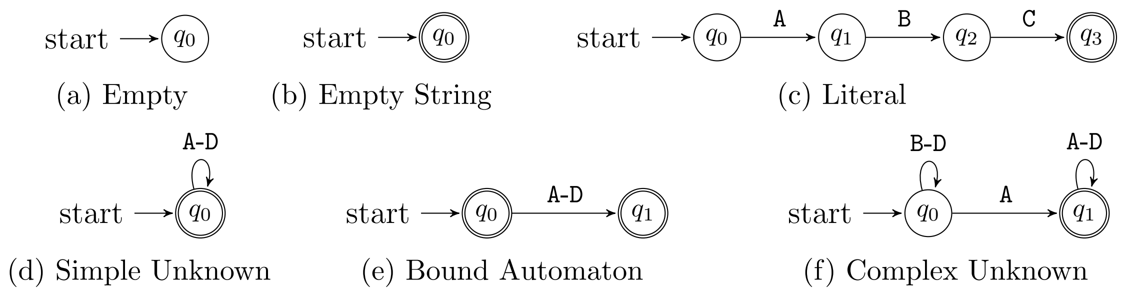
Figure 2.6: Initial Symbolic String Characteristics

In order to explain the initialization of an automata as a symbolic string variable, it is necessary to explain some characteristics of different string solution sets. When a string variable is initialized in a program, it will have three different characteristics: an *alphabet*, an *initial bounding length*, and a *string type*. Figure 2.6 lists each of these characteristics and briefly describes the characteristic.

A lphabet The alphabet of a symbolic string variable is most often specified as an input parameter to the PSE analysis. While individual symbols can be specified, it is often easier to specify a range of symbols in an existing symbol system such as the Unicode symbols.

Initial Bounding Length The initial bounding length of a symbolic string variable is also usually specified as an input parameter to the PSE analysis. This bounding length k is and upper bound on the initial length of strings in the solution set of symbolic string values. This bound is applied due to the exponential increase in solution set size for each linear increase in string length. To better demonstrate this increasing complexity, the solution set for a string with $\Sigma = \{\texttt{A}, \texttt{B}\}, k \leq 1$ is $\{\varepsilon, \texttt{A}, \texttt{B}\}$ where a similar solution set for a string with of $\Sigma = \{\texttt{A}, \texttt{B}\}, k \leq 2$ is $\{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$, an increase in solution set size from 3 to 7 with only a corresponding length increase of 1.

String Type The string type of a symbolic string value is a category assigned after observing the behavior of different string variables as symbolic strings in this analysis of symbolic string values. The three categories of string types are assigned based upon the membership of the solution set for the corresponding symbolic string variables and are the following string value types: null, concrete, and unknown. These categories can be further refined based on the corresponding automaton construction procedures and are the following five string types: empty, empty string, literal, simple unknown, and complex unknown. The *empty* string type is the only null value type and represents the the empty solution. This string type is only created when a string variable represents a null value, e.g. `str = null`. The *empty string* string type is a concrete string value type and represents the solution set containing only the empty string. This string type is most often is generated as part of a predicate where a string is checked to determine if it is the empty string, e.g. implicitly created for `str.isEmpty()`, but can also often occur as string literals, e.g. `str = ""`. A *literal* string type is a concrete string value type and represents the solution set containing only one non-empty string value. A literal type is most often used for string literals



(a) Empty (b) Empty String (c) Literal

(d) Simple Unknown (e) Bound Automaton (f) Complex Unknown

Figure 2.7: FSA Creation Examples.

in a program, e.g. `println("Hello World")`. A *simple unknown* string type is an unknown string value type with a solution set containing more all possible string values for the specified alphabet and bounding length. The simple unknown string type is used when initializing a string variable, often from an input source external to the program under analysis, e.g. `void func(String str)`. Finally, a *complex unknown* string type is an unknown string value type with a solution set containing more than one possible string value but not all possible string values for the specified alphabet and bounding length. The complex unknown string type usually occurs as the result of one or more string operations or as the result of applying a string predicate. Because of this additional complexity of only containing some of the possible string values in the solution set, the corresponding automaton representations of complex unknown string types have a much more complex structure than any of the other four types and are the main focus of this research of automata-based symbolic string constraints.

When initializing a FSA as a symbolic string value, the FSA is created in one of several different ways depending on the type of string represented. Figure 2.7 shows examples of six different automata representing the different construction methods for automata-based symbolic string models. Figure 2.7a is a single non-accepting state which is also the start state in the automata. This automaton is created for all *empty* string type variables. Figure 2.7b shows an automaton consisting of only a single accepting start state. This automaton is created for all *empty string* string type variables. Figure 2.7c is an automaton which accepts only the string value `ABC`. This is an example automaton created for a *literal* string type variable. The construction of such an automaton begins with a non-accepting start state, q_0 in the example, and an additional state, q_1 in the example, is added to the automaton with a transition from q_0 to q_1 for the first symbol of the string value, `A` in the example. This process is repeated for each subsequent symbol in the string value until all the symbols of the string value are reflected in the automaton. To complete the *literal* type automaton, the last added state, q_3 in the example, is then made an accepting state. Figure 2.7d is an automaton which accepts any string of any length in the with the alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$. This is an example automaton created for a *simple unknown* string type variable. The creation of such automata requires only a two part process. First, a single accepting start state is used. Then, a transition is added to the automaton from the start state to itself for every symbol in the alphabet Σ. Figure 2.7e is a *bounded* automaton which in this example is a modification of the previous *simple unknown* automaton in Figure 2.7d. This example automaton is bounded by an initial bounding length k where $k \leq 1$. Bounded automata like this are useful in some analyses where in some operations and predicates an infinite automaton can not be used. Finally, Figure 2.7f is an automaton that accepts any non-empty string with the alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ where the accepted string must contain the `A` symbol. This is just one example of an automaton representing a *complex unknown* string type variable. This string type does not have a defined construction since such types are only produced as a result from string operations and string predicates. To demonstrate this, the example automaton in Figure 2.7f was constructed by asserting the Contains(A) predicate for a *simple unknown* string type for the alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ rather than through any specific construction procedure. While *complex unknown* string type variables are important to understanding how

Both following the creation of an automaton as symbolic string models and following any string operations or string predicates, an automaton is evaluated for determinism and converted into a deterministic automaton if it is not already. Then the automaton is minimized. This minimal DFA is optimized for space by utilizing the fewest number of states and transitions possible to accept the language of the automaton. Additionally, the minimal DFA is optimized for future algorithms which emulate string operations and predicates since the performance complexity of those algorithms is dependent upon the number of states and transitions in automata.

String Operations

Category:	**Examples**
Injective	ToString(str), Reverse(str)
Additive	Concatenate(str_1, str_2), Insert(str_1, int, str_2)
Subtractive	Delete(str_1, int_1, int_2), Substring(str_1, int_1, int_2)
Substitutive	Replace($str_1, char_1, char_2$), Replace(str_1, str_2, str_3)
Mixed Constraint	Length(str), IndexOf($char$)
Infeasible	Format($str, obj[]$), HashCode(str)

Figure 2.8: String Operation Categories

In order to discuss the simulating of string operations for automata-based symbolic string models, six categories of string operations will first be explored. We assigned these six categories to the different string operations we encountered in this analysis of automata-based symbolic string models. These six categories are as follows: injective, additive, subtractive, substitutive, mixed constraint, and infeasible. These categories are listed with example operations in the table in Figure 2.8. For the purposes of this discussion, the string variable upon which the string operation is being performed is always specified as the primary string argument for an operation.

Injective We identified *injective* string operations which are characterized by the guaranteed one-to-one correspondence between the string values in the solution sets of the resulting string and the primary string argument of the operation. In practice, only the most simple of string operations have been identified as *injective* operations. An example of an *injective* string operation is the ToString(str) operation where the string returns a string representation of its value which is just the string itself for a string data type. A more important example of an *injective* string operation is the assignment of a string value to a string variable. When this type of operation is simulated for automata-based symbolic string models, the algorithms simply create a copy of the primary string argument automaton and returns the copy as the result of the operation.

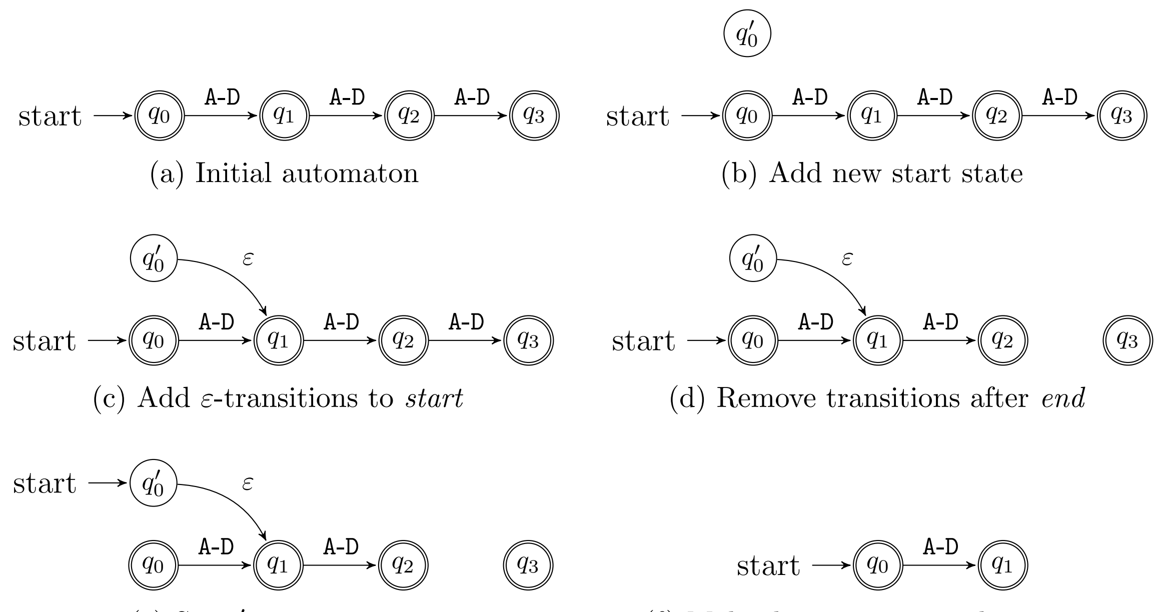
Additive We identified *additive* string operations where additional symbols appear in the string returned by the operation compared to the primary string argument. This category of operation does not discriminate based upon where in the primary string argument the new symbols have been added. For example, the operation Concatenate(str_1, str_2) adds the new symbols of str_2 after the existing symbols of the primary string argument str_1. However, the Insert(str_1, int, str_2) operation adds the new symbols of str_2 to the primary string argument str_1 as the index specified by int. Both of these example operations add new symbols to primary string argument, but not necessarily at the same location within the string. While these differences in *additive* operations requires different simulating algorithms for each operation instead of relying on a single algorithm as is the case with *injective* operations.

Subtractive We identified *subtractive* string operations where symbols have been removed in the string returned by the operation compared to the primary string argument. This category of operation includes all operations which remove symbols from the primary string argument regardless of the location at which the symbols are removed. For example, the $\textsc{Delete}(str_1, int_1, int_2)$ operation removes all the symbols in the primary string argument str_1 from the index specified by the first integer argument int_1 to the index specified by the second integer argument int_2. While the $\textsc{Substring}(str_1, int_1, int_2)$ operation accepts the same arguments, it removes all symbols from the primary string argument str_1 both before the index specified by the first integer argument int_1 and after the second integer argument int_2. Just as with *additive* operations, simulating *subtractive* operations requires unique algorithms for each specific operation.

Substitutive We identified *substitutive* string operations where symbols have been substituted in the string returned by the operation compared to the primary string argument. There are two sub-types of *substitutive* string operations: simple and complex. The simple variant of the *substitutive* operation performs the substitution of symbols without altering the length of the string, i.e., in a simple *substitutive* operation, the lengths of the primary string argument and the string resulting from the operation are the equal. An example of this is the $\textsc{Replace}(str_1, char_1, char_2)$ operation where all instances of the first symbol argument $char_1$ in the primary string argument str_1 are replaced with the second symbol argument $char_2$. Because individual symbols are being replaced by other individual symbols in the primary string, the total length of the string does not change. The complex variant of the *substitutive* operation performs the substitution of symbols without being able to guarantee that the length of the string will be unaltered, i.e., in a complex *substitutive* operation, the lengths of the primary string argument and the string resulting form the operation may or may not be equal. An example of this is the REPLACE(str_1, str_2, str_3) operation all instances of the second string argument str_2 within the primary string argument str_1 are substituted for the third string argument str_3. Because the lengths of the second and third string arguments may differ, the replacement of the substrings within the primary string can result in string with a different length. Despite the differences between the simple and complex *substitutive* operations, the simulation algorithms for these operations are similar enough to group them within the same category as we have done.

Mixed Constraint We identified *mixed constraint* string operations where the result of the operation was not a string data type. An example of this operation type is the LENGTH(str) operation which returns an integer value representing the length of the string. While the LENGTH(str) operation could simulated by a fairly simple algorithm, the return value would need to be a symbolic integer which could be represented by one of many different symbolic integer models. Any algorithm used to simulate a *mixed constraint* operation for automata-based symbolic strings would also need to be specialized to return a symbolic model for the return data type of the operation.

Infeasible We identified *infeasible* string operations where the complexity requirements of simulating the operation for automata-based symbolic string models is not feasible by any known algorithms. An example of such an operation is the FORMAT($str, obj[]$) operation where the primary string argument automaton would need to be processed for a multitude of special character sequences for each possible string value in the solution set of the automaton. Similarly, the *mixed constraint*



(a) Initial automaton

(b) Add new start state

(c) Add ε-transitions to *start*

(d) Remove transitions after *end*

(e) Set q'_0 as start state

(f) Make deterministic and minimize

Figure 2.9: FSA SUBSTRING($\mathcal{A}, 1, 2$) operation example

operation HASHCODE(*str*) is also an *infeasible* operation due to the complexity required to compute the hash code for each possible string value in the primary string argument solution set.

Figure 2.9 demonstrates how a string operation can be simulated for an automata-based symbolic string model. In this example, the SUBSTRING operation with a start index of 1 and an end index of 2. This operation performed for a *simple unknown* string type variable, shown in Figure 2.9a, with an alphabet $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ and initial bounding length $k \leq 3$. The first step to simulate the operation is the addition of a new state q'_0 to the automaton as shown in Figure 2.9b. Next Figure 2.9c show that an epsilon-transition is created from the new state q'_0 to the state at a length of 1 from the start state, the state q_1 in this case. This epsilon-transition represents the removal of the symbols not captured in the substring operation, i.e. the symbols occurring before the *start* index. Figure 2.9d illustraits the next step where the transitions leaving the states at the *end* length in the automaton have their outgoing transitions removed and are made accepting states if they are not already. This step emulates the removal of all symbols following the *end* index in original string. Figure 2.9e shows that the start state of the automaton is changed from the previous start state q_0 to the newly added start state q_0'. Finally, Figure 2.9f is an automaton which is the result of making the previous automaton deterministic and then minimizing the resulting automton. It is in this manner that the SUBSTRING string operation is simulated so that an automaton accepting the language $\mathcal{L}(\mathcal{A}) = \{\varepsilon, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ is created and returned from the algorithm. Other string operation simulation algorithms operate similarly by altering the structure of automata, usually by adding states, adding and removing transitions, and adding or removing states from the set of accepting states.

```
1: procedure FastSubstring($\mathcal{A}$)
2:     $\mathcal{A}' \leftarrow$ Copy($\mathcal{A}$)
3:     for all $q_i \in Q'$ do
4:         $\delta'(q_0', \epsilon) \rightarrow q_i$
5:         for all $q_j \in F'$ do
6:             $\delta'(q_i, \epsilon) \rightarrow q_j$
7:         end for
8:     end for
9:     return $\mathcal{A}$
10: end procedure
```

(a) Fast SUBSTRING Algorithm

```
1: procedure FollowTransitions($Q$)
2:     $Q_r \leftarrow \varnothing$
3:     for all $q \in Q$ do
4:         for all $\langle q, \alpha, q_t \rangle \in \delta_q$ do
5:             $Q_r \leftarrow Q_r \cup \{q_t\}$
6:         end for
7:     end for
8:     return $Q_r$
9: end procedure
```

(b) FOLLOWTRANSITIONS

```
1: procedure PreciseSubstring($\mathcal{A}, start, end$)
2:     $\mathcal{A}' \leftarrow$ Copy($\mathcal{A}$)
3:     $i \leftarrow 0, Q_i \leftarrow \{q_0'\}$
4:     while $i < start$ do
5:         $Q_i \leftarrow$ FollowTransitions($Q_i$)
6:         $i \leftarrow i + 1$
7:     end while
8:     $Q_s \leftarrow Q_i$
9:     while $i < end$ do
10:        $Q_i \leftarrow$ FollowTransitions($Q_i$)
11:        $i \leftarrow i + 1$
12:    end while
13:    $Q_e \leftarrow Q_i$
14:    RemoveTransitions($Q_e$)
15:    $q_0' \leftarrow q_{new}$
16:    $F' \leftarrow Q_e$
17:    for all $q_i \in Q_s$ do
18:        $\delta'(q_0', \epsilon) \rightarrow q_i$
19:    end for
20:    return $\mathcal{A}$
21: end procedure
```

(c) Precise SUBSTRING Algorithm

Figure 2.10: Two SUBSTRING operation algorithms

An important consideration when selecting an appropriate simulation algorithm for a string operation is the balance of precision and performance of the algorithm. Most string operations can be simulated by many different algorithms some of which have less computational complexity at the cost of over-approximating the solution sets for the resulting automata-based symbolic string models. An example of this performance difference can be seen in the two algorithms for the SUBSTRING operation shown in Figure 2.10. The FASTSUBSTRING (2.10a) algorithm does not utilize any of the indices which are arguments for a SUBSTRING operation, instead the algorithm returns an over-approximated automaton where the solution set contains substring values for all possible start and end indices to the SUBSTRING operation. The PRECISESUBSTRING (2.10c) algorithm makes use of the *start* and *end* indices to provide a more precise resulting automaton. This algorithm is the version of the SUBSTRING operation used in the previous example shown in Figure 2.9. This precision comes at the performance cost incurred by having to traverse along the transitions of the automaton up to the *start* and *end* lengths. It is clear that PRECISESUBSTRING is an algorithm with much greater precision than the FASTSUBSTRING algorithm, but this precision does come at the cost of performance. The computational complexity of the FASTSUBSTRING algorithm is depends upon the number of states in the automaton and the number of those that are accepting due to the nested loops on lines 3 and 5. In comparison, the PRECISESUBSTRING algorithm contains two loops on lines 4 and 9 with a call to the FOLLOWTRANSITIONS (2.10b) sub-algorithm nested within the loop. This FOLLOWTRANSITIONS sub-algorithm itself contains nested loops on lines 3 and 4 which result in a PRECISESUBSTRING algorithm which contains two triple nested loops and two single loops on lines 14 and 17. The improvement in computational complexity of the FASTSUBSTRING algorithm compared to the

PRECISESUBSTRING algorithm is clear, but at the cost of over-approximating the solution set.

There do exist some string operations which will always have the potential to over-approximate the solution set of a string variable. These are of subset of string operations under which regular languages are not closed. Because the languages accepted by automata are only regular languages, any operation under which regular languages are not closed cannot be precisely modeled by an automaton. Thus, when some string functions are modeled by an algorithm, the resulting automaton can be an over-approximation of the actual solution set. For example, some algorithms which model the REPLACE operation produce an over-approximated solution set [38, 39]. The choice to use an over-approximating modeling algorithm is made because either it is much more efficient than a precise algorithm or no known precise modeling algorithm is known. Which approach to utilize when choosing a string operation simulation algorithm depends upon the analysis in which such algorithms are used.

String Predicates

Category:	**Examples**
Full Match	$str_1 = str_2$, $\textsc{EqualsIgnoreCase}(str_1, str_2)$
Partial Match	$\textsc{Contains}(str_1, str_2)$, $\textsc{StartsWith}(str_1, str_2)$
Mixed Constraint	$\textsc{Length}(str) > int$, $\textsc{IndexOf}(char) < int$
Infeasible	$\textsc{Matches}(str_1, str_{regex})$, $str_1 = \textsc{Format}(str_2, obj[])$

Figure 2.11: String Predicate Categories

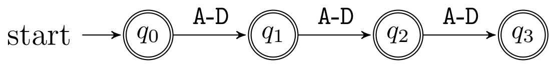
In order to examine the application of string predicates for automata-based symbolic string models, four categories of string predicates must first be reviewed. We assigned these four categories to the different string predicates we encountered in this analysis of automata-based symbolic string models. These four categories are as follows: full match, partial match, mixed type, and infeasible. These categories are listed with example predicates in the table in Figure 2.11.

Full Match We identified *full match* string predicates where two string values are compared to determine equality. Some of these predicates include some prepossessing of the string arguments such that the string returned from this prepossessing is used in the equality comparison. An example of a string predicate which includes some prepossessing of the string arguments is the EqualsIgnoreCase(str_1, str_2) predicate where both string arguments str_1 and str_2 have their symbols converted into the same case before being compared for equality. This is an extra step compared to the much simpler predicate $str_1 = str_2$ which requires no prepossessing step before comparing str_1 and str_2 for equality. *Full match* string predicates are very simple to simulate for automata-based symbolic string models because the equality predicate is just the automaton Intersect operation which has many known algorithms which can be implemented [32]. Similarly, to check for inequality the Intersect operation is used between the first automaton and the automaton returned from the Complement operation performed on the automaton of the second string.

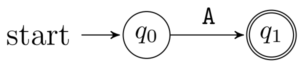
Partial Match We identified *partial match* string predicates where two string values are compared to determine if a portion of the first string value is equal to either a second string value or a portion of a second string value. One example of this kind of predicate is Contains(str_1, str_2) where the first string str_1 is checked to see if it contains a sequence of symbols equal to the second string str_2. Similarly, the *partial match* predicate StartsWith(str_1, str_2) where the first string str_1 is checked to see if it begins with the sequence of symbols equal to the second string str_2. *Partial match* string predicates like *full match* predicates are fairly simple to simulate for automata-based symbolic string models. Just like *full match* predicates, *partial match* predicates use the automaton Intersect operation to check for partial equality, however this check for partial equality requires the concatenation of simple unknown string automata such that the new automata is added to the beginning and/or the end of the second automaton in the predicate depending on the specific predicate used. The check for partial inequality for *partial match* predicates is handled similarly to *full match* types where the previously described concatenated second automaton is processed by the Complement automaton operation before intersected with the first automaton argument.

Mixed Constraint We identified *mixed constraint* string predicates where the operations contained in the predicate where *mixed constraint* string operations. An example of this kind of predicate is $\textsc{Length}(str) > int$ where the length of a string value is compared to an integer value. Just like *mixed constraint* string operations, any algorithm which can simulate a *mixed constraint* string predicate would need to be specialized to handle both automata-based symbolic strings and symbolic values for the other data types included in the predicate.

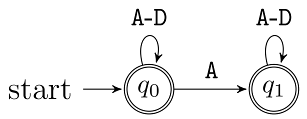
Infeasible We identified *infeasable* string predicates where the complexity requirements of simulating the predicate for automata-based symbolic string models is not feasible by any known algorithms. This infeasibility can occur due to either the logic of the predicate itself being to complex or due to the inclusion of *infeasable* string operations in the predicate. An example of an *infeasible* complex predicate is $\textsc{Matches}(str_1, str_{regex})$ which would require the processing of the automaton representing the regular expression string to be processed for all the possible regular expressions such a string could have, a very complex task. An example of an *infeasible* predicate due to included *infeasible* operations would be $str_1 = \textsc{Format}(str_2, obj[])$ where equality is checked between one string str_1 and a formatted string returned from the *infeasible* string operation FORMAT($str_2, obj[]$).



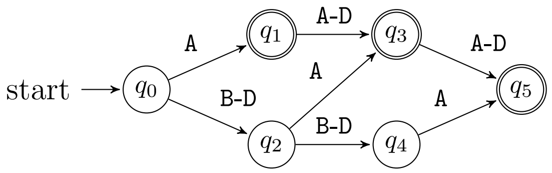
(a) Primary automaton



(b) `A` automaton



(c) `A` automaton prepared for INTERSECT



(d) Resulting automaton

Figure 2.12: FSA CONTAINS($\mathcal{A}_1$, `A`) *true* predicate example

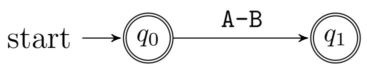
Figure 2.12 how a string predicate can be simulated for an automata-based symbolic string model. In this example, the *true* branch outcome is asserted for the CONTAINS($\mathcal{A}_1$, `A`) predicate. This predicate is performed for a *simple unknown* string type variable, shown in Figure 2.12a, with an alphabet $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ and initial bounding length $k \leq 3$. Figure 2.12b shows the *literal* string type automaton that is created for the `A` argument of the CONTAINS predicate. In order for the simulate the predicate, the argument automaton must be modified to represent any string containing `A` so that it can be used in an INTERSECT automaton operation. Figure 2.12c illustrates how this is done by concatenating two *simple unknown* string type automata to the argument automaton, one preceding the argument automaton and one following it. Figure 2.12d is the automaton which is returned by the intersection of the primary automaton (Figure 2.12a) and the prepared argument automaton (Figure 2.12c). This automaton accepts any string up to length 3 which also contains the symbol `A` which reflects only the string values for which the predicate CONTAINS($\mathcal{A}_1$, `A`) is *true*.

Model Counting

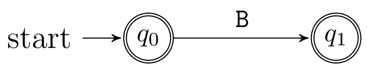
When modeling string constraints with automata in a quantitative string analysis such as PSE, the MC must be calculated using the automaton quintuple. Fortunately, research has shown that model counting an automaton can be reduced to exactly counting the accepting paths of the automaton [1]. This means that a simple algorithm for model counting automata requires a straightforward traversal of the automata graph until no more transitions can be explored or up to a specified bounding length k if the automaton is contains one or more infinite cycles. This MC procedure will be examined in detail in Chapter 3. The calculation of MC from an automata is an area of ongoing research aimed at reducing the performance cost of model counting automata [1, 27]. While the automaton MC calculation procedure could be optimized for better performance, the additional performance cost that is required by even the most efficient algorithms when added to the other additional performance costs required for using a PSE analysis results in a much more costly analysis than an SE analysis. In addition to these performance issues, other problems have been observed during quantitative string analyses which can render such analyses invalid.

2.4.3 Limitations

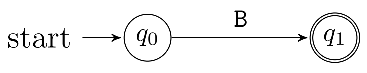
The two significant limitations which have been observed in both our research and other existing research using automata in SE and PSE are the collapse problem as discussed in Section 2.3.2 and over-approximation as discussed in Section 2.4.2. The occurrence of either one of these problems in a quantitative analysis can be enough to alter the MC and the subsequent probability calculations which can result in an invalid PSE analysis.



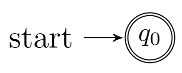
(a) FSA for $\Sigma = \{\texttt{A}, \texttt{B}\}$ and $k = 1$.



(b) FSA for Replace(X, A, B)



(c) PC_4 Automaton



(d) PC_6 Automaton

Figure 2.13: An sample Java method that uses PSE with String variables.

To illustrate how the collapse problem occurs for automata-based symbolic string models, the example from Figure 2.4a will be revisited. The automata in Figure 2.13 correspond to the symbolic value X shown in the PSE tree in Figure 2.4c. This automaton accepts ε since the initial state q_0 is accepting and reaches the second accepting state q_1 by reading either an A or B symbol. The automaton therefore accepts three strings: ε, A, and B. When the initial string value undergoes a Replace operation on line 2, it produces the automaton shown in Figure 2.13a. This automaton accepts ε since the initial state q_0 is accepting and reaches the second accepting state q_1 by reading a B symbol. This automaton therefore accepts only two strings: ε and B. It is clear that the non-injective Replace operation produces an automaton which accepts fewer strings than the initial automaton, demonstrating the collapse problem for automaton-based symbolic string models. This is also seen in the automata for PC_4 and PC_6 in Figures 2.13c and 2.13d respectively where both automaton only accept a single string for a total of two possible execution paths rather than the three which actually exist.

CHAPTER 3

AUTOMATA-BASED SYMBOLIC STRING MODELS

This chapter details the theoretical work required to complete the analysis of automata-based symbolic string models. An automata-based symbolic string model, shortened as automata model and denoted with the symbol $\mathcal{M}$, encapsulates one or more FSAs as well as any other important data and data structures required to represent a solution set. Two instances of important data for each of the four automata models used in this analysis are the alphabet and the length of the automaton. The alphabet of an automata model is used to represent all the symbols possible in a quantitative analysis rather than just the symbols appearing in the solution set of the symbolic string. These additional symbols in the alphabet are needed to simulate string operations and predicates which require the construction of automata representing every possible symbol rather than just the subset of symbols which appear in the language of a specific automaton. The length of an automata model is used differently depending on the version of the automata model but ultimately is used to ensure the finite length of the automata when used in algorithms for string operations, string predicates, or model counting. The automaton structure itself remains the same for each type of automata model, using the quintuple defined in Section 2.4.1.

For our analysis of automata-based symbolic string models we utilized four distinct versions of automata models: *unbounded*, *bounded*, *aggregate bounded*, and *weighted-transition aggregate*. Section 3.1 details the *unbounded* automata model which serves as the basis for comparison to the other modified automata models. Section 3.2 describes the *bounded* automata model which is a fairly simple modification of the *unbounded* version. Section 3.3 describes the *aggregate bounded* automata model which is a further modification of the *bounded* version. Finally, Section 3.4 covers the *weighted-transition aggregate* automata model which incorporates a redesigned FSA using weighted-transitions as a further refinement of the *aggregate* automata model. Each of these automata models will be defined and three important uses of the automata model will be examined: model counting the automata model, string operations which result in an over-approximation of the solution set for the automata model, and string operations which result in a collapse problem for the automata model.

3.1 Unbounded Automata Model

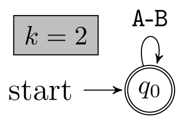


Figure 3.1: Unbounded Automata Model:
$\Sigma = \{\texttt{A}, \texttt{B}\}$, $k = 2$

The *unbounded* automata model consists of a FSA, an alphabet, and a length. Figure 3.1 is an example of such an automata model with an alphabet $\Sigma = \{\texttt{A}, \texttt{B}\}$ and a length value $k = 2$. The length value of the *unbounded* automata model is used as a maximum length counter when traversing automata when performing algorithms to ensure that automaton cycles do not cause endless loops. The maximum length value must be modified as part of *additive*, *subtractive*, and *complex substitutive* string operations to accurately reflect the way these operations alter the lengths of the resulting strings. The creation process for the FSA component of the *unbounded* automata model is completed as described in Section 2.4.2 for each of the *empty*, *empty string*, *literal*, *simple unknown*, and *complex unknown* string types. The *unbounded* automata model is equivalent to the automata models used in known string analysis research [1, 7, 18, 19, 34, 36–41].

3.1.1 Model Counting

```
1:  procedure CountUnbounded($\mathcal{M}$)
2:      $k \leftarrow k \in \mathcal{M}$
3:      $\mathcal{A} \leftarrow \mathcal{A} \in \mathcal{M}$
4:      $q_0 \leftarrow q_0 \in \mathcal{A}$
5:      $mc_r \leftarrow$ MCUnbounded($q_0, k$)
6:      if $q_0 \in F$ then
7:          $mc_r \leftarrow mc_r + 1$
8:      end if
9:      return $mc_r$
10: end procedure
```

(a) Coordinating algorithm

```
1:  procedure MCUnbounded($q$, $i$)
2:      if $i \geq 0$ then
3:          $i \leftarrow i - 1$
4:      end if
5:      $mc_r \leftarrow 0$
6:      for $q_d \in \{q_e \mid \delta(q, \alpha) \rightarrow q_e\}$ do
7:          if $q_d \in F$ then
8:              $mc_r \leftarrow mc_r + 1$
9:          end if
10:         if $i > 0$ then
11:             $mc_r \leftarrow mc_r +$ MCUnbounded($q_d, i$)
12:         end if
13:     end for
14:     return $mc_r$
15: end procedure
```

(b) Recursive model counting algorithm

Figure 3.2: Model counting algorithms for *unbounded* automata models

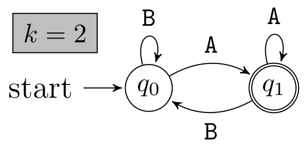
Figure 3.2 shows the model counting algorithms for the *unbounded* automata models where the coordinating algorithm is Figure 3.2a and the recursive algorithm is Figure 3.2b. To count an *unbounded* automata model $\mathcal{M}$, the model is passed as the argument to the coordinating CountUnbounded algorithm. Next, the length k and the start state q_0 of the FSA $\mathcal{A}$ are retrieved for the call of the MCUnbounded recursive algorithm. The initial call to this algorithm is made using the start state

q_0 as the state argument q and using the length value k as the counter i. Lines 1-3 of MCUnbounded decrements the counter i so that the recursive loops of this algorithm will not continue endlessly. Line 4 initializes the model count value to be returned by the algorithm mc_r. Line 5 of MCUnbounded begins an iteration through all the outgoing transitions of the state q provided as an input argument to the algorithm, the tuple $\langle \alpha_t, q_d \rangle$ which are the symbol and destination state of each outgoing transition are the values used in this for loop. Lines 6-8 of MCUnbounded check to see if the transition destination state q_d is an accepting state, i.e. $q_d \in F$. If q_d is accepting, then Line 7 of MCUnbounded increments the currently tracked model count mc_r of transitions outgoing from the initial state q. Next, lines 9-11 of MCUnbounded check the counter i to determine if recursion should continue. If so, the result of a recursive call to the MCUnbounded algorithm is added to the currently tracked number of model counts mc_r. The recursive call to the algorithm on line 10 of MCUnbounded is made specifying the transition destination state q_d and the updated length counter i. As a result of the loop covering Lines 5-12 of MCUnbounded, the number of strings accepted by the automaton up to length i for outgoing transitions from the state q will be tracked by the mc_r variable and is returned from the algorithm on Line 13 of MCUnbounded. When the recursive calls to MCUnbounded finally complete, line 5 of CountUnbounded stores the returned model count as mc_r. Lines 6-8 of CountUnbounded then check if the start state q_0 is accepting and increments the model count mc_r if so, this accounts for the possible empty string value.

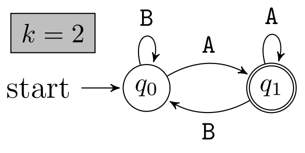
While there is ongoing research into more efficient model counting algorithms for automata [1, 27] as discussed in Section 2.4.2, the straightforward and easy to understand Algorithm 3.2 is sufficient for our analysis of automata-based symbolic string models. This particular model counting algorithm will serve as a basis of comparison when discussing the corresponding algorithms for *bounded*, *aggregate*, and *weighted* automata models.

3.1.2 Over-Approximation

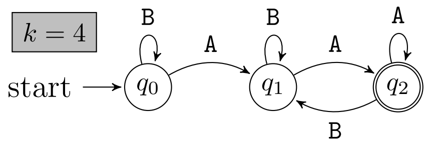
The over-approximation of a solution set due to the string operation simulating algorithms has been discussed previously in Sections 2.4.2 and 2.4.3. However, there is another possible source of over-approximation of a solution set for automata-based symbolic string models, the composition automata model itself. The string operation Concatenation is used to demonstrate this type of over-approximation for an *unbounded* automata model. The later Sections 3.2.2, 3.3.3, and 3.4.5 will examine how the model caused over-approximation does or does not occur for *bounded*, *aggregate*, and *weighted* automata models for the same Concatenation operation scenario.



(a) $\mathcal{M}_1$ *unbounded* automata model



(b) $\mathcal{M}_2$ *unbounded* automata model



(c) $\mathcal{M}_C$ *unbounded* automata model
resulting from Concatenation operation

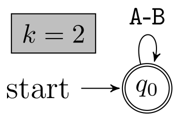
Figure 3.3: Concatenation($\mathcal{M}_1, \mathcal{M}_2$) example for *unbounded* automata model

The example shown in Figure 3.3 demonstrates how over-approximation due to model structure can occur for *unbounded* automata models. This example presents the string operation $\textsc{Concatenation}(str_1, str_2)$ simulated for *unbounded* automata models $\mathcal{M}_1$ and $\mathcal{M}_2$ representing the string variables str_1 and str_2 respectively. Figure 3.3a shows the model diagram corresponding to $\mathcal{M}_1$ which accepts the language of strings up to a length 2 which contain only the symbols `A` and `B` and must end in the symbol `A`, i.e., $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\mathtt{A}, \mathtt{B}\}$ and maximum length $k_{\mathcal{M}_1} = 2$ and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\mathtt{A}, \mathtt{AA}, \mathtt{BA}\}$. This *unbounded* automata model structure can appear often in a software program by asserting the $\textsc{EndsWith}(\mathcal{M}, \mathtt{A})$ predicate. Figure 3.3b shows the model diagram corresponding to $\mathcal{M}_2$ which has an identical structure to the $\mathcal{M}_1$ *unbounded* automata model, i.e., $\Sigma_{\mathcal{M}_2} = \{\mathtt{A}, \mathtt{B}\}$, $k_{\mathcal{M}_2} = 2$, and $\mathcal{S}_{\mathcal{M}_2} = \{\mathtt{A}, \mathtt{AA}, \mathtt{BA}\}$. Figure 3.3c shows the diagram of the *unbounded* automata model $\mathcal{M}_C$ which is the result of the $\textsc{Concatenation}$ operation. This resulting model has the same alphabet as both the $\mathcal{M}_1$ and $\mathcal{M}_2$ models since the alphabet of the automata model resulting from the $\textsc{Concatenation}$ operation will be the $\textsc{Union}$ of the two argument automata models. The maximum length value of the $\mathcal{M}_C$ model has been set as $k_{\mathcal{M}_C} = 4$ to reflect the addition of two length 2 automata models in the *additive* string operation. The cause of the over-approximation for the $\mathcal{M}_C$ *unbounded* automata model is this maximum length value and the fact that the length is tracked separately from the FSA. This can be seen by comparing the expected solution set $\mathcal{S}_e$ and actual solution sets for the $\mathcal{M}_C$ model $\mathcal{S}_{\mathcal{M}_2}$. It is expected that the $\textsc{Concatenation}$ of $\mathcal{M}_1$ with solution set $\mathcal{S}_{\mathcal{M}_1} = \{\mathtt{A}, \mathtt{AA}, \mathtt{BA}\}$ and $\mathcal{M}_2$ with solution set $\mathcal{S}_{\mathcal{M}_2} = \{\mathtt{A}, \mathtt{AA}, \mathtt{BA}\}$ will produce the solution set $\mathcal{S}_e = \{\mathtt{AA}, \mathtt{AAA}, \mathtt{ABA}, \mathtt{BAA}, \mathtt{AAAA}, \mathtt{AABA}, \mathtt{BAAA}, \mathtt{BABA}\}$ which has a model count $MC(\mathcal{S}_e) = 8$. However, the actual solution set produced for the $\mathcal{M}_C$ *unbounded* automata model is $\mathcal{S}_a = \{\mathtt{AA}, \mathtt{AAA}, \mathtt{ABA}, \mathtt{BAA}, \mathtt{AAAA}, \mathtt{AABA}, \mathtt{BAAA}, \mathtt{BABA}, \mathtt{ABAA}, \mathtt{ABBA}, \mathtt{BBAA}\}$ which has a model count $MC(\mathcal{S}_{\mathcal{M}_C}) = 11$. This over-approximation occurs to the inclusion of the three string values `ABAA`, `ABBA`, and `BBAA` which cannot be produced from the Concatenation of $\mathcal{M}_1$ and $\mathcal{M}_2$ since `AB` $\notin \mathcal{S}_{\mathcal{M}_1}$ and `BB` $\notin \mathcal{S}_{\mathcal{M}_1}$. These three string values only exist in the actual solution set $\mathcal{S}_{\mathcal{M}_C}$ due to the concatenation of the $\mathcal{M}_1$ and $\mathcal{M}_2$ automata as infinite automata with a independently tracked length value. Because this type of over-approximation for *unbounded* automata models is due to the alteration of maximum length value independent of the FSA, it is possible to similarly over-approximate the resulting models for any of the *additive*, *subtractive*, or *complex substitutive* string operations.

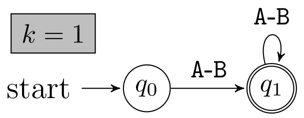
3.1.3 Collapse

In addition to the problems of over-approximation the *unbounded* automata model suffers from, the model is also susceptible to collapse problems as described in Sections 2.2.3 and 2.3.2. These collapse problems can manifest as the result of either *subtractive* or *substitutive* (both *simple* and *complex*) string operations. The Delete operation is demonstrated to illustrate how collapse problems can occur due to the use of *subtractive* operations for *unbounded* automata models. Similarly, the Replace operation is examined to explain how collapse problems can occur when simulating *substitutive* operations for *unbounded* automata models. The specific Replace operation chosen for this example is the *simple substitutive* version of the operation Replace$(\mathcal{M}, \alpha_1, \alpha_2)$ where α_1 is a single symbol to be replaced in the primary string and α_2 is a single symbol which replaces the α_1 symbols in the resulting string. The later Sections 3.2.3, 3.3.4, and 3.4.6 will examine how the collapse problem does or does not occur for *bounded*, *aggregate*, and *weighted* automata models in the same Delete and Replace scenarios.

The example shown in Figure 3.4 demonstrates how the collapse problem can occur



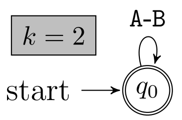
(a) $\mathcal{M}_1$ *unbounded* automata models



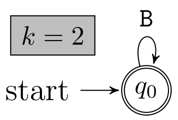
(b) $\mathcal{M}_D$ *unbounded* automata model resulting from DELETE operation

Figure 3.4: DELETE$(\mathcal{M}_1, 1, 2)$ example for *unbounded* automata model

for *unbounded* automata models due to *subtractive* string operations. This example presents the string operation DELETE(str_1, int_s, int_e) simulated for the *unbounded* automata model $\mathcal{M}_1$ and the start and end indices 1 and 2. Figure 3.4a shows the diagram corresponding to $\mathcal{M}_1$ such that $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\mathtt{A}, \mathtt{B}\}$, a maximum length $k_{\mathcal{M}_1} = 2$, and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \mathtt{A}, \mathtt{B}, \mathtt{AA}, \mathtt{AB}, \mathtt{BA}, \mathtt{BB}\}$. Figure 3.4b shows the diagram of the *unbounded* automata model $\mathcal{M}_D$ which is returned from the DELETE$(\mathcal{M}_1, 1, 2)$ operation. This automata model $\mathcal{M}_D$ has the same alphabet as $\mathcal{M}_1$ such that $\Sigma_{\mathcal{M}_D} = \{\mathtt{A}, \mathtt{B}\}$. The maximum length value k for $\mathcal{M}_D$ has been reduced to $k = 1$ to reflect the deletion of symbols. The collapse problem is identified when comparing the model count for the original automata model $MC(\mathcal{M}_1) = 7$ to the model count of the resulting automata model $MC(\mathcal{M}_D) = 2$ instead of the expected model count value of 6 (the empty string ε does not contain a symbol at index 1 and therefore would return an error from the DELETE$(str_1, 1, 2)$ operation). Because of the DELETE operation, each of the A, AA, and AB string values in $\mathcal{S}_{\mathcal{M}_1}$ correlates to the single A string value in $\mathcal{S}_{\mathcal{M}_D}$. The same relationship exists for the B, BA, and BB string values in $\mathcal{S}_{\mathcal{M}_1}$ and the single B string value in $\mathcal{S}_{\mathcal{M}_D}$. Thus, the original solution set values $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \mathtt{A}, \mathtt{B}, \mathtt{AA}, \mathtt{AB}, \mathtt{BA}, \mathtt{BB}\}$ collapse into the solution set values $\mathcal{S}_{\mathcal{M}_D} = \{\mathtt{A}, \mathtt{B}\}$ as a result of the DELETE$(\mathcal{M}_1, 1, 2)$ operation. This particular type of collapse behavior for *unbounded* automata models can occur for any *subtractive* string operation.



(a) $\mathcal{M}_1$ *unbounded* automata model



(b) $\mathcal{M}_R$ *unbounded* automata model resulting from Replace operation

Figure 3.5: Replace($\mathcal{M}_1, \texttt{A}, \texttt{B}$) example for *unbounded* automata model

The example shown in Figure 3.5 demonstrates how the collapse problem can occur for *unbounded* automata models due to *substitutive* string operations. This example presents the string operation Replace($str_1, char_1, char_2$) simulated for the *unbounded* automata model $\mathcal{M}_1$ representing the string argument str_1 and the symbols A and B as the symbol arguments $char_1$ and $char_2$. Figure 3.5a shows the diagram corresponding to $\mathcal{M}_1$ such that $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\texttt{A}, \texttt{B}\}$, a maximum length $k_{\mathcal{M}_1} = 2$, and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$. Figure 3.4b shows the diagram of the *unbounded* automata model $\mathcal{M}_R$ which is returned from the Replace($\mathcal{M}_1, \texttt{A}, \texttt{B}$) operation. This automata model $\mathcal{M}_R$ has the same maximum length as $\mathcal{M}_1$, $k_{\mathcal{M}_R} = 2$, since the symbols to be replaced (A) and the replacing symbols (B) are both single symbols. However, the alphabet of the resulting model only contains a single symbol $\Sigma_{\mathcal{M}_R} = \{\texttt{B}\}$ since all A symbols were replaced in the operation. The collapse problem can be seen when comparing the model count of the original automata model $MC(\mathcal{M}_1) = 7$ to the model count of the resulting automata model $MC(\mathcal{M}_R) = 3$. Due to the `Replace` operation, both the A and B string values in $\mathcal{S}_{\mathcal{M}_1}$ correlate to the B string value in $\mathcal{S}_{\mathcal{M}_R}$. Similarly, each of the string values AA, AB, BA, and BB in $\mathcal{S}_{\mathcal{M}_1}$ correlate to the BB string value in $\mathcal{S}_{\mathcal{M}_R}$. The empty string value ε in $\mathcal{S}_{\mathcal{M}_1}$ correlates to the empty string value in the result solution set $\mathcal{S}_{\mathcal{M}_R}$. Thus, the original solution set values $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$ collapse into the solution set values $\mathcal{S}_{\mathcal{M}_R} = \{\varepsilon, \texttt{B}, \texttt{BB}\}$ as a result of the Replace($\mathcal{M}_1, \texttt{A}, \texttt{B}$) operation.

This type of collapse behavior for *unbounded* automata models can occur for any *substitutive* string operations.

3.2 Bounded Automata Model

$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[r, "\texttt{A-B}"] & q_1 \arrow[r, "\texttt{A-B}"] & q_2
\end{tikzcd}$$

Figure 3.6: Bounded Automata Model:
$\Sigma = \{\texttt{A}, \texttt{B}\}$, $k = 2$

The *bounded* automata model was created to solve the issue of model over-approximation as described in Section 3.1.2 for the *unbounded* automata model. Figure 3.6 is an example of a *bounded* automata model with an alphabet $\Sigma = \{\texttt{A}, \texttt{B}\}$ and a length value $k = 2$. The *bounded* automata model consists of a FSA, an alphabet, and a length. The length value for this model is used to bound the length of the FSA to ensure that no cycles can exist within FSA, this allows the *bounded* model to avoid the problem of tracking the maximum length value external to the FSA which caused the over-approximation for the *unbounded* model.

```
procedure CreateBoundedFSA($\mathcal{A}_{init}, k, \Sigma$)
    $Q \leftarrow F \leftarrow \{q_0\},\ \delta \leftarrow \varnothing$
    $\mathcal{A} \leftarrow \langle Q, \Sigma, \delta, q_0, F \rangle$
    $q \leftarrow q_0$
    for $i \leftarrow 0$ to $k$ do
        $Q \leftarrow Q \cup \{q_n\},\ F \leftarrow F \cup \{q_n\}$
        for $\alpha \in \Sigma$ do
            $\delta(q, \alpha) \rightarrow q_n$
        end for
        $q \leftarrow q_n$
    end for
    $\mathcal{A}_r \leftarrow \mathcal{A}_{init} \cap \mathcal{A}$
    return $\mathcal{A}_r$
end procedure
```

Figure 3.7: Creation algorithm for *bounded* automata model

The FSA creation process for a *bounded* automata model begins just as the *unbounded* process, creating an initial FSA $\mathcal{A}_{init}$ as one of the five initial FSAs described in Section 2.4.2: *empty*, *empty string*, *literal*, *simple unknown*, and *complex unknown*. While only the *unknown* string types have a corresponding FSA which can contain cycles and would therefore need to be bounded by the length value k, the bounding process can still be performed for the other string type FSAs without issue. The bounding process is shown in Figure 3.7 where the initial FSA $\mathcal{A}_{init}$, the bounding length k, and the alphabet Σ are required parameters. Lines 2 and 3 initialize the bounding FSA $\mathcal{A}$ with only the accepting start state q_0 which is then stored as the current state q on line 4. The loop on lines 5-11 is used to create the bounding FSA which accepts all strings up to length k in the alphabet Σ. Line 6 creates a new state q_n and adds this state to the FSA as an accepting state. The loop on lines 7-9 creates a transition from the current state q to the new state q_n for every symbol $\alpha \in \Sigma$. On line 10 the current state q_b is updated as the new state q_n for the next iteration of the loop. After the loop is completed, the FSA $\mathcal{A}$ is now the desired bounding FSA accepting all strings up to the length k. Line 12 intersects the initial FSA $\mathcal{A}_{init}$ and the bounding FSA $\mathcal{A}$ to produce the desired bounded initial FSA $\mathcal{A}_r$. This FSA $\mathcal{A}_r$ has no cycles and accepts the same language as the initial FSA $\mathcal{A}_{init}$ such that $\mathcal{L}(\mathcal{A}_i) = \mathcal{L}(\mathcal{A}_r)$ for all strings with lengths less than or equal to the bounding length k. This bounded FSA $\mathcal{A}_r$ is added to the bounding length k and the alphabet Σ to complete the construction of the new *bounded* automata model.

3.2.1 Model Counting

Figure 3.8 shows the model counting algorithms for the *bounded* automata models where the coordinating algorithm is shown in Figure 3.8a and the recursive algorithm

```
procedure CountBounded($\mathcal{M}$)
    $\mathcal{A} \leftarrow \mathcal{A} \in \mathcal{M}$
    $q_0 \leftarrow q_0 \in \mathcal{A}$
    $mc_r \leftarrow$ MCBounded($q_0$)
    if $q_0 \in F$ then
        $mc_r \leftarrow mc_r + 1$
    end if
    return $mc_r$
end procedure
```

(a) Coordinating algorithm

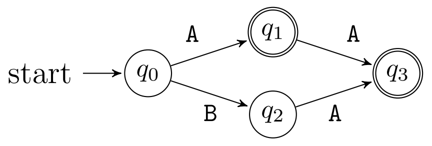
```
procedure MCBounded($q$)
    $mc_r \leftarrow 0$
    for $q_d \in \{q_e \mid \delta(q, \alpha) \rightarrow q_e\}$ do
        if $q_d \in F$ then
            $mc_r \leftarrow mc_r + 1$
        end if
        $mc_r \leftarrow mc_r +$ MCBounded($q_d$)
    end for
    return $mc_r$
end procedure
```

(b) Recursive model counting algorithm

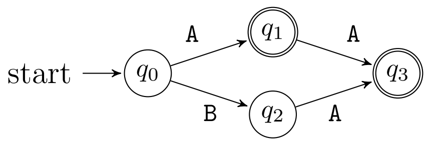
Figure 3.8: Model counting algorithms for *bounded* automata models

is shown in Figure 3.8b. This version of the model counting algorithm works nearly the same as the *unbounded* version except that the *bounded* version does not include a counter parameter to the recursive algorithm since a *bounded* FSA cannot contain cycles.

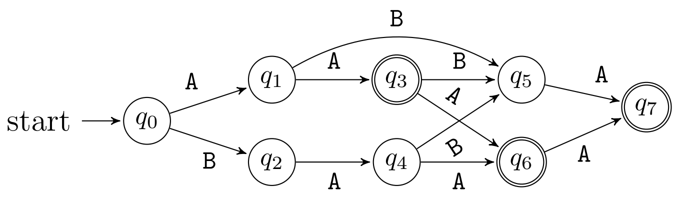
3.2.2 Over-Approximation



(a) $\mathcal{M}_1$ *bounded* automata model



(b) $\mathcal{M}_2$ *bounded* automata model



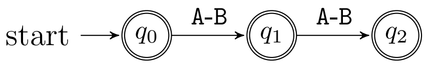
(c) $\mathcal{M}_C$ *bounded* automata model resulting from Concatenation operation

Figure 3.9: Concatenation($\mathcal{M}_1, \mathcal{M}_2$) example for *bounded* automata model

The example shown in Figure 3.9 demonstrates how over-approximation due to model structure is prevented for *bounded* automata models. This example presents the string operation $\textsc{Concatenation}(str_1, str_2)$ simulated for *bounded* automata models $\mathcal{M}_1$ and $\mathcal{M}_2$ representing the string variables str_1 and str_2 respectively. Figures 3.9a and 3.9b show the diagrams corresponding to $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $\mathcal{M}_1$ and $\mathcal{M}_2$ both have an alphabet $\Sigma_{\mathcal{M}_1} = \Sigma_{\mathcal{M}_2} = \{\mathtt{A}, \mathtt{B}\}$ and bounding length $k_{\mathcal{M}_1} = k_{\mathcal{M}_1} = 2$ and both represent the solution set $\mathcal{S}_{\mathcal{M}_1} = \mathcal{S}_{\mathcal{M}_2} = \{\mathtt{A}, \mathtt{AA}, \mathtt{BA}\}$. The string operation and automata model parameters are identical to those discussed in Section 3.1.2 for the *unbounded* automata model where over-approximation occurred. Figure 3.9c shows the diagram of the *bounded* automata model $\mathcal{M}_C$ which is the result of the $\textsc{Concatenation}$ operation and has the same alphabets as $\mathcal{M}_1$ and $\mathcal{M}_2$, that is $\Sigma_{\mathcal{M}_C} = \{\mathtt{A}, \mathtt{B}\}$. This resulting model $\mathcal{M}_C$ has a solution set $\mathcal{S}_{\mathcal{M}_\rfloor} = \{\mathtt{AA}, \mathtt{AAA}, \mathtt{ABA}, \mathtt{BAA}, \mathtt{AAAA}, \mathtt{AABA}, \mathtt{BAAA}, \mathtt{BABA}\}$ which matches the expected solution set $\mathcal{S}_e = \{\mathtt{AA}, \mathtt{AAA}, \mathtt{ABA}, \mathtt{BAA}, \mathtt{AAAA}, \mathtt{AABA}, \mathtt{BAAA}, \mathtt{BABA}\}$ resulting from the $\textsc{Concatenation}(str_1, str_2)$ operation the alphabet $\Sigma_{\mathcal{M}_1} = \Sigma_{\mathcal{M}_2} = \{\mathtt{A}, \mathtt{B}\}$ and bounding length $k_{\mathcal{M}_1} = k_{\mathcal{M}_1} = 2$. Since the expected and actual solution sets are equivalent, the expected and actual model counts are equal, i.e. $MC(\mathcal{S}_e) = MC(\mathcal{M}_\rfloor) = 8$. Because the length is incorporated into the structure of the FSA by bounding the FSA in a *bounded* automata model, the over-approximation due to the model which can occur for *unbounded* automata models cannot occur for *bounded* automata models.

3.2.3 Collapse

While the *bounded* automata model prevents the over-approximation experienced by *unbounded* models, the collapse problems due to *subtractive* and *substitutive* operations still can occur for *bounded* automata models.


(a) $\mathcal{M}_1$ *bounded* automata model

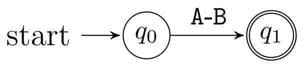
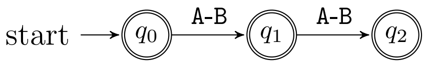

(b) $\mathcal{M}_D$ *bounded* automata model resulting from DELETE operation

Figure 3.10: DELETE($\mathcal{M}_1, 1, 2$) example for *bounded* automata model

The example shown in Figure 3.10 demonstrates how the collapse problem can occur for *bounded* automata models due to *subtractive* string operations. This example presents the string operation DELETE(str_1, int_s, int_e) simulated for the *bounded* automata model $\mathcal{M}_1$ and the start and end indices 1 and 2. Figure 3.10a shows the diagram corresponding to $\mathcal{M}_1$ such that $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\texttt{A}, \texttt{B}\}$ and maximum length $k = 2$ and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$. Figure 3.10b shows the diagram of the *bounded* automata model $\mathcal{M}_D$ which is returned from the DELETE($\mathcal{M}_1, 1, 2$) operation and has the same alphabet as $\mathcal{M}_1$, $\Sigma_{\mathcal{M}_D} = \{\texttt{A}, \texttt{B}\}$. As with the *unbounded* DELETE operation in Section 3.1.3, the solution set of the *bounded* automata model produced by simulating the DELETE operation $\mathcal{S}_{\mathcal{M}_D} = \{\texttt{A}, \texttt{B}\}$ and the corresponding model count $MC(\mathcal{M}_D) = 2$ does not match the expected model count of 6 (the empty string should not be represented in the model after the operation). This example demonstrates that *bounded* automata models do not prevent collapses due to *subtractive* string operations.


(a) $\mathcal{M}_1$ *bounded* automata model

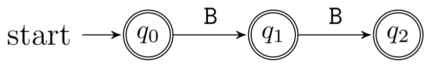

(b) $\mathcal{M}_R$ *bounded* automata model resulting from REPLACE operation

Figure 3.11: REPLACE($\mathcal{M}_1, \texttt{A}, \texttt{B}$) example for *bounded* automata model

The example shown in Figure 3.11 demonstrates how the collapse problem can occur for *bounded* automata models due to *substitutive* string operations. This example presents the string operation Replace($str_1, char_1, char_2$) simulated for the *bounded* automata model $\mathcal{M}_1$ representing the string argument str_1 and the symbols A and B as the symbol arguments $char_1$ and $char_2$. Figure 3.11a shows the diagram corresponding to $\mathcal{M}_1$ such that $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\texttt{A}, \texttt{B}\}$ and a bounding length $k_{\mathcal{M}_1} = 2$ and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$. Figure 3.11b shows the diagram of the *bounded* automata model $\mathcal{M}_R$ which is returned from the Delete($\mathcal{M}_1, 1, 2$) operation. As with the *unbounded* Replace operation in Section 3.1.3, the solution set for the *bounded* automata model produced by simulating the Replace operation $\mathcal{S}_{\mathcal{M}_\nabla} = \{\varepsilon, \texttt{B}, \texttt{B}\}$ and the corresponding model count $MC(\mathcal{M}_R) = 3$ does not match the expected model count of 7. As with the Delete example, this Replace example demonstrates that the *bounded* automata model does not prevent collapses due to *substitutive* string operations.

3.3 Aggregate Bounded Automata Model

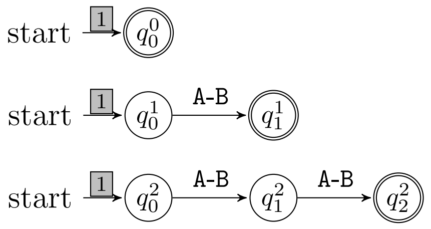


Figure 3.12: Aggregate Bounded Automata Model:
$\Sigma = \{\texttt{A}, \texttt{B}\}$, $k = 2$

The *aggregate bounded* automata model, shortened henceforth as *aggregate* automata model, was created to prevent collapse problems which occur due to *subtractive* string operations as described in Sections 3.1.3 and 3.2.3. Figure 3.12 is an example of an *aggregate* automata model with an alphabet $\Sigma = \{\texttt{A}, \texttt{B}\}$ and a length value $k = 2$. The *aggregate* model includes a sequence of FSAs, a sequence of positive integers called factors, an alphabet, and a length. The factors $(f_n)_{n=0}^k$ are updated in string operation simulation algorithms and are used as a multiplying factors when computing the model count of an *aggregate* model. As with the *bounded* model, the *aggregate* automata model uses the length value to bound the length of FSAs. However, unlike the *bounded* automata model, the *aggregate* automata model can contain more than one FSA. These multiple FSAs are collected into a sequence such that each FSA in the sequence correlates to a FSA which accepts a language of strings with precisely the length of the corresponding index in the sequence, i.e. $(\mathcal{A}_n)_{n=0}^k$ where $\mathcal{L}(\mathcal{A}_n)$ only contains strings of length n. In the example *aggregate* automata model, the FSA with start state q_0^0 is at index 0 in the FSA sequence and only accepts the empty string ε which is a string with length 0. Similarly, the FSA with start state q_0^1 is at index 1 in the FSA sequence and only accepts strings of length 1. The process of FSA creation is similar to that used for *unbounded* automata models, starting with the creation of an initial FSA $\mathcal{A}_{init}$ as described in Section 2.4.2.

```
1: procedure FSASequence($\mathcal{A}_{init}, k, \Sigma$)
2:     for $\mathcal{A}_r^i \in (\mathcal{A}_r^n)_{n=0}^k,\ f_r^i \in (f_r^n)_{n=0}^k$ do
3:         $\mathcal{A}_b \leftarrow$ LengthFSA($i, \Sigma$)
4:         $\mathcal{A}_r^i \leftarrow \mathcal{A}_{init} \cap \mathcal{A}_b$
5:         $f_r^i \leftarrow 1$
6:     end for
7:     return $(\mathcal{A}_r^n)_{n=0}^k,\ (f_r^n)_{n=0}^k$
8: end procedure
```

(a) Creation algorithm for FSA sequence

```
1: procedure LengthFSA($k, \Sigma$)
2:     $Q \leftarrow F \leftarrow \{q_0\},\ \delta \leftarrow \varnothing$
3:     $\mathcal{A} \leftarrow \langle Q, \Sigma, \delta, q_0, F \rangle$
4:     $q \leftarrow q_0$
5:     for $i \leftarrow 0$ to $k$ do
6:         $Q \leftarrow Q \cup \{q_n\}$
7:         for $\alpha \in \Sigma$ do
8:             $\delta(q, \alpha) \rightarrow q_n$
9:         end for
10:        $q \leftarrow q_n$
11:    end for
12:    $F \leftarrow F \cup \{q_n\}$
13:    return $\mathcal{A}$
14: end procedure
```

(b) Creation algorithm for length bounded FSA

Figure 3.13: Algorithms to create FSA sequence for *aggregate* automata model

Figure 3.13 contains the algorithms used to create the FSA and factor sequences for an new *aggregate* automata model. The sequence creation process begins using the FSASequence algorithm shown in Figure 3.13a which requires an initial FSA $\mathcal{A}_{init}$, an initial bounding length k, and an alphabet Σ as parameters. Lines 2-6 of the FSASequence algorithm initializes both the return FSA sequence $(\mathcal{A}_r^n)_{n=0}^k$ and the return factor sequence $(f_r^n)_{n=0}^k$ and then loops from 0 until k. Line 3 of the FSASequence algorithm uses the LengthFSA algorithm create a bounding FSA which only accepts strings of length i. The LengthFSA algorithm is shown in Figure 3.13b and accepts the length integer k and the alphabet Σ as parameters. Lines 2-4 of LengthFSA initialize the bounding FSA $\mathcal{A}$ with only the non-accepting start state q_0 which is then stored as the current state q. The loop on lines 5-11 of LengthFSA is used to create the bounding FSA which accepts only of length k in the alphabet Σ. Line 6 of LengthFSA creates a new state q_n and adds this state to the FSA set of states $Q \in \mathcal{A}$. The loop on lines 7-9 of LengthFSA creates a transition from the current state q to the new state q_n for every symbol $\alpha \in \Sigma$. On line 10 of LengthFSA the current state q_b is updated as the new state q_n for the next iteration of the loop. After the loop is completed, line 12 of LengthFSA adds the last newly added state q_n to the set of accepting states $F \in \mathcal{A}$. Line 13 of LengthFSA returns the FSA $\mathcal{A}$ which is now the desired bounding FSA accepting all strings of length k as $\mathcal{A}_b$ on Line 3 of FSASequence. Line 4 of FSASequence intersects the initial FSA $\mathcal{A}_{init}$ and the bounding FSA $\mathcal{A}_b$ producing the intersected FSA $\mathcal{A}_r^i$ at index i in the FSA sequence $(\mathcal{A}_r^n)_{n=0}^k$. Line 5 of FSASequence sets the factor at index i to 1 for the factor sequence $(f_r^n)_{n=0}^k$. The loop on lines 2-6 of FSASequence continues until all the FSAs in the FSA sequence $(\mathcal{A}_r^n)_{n=0}^k$ are created and all the factors in the factor sequence $(f_r^n)_{n=0}^k$ are set to 1. Finally, line 7 of FSASequence returns both the completed FSA and factor sequences. These FSA and factor sequences are then added to the bounding length k and the alphabet Σ to complete the creation of an *aggregate* automata model.

3.3.1 String Operations

```
1: procedure ABinaryOp(Op, $\mathcal{M}_1, \mathcal{M}_2, \Sigma$)
2:     $(\mathcal{A}_1^n)_{n=0}^k \in \mathcal{M}_1, (\mathcal{A}_2^n)_{n=0}^k \in \mathcal{M}_2$
3:     $\mathcal{A}_{U_2} \leftarrow \bigcup_{n=0}^k \mathcal{A}_2^n$
4:     for $\mathcal{A}_{op}^i \in (\mathcal{A}_{op}^n)_{n=0}^k, f_{op}^i \in (f_{op}^n)_{n=0}^k$ do
5:         $\mathcal{A}_{op}^i, f_{op}^i \leftarrow$ Op$(\mathcal{A}_1^i, f_1^n, \mathcal{A}_{U_2})$
6:     end for
7:     $\mathcal{M}_{op} \leftarrow \langle(\mathcal{A}_{op}^n)_{n=0}^k, (f_{op}^n)_{n=0}^k, k, \Sigma\rangle$
8:     return $\mathcal{M}_{op}$
9: end procedure
```

(a) String operation and predicate coordination algorithm for two *aggregate* automata model arguments

```
1: procedure RestructFSAs($\mathcal{M}_{op}$)
2:     $(\mathcal{A}_{op}^n)_{n=0}^k \in \mathcal{M}, (f_{op}^n)_{n=0}^k \in \mathcal{M}, \Sigma \in \mathcal{M}$
3:     $k_r \leftarrow$ GetMaxLength$((\mathcal{A}_n)_{n=0}^k)$
4:     for $\mathcal{A}_r^i \in (\mathcal{A}_r^m)_{m=0}^{k_r}, f_r^i \in (f_r^m)_{m=0}^{k_r}$ do
5:         $f_r^i \leftarrow 0$
6:         $\mathcal{A}_b \leftarrow$ LengthFSA$(i, \Sigma)$
7:         for $\mathcal{A}_I^j \in (\mathcal{A}_t^n)_{n=0}^k$ do
8:             $\mathcal{A}_I^j \leftarrow \mathcal{A}_{op}^j \cap \mathcal{A}_b$
9:             if $\mathcal{L}(\mathcal{A}_I^j) \neq \varnothing$ then
10:                $f_r^i \leftarrow f_r^i + f_i$
11:            end if
12:        end for
13:        $\mathcal{A}_r^i \leftarrow \bigcup_{n=0}^{k_r} \mathcal{A}_I^n$
14:    end for
15:    $\mathcal{M}_r \leftarrow \langle(\mathcal{A}_r^n)_{n=0}^{k_r}, (f_r^n)_{n=0}^{k_r}, k_r, \Sigma\rangle$
16:    return $\mathcal{M}_r$
17: end procedure
```

(b) Restructuring algorithm for *aggregate* automata models

Figure 3.14: *Aggregate* automata model utility algorithms for string operations and predicates

Since the *aggregate* automata model contains a sequence of FSAs instead of a single FSA, the simulation of string operations and predicates with more than one *aggregate* automata model argument must now manage how the operation or predicate will be simulated using two or more FSA sequences. The strategy we use in our work for these multiple FSA sequences is to merge each non-primary FSA sequence into a single FSA which can be used to simulate a string operation or predicate. Figure 3.14a is an algorithm for binary operations between two *aggregate* automata models which details this merging of non-primary FSA sequences and the subsequence string operation or predicate simulation. The ABinaryOp algorithm takes an operation or predicate simulation algorithm Op as a parameter in addition to two automata model parameters $\mathcal{M}_1$ and $\mathcal{M}_2$ and the alphabet Σ. Line 2 of the algorithm extracts the FSA sequences for both the $\mathcal{M}_1$ and $\mathcal{M}_2$ models for later use in the algorithm. Line 3 merges all the FSAs in the FSA sequence for $\mathcal{M}_2$ using the Union operation and producing the merged FSA $\mathcal{A}_{U_2}$. Line 4 initializes the operation or predicate result FSA sequence $(\mathcal{A}^n_{op})^k_{n=0}$ and operation result factor sequence $(f^n_{op})^k_{n=0}$ before the loop on lines 4-6. Line 5 of this loop performs the Op operation or predicate simulation algorithm using the FSA and factor at index i of their respective sequences for $\mathcal{M}_1$ and the merged FSA $\mathcal{A}_{U_2}$. The FSA $\mathcal{A}^i_{op}$ and factor f^i_{op} produced by the Op algorithm are set as the respective FSA and factor at index i in the result sequences $(\mathcal{A}^n_{op})^k_{n=0}$ and $(f^n_{op})^k_{n=0}$. Lines 7 and 8 finish the algorithm by creating the operation or predicate result *aggregate* automata model $\mathcal{M}_{op}$ on line 7 and returning that model from the algorithm on line 8.

Unfortunately, the Union of non-primary FSA sequences for string operations and predicates can result in an *aggregate* automata model which is vulnerable to *subtractive* collapses due to produced FSA sequence containing individual FSAs which accept strings of different lengths. Since the *aggregate* model was created specifically for the prevention of *subtractive* collapses, this continuing vulnerability must be removed by restructuring the FSA sequence produced by the simulation of the string operation or predicate $(\mathcal{A}^n_{op})^k_{n=0}$. Figure 3.14b shows the RestructFSAs algorithm which performs this restructuring and accepts the *aggregate* automata model $\mathcal{M}_{op}$ as its only parameter. Line 2 of the algorithm extracts the FSA and factor sequences $(\mathcal{A}^n_{op})^k_{n=0}$ and $(f^n_{op})^k_{n=0}$ and the alphabet Σ from the model $\mathcal{M}_{op}$. Line 3 uses the

GetMaxLength algorithm to determine the length k_r longest accepted string in the FSA sequence $(\mathcal{A}_n)_{n=0}^{k}$. The loop on lines 4-14 begins by initializing the return FSA and factor sequences $(\mathcal{A}_r^m)_{m=0}^{k_r}$ and $(f_r^m)_{m=0}^{k_r}$ before iterating from 0 to k_r. Line 5 within the loop sets the factor f_r^i of the sequence $(f_r^m)_{m=0}^{k_r}$ at index i to 0. On line 6, the algorithm LengthFSA seen in Figure 3.13b is used to create bounding FSA $\mathcal{A}_b$ accepting only strings of length i. The loop on lines 7-12 begins by initializing the FSA sequence $(\mathcal{A}_I^n)_{n=0}^{k}$ before iterating from 0 to k. Line 8 utlizes the bounding FSA created on line 6 $\mathcal{A}_b$ to bound the FSA $\mathcal{A}_{op}^j$ using the Intersection automata operation to produce the resulting FSA $\mathcal{A}_I^j$. Lines 9-11 are an if condition which is used to increment the factor f_r^i. After the completion of the loop from lines 7-12, all the intersected FSAs in the sequence $(\mathcal{A}_I^n)_{n=0}^{k}$ are merged using the Union operation on line 13 to produce a single FSA $\mathcal{A}_r^i$ which accepts all strings with a length i accepted by the sequence $(\mathcal{A}_{op}^n)_{n=0}^{k}$. Due to the if condition on lines 9-11, the factor f_r^i at the time line 13 is performed will equal the sum of all factors from the sequence $(f_{op}^n)_{n=0}^{k}$ which correspond to FSAs in the sequence $(\mathcal{A}_{op}^n)_{n=0}^{k}$ accepting strings of length i. This summation of factors is done to preserve the adjustments made to factors due to string operations and predicates expressed in the $(f_{op}^n)_{n=0}^{k}$ sequence. Line 15 of the RestructFSAs algorithm creates a new *aggregate* automata model $\mathcal{M}_r$ from the sequence of merged FSAs $(\mathcal{A}_r^n)_{n=0}^{k_r}$, the sequence of summed factors $(f_r^n)_{n=0}^{k_r}$, the newly computed bounding length k_r, and the alphabet Σ. Line 16 finishes the algorithm by returning this newly created $\mathcal{M}_r$ model.

While this process of restructuring the FSA sequence does ensure that *subtractive* collapses do not occur, the additional performance costs are incurred due to the restructuring. Specifically, more time is required to complete operation and predicate simulations, additional temporary space is required to create the sequence of sequences, and the restructured sequence of FSAs becomes vulnerable to collapse due to the UNION of the bounded FSAs (this type of collapse is discussed in more detail in Section 3.3.4).

3.3.2 Model Counting

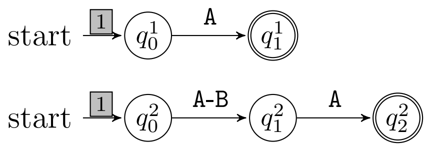
```
 1: procedure CountAggregate($\mathcal{M}$)
 2:     $mc_r \leftarrow 0$
 3:     $(\mathcal{A}_n)_{n=0}^{k} \in \mathcal{M},\ (f_n)_{n=0}^{k} \in \mathcal{M}$
 4:     for $\langle Q_i, \Sigma_i, \delta_i, q_0^i, F_i \rangle = \mathcal{A}_i \in (\mathcal{A}_n)_{n=0}^{k}$ do
 5:         $mc \leftarrow$ MCBounded($q_0^i$)
 6:         if $q_0^i \in F$ then
 7:             $mc \leftarrow mc + 1$
 8:         end if
 9:         $mc_r \leftarrow mc_r + (mc \times f_i)$
10:     end for
11:     return $mc_r$
12: end procedure
```

Figure 3.15: Coordinating algorithm *aggregate* automata models

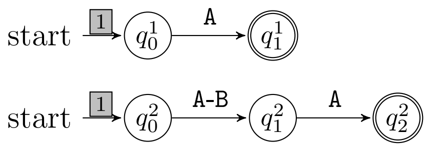
The model counting algorithm for the *aggregate* automata model is shown in Figure 3.15 utilizing the recursive MCBOUNDED algorithm shown previously in Figure 3.8b and discussed in Section 3.2.1. The coordinating algorithm begins by initializing the return model count mc_r as 0. Next, the FSA sequence and the factor sequence are retrieved from the *aggregate* automata model on Lines 3 and 4. Lines 5-11 contain an iteration through each FSA quintuple $\langle Q_i, \Sigma_i, \delta_i, q_0^i, F_i \rangle = \mathcal{A}_i \in \mathcal{A}_i$ in the *aggregate* model $\mathcal{M}$. For each of these FSAs, the model count is determined with a call to the recursive algorithm MCBOUNDED and stored as the current model count mc as shown on Line 6. Lines 7-9 account for an empty string with an if condition which checks if the current start state q_0^i is accepting and increments the current model count mc. Line 9 is where the return model count mc_r is updated by the addition of the current model count mc multiplied by the factor f_i corresponding to the current

FSA $\mathcal{A}_i$. After iterating through all the FSAs in the *aggregate* automata model, the return model count mc_r is returned from the algorithm.

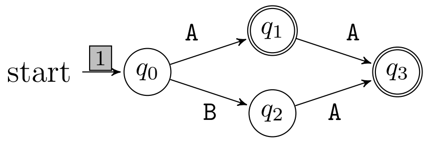
3.3.3 Over-Approximation



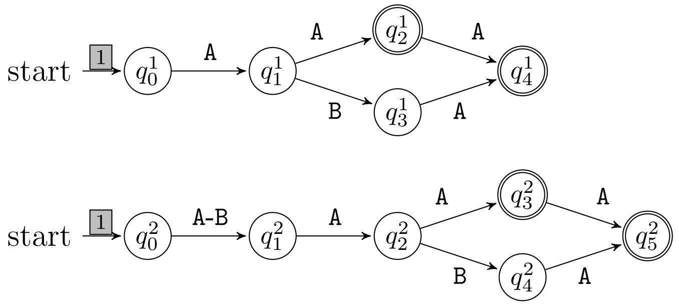
(a) $\mathcal{M}_1$ *aggregate* automata model



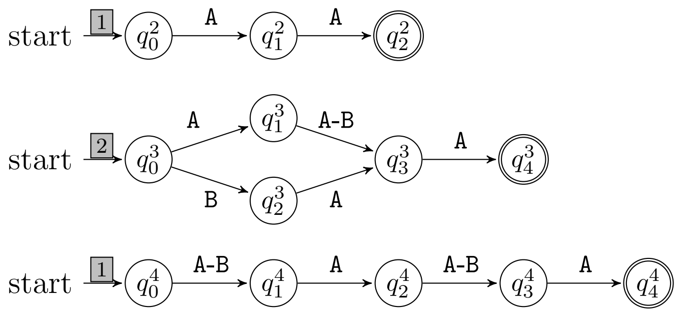
(b) $\mathcal{M}_2$ *aggregate* automata model



(c) $\mathcal{M}_{U_2}$ *aggregate* automata model



(d) $\mathcal{M}_C$ *aggregate* automata model resulting CONCATENATION operation



(e) $\mathcal{M}_R$ *aggregate* automata model after RESTRUCTFSAS

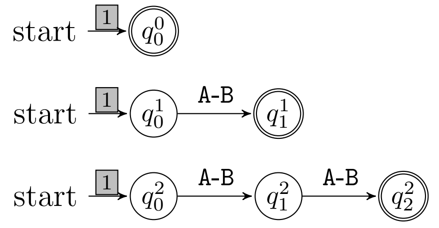
Figure 3.16: CONCATENATION($\mathcal{M}_1, \mathcal{M}_2$) example for *aggregate* automata model

The example shown in Figure 3.16 demonstrates how over-approximation due to model structure is prevented for *aggregate* automata models. This example presents the string operation CONCATENATION(str_1, str_2) simulated for *aggregate* automata models $\mathcal{M}_1$ and $\mathcal{M}_2$ representing the string variables str_1 and str_2 respectively. Figures 3.16a and 3.16b show the diagrams corresponding to $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $\mathcal{M}_1$ and $\mathcal{M}_2$ both have an alphabet $\Sigma_{\mathcal{M}_1} = \Sigma_{\mathcal{M}_2} = \{\mathtt{A}, \mathtt{B}\}$ and bounding length $k = 2$ and both represent the solution set $\mathcal{S}_{\mathcal{M}_1} = \mathcal{S}_{\mathcal{M}_2} = \{\mathtt{A}, \mathtt{AA}, \mathtt{BA}\}$. Figure 3.16c shows the diagram of the *aggregate* automata model $\mathcal{M}_{U_2}$ which is produced on line 3 of the ABinaryOp procedure from the FSA sequence in the $\mathcal{M}_2$ model. Figure 3.16d shows the *aggregate* model $\mathcal{M}_C$ produced by simulating the Concatenation operation for $\mathcal{M}_1$ and $\mathcal{M}_{U_2}$. Figure 3.16e shows the *aggregate* model $\mathcal{M}_R$ which was created by the RestructFSAs algorithm performed on the $\mathcal{M}_C$ model. While there are 2 additional intermediary steps in the *aggregate* version of this Concatenation operation compared to the *bounded* version from Section 3.2.2, the *aggregate* automata model also does not over-approximate the solution set. The 8 values in the *aggregate* model's solution set match the expected solution set of 8, i.e. $\mathcal{S}_e = \mathcal{S}_{\mathcal{M}_R} = \{\mathtt{AA}, \mathtt{AAA}, \mathtt{ABA}, \mathtt{BAA}, \mathtt{AAAA}, \mathtt{AABA}, \mathtt{BAAA}, \mathtt{BABA}\}$. This provides an example of how over-approximation of a solution set due to an automata model is prevented in *aggregate* automata models.

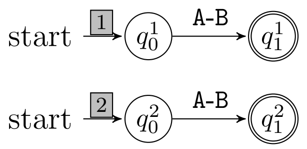
While *aggregate* automata models avoid over-approximation of the solution set, the use of factors in the calculation of the model count as seen in the CountAggregate algorithm in Figure 3.15 can result in over-approximation of the model count calculated for the model. The previous example in Figure 3.16 shows how this type of model count over-approximation can occur. The *aggregate* automata models $\mathcal{M}_1$ and $\mathcal{M}_2$, shown in figures 3.16a and 3.16a respectively, both have a model count $MC(\mathcal{M}_1) = MC(\mathcal{M}_2) = 3$ which should produce an *aggregate* model from the Concatenation operation with a model count of 9. However, the model count of the restructured *aggregate* model $MC(\mathcal{M}_R) = 11$. This over-approximation occurs due to the summation of the factors on lines 9-11 of the RestructFSAs algorithm in Figure 3.14b. When this condition on lines 9-11 occurs with an i value of 3, the factors from $\mathcal{A}_C^1$ and $\mathcal{A}_C^2$ FSAs are summed to produce a factor of 2 for the $\mathcal{A}_R^3$. This new factor of 2 is used to reflect the merging both of the `AAA` string value from the $\mathcal{A}_C^1$ FSA and the `AAA` string value from the $\mathcal{A}_C^2$ FSA into the $\mathcal{A}_R^3$ FSA. However, since the the $\mathcal{A}_R^3$ FSA also represents `ABA` string value from $\mathcal{A}_C^1$ but not $\mathcal{A}_C^2$ and the `BAA` string value from $\mathcal{A}_C^2$ and not $\mathcal{A}_C^1$, both the `ABA` and `BAA` string values are over-approximated in the $\mathcal{A}_R^3$ FSA. It is due to this merging of FSAs and the corresponding summation of factors which results in model count over-approximation for *aggregate* automata models.

3.3.4 Collapse

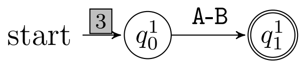
While the *aggregate* automata model does improve upon the *bounded* model by preventing collapses due to *subtractive* string operations, *aggregate* automata models are still susceptible to collapses from *substitutive* string operations.



(a) $\mathcal{M}_1$ *aggregate* automata model



(b) $\mathcal{M}_D$ *aggregate* automata model resulting from Delete operation

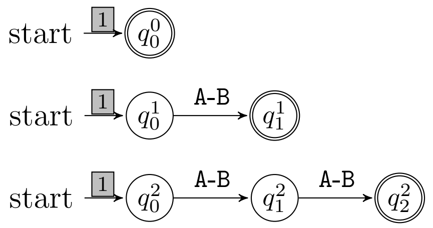


(c) $\mathcal{M}_R$ *aggregate* automata model after restructuring the FSA sequence

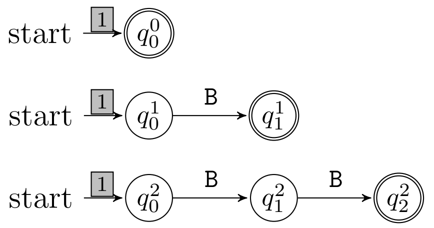
Figure 3.17: Delete($\mathcal{M}_1, 1, 2$) example for *aggregate* automata model

The example shown in Figure 3.17 demonstrates how the collapse problem is prevented for *aggregate* automata models due to *subtractive* string operations. This example presents the string operation Delete(str_1, int_s, int_e) simulated for the *aggregate* automata model $\mathcal{M}_1$ and the start and end indices 1 and 2. Figure 3.17a shows the diagram corresponding to $\mathcal{M}_1$ such that $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\texttt{A}, \texttt{B}\}$ and maximum length $k = 2$ and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$. Figure 3.17b shows the diagram of the *aggregate* automata model $\mathcal{M}_D$ which is returned from the DELETE$(\mathcal{M}_1, 1, 2)$ operation and has the same alphabet as $\mathcal{M}_1$, $\Sigma_{\mathcal{M}_D} = \{\texttt{A}, \texttt{B}\}$. The prevention of the collapse due to the DELETE operation can be seen by comparing the $\mathcal{M}_1$ model count $MC(\mathcal{M}_1) = 7$ and the $\mathcal{M}_D$ model count $MC(\mathcal{M}_D) = 6$ which is the expected outcome due to the empty string ε not being a valid argument for the DELETE operation. This prevention of *subtractive* collapse is accomplished first by the DELETE algorithm which preserves the 4 distinct string values AA, AB, BA, and BB from the FSA $\mathcal{A}_1^2$ as the 2 distinct string values A and B accepted by the FSA $\mathcal{A}_D^2$ and the factor $f_D^2 = 2$. The 6 expected distinct string values represented by $\mathcal{M}_D$ are maintained in $\mathcal{M}_R$ produced from the RESTRUCTFSAS algorithm with the model count $MC(\mathcal{M}_R) = 6$. This example demonstrates why the factor summation on lines 9-11 of the RESTRUCTFSAS algorithm is necessary to prevent *subtractive* collapses where the merging of the FSAs $\mathcal{A}_D^1$ and $\mathcal{A}_D^2$ requires the summation of their respective factors $f_D^1 = 1$ and $f_D^2 = 2$ to preserver the model count $MC(\mathcal{M}_R) = 6$ calculated from the 2 string values accepted by $\mathcal{A}_R^1$ multiplied by the factor $f_R^1 = 3$. It is this use of an FSA sequence and summation of factors which allows *aggregate* automata models to prevent collapse problems due to *subtractive* string operations.

The example shown in Figure 3.18 demonstrates how the collapse problem can occur for *aggregate* automata models due to *substitutive* string operations. This example presents the string operation REPLACE$(str_1, char_1, char_2)$ simulated for the *aggregate* automata model $\mathcal{M}_1$ representing the string argument str_1 and the symbols A and B as the symbol arguments $char_1$ and $char_2$. Figure 3.18a shows the diagram



(a) $\mathcal{M}_1$ *aggregate* automata model



(b) $\mathcal{M}_R$ *aggregate* automata model resulting from REPLACE operation

Figure 3.18: REPLACE($\mathcal{M}_1, \texttt{A}, \texttt{B}$) example for *aggregate* automata model

corresponding to $\mathcal{M}_1$ such that $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\texttt{A}, \texttt{B}\}$ and maximum length $k = 2$ and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$. Figure 3.18b shows the diagram of the *aggregate* automata model $\mathcal{M}_R$ which is returned from the REPLACE($\mathcal{M}_1, \texttt{A}, \texttt{B}$) operation and has the same alphabet as $\mathcal{M}_1$, $\Sigma_{\mathcal{M}_R} = \{\texttt{A}, \texttt{B}\}$. There is no need to use the RESTRUCTREFSAS procedure on $\mathcal{M}_R$ because *simple substitutive* operations do not alter the length of the strings accepted by FSAs which is the reason such a procedure is required. It is clear that the 7 distinct string values represented in $\mathcal{M}_1$ by the model count $MC(\mathcal{M}_1) = 7$ are not properly represented in the $\mathcal{M}_R$ model with the model count $MC(\mathcal{M}_R) = 3$. While the empty string ε is properly represented in both the $\mathcal{A}_1^0$ and $\mathcal{A}_R^0$ models, the A and B string values in $\mathcal{A}_1^1$ are only represented as the single value B in $\mathcal{A}_R^1$ and the AA, AB, BA, and BB string values in $\mathcal{A}_1^2$ are only represented by the single string value BB in $\mathcal{A}_R^2$. It is in this way that collapses due to *substitutive* string operations can occur for *aggregate* automata models.

3.4 Weighted Transition Automata Model

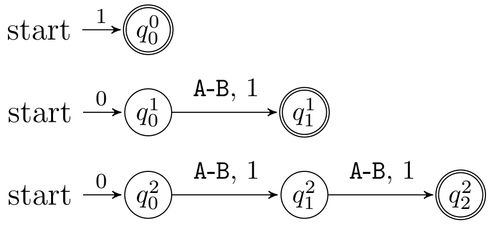


Figure 3.19: Weighted-Transition Aggregate Bounded Automata Model

The *weighted-transition aggregate* automata model, shortened henceforth as *weighted* automata model, was created primarily to prevent collapse problems which occur due to *substitutive* string operations as described in Sections 3.1.3, 3.2.3, and 3.3.4. The *weighted* automata model was also created with the intent of preventing the model count over-approximation and UNION collapses observed in *aggregate* automata models. Figure 3.19 is an example of a *weighted* automata model with an alphabet $\Sigma = \{\texttt{A}, \texttt{B}\}$ and a length value $k = 2$. As the example shows, the *weighted* automata model is nearly identical to the *aggregate* automata model due to the *weighted* model building upon the improvements made in the *aggregate* model for preventing *subtractive* collapse. In fact, there are only two differences between the two automata models: the *weighted* model uses a weighted-transition finite state automaton (WFSA) sequence instead of the FSA sequence used in the *aggregate* model and the lack of a factor sequence in the *weighted* model. The factor sequence of the *aggregate* model is incorporated as transition weights of the new WFSAs which prevents the model count over-approximation in the *weighted* model. Before detailing the creation process for a *weighted* automata model in Section 3.4.2, the new type of automaton, the WFSA, will be detailed in Section 3.4.1.

3.4.1 Weighted-Transition Finite State Automaton

A weighted-transition finite state automaton (WFSA) is a modification of FSA so that in addition to accepting or rejecting input strings, the actual number of identical input strings accepted can be determined. Due to this ability to quantify accepted input strings, the language accepted by the WFSA $\mathcal{W}$, called the language of the WFSA and denoted as $\mathcal{L}(\mathcal{W})$, is not a set of strings but is instead a multiset or bag of strings. Because the language of the WFSA is a multiset rather than a set of strings, a WFSA can accept and therefore represent multiple instances of the same input string. More formally, a weighted-transition finite state automaton $\mathcal{W}$ is defined as the 6-tuple $(Q, \Sigma, \delta, q_0, F, i)$ where

- Q is a finite set of states, $Q \neq \varnothing$
- Σ is a finite set of symbols called the alphabet, $\Sigma \neq \varnothing$
- δ is the transition function, $\delta : Q \times \Sigma \times \mathbb{Z} \to Q$
- q_0 is the start state, $q_0 \in Q$
- F is the set of accepting or final states, $F \subseteq Q$
- i_ε is an integer counting the number of empty strings represented by the start state q_0 if the start state is an accepting state $q_0 \in F$

There are two notable differences between the FSA quintuple and the WFSA 6-tuple: the addition of the empty string counter i_ε and the added positive integer argument in the transition function. The addition of the empty string counter allow a WFSA to represent more than a single empty string value which can be required due to either a *subtractive* or a *complex substitutive* string operation. For example, the string operation $\textsc{Delete}(str, 0, 1)$ could create one or more empty strings when deleting strings of length 1, but would only be represented in an FSA by a single accepting start state. Since a WFSA can represent more than a single empty string via the empty string counter i_ε, the Delete operation can be represented accurately. The transition function of a WFSA $\delta_{WFSA} : (q, \alpha, i_w) \rightarrow q_d$ differs from the transition function of a FSA $\delta_{WFSA} : (q, \alpha) \rightarrow q_d$ due to the addition of the i_w positive integer weight parameter. It is this weight parameter in the WFSA transition function that allows a WFSA to accept a multiset of string values rather than just a set. Since collapse problems for symbolic string models occur due to using set based models to reprent multisets of string values in quantitative analyses, the ability of WFSAs to accept a multiset of strings is essential to solving collapse problems for automata-based symbolic string models.

In addition to these differences in the WFSA 6-tuple compared to the FSA quintuple, the automata operation for the WFSA must be changed from their FSA counterparts. Specifically, the minimization algorithm (whether Hopcroft, Moore, or Brzozowski), SubsetConstruction (determinization), Intersection, Union, Subtraction, Concatenation, and Complement automata operations must be modified so that they perform appropriately with the new weighted transitions $\delta : (q, \alpha, i_w) \rightarrow q_d$ and empty string counter i_ε. In particular, the SubsetConstruction and minimization algorithms must ensure that transitions are duplicates according to states, symbols, and weights before reducing such duplicate transitions to single transitions with increased weights.

3.4.2 Model Creation

The creation process for WFSA sequence of the *weighted* model is shown in Figure 3.20 and starts with the WFSASequence algorithm shown in Figure 3.20a. WFSASequence requires an initial WFSA $\mathcal{W}_{init}$ similar to the five initial FSAs

```
1: procedure WFSASequence($\mathcal{W}_{init}, k, \Sigma$)
2:     for $\mathcal{W}_r^i \in (\mathcal{W}_r^n)_{n=0}^k$ do
3:         $\mathcal{W}_b \leftarrow$ LengthFSA($i, \Sigma$)
4:         $\mathcal{W}_r^i \leftarrow \mathcal{W}_{init} \cap \mathcal{W}_b$
5:     end for
6:     return $(\mathcal{W}_r^n)_{n=0}^k$
7: end procedure
```

(a) Creation algorithm for FSA sequence

```
1: procedure LengthWFSA($k, \Sigma$)
2:     $Q \leftarrow F \leftarrow \{q_0\},\ \delta \leftarrow \varnothing,\ i_\varepsilon \leftarrow 0$
3:     $\mathcal{W} \leftarrow \langle Q, \Sigma, \delta, q_0, F, i_\varepsilon \rangle$
4:     $q \leftarrow q_0$
5:     for $i \leftarrow 0$ to $k$ do
6:         $Q \leftarrow Q \cup \{q_n\}$
7:         for $\alpha \in \Sigma$ do
8:             $\delta(q, \alpha, 1) \rightarrow q_n$
9:         end for
10:        $q \leftarrow q_n$
11:    end for
12:    $F \leftarrow F \cup \{q_n\}$
13:    if $k = 0$ then
14:        $i_\varepsilon \leftarrow 1$
15:    end if
16:    return $\mathcal{W}$
17: end procedure
```

(b) Creation algorithm for length bounded FSA

Figure 3.20: Algorithms to create WFSA sequence for *weighted* automata model

described in Section 2.4.2: *empty*, *empty string*, *literal*, *simple unknown*, and *complex unknown*. Also required by the WFSASequence procedure are an initial bounding length k and an alphabet Σ. WFSASequence begins by initializing the WFSA sequence $(\mathcal{W}_r^n)_{n=0}^k$ on line 2 before iterating the loop on lines 2-5 from 0 to k. Each iteration of the loop creates a bounding WFSA on line 3 using the LengthWFSA algorithm with the current i value and the alphabet Σ. Similar to the LengthFSA algorithm for *aggregate* automata models, the LengthWFSA algorithm shown in Figure 3.20b is used to create a WFSA which accepts all strings of the specified length k using every combination of symbols in the alphabet Σ. LengthWFSA initializes the return WFSA on lines 2-3 with the single non-accepting start state q_0, not transitions, and an initial empty string count $i_\varepsilon = 0$. Line 4 of LengthWFSA sets the current state q as the WFSA $\mathcal{W}$ start state q_0. Next in the LengthWFSA algorithm, the loop on lines 5-11 creates the structure of the bounding WFSA $\mathcal{W}$

while iterating from 0 to k. Line 6 within the loop creates a new state q_n and adds it to the set of states $Q \in \mathcal{W}$. Lines 7-9 of LengthWFSA then create a transition from the current state q to the new non-accepting state q_n with a weight of 1 for every symbol in the specified alphabet Σ. The last part of this loop updates the current state q by setting it to the newly added state q_n allowing future loop iterations to add transitions from that state. After the loop in LengthWFSA used to create the WFSA structure, line 12 adds the current state q to the set of accepting states $F \in \mathcal{W}$ making it the single accepting state in the WFSA. The if condition on lines 13-15 of LengthWFSA checks if k is 0 and if so updates the empty string counter i_ε to 1. LengthWFSA completes on line 16 by returning the created length bounding WFSA W from the algorithm. WFSASequence resumes its iteration using the new bounding WFSA W_b from line 3 in the Intersection operation on line 4 between it the initial WFSA $\mathcal{W}_{init}$ producing a WFSA stored as $\mathcal{W}_r^i$ at index i in the WFSA sequence $(\mathcal{W}_r^n)_{n=0}^k$. After WFSASequence finishes the loop from lines 2-5, the WFSA sequence $(\mathcal{W}_r^n)_{n=0}^k$ which is now complete is returned on line 6. This returned WFSA sequence is used along with the bounding length k and the alphabet Σ to create the new *weighted* automata model.

3.4.3 Model Counting

Figure 3.21 shows the model counting algorithm for the *weighted* automata model. The coordinating algorithm CounteWeighted shown in 3.21a utilizes the recursive model counting algorithm MCWeighted which is shown in 3.21b. The model counting process begins using the CountWeighted procedure which accepts the *weighted* automata model $\mathcal{M}$ as its only parameter. CountWeighted starts by initializing the result model count mc_r as 0 on line 2 and extracting the WFSA

```
1: procedure CountWeighted($\mathcal{M}$)
2:     $mc_r \leftarrow 0$
3:     $(\mathcal{W}_n)_{n=0}^{k} \in \mathcal{M}$
4:     for $\langle Q_i, \Sigma_i, \delta_i, q_0^i, F_i, i_\varepsilon^i \rangle \in (\mathcal{W}_n)_{n=0}^{k}$ do
5:         $mc_r \leftarrow mc_r +$ MCWeighted$(q_0, 1)$
6:         if $q_0^i \in F_i$ then
7:             $mc_r \leftarrow mc_r + i_\varepsilon^i$
8:         end if
9:     end for
10:    return $mc_r$
11: end procedure
```

(a) Coordinating algorithm

```
1: procedure MCWeighted($q, i$)
2:     $mc_r \leftarrow 0$
3:     for $\langle i_t, q_d \rangle \in \{\langle i, q_e \rangle \mid \delta(q, \alpha, i) \rightarrow q_e\}$ do
4:         $i_w \leftarrow i \times i_t$
5:         if $q_d \in F$ then
6:             $mc_r \leftarrow mc_r + i_w$
7:         end if
8:         $mc_r \leftarrow mc_r +$ MCWeighted$(q_d, i_w)$
9:     end for
10:    return $mc_r$
11: end procedure
```

(b) Recursive model counting algorithm

Figure 3.21: Model counting algorithms for *weighted* automata models

sequence $(\mathcal{W}_n)_{n=0}^{k}$ from the $\mathcal{M}$ model on line 3. The loop on lines 4-9 of CountWeighted are responsible for calculating the model count for each WFSA in the $(\mathcal{W}_n)_{n=0}^{k}$ sequence. Line 4 of CountWeighted begins the loop by extracting the WFSA 6-tuple $\langle Q_i, \Sigma_i, \delta_i, q_0^i, F_i, i_\varepsilon^i \rangle$ at the corresponding index i for use in each iteration of the loop. Line 5 of CountWeighted utilizes the MCWeighted recursive algorithm with the start state $q_0^i \in \mathcal{W}_i$ and initial transition weight 1 as parameters to count the number of string values represented by WFSA $\mathcal{W}_i$. The MCWeighted recursive algorithm takes two parameters, a WFSA state q and an initial transition weight i. MCWeighted begins by initializing its return model count mc_r as 0. The MCWeighted procedure then loops through lines 3-9 for each transition $\delta(q, \alpha, i) \rightarrow q_e$ leaving the WFSA state q using weight i and destination state q_e of the transition as i_t and q_d for each iteration of the loop. Line 4 of this loop multiplies the transition weight parameter i by the weight i_t for the loop iteration's transition to produce a new transition weight i_w. Lines 5-7 of MCWeighted check if the destination state q_d of the loop iteration's transition is an accepting state and if so the the return model count mc_r is incremented by the new transition weight i_w. Line 8 of MCWeighted then makes the recursive call to itself specifying the destination state q_d of loop iteration's transition and the new transition weight i_w. After the loop in MCWeighted is complete, line 10 returns the updated return model count mc_r. After CountWeighted increments the return model count mc_r by model count returned from MCWeighted, lines 6-8 of CountWeighted ensure that the correct number of empty string values are accounted for by checking if q_0^i is an accepting state and incrementing the return model count mc_r by the empty string counter i_ε^i if it is an accepting state. After CountWeighted completes the iteration through the loop on lines 4-9, the computed model count mc_r is returned on line 10. While this process of model counting *weighted* automata models is very similar to the same process for *aggregate* models, the use of WFSA does alter alter the model counting process. This alteration is most noticeable on line 6 of MCWeighted which uses the newly computed transition weight i_w to increment the model count instead of an increment of 1 per transition as is the case when model counting *unbounded*, *bounded*, and *aggregate* automata models.

3.4.4 String Operations and Predicates

The use of WFSAs for the *weighted* automata model requires the creation of additional string operation and predicate simulation algorithms since the existing FSA algorithms can not be used. These WFSA operation and predicate simulation algorithms are altered from their existing FSA versions in three primary ways: utilizing the WFSA specific automata operation algorithms, including the empty string counter, and adjusting WFSA transitions to incorporate weight adjustments. The use of the WFSA specific auotmata operation such as Union and Intersection is a very simple change needed since the WFSAs is used instead of FSAs. The inclusion of the empty string counter for adjustments is needed primarily for *subtractive* and *complex* *substitutive* operations which can produce an empty string value corresponding to more than one string values prior to the operation. The adjustments of transition weights by the simulation algorithms is is the means by which collapse problems are prevented in *weighted* automata models. Each of these three alterations of existing FSA operation and predicate simulation algorithms allows the performance of these algorithms to be compared between the *weighted* model and each of the *unbound*, *bounded*, and *aggregate* models with a focus on model differences rather than algorithm differences.

```
1: procedure WBinaryOp($\textsc{Op}, \mathcal{M}_1, \mathcal{M}_2, \Sigma$)
2:     $(\mathcal{W}_1^n)_{n=0}^k \in \mathcal{M}_1, (\mathcal{W}_2^n)_{n=0}^k \in \mathcal{M}_2$
3:     $\mathcal{W}_{U_2} \leftarrow \bigcup_{n=0}^k \mathcal{W}_2^n$
4:     for $\mathcal{W}_{op}^i \in (\mathcal{W}_{op}^n)_{n=0}^k$ do
5:         $\mathcal{W}_{op}^i \leftarrow \textsc{Op}(\mathcal{W}_1^i, \mathcal{W}_{U_2})$
6:     end for
7:     $\mathcal{M}_{op} \leftarrow \langle (\mathcal{W}_{op}^n)_{n=0}^k, k, \Sigma \rangle$
8:     return $\mathcal{M}_{op}$
9: end procedure
```

(a) String operation and predicate coordination algorithm for two *weighted* automata model arguments

```
 1: procedure RestructWFSAs($\mathcal{M}_{op}$)
 2:     $(\mathcal{W}_{op}^n)_{n=0}^k \in \mathcal{M}, \Sigma \in \mathcal{M}$
 3:     $k_r \leftarrow \textsc{GetMaxLength}((\mathcal{W}_n)_{n=0}^k)$
 4:     for $\mathcal{W}_r^i \in (\mathcal{W}_r^m)_{m=0}^{k_r}$ do
 5:         $\mathcal{W}_b \leftarrow \textsc{LengthFSA}(i, \Sigma)$
 6:         for $\mathcal{W}_I^j \in (\mathcal{W}_t^n)_{n=0}^k$ do
 7:             $\mathcal{W}_I^j \leftarrow \mathcal{W}_{op}^j \cap \mathcal{W}_b$
 8:         end for
 9:         $\mathcal{W}_r^i \leftarrow \bigcup_{n=0}^{k_r} \mathcal{W}_I^n$
10:     end for
11:     $\mathcal{M}_r \leftarrow \langle (\mathcal{W}_r^n)_{n=0}^{k_r}, k_r, \Sigma \rangle$
12:     return $\mathcal{M}_r$
13: end procedure
```

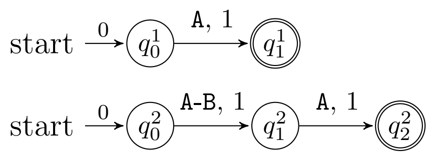
(b) Restructuring algorithm for *weighted* automata models

Figure 3.22: *Weighted* automata model utility algorithms for string operations and predicates

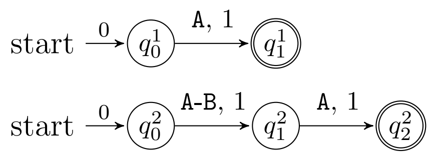
Since the *weighted* automata model uses a WFSA sequence in the same way that the *aggregate* model uses its FSA sequence, the same problems expressed in Section 3.3.1 for operations and predicate simulation are also problems that need addressing for *weighted* automata models. Figure 3.22 contains the WBinaryOp procedure in Figure 3.22a and the RestructWFSAs procedure in Figure 3.22b. The WBinaryOp procedure is used to coordinate the simulation of string operations or predicates which require two *weighted* automata models. The WBinaryOp procedure functions very similarly to the ABinaryOp procedure where the only differences in the WBinaryOp version are the use of WFSAs instead of FSAs and the removal of factors. The RestructWFSAs procedure also functions very similarly to the RestructFSAs procedure for *aggregate* models where the only differences in the RestructWFSAs procedure for *weighted* models is the use of WFSAs instead of FSAs and the removal of factors. While the restructuring process for the WFSA sequence of a *weighted* automata model should not be nessesary to prevent *subtractive* collapses, the similar RestructWFSAs procedure is used to preserve consistency between the two models.

3.4.5 Over-Approximation

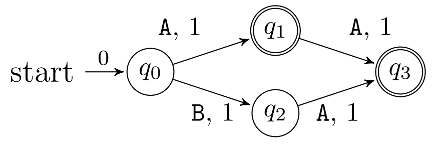
The example shown in Figure 3.23 demonstrates how over-approximation due to model structure is prevented for *weighted* automata models. This example presents the string operation Concatenation(str_1, str_2) simulated for *weighted* automata models $\mathcal{M}_1$ and $\mathcal{M}_2$ representing the string variables str_1 and str_2 respectively. Figures 3.23a and 3.23b show the diagrams corresponding to $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $\mathcal{M}_1$ and $\mathcal{M}_2$ both have an alphabet $\Sigma_{\mathcal{M}_1} = \Sigma_{\mathcal{M}_2} = \{\texttt{A}, \texttt{B}\}$ and bounding length $k = 2$ and both represent the solution set $\mathcal{S}_{\mathcal{M}_1} = \mathcal{S}_{\mathcal{M}_2} = \{\texttt{A}, \texttt{AA}, \texttt{BA}\}$. Figure 3.23c shows the diagram of the *weighted* automata model $\mathcal{M}_{U_2}$ which is produced on line 3 of the WBinaryOp procedure from the WFSA sequence in the $\mathcal{M}_2$ model. Figure 3.23d shows the *weighted* automata model $\mathcal{M}_C$ produced by simulating the Concatenation algorithm for $\mathcal{M}_1$ and $\mathcal{M}_{U_2}$. Figure 3.23e shows the *weighted* automata model returned from the RestructWFSAs procedure performed on the $\mathcal{M}_C$ model. Just as with the *bounded* and *aggregate* automata models, the *weighted* model prevents the



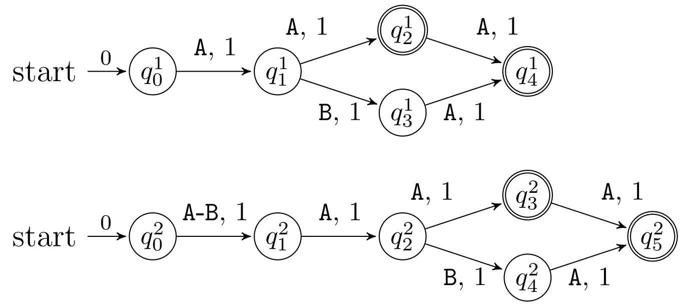
(a) $\mathcal{M}_1$ *weighted* automata model



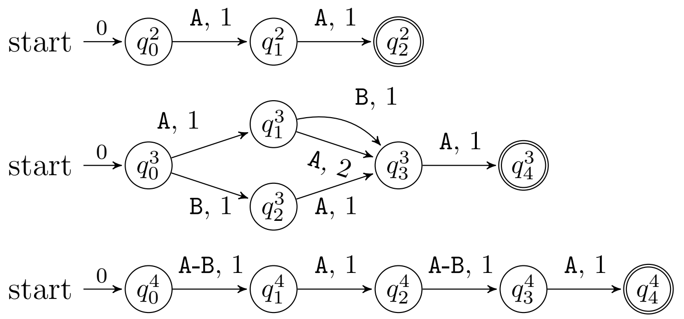
(b) $\mathcal{M}_2$ *weighted* automata model



(c) $\mathcal{M}_{U_2}$ *weighted* automata model



(d) $\mathcal{M}_C$ *weighted* automata model resulting CONCATENATION operation



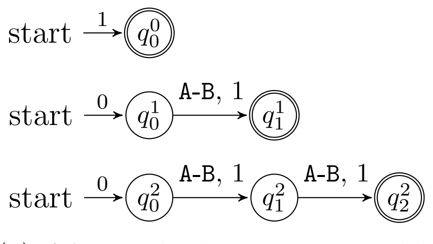
(e) $\mathcal{M}_r$ *weighted* automata model after restructuring the WFSA sequence

Figure 3.23: CONCATENATION($\mathcal{M}_1, \mathcal{M}_2$) example for *weighted* automata model

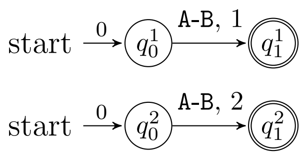
over-approximation of its solution set for the CONCATENATION($\mathcal{M}_1, \mathcal{M}_2$) operation since the expected solution set $\mathcal{S}_e$ and the solution set of the restructured model $\mathcal{S}_{\mathcal{M}_R}$ are equal, i.e. $\mathcal{S}_e = \mathcal{S}_{\mathcal{M}_R}\{\texttt{AA}, \texttt{AAA}, \texttt{ABA}, \texttt{BAA}, \texttt{AAAA}, \texttt{AABA}, \texttt{BAAA}, \texttt{BABA}\}$. This example demonstrates how over-approximation of a solution set due to an automata model is prevented in *weighted* automata models.

The example in Figure 3.23 also demonstrates that *weighted* automata models do not suffer from model count over-approximation observed in *aggregate* models as described in Section 3.3.3. The expected model count is 9 for both the model produced by the CONCATENATION $\mathcal{M}_C$ and the restructured model $\mathcal{M}_R$ which matches the actual model counts $MC(\mathcal{M}_C) = MC(\mathcal{M}_R) = 9$. *Weighted* automata models do not suffer from model count over-approximation because the factor sequence in *aggregate* models which is the cause for the over-approximation is not present in *weighted* models.

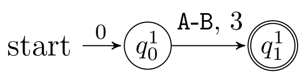
3.4.6 Collapse



(a) $\mathcal{M}_1$ *weighted* automata model



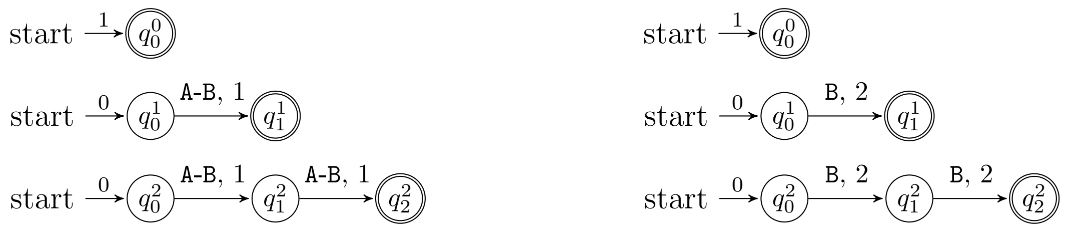
(b) $\mathcal{M}_D$ *weighted* automata model resulting from DELETE operation



(c) $\mathcal{M}_r$ *weighted* automata model after restructuring the WFSA sequence

Figure 3.24: DELETE($\mathcal{M}_1, 1, 2$) example for *weighted* automata model

The example shown in Figure 3.24 demonstrates how the collapse problem is prevented for *weighted* automata models due to *subtractive* string operations. This example presents the string operation DELETE(str_1, int_s, int_e) simulated for the *weighted* automata model $\mathcal{M}_1$ and the start and end indices 1 and 2. Figure 3.24a shows the diagram corresponding to $\mathcal{M}_1$ such that $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\texttt{A}, \texttt{B}\}$ and maximum length $k = 2$ and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$. Figure 3.24b shows the diagram of the *weighted* automata model $\mathcal{M}_D$ which is returned from the DELETE($\mathcal{M}_1, 1, 2$) operation and has the same alphabet as $\mathcal{M}_1$, $\Sigma_{\mathcal{M}_D} = \{\texttt{A}, \texttt{B}\}$. The prevention of a collapse due to the DELETE operation can be seen by comparing the expected model count of 6 (empty string is not a valid target of the DELETE(str_1, int_s, int_e) operation) to the model counts of both the model returned from the delete operation $MC(\mathcal{M}_D) = 6$ and the restructured model $MC(\mathcal{M}_R) = 6$. This is an example of the prevention of collapse due to *subtractive* operations using *weighted* automata models.



(a) $\mathcal{M}_1$ *weighted* automata model

(b) $\mathcal{M}_R$ *weighted* automata model resulting from REPLACE operation

Figure 3.25: REPLACE($\mathcal{M}_1, \texttt{A}, \texttt{B}$) example for *weighted* automata model

The example shown in Figure 3.25 demonstrates how the collapse problem is prevented for *weighted* automata models due to *substitutive* string operations. This example presents the string operation REPLACE($str_1, char_1, char_2$) simulated for the *weighted* automata model $\mathcal{M}_1$ representing the string argument str_1 and the symbols A and B as the symbol arguments $char_1$ and $char_2$. Figure 3.25a shows the diagram corresponding to $\mathcal{M}_1$ such that $\mathcal{M}_1$ has an alphabet $\Sigma_{\mathcal{M}_1} = \{\texttt{A}, \texttt{B}\}$ and maximum length $k = 2$ and represents the solution set $\mathcal{S}_{\mathcal{M}_1} = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{AA}, \texttt{AB}, \texttt{BA}, \texttt{BB}\}$. Figure 3.25b shows the diagram of the *weighted* automata model $\mathcal{M}_R$ which is returned from the DELETE($\mathcal{M}_1, 1, 2$) operation and has the same alphabet as $\mathcal{M}_1$, $\Sigma_{\mathcal{M}_R} = \{\texttt{A}, \texttt{B}\}$. The prevention of the collapse due to the REPLACE($\mathcal{M}_1, \texttt{A}, \texttt{B}$) operation can be seen when comparing the expected model count 7 with the actual model count after the REPLACE operation $MC(\mathcal{M}_R) = 7$. This collapse prevention is accomplished due to the use of weighted transitions in the WFSA where the REPLACE algorithm changes the one A transition in $\mathcal{W}_1^1$ and the two A transitions in $\mathcal{W}_1^2$ to one and two B transitions respectively. Since these B transitions are duplicates of already exiting transitions, the minimize automata operation merges these duplicate transitions. However, since WFSAs have weighted transitions, the merging of the duplicates in the WFSA minimization algorithm requires the summation of the duplicate transition weights to produce the weight 2 `B` transitions seen in WFSA $\mathcal{W}_R^1$ and $\mathcal{W}_R^2$. Thus the `A` and `B` string values accepted by $\mathcal{W}_1^1$ are represented by the two `B` string values accepted by $\mathcal{W}_R^1$. Similarly, the `AA`, `AB`, `BA`, and `BB` string values accepted by $\mathcal{W}_1^2$ are represented by the four `BB` values accepted by $\mathcal{W}_R^2$. It is through the use of these weighted transitions that *weighted* automata model prevent collapses due to *substitutive* string operations.

CHAPTER 4

STRING CONSTRAINT SOLVER FRAMEWORK

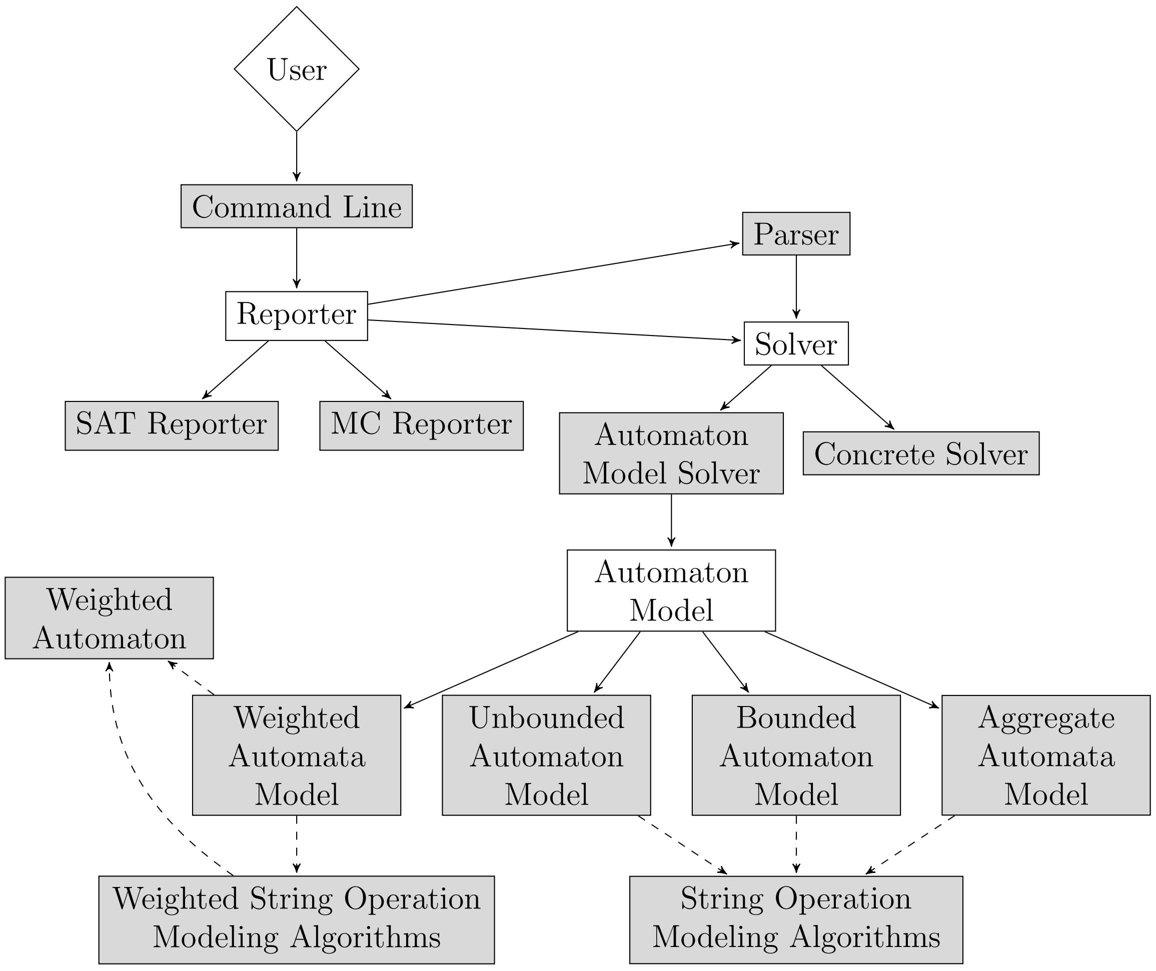


Figure 4.1: The String Constraint Solver Framework (SCSF)

The String Constraint Solver Framework (SCSF) is a Java software program which is used to solve a specified string constraint directed graph and determine either the satisfiability of the constraint or its model count depending on which option is specified. SCSF was adapted from the *Processor* component of the String Solver Analysis Framework detailed by Kausler [21]. This chapter will cover the different components used in the construction of this software analysis tool. Section 4.1 reports the third party software libraries upon which SCSF depends. Section 4.2 details the user interface to the SCSF tool. Section 4.3 explores the string constraint graphs (SCG) used as input to the SCSF tool. Section 4.4 describes the different components and their role in the software architecture of the SCSF tool. Section 4.5 outlines the test suite which ensures the correct functioning of the different components of the SCSF tool. Finally, Section 4.6 covers the various utility scripts used to orchestrate and supplement the SCSF tool in the analysis produced by this work.

4.1 Third Party Dependencies

Library	**Copyright Holder**	**Version**
Apache Maven	The Apache Software Foundation	
Apache Commons CLI	The Apache Software Foundation	1.3.1
Apache log4j	The Apache Software Foundation	1.2.17
Jackson Project	FasterXML, LLC	2.7.2
jgrapht	Barak Naveh and Contributors	0.9.1
dk.brics.automaton	Anders Møller	1.11-8
Java String Analyzer	Anders Møller, Aske Simon, and Asger Feldthaus	2.1-1
JUnit	JUnit	4.12
mockito	Mockito contributors	1.10.19
hamcrest	www.hamcrest.org	1.3
NumPy	NumPy Developers	1.9.2
SciPy	Enthought, Inc.	0.16.0

Table 4.1: SCSF Third Party Dependencies

In the creation of the SCSF tool and its associated utility scripts, we utilized many third party software libraries to speed the development process of the tool and to leverage the domain expertise embedded in these libraries. Table 4.1 lists each of these third party dependencies with the copyright holder associated with the library and the version used in the SCSF tool. The uses of each of these third party dependencies in the SCSF tool and associated scripts is listed below:

- The Apache Maven build automation tool allows easy tracking and updating of the third party dependencies in the SCSF tool.
- The Apache Commons CLI is used easily create a robust and standardized command line interface to the SCSF tool.
- The Apache log4j library is used to provide logging messages for the SCSF tool.
- The Jackson Project JSON library and parser which is used to parse the input SCG files which are expected to be in the JSON file format.
- The jgrapht graph theory library is used to construct the SCGs as Java objects in memory for processing in the SCSF tool.
- dk.brics.automaton is an automaton library used to create FSA for the *unbounded*, *bounded*, and *aggrgate* automata model implementations in the SCSF tool.
- Java String Analyzer is a string analysis tool which it includes a library of string operation and predicate simulation algorithms for dk.brics.automaton FSAs.
- JUnit is a unit testing framework used to create the unit test suite that accompanies the SCSF tool.
- mockito is a mocking framework used to create the test doubles for the unit tests in the test suite that accompanies the SCSF tool.
- hamcrest is library which is used to create easy to understand matching assertion critera for the unit tests in the test suite that accompanies the SCSF tool.

- NumPy is a python library which enables the use efficient use of large arrays and matrices, it is used in the python utility scripts to gather and analyze the results of the evaluation described in Chapter 5.
- SciPy is a python library used for scientific and technical computing, it is used in the python utility scripts to perform statistical tests for the result analysis in the evaluation described in Chapter 5.

4.2 Interface

SCG File Path	$\langle \mathit{filepath} \rangle$	**Initial Bounding Length:**	$\langle \mathit{integer} \rangle \geq 0$
Reporter Type:	Satisfiability	**Automata Model Type:**	Unbounded
	Model Count		Bounded
Solver Type:	Concrete		Aggregate
	Automaton Model		Weighted

Figure 4.2: SCSF Command Line Parameters

The command line processor component of SCSF uses Apache Commons CLI library to parse command line options and perform the appropriate configuration actions, e.g., converting the SCG file into an in-memory directed graph data structure. SCSF has one required command line parameter, a file path to a SCG file. The SCG file is expected to be a `.json` file adhering to a particular JSON schema and will be discussed in more detail in Section 4.3.1. SCSF also accepts five additional optional parameters which can be used to change the behaviour of the SCSF tool when solving the string constraints in the specified SCG. Without these optional parameters specified, default values are supplied instead. These five total command line parameters are listed and enumerated in Figure 4.2. The *initial bounding length* parameter can be used to refine the solver behaviour. The *initial bounding length* parameter specifies a maximum length for *unknown* string types when represented by symbolic string models with a default bounding length of 10. The *reporter type* parameter determines whether the SCSF will solve the constraints for either satisfiability which is the default option or for probability, i.e., SE vs PSE. The *solver type parameter* specifies which string constraint solver to utilize, the options being either the default option of the automata model solver or the concrete solver oracle. Finally, if the automaton model *solver type* is chosen, an *automta model type* parameter can be used to choose an automata model with the default model as the *unbounded* automata model.

4.3 Inputs

The only input to the SCSF tool is a string constraint graph (SCG) for which the chosen string analysis will be performed. As mentioned in Section 4.2, SCSF requires the SCG file to be provided as a `.json` file. This file must also adhere to a specific JSON schema so that the edge and node data of the graph can be accurately converted into an in-memory graph data structure. This graph schema has emerged from the conversion of existing serialized graph files from the work of Kausler [21] into JSON files and thus reflects a string constraint SCG obtained via Dynamic Symbolic Execution (DSE) [11,33], i.e. each vertex in the SCG was recorded during the symbolic execution of a concrete software program. Section 4.3.1 describes this schema by detailing the required structure of an SCG. Section 4.3.2 then describes the creation of multiple synthetic SCGs which are used as the primary dataset for this analysis of automata-based symbolic string models.

4.3.1 Constraint Graph Structure

Alphabet:	Size	$\langle \textit{integer} \rangle \geq 0$	
	Alphabet Declaration	$\langle \textit{Alphabet Declaration} \rangle$	
Verticies:	Incoming Edges		
		Source Vertex Id	$\langle \textit{integer} \rangle \geq 0$
		Source Type	$\{\texttt{t}, \texttt{s1}, \texttt{s2}, \texttt{s3}, \texttt{s4}, \texttt{s5}, \texttt{s6}\}$
	Source Constraints		
		Vertex Id	$\langle \textit{integer} \rangle \geq 0$
	Time Stamp	$\langle \textit{integer} \rangle \geq 0$	
	Value	$\langle \textit{StringConstraint} \rangle$	
	Number	$\langle \textit{integer} \rangle \geq 0$	
	Actual Value	$\langle \textit{string} \rangle$	
	Type	$\langle \textit{integer} \rangle \geq 0$	
	Id	$\langle \textit{integer} \rangle \geq 0$	

Figure 4.3: SCG File Components

The schema for a graph file specifies two main components: an optional alphabet specification and an array of graph vertices. Both of these components are encapsulated in an outer JSON object. The characteristics of both the alphabet and vertices components are listed in Figure 4.3. The alphabet specification is a JSON object named `alphabet` and contains a maximum length and a symbol set specification. The maximum length has the key `size` and an integer value, e.g. `"size":4`. The symbol set specification in the JSON file has the key `declaration` and a string value which expresses the characters in the alphabet in the same way as bracket expression in a regular expression, e.g. `"declaration":"A-Ca-C"` which specifies only the characters `A`, `B`, `C`, `a`, `b`, and `c` are in the alphabet. If an alphabet specification is not included in the graph file, one is generated from the known string values which appear within the constraint graph. The vertices in the graph file are JSON objects which are contained in an array. Each vertex object contains the following items: an incoming edges array, a source constraints array, a time stamp, a value, a number, an actual value, a type, and an id. The array of incoming edges contains objects with two data items: a source vertex id corresponding the the id of another vertex object and a source time string value indicating that the current vertex object is either the primary (`t`) or non-primary (`s1`-`s6`) parameter of a operation or predicate. The source constraints array contains the ids of other vertex objects visited before the current vertex was recorded. The time stamp value is the time at which the current vertex was recorded during the SCG creation process. The value is a string representation of the string constraint. The number is an integer indicating the number of previous visited constraints along the execution path before the current vertex. The actual value is the record of the actual string value the string variable corresponding to the current vertex recorded during the dynamic symbolic execution. The type is a integer value from 0 to 11 indicating a category of string constraint applying to the vertex, this value is not used in the SCSF tool and is a legacy of the previous work [21] upon which the SCSF tool is built. The id value is a positive integer which uniquely identifies the vertex object within the array of vertices. Both the alphabet specification and the vertices array are the only components of a JSON graph file expected as the input to the SCSF tool.

4.3.2 Synthetic Constraint Graphs

Alphabet:	$\langle$*Alphabet Declaration*$\rangle$	**Input String(s):**	Specified Concrete String(s)
Maximum Consecutive			Empty String (ε)
String Operations:	$\langle integer \rangle > 0$		Unknown String

Figure 4.4: Synthetic SCG Generation Script Options

The synthetic SCG dataset is generated from a Python script, which produces one or more SCGs depending on the specified options. The script accepts a number of options, which configure the resulting SCGs. A user can configure which string operations and predicates are used to produce the SCGs. Other important script options include input string options, an alphabet specification, and the initial string length k. The input string options allows a user to specify one or more concrete string values as the initial strings values appearing at the root node in the SCGs. Additionally, two unique string values can be chosen as input strings: the empty string and an unknown string value (represents any string in the the specified alphabet up to k). The alphabet specification is used to generate both appropriate arguments for string operations which appear in the string constraints as well as generating random strings for the actual value field of the SCGs. The initial string length is used in conjunction with the unknown string value to limit the possible length of the random string generated for the actual value string.

The script is used to produce multiple string constraint graphs containing a thorough combination of possible string operations and predicates. The script begins by producing root nodes for the constraint graph corresponding to each of the specified input string options. Next, each configured string operation is used to produce an array of string operation nodes where each operation produces a thorough set of operation nodes for each possible argument configuration. For example, the `substring` operation will produce an operation for each of the following argument configurations when the initial string length is 2: $\{(0,0),(0,1),(0,2),(1,1),(1,2),(2,2)\}$. Each of these string operation nodes are collected into a set for later use. String predicates undergo a similar process and are collected into a separate set. The set of predicate nodes are then used to produce an n-fold Cartesian product up to a depth (n) specified as a script option. The resulting Cartesian product set contains all possible combinations of string operations in all possible operation orders. This set is then combined with the set of string predicates, producing a set where all possible string operation configurations are constrained by all possible string predicates. In this way, the script attempts to anticipate all possible string constraint graphs which can occur in a software program for the specified alphabet, initial strings, and operation depth.

4.4 Components

The three primary components of the SCSF tool are the reporter which is detailed in Section 4.4.1, the parser which is discussed in Section 4.4.2, and the solver which is explored in Section 4.4.3. Additional major components of the SCSF tool include the automata models which are detailed in Section 4.4.4, the weighted-transition finite state automaton implementation explored in Section 4.4.5, and the symbolic string model algorithms discussed in Section 4.4.6.

4.4.1 Reporters

The reporter component of SCSF is responsible for gathering the result of solving each branch PC in the SCG and reporting these results to standard output. Due to this central role, the reporter becomes the primary orchestrating component. The reporter retrieves each node from the SCG and must determine if the node represents a root node (a new string variable in the SCG) an operation node (a string operation) or a constraint node (a string predicate). For each of these node types, the node is then passed to the parser using the appropriate method for the node type. If the node type was a constraint node, the reporter gathers the results from the parser's semantic action (solving the *true* and *false* branch *PC*s). Once all nodes in the SCG have been parsed and all semantic actions have been performed, the reporter reports the gathered results to standard output. The abstract class `Reporter` is implemented in the SCSF tool by two different reporters: `SATReporter` for satisfiability analyses and `MCReporter` for quantitative analyses.

Satisfiability Reporter

The `SATReporter` is an implementation of the `Reporter` abstract class in the SCSF tool and is responsible for coordinating and reporting the results of a SE analysis. This reporter requires a jgrapht directed graph of string constraint vertices, a `Parser` implementation, and an implementation of the `ExtendedSolver` abstract solver (satisfiability solver). The `SATReporter` outputs the following information for each predicate graph vertex:

- The integer id of the vertex.
- The "actual value" string value for the predicate vertex.
- If the predicate is a singleton branching point, i.e. all involved string variables only represent single concrete string values.
- The satisfiability of the *true* predicate branch.
- The satisfiability of the *false* predicate branch.

- Whether the predicate branches are disjoint, i.e. branches are disjoint if there is no overlap in their solution sets: $\mathcal{S}_t \cap \mathcal{S}_f = \varnothing$.
- A string representation of the previous string initializations, operations, and predicates.

Model Count Reporter

The `MCReporter` is an implementation of the `Reporter` abstract class in the SCSF tool and is responsible for coordinating and reporting on the results of a PSE analysis. This reporter requires a jgrapht directed graph of string constraint vertices, a `Parser` implementation, and an implementation of the `ModelCountSolver` solver interface. The `MCReporter` outputs the following information for each predicate graph vertex:

- The integer id of the vertex.
- The "actual value" string value for the predicate vertex.
- If the predicate is a singleton branching point.
- The satisfiability of the *true* predicate branch.
- The satisfiability of the *false* predicate branch.
- Whether the predicate branches are disjoint
- The accumulated time of previous variable initializations, operations, and predicates.
- The id of the previous initialization, operation, or predicate vertex for the primary string argument in the predicate.
- The model count corresponding to the symbolic string model of the incoming primary string variable in the predicate.
- The time to perform the previous variable intialization, operation, or predicate for the primary string variable in the predicate.

- The model count of the *true* predicate branch.
- The time required to compute the model count for the *true* predicate branch.
- The time required to simulate the predicate for the *true* predicate branch.
- The model count of the *false* predicate branch.
- The time required to compute the model count for the *false* predicate branch.
- The time required to simulate the predicate for the *false* predicate branch.
- The amount of overlap between the *true* and *false* predicate branches, i.e. model count of the disjoint symbolic string model.
- A string representation of the previous string initializations, operations, and predicates.

4.4.2 Parser

The parser component of SCSF is responsible for determining the appropriate semantic action for each node in the SCG. The parser accomplishes this in three different ways, one way for each of the different node types: root, operation, and constraint. For root nodes, the parser determines the string type of the new string value so that the appropriate symbolic string model can be created by the solver. For operation nodes, the parser determines the string operation so that the appropriate operation simulating algorithm can be chosen by the solver. For constraint nodes, the parser determines the string predicate so that the appropriate predicate simulating algorithm can be chosen by the solver. Through interpreting these three different node types in the SCG, the graph structure is converted into a series of action to be performed by the solver.

4.4.3 Solvers

The solver component of SCSF is responsible for managing the symbolic string values through a variety of actions which correspond to the parser's semantic actions. These actions include creating new symbolic string models, simulating string operations for symbolic string models, determining satisfiability of a symbolic string model, and determining the model count of a symbolic string model. This solver component implements the bridge design pattern so that the actions can be applied to a variety of possible symbolic string models and allowing the symbolic string model implementation to vary independent of the solver [15]. Two different solver implementations are available: a concrete solver and an automaton model solver.

Concrete Solver

The concrete solver `ConcreteSolver` provides the string constraint oracle and implements the `ModelCountSolver` interface and extends the `ExtendedSolver` abstract class allowing `ConcreteSolver` to serve as the solver component for either type of reporter in the SCSF tool. This concrete solver can provide an oracle for either SAT or MC solvers due to two distinct differences from other abstract solver implementations: it models symbolic strings using arrays of concrete string values and it invokes the actual string operations and predicates rather than simulating such operations using algorithms. The `ConcreteSolver` models symbolic strings using an array of all possible concrete string values. These arrays of string values are created according to the different string types described in Section 2.4.2 where the *empty* string type is an empty array, the *empty string* string type is a 1 element array where the only element is the empty string `""`, and the *literal* string type is a 1 element array where the only element is the string literal. The *complex unknown* string type is only created as the result of string operations and predicates and so is not needed. The *simple unknown* string type is created by creating an array filled with only one of each and every possible string value for the specified alphabet and bounding length. This requires significantly more memory than string constraint solvers which employ symbolic models, but guarantees accuracy since each possible string value is accounted for in the array. The solver performs string operations on each concrete string which precisely simulates the results of string operations on the possible concrete string values. However, this does require a significant performance cost due to performing operations for each possible combination of strings and operation arguments. Both the performance and memory cost associated with the concrete solver makes it an impractical choice for real world analysis, but sufficient to provide the satisfiability and model count oracle for our analysis of automata-based symbolic string models.

Automaton Model Solver

The automata model solver `AutomtonModelSolver` provides the string constraint oracle and implements the `ModelCountSolver` interface and extends the `ExtendedSolver` abstract class allowing `ConcreteSolver` to serve as the solver component for either type of reporter in the SCSF tool. The `AutomatonModelSolver` requires an initial bounding length and an `Alphabet` object which are used when creating initial symbolic string models. This solver also requires an implementation of the abstract `AutomatonModelManager` which is an abstract factory for producing `AutomatonModel` objects which are further detailed in Section 4.4.4. The `AutomtonModelSolver` is responsible for coordinating the semantic actions assigned by the reporter, this means that the solver coordinates the initialization of a new symbolic string model, simulates a string operation, or simulates either the *true* or *false* branch of a string predicate. The initialization of new symbolic string models is accomplished by requesting the appropriate model from the `AutomatonModelManager` object. The simulation of string operations is accomplished by requesting the appropriate operation simulation from the `AutomatonModel` corresponding to the primary string argument of the operation. The simulation of either *true* or *false* branches of a predicate is accomplished by requesting the appropriate operation simulation from the `AutomatonModel` corresponding to the primary string argument of the predicate. This coordination role of a `AutomatonModelSolver` allows the implementations of both the `AutomatonModel` factory and the `AutomatonModel` itself to vary independent of the solver, allowing the addition of future automaton models.

4.4.4 Automaton Models

The automaton model sub-component of SCSF is responsible managing automata models. The responsibilities of these automata models include the creation of new automata models, model counting, and simulating string operations and predicates for automata models. The automata model sub-component consists of an abstract automata model interface as well as concrete implementations of this interface for the four automata models discussed in detail in Chapter 3. The abstraction of an automata model is another implementation of the bridge design pattern allowing the concrete automata model implementations to vary independent of the `AutomatonModelSolver` [15]. In this manner, the `AutomatonModelSolver` uses the abstract `AutomatonModel` as its symbolic string model rather than any specific automata model implementation allowing for greater flexibility when modifying or adding automata model implementations. This flexibility also allows future automata models to be implemented with little extra overhead. The four automata models implemented for the `AutomatonModel` interface are as follows: `UnboundedAutomatonModel`, `BoundedAutomatonModel`, `AggregateAutomata` and `WeightedAutomatonModel`

Unbounded Automata Model

The `UnboundedAutomatonModel` is an implementation of the *unbounded* automata model described in Section 3.1. An `UnboundedAutomatonModel` contains an integer to track the maximum length, an `Alphabet` object to encapsulate the alphabet for the model, and a `dk.brics.automaton.Automaton` object as the FSA implementation. An `UnboundedAutomatonModel` object is responsible for determining its model count, simulating string operations where it is the primary argument, and simulating string predicates where it is the primary argument. An `UnboundedAutomatonModel` object does utilize algorithms external to the itself for model counting, operation simulation, and predicate simulation. These external algorithms operate upon `dk.brics.automaton.Automaton` objects and therefore require only `dk.brics.automaton.Automaton` objects instead of `AutomatonModel` objects. Therefore, the aspects of model counting, operation simulation, or predicate simulation specific to *unbounded* automata models are handled within the `UnboundedAutomatonModel` object.

Bounded Automata Model

The `BoundedAutomatonModel` is an implementation of the *bounded* automata model described in Section 3.2. A `BoundedAutomatonModel` is comprised of an integer bounding length of the FSA, an `Alphabet` object to encapsulate the alphabet for the model, and a `dk.brics.automaton.Automaton` object as the FSA implementation. A `BoundedAutomatonModel` object is responsible for determining its model count, simulating string operations where it is the primary argument, and simulating string predicates where it is the primary argument. A `BoundedAutomatonModel` object also uses external algorithms for model counting, operation simulation, and predicate simulation. These external algorithms are the same algorithms used by the `UnboundedAutomatonModel` since both it and `UnboundedAutomatonModel` use the same `dk.brics.automaton.Automaton` objects for their FSAs. Similarly, the aspects of model counting, string operation simulation, or string predicate simulation specific to a *bounded* automata model are handled within the `UnboundedAutomatonModel` object.

Aggregate Automata Model

The `AggregateAutomataModel` is an implementation of the *aggregate* automata model described in Section 3.3. An `AggregateAutomataModel` object includes an integer to track the bounding length of the model, an `Alphabet` object to encapsulate the alphabet for the model, and an array of a `dk.brics.automaton.Automaton` objects as the FSA sequence implementation. An `AggregateAutomataModel` object is responsible for determining its model count, simulating string operations where it is the primary argument, and simulating string predicates where it is the primary argument. An `AggregateAutomataModel` object also uses external algorithms for model counting, operation simulation, and predicate simulation. These external algorithms are the same algorithms used by both the `UnboundedAutomatonModel` and the `BoundedAutomatonModel` since all three use the same `dk.brics.automaton.Automaton` objects for their FSAs. Similarly, the aspects of model counting, string operation simulation, or string predicate simulation specific to a *aggregate* automata model are handled within the `AggregateAutomataModel` object.

Weighted Automata Model

The `WeightedAutomatonModel` is an implementation of the *weighted* automata model described in Section 3.4. A `WeightedAutomatonModel` contains an integer to track the bounding length of the model, an `Alphabet` object to encapsulate the alphabet of the model, and an array of `WeightedAutomaton` objects as the WFSA sequence implementation. A `WeightedAutomatonModel` object is responsible for determining its model count, simulating string operations where it is the primary argument, and simulating string predicates where it is the primary argument. A `WeightedAutomatonModel` also uses external algorithms for model counting, operation simulation, and predicate simulation. Unlike the other three automata model implementations, a `WeightedAutomatonModel` object uses external algorithms which operate upon `WeightedAutomaton` objects. The the aspects of model counting, string operation simulation, or string predicate simulation specific to a *weighted* automata model are handled within the `WeightedAutomatonModel` object.

4.4.5 Weighted-Transition Finite State Automaton

The `WeightedAutomaton` is an implementation of the weighted-transition finite state automaton defined in Section 3.4.1. A `WeightedAutomaton` object includes both an integer serving as the empty string counter i_ε and a `WeightedState` object which servers as the initial state q_0 in the WFSA. This initial state object is the only state contained within `WeightedAutomaton` object, the other `WeightedState` objects acting as states in the WFSA are reached from this initial `WeightedState`. A `WeightedState` includes a `boolean` variable to mark the state as accepting and a set of `WeightedTransition` objects which are the outgoing transitions from the WFSA state. A `WeightedTransition` includes a minimum `char`, a maximum `char`, an integer transition weight, and a destination `WeightedState`. By using both minimum and maximum `char` values, a single `WeightedTransition` can represent multiple WFSA transitions for adjacent symbols. It is the combination of the `WeightedAutomaton` class, the `WeightedState` class, and the `WeightedTransition` class that the implementation of a WFSA is achieved in the SCSF tool. While this WFSA structure is useful in the SCSF tool, using WFSAs as part of an automata model requires working automata operation algorithms for some essential operations. The algorithms implemented for the SCSF tool include: Brzozowski's minimization [6], DETERMINIZATION (subset construction), INTERSECTION, UNION, SUBTRACTION, and CONCATENATION.

4.4.6 Symbolic String Model Algorithms

The collection of symbolic string constraint algorithms is best divided into three separate types of algorithms: model counting algorithms (Section 4.4.6), string and predicate simulation algorithms using FSAs, and string and predicate simulation algorithms using WFSAs.

Model Counting Algorithms

Because the model counting algorithms discussed in Sections 3.1.1, 3.2.1, 3.3.2, and 3.4.3 are specific to each automata model, the portions of each procedure specific to an automata model, i.e. COUNTUNBOUNDED, COUNTBOUNDED, COUNTAGGREGATE, and COUNTWEIGHTED, are encapsulated within their respective `AutomatonModel` implementations. However, the algorithms specific only to either FSAs ore WFSAs, i.e. MCUNBOUNDED, MCBOUNDED, and MCWEIGHTED, were implemented as algorithms separate from the `AutomatonModel` classes. The `StringModelCounter` class in the SCSF tool provides implementations for each of these FSA or WFSA specific algorithms. These model counting algorithms each utilize and return `BigInteger` object to allow for much larger integer values which can occur due to longer string lengths and larger alphabets.

FSA Algorithms

While the existing string and predicate operation simulation algorithms from the Java String Analyzer library [12] were sufficient for SE analyses, some operations and predicates required more precise algorithms to be useful in PSE analyses. The SCSF tool includes more precise versions of the following string operation simulation algorithms: Delete, Insert, Prefix, SetCharAt, SetLength, Substring, Suffix, and Trim.

WFSA Algorithms

Since `WeightedAutomaton` objects did not have either the Java String Analyzer library to provide string operation and predicate simulation algorithms, such algorithms were created to operate on `WeightedAutomaton` objects. Some operations and predicates have both precise and imprecise simulation algorithms for `WeightedAutomaton` objects, specifically: Prefix, Replace (for single symbol arguments), Substring, and Suffix. Other operations and predicates with precise simulation algorithms for for `WeightedAutomaton` objects include: Delete, Insert, SetCharAt, SetLength, and Trim. In addition to these precise algorithms, the following string operations and predicates have simulation algorithms for `WeightedAutomaton` objects included in the SCSF tool: Replace (for string arguments), Reverse, ToLowerCase, and ToUpperCase.

4.5 Test Suite

The SCSF test suite contains 6635 unit tests covering the concrete solver, the four automata models, the WFSA implementation, and the symbolic string algorithms created for the SCSF tool. The test suite does not cover the other components of the SCSF tool either because the component is a legacy component from the String Solver Analysis Framework [21] or because the component is primarily serving a straightforward coordinator which is not complex enough to benefit from unit testing coverage. The unit test in the SCSF test suite were created primarily as part of a test-driven-development [2] and follow a *Given-When-Then* class and method style [14] to aid comprehension of each unit test. To ensure the independence of the objects under test, the mockito library is used to provide test doubles for the interfaces and abstract classes upon which many components within SCSF tool depend. This test suite is able to provide easy to understand verification that the components are correct and perform as expected indecent of the evaluation detailed in Chapter 5.

4.6 Utility Scripts

The final portion of the SCSF tool is actually external to the tool. The utility scripts accompany the SCSF tool and consist of five primary Python script files and other supporting utility Python scripts which automate the evaluation described in Chapter 5. One of these scripts, `run.py`, orchestrates the other four automation scripts allowing the entire evaluation process to be invoked using a single script with appropriate command line arguments specified (documentation of the command line arguments is built into this and all other scripts). The four other primary scripts are: `generate_graphs.py`, `run_solvers_on_graphs.py`, `gather_results.py`, and `analyze_results.py`. The `generate_graphs.py` script automates the generation of the synthetic SCG dataset discussed in Section 4.3.2. The `run_solvers_on_graphs.py` script automates the execution of the SCSF tool on the specified set of graphs using the specified different reporters and solvers. The `gather_results.py` script automates the collection of the result data from the SCSF tool and generates a single result data file fore each automata model. Finally, the `analyze_results.py` script automates the statistical analysis of the result data and generates the appropriate LaTeX output data tables and gnuplot plots for this data.

CHAPTER 5

EVALUATION

In order to evaluate the suitability of different automata models for modeling symbolic string constraints in PSE, each automata model is evaluated in a set of controlled experiments. These experiments consist of each automata model and a model counting oracle solving a series of synthetic string constraints. The result data from these experiments is used to measure the accuracy and performance of each automata model in PSE analyses. Additionally, the independent variables of the evaluation (detailed later in Section 5.2) are analyzed in isolated experiments to determine their effects on accuracy and performance for the automata models.

This chapter is organized as follows. Section 5.1 details the procedure used to conduct the evaluation. Section 5.2 explains the independent variables of the evaluation. After that, the next two sections of this chapter describe the measurements recorded in the evaluation experiments which are be used to determine the suitability of the automata-based constraint solvers: Section 5.3 describes the measurement of accuracy and Section 5.4 describes the measurement of performance. Section 5.5 reports the additional data analyses which compare accuracy and performance metrics. Finally, Section 5.6 concludes the chapter by detailing the evaluation environment.

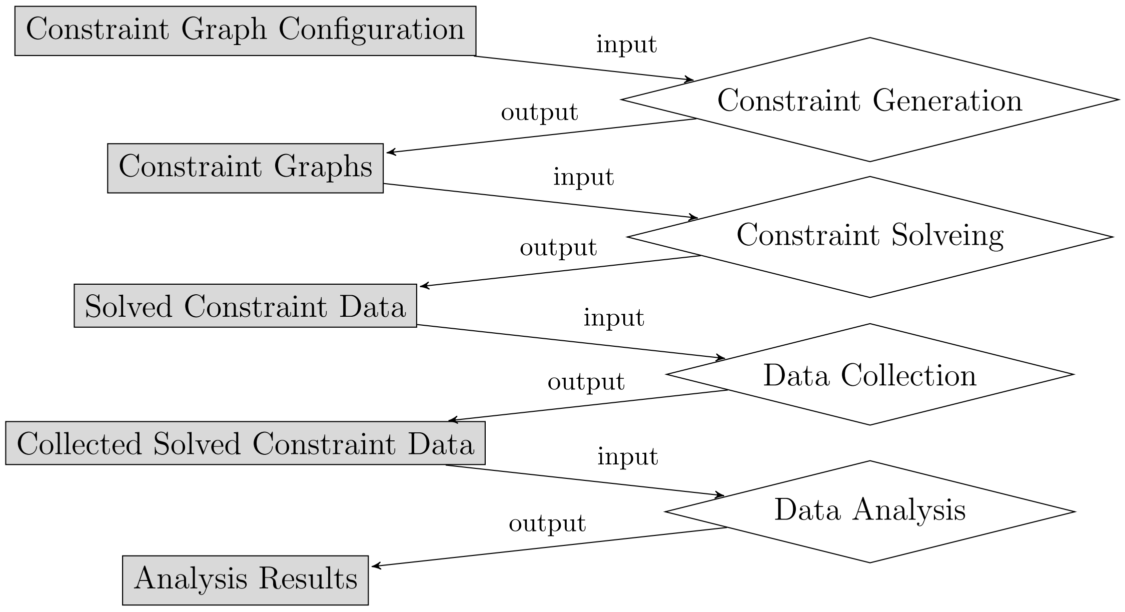


Figure 5.1: The Evaluation Procedure

5.1 Evaluation Procedure

Figure 5.1 is a diagram illustrating the evaluation procedure and the associated input and output data for each step in the procedure. The four diamonds on the right side of the diagram and consist of the *Constraint Generation*, *Constraint Solving*, *Data Collection*, and *Data Analysis* steps in the evaluation. The five grey rectangles on the left side of the diagram represent the different inputs and outputs for each step of the evaluation procedure where the output of one step is becomes the input of the next step in the evaluation, except for the the output of the *Data Analysis* step which are the results of the evaluation and are discussed in Chapter 6.

5.1.1 Constraint Generation

The *Constraint Generation* step is responsible for creating the inputs to the evaluation's controlled experiments. These inputs are the *Constraint Graphs* and consist of multiple sets of synthetic string constraints that are generated as graphs in the format described in Section 4.3. A constraint graph is created from a collection of independent constraint subgraphs. These sub-constraints are in turn created from a combination of parameters as specified by the *Constraint Graph Configuration*. These parameters are the independent variables in the evaluation and are later explained in Section 5.2. The graph generation begins by reading each the *Constraint Graph Configuration* into memory. Next, nodes representing operations and predicates and edges representing the targets and arguments of the operations and predicates are created according to the configuration parameters. These nodes and edges form string constraint graph structures which correspond to the individually constraints to be analyzed in the PSE analysis. These constraints are then collected into the *Constraint Graphs* which consist of multiple JSON files. The *Constraint Generation* step is automated by a python script for repeatability and ease of use. Next, constraint solvers use these constraint graphs in the *Constraint Solving* step of the evaluation procedure.

5.1.2 Constraint Solving

The *Constraint Solving* step is responsible for performing the PSE analyses on each of the *Constraint Graphs*. Each PSE analysis of a graph is conducted for each of the four automa-based constraint solvers as well as the concrete solver oracle. These PSE analysis are performed by SCSF (Chapter 4). This process invokes SCSF for each combination of generated graph and solver. Each invocation of SCSF produces a tab-delimited text file as *Solved Constraint Data*. The data columns reported by SCSF are listed and explained in Table 5.1. The *Constraint Sequence* column values are of special note because of the additional data embedded in these values which are

Column	**Explanation**
Constraint Id	The integer which uniquely identifies the constraint within its parent constraint graph. This id corresponds to the node id of branch predicate of the constraint graph.
Constraint Sequence	A string value specifying the sequence of operations and predicates which comprise the sub-constraint. This value contains additional information about the operations and predicates in the sequence.
Cumulative Time	The total time required to perform all dependent string operations and string predicates.
Before MC	The model count before the terminal predicate of its target symbolic string. This model count represents the number of execution paths reaching the predicate.
false MC	The model count of the *false* branch of the constraint.
false MC Time	The time required to calculate the model count of the symbolic string model representing the *false* branch of the constraint.
false Predicate Time	The time required to model the *false* branch predicate for the two symbolic string arguments.
true MC	The model count of the *true* branch of the constraint.
true MC Time	The time required to calculate the model count of the symbolic string model representing the *true* branch of the constraint.
true Predicate Time	The time required to model the *true* branch predicate for the two symbolic string arguments.

Table 5.1: Data Columns Reported For Solved Constraints

extracted later in the evaluation procedure. The *Constraint Solving* step is automated by another python script and produces a text file for each combination of generated graph and symbolic model.

5.1.3 Data Collection

The *Data Collection* step is responsible for gathering all of the individual result text files which constitute the *Solved Constraint Data* and partitions them into three data groups: model counting accuracy, constraint solving performance, and model counting performance. The collection step begins by reading the result text files into memory. Next, the data tables are grouped by the constraint graphs from which the result tables were produced. This grouping ensures that the *Constraint Id* values in

Column	**Explanation**
File	The name of the result file containing the data result. This corresponds to the generated graph solved to produce the data result.
String Type	The type of the initial input string of the constraint as described in Section 2.4.2.
Operation 1	The first operation in the constraint.
Operation 1 Arg	The category of argument combination received by the first operation in the constraint.
Operation 2	The second operation in the constraint.
Operation 2 Arg	The category of argument combination received by the second operation in the constraint.
Predicate	The branch predicate of the constraint.
Predicate Arg	The input string type of the second string argument for the branch predicate of the constraint.

Table 5.2: New Data Columns Extracted from *Constraint Sequence* Values

each group can be used to identify the same sub-constraint solved using each of the five solvers. Next, an intermediate processing step parses each *Constraint Sequence* value for each data row and extracts the values for 7 new data columns which are added to the results tables. An additional *File* data column is also added to the result tables. These 8 new data columns are defined and explained in Table 5.2. Next, the result tables in each constraint graph group are partitioned into three subtables: a model count data table, a constraint solving performance data table, and an operation and predicate performance data table. In both the model count table and the constraint solving table, each data rows corresponds to a single sub-constraint solved during the PSE analysis. The separate model count data for each of the five solvers are appended to a single output data row under separate data columns. Similarly, the separate constraint solving performance results are appended for each of the solvers. The operation and predicate performance data is also appended to a row, but in this output data table, each row corresponds to a specific operation or predicate in each constraint instead of the constraint itself. Finally, the groups of these subtables are joined to form three result data tables as the *Collected Solved Constraint Data*. This *Data Collection* step is automated as another python script and produces three tab-delimited text files for processing in the *Data Analysis* step.

5.1.4 Data Analysis

The *Data Analysis* step is responsible for transforming the *Collected Solved Constraint Data* into information useful for determining the suitability of the four automata-based solvers in PSE analyses. This *Data Analysis* will also evaluate the effects of the independent variables (detailed later in Section 5.2) on the suitability of each solver. The *Data Analysis* begins by reading the three text files into memory. Next, informative measurements are taken for solver accuracy and performance. The details of these measurements are provided later in Section 5.3 for accuracy and Section 5.4.2 for performance. The *Data Analysis* is also automated by a python script to ensure repeatability and produces *Analysis Results*.

5.2 Independent Variables

The five independent variables of the evaluation listed along with their values in Table 5.3. These variables are introduced in the evaluation through representations in the synthetic *Constraint Graphs*. These independent variables correspond to five characteristics of string constraints discussed in Sections 2.4.2, 2.4.2, and 2.4.2. So that this evaluation is relevant for PSE analyses on actual software programs, the synthetic graphs are generated according to the format described in Section 4.3. The nodes and edges of the graphs are generated representing the independent variable values specified as the *Constraint Graph Configuration* parameters. The representation of

Variable	**Values**
String Alphabet	{`A`,`B`}, {`A`,`B`,`C`}, {`A`,`B`,`C`,`D`}, {`A`,`B`,`C`,`D`,`E`}
Initial Maximum String Length	$1, 2, 3, 4$
Initial Input String Type	*Literal*, *Simple*, *Complex*
Constraint Operations	$\begin{pmatrix} \texttt{reverse}() \\ \texttt{concat}(\mathit{Literal}) \\ \texttt{concat}(\mathit{Simple}) \\ \texttt{concat}(\mathit{Complex}) \\ \texttt{delete}(\mathit{Same}) \\ \texttt{delete}(\mathit{Different}) \\ \texttt{replace}(\mathit{Same}) \\ \texttt{replace}(\mathit{Different}) \end{pmatrix} \times \begin{pmatrix} \mathit{none} \\ \texttt{reverse}() \\ \texttt{concat}(\mathit{Literal}) \\ \texttt{concat}(\mathit{Simple}) \\ \texttt{concat}(\mathit{Complex}) \\ \texttt{delete}(\mathit{Same}) \\ \texttt{delete}(\mathit{Different}) \\ \texttt{replace}(\mathit{Same}) \\ \texttt{replace}(\mathit{Different}) \end{pmatrix}$
Constraint Predicate	`contains`(*Literal*), `contains`(*Simple*), `contains`(*Complex*), `equals`(*Literal*), `equals`(*Simple*), `equals`(*Complex*)

Table 5.3: The Independent Variables in the Evaluation

these independent variables in the graph nodes and edges is described in this section as follows: the *string alphabet* in Section 5.2.1, the *initial maximum string length* in Section 5.2.2, the *initial input string type* in Section 5.2.3, the *string operations* in Section 5.2.4, and finally, the string predicates in Section 5.2.5. Additionally, Section 5.2.6 describes the distribution of the independent variables in the synthetic graphs.

5.2.1 String Alphabet

The *string alphabet* for a constraint is the set of symbols in the language of the strings in the constraint. Section 2.4.2 discusses the properties of the *string alphabet* which characterizes a string constraint. The four *string alphabet* values which appear in this evaluation are listed in Table 5.3. In this evaluation, the *string alphabet* is treated as an interval quantitative variable for the size of the alphabet value. This is because the the algorithms which are used in the *Constraint Solving* step are limited by the size of the alphabet, not the actual symbols in the alphabet.

The alphabet variable is represented in constraint generation by restricting the concrete strings in constraints to the language of strings over the *string operations*. This variable is also represented in the *Constraint Solving* step by restricting the language of string values represented by the symbolic string models of the solvers.

5.2.2 Initial String Length

The *initial maximum string length* of an input strings is the maximum possible length for the string values represented by the primary input string in the constraint. For example, a string input variable is assigned from an external source in a program, the corresponding symbolic string value for this variable will represent all possible string values with a length between 0 and the initial maximum length. Section 2.4.2 describes the *initial maximum string length* as a property of the string length characteristic in a string constraint. The four *initial maximum string length* values which are used in this evaluation are listed in Table 5.3. In this evaluation, the *initial maximum string length* is an interval quantitative variable.

The length variable is represented in the synthetic graphs by ensuring the concrete strings in the graph are not longer than the maximum length. This variable is also used in the *Constraint Solving* step by restricting the input string values represented by the symbolic string models of the solvers to the initial maximum length.

5.2.3 Input String Type

The input string type of a constraint defines the distribution of the string values in the set represented by the symbolic string value. Section 2.4.2 explains the three types of input strings in detail. Table 5.3 lists these three types which are used in this evaluation. Table 5.3 shows these three values used specifically for the initial input string in a constraint. The *initial input string type* in a constraint is the type for the string variable which is the primary input string variable in the constraint. For example, in the constraint `contains`(`concat`(s_1, s_2), s_3), the string variable s_1 is this primary input string variable. In this evaluation, the input string type is a categorical variable because of its three unordered categories.

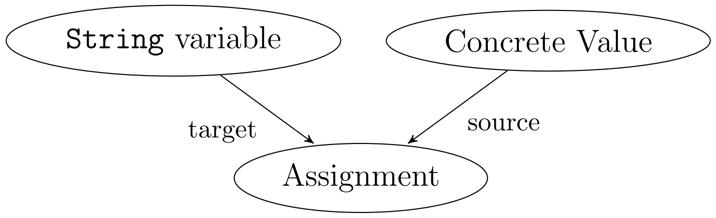


Figure 5.2: *Literal* Input String Type Graph Representation

Literal The created graph structure representing an *Literal* input string type is shown in Figure 5.2. There are three nodes in the structure: one for a string variable, one for an assignment operation, and one for the concrete string value. Two edges connect the string variable and concrete value nodes to the assignment operation node as target and source edges respectively. This structure represents the assignment of the concrete value to the string variable in a program.

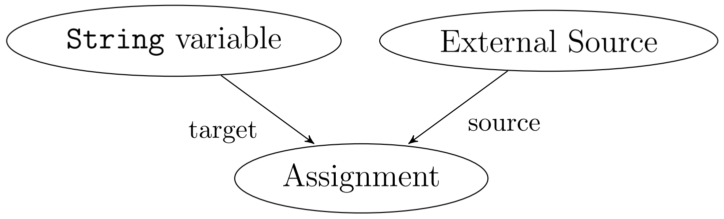


Figure 5.3: *Simple* Input String Type Graph Representation

Simple The created graph structure representing an *Simple* input string type is shown in Figure 5.3. There are three nodes in the structure: one for the string variable, one for an assignment operation, and one for the call to an external process. An external process is any external source of a string value such as standard input, another program, etc.. The target and source edges connect the string variable and external source nodes respectively to the to the assignment operation. This structure represents an assignment of an unknown string value to the input string variable in a program.

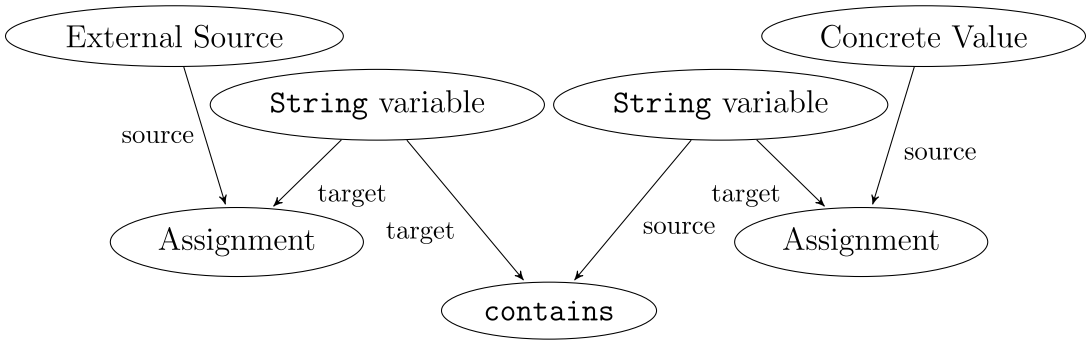


Figure 5.4: *Complex* Input String Type Graph Representation

Complex The created graph structure representing an *Complex* input string type is shown in Figure 5.4. The *Complex* string is constructed using an *Simple* string, a *Literal* string, and a `contains` operation node. The string variable node from the *Simple* structure is connected with an edge to the `contains` operation node as the primary string argument (target). The string variable of the *Literal* structure is connected with an edge to the `contains` operation as the second string argument (source). This structure for the *Complex* input string type results in the contains operation applied to the input string variable from the *Simple* string structure and therefore creating an complex distribution of string values represented for that string variable. As the creation process indicates, the *Complex* input string type is not strictly an initial string input type since it requires both a *Literal* and an *Simple* string type to construct. However, in this evaluation, the entire *Complex* structure is considered as representing the single *Complex initial input string type.*

5.2.4 Operations

The *string operations* in a constraint are the sequence of either one or two operations in the constraint. Section 2.4.2 explains how these operations characterize a constraint and describes the four string operation types included in the generated synthetic graphs: *Injective*, *Additive*, *Subtractive*, and *Substitutive.* Since each string operation in an operation category has similar effects on a symbolic string value, then only one operation from each category is represented in the constraints for this evaluation. These chosen operation are `reverse` for *Injective* operations, `concat` for *Additive* operations, `delete` for *Subtractive* operations, and `replace` with `char` arguments for *Substitutive* operations. Additionally, The `concat`, `delete`, and `replace` operations each takes a sequence of arguments. For each operation, the argument sequences can be categorized into distinct argument configurations based on the similar effects on a symbolic string value. The `concat` operation has three configurations for its single string argument: *Literal*, *Simple*, and *Complex.* The `delete` operation has two configurations for its two integer arguments: *Same* and *Different.* The `replace` operation has two configurations for its two character arguments: *Same* and *Different.* Table 5.3 shows values of the *string operations* independent variable in this evaluation where the variable values are the sequence of either one or two operations produced from the cross product shown in the table. The *string operations* variable is a nominal categorical variable due to the operation sequences being unordered category values.

One of the goals of this evaluation is to determine the effects of the different string operation types on the automata-based solvers. Unfortunately, the *string operations* variable values are operation sequences and can not be used directly to isolate the effects of specific operation types. Instead, different PSE constraint graph sets are used to determine these effects: graphs only containing one operation or operation argument configuration for all constraints and graphs including one operation or operation configuration for all constraints. These two series of experiments allows the effects of the string operation type to be isolated. While the string operation variable is used differently in these two series of experiments, it is still a nominal categorical variable because of the unordered category values.

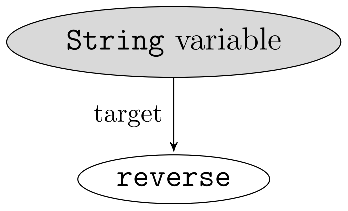


Figure 5.5: `reverse` Operation Graph Representation

`reverse` The created graph structure representing the `reverse` operation is shown in Figure 5.5. There are only two nodes in the structure: one for the target string variable and one for the `reverse` operation. There is a single target edge from the string variable to the `reverse` operation. This structure represents a `reverse` operation in the synthetic constraint graphs.

`concat` The created graph structure representing the `concat` operation is shown in Figure 5.6. There are three nodes in this structure: one for the target string variable,

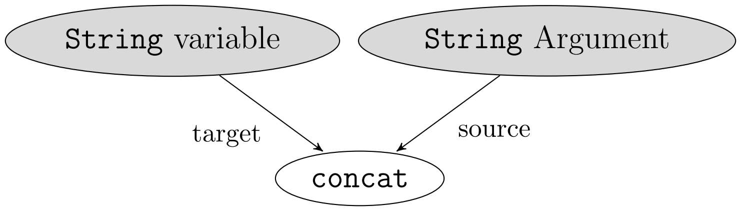


Figure 5.6: `concat` Operation Graph Representation

one for the argument string variable, and one for the `concat` operation. There are two edges in the structure: a target edge connecting the target string variable to the operation and an argument edge (source) connecting the string argument to the operation. The string argument node in this structure represent one of the three input string structures shown in Section 5.2.3: *Literal*, *Simple*, or *Complex*. This is the structure representing a `concat` operation in the synthetic graphs.

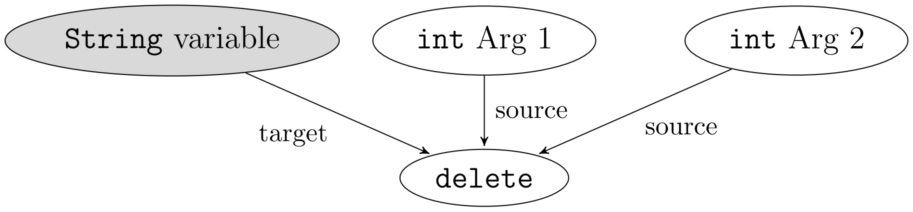


Figure 5.7: `delete` Operation Graph Representation

`delete` The created graph structure representing the `delete` operation is shown in Figure 5.7. There are four nodes in this structure: one for the target string variable, one for the first `int` argument (the start index), one for the second `int` argument (the end index), and one for the `delete` operation. The target edge connects the string variable to the operation node. Two source edges connect the two argument nodes to the operation node. These source edges are labeled `s1` and `s2` in the synthetic graph files.

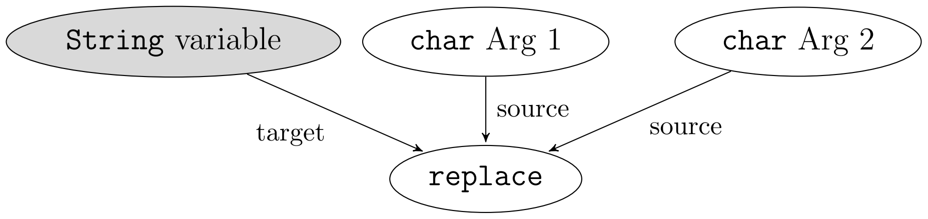


Figure 5.8: `replace` Operation Graph Representation

`replace` The created graph structure representing the `replace` operation is shown in Figure 5.8. This structure contains four nodes: one for the target string variable, one for the first `char` argument, one for the second `char` argument, and one for the `replace` operation. There is a target edge connecting the string variable and the operation. There are two source edges connecting the two argument nodes and the operation node. These source nodes are labeled `s1` and `s2` in the generated graph files.

5.2.5 Predicates

The *predicates* in a constraint are the terminal predicate as well as any other predicate in the constraint. In this evaluation, only the terminal predicates will be considered *predicates* since all other predicates in the *Constraint Graphs* are the `contains` predicates needed to create *Complex* input strings. Section 2.4.2 explains how predicates can characterize constraints and describes the two types of predicates which are used in this evaluation: *Partial Match* and *Full Match* predicates. One predicate is chosen from both categories to be included in the *Constraint Graphs*. The `contains` predicate represents the *Partial Match* category and the `equals` predicate represents *Full Match* category. Both of these predicates have a target a string and takes one string argument. The string argument in this evaluation is always one of three input string types: *Literal*, *Simple*, and *Complex*. Table 5.3 shows these six predicate configurations as the *predicates* variable values. The *predicates* variable is a nominal categorical variable in this evaluation due to the predicate configurations being unordered category values.

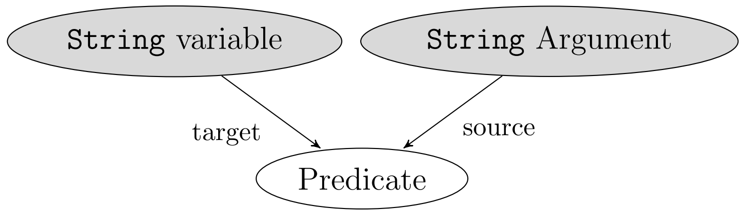


Figure 5.9: Predicate Graph Representation

Figure 5.9 shows the created graph structure for both the `contains` and `equals` predicates. This structure contains three nodes: one for the target string variable, one for the argument string variable, and one for the predicate. An edge connects the target string variable to the predicate while a source edge connects the argument string variable to the predicate. to the operation and an argument edge (source) connecting the string argument to the operation. The string argument node in this structure represent one of the three input string structures shown in Section 5.2.3. This structure is created to represent both the `contains` and the `equals` predicates in the synthetic graphs.

5.2.6 Distribution of Independent Variables in Graphs

Four of the five independent variables in this evaluation are evenly distributed through the *Constraint Graphs* so that the results of the experiments are not skewed by the effects of any particular variable value. These four variables are the *string alphabet*, the *initial maximum string length*, the *initial input string type*, and the *predicates*. For each *string alphabet* and *initial maximum string length*, a constraint graph file is generated where all constraints in the graph file have one of three initial input string variables with the corresponding alphabet and maximum string length. The three initial variables correspond to the three *initial input string type* values. Each of these graphs a all combinations of string operation sequences. Each of these sequences is duplicated twelve times so that each *string predicate* configuration is applied to all operation sequences.

Operation	**Distinct Operations**			
`reverse`()	1			
	Distinct Operations per Alphabet			
	{`A`, `B`}	{`A`, `B`, `C`}	{`A`, `B`, `C`, `D`}	{`A`, `B`, `C`, `D`, `E`}
`concat`($Literal$)	2	3	4	5
`concat`($Simple$)	1	1	1	1
`concat`($Complex$)	1	1	1	1
	Distinct Operations per Length			
	1	**2**	**3**	**4**
`delete`($Same$)	2	3	4	5
`delete`($Different$)	1	3	6	10
Operation	**Distinct Operations per Alphabet**			
	{`A`, `B`}	{`A`, `B`, `C`}	{`A`, `B`, `C`, `D`}	{`A`, `B`, `C`, `D`, `E`}
`replace`($Same$)	2	3	4	5
`replace`($Different$)	2	6	9	16

Table 5.4: The Number of Distinct Argument Combinations for String Operations

Unlike the other independent variables in the evaluation, the *string operations* variable is not evenly distributed through the *Constraint Graphs*. This is due to the different number of distinct operation configurations for the different chosen operations. This is further complicated by the different number of distinct operation argument configurations for different *string alphabet* and *initial maximum string length* values. These differences for the chosen operations is shown in Table 5.4. The table illustrates that the number of distinct `replace` and `concat` operations in a synthetic graph is dependent on the *string alphabet*. The number of distinct `delete` operations in turn is dependent on the *initial maximum string length.*

To ensure that any one *string operations* value does not skew the evaluation results, a data normalization process is applied the *Collected Solved Constraint Data.* This normalization applies integer weights to each result data row corresponding to the operations and argument configuration in the constraint solved to produce the row. For example, the constraint $\texttt{contains}(\texttt{delete}(\texttt{replace}(s_1, \texttt{A}, \texttt{B}), 2, 2), s_4)$ has the *string operations* $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}\}$ and *initial maximum string length* of 4. The corresponding data row will be given a weight of 36. This weight is the product of the weights for the two individual operation argument configurations, 3 for the `replace` operation with *Different* character arguments and 12 for the `delete` operation with *Same* integer arguments. This data normalization allows each of the distinct operation argument configurations to be included in a graph without skewing the result data.

5.3 Accuracy

The extent to which a solver can accurately represent the concrete values of concrete execution is the most important measure for determining its suitability in PSE. In this evaluation, two different measurements of model counting accuracy are taken by comparing the actual model count of the oracle solver to the model counts reported by the four automata-based solvers. This oracle is described in Section 5.3.1. Section 5.3.2 then explains the two different measurements of model counting accuracy. Additionally, Section 5.3.3 addresses the application of these metrics to the independent variable effect isolation experiments.

5.3.1 Oracle / `ConcreteSolver`

In order to determine the accuracy of the model count reported by a solver, an actual model count is needed. In thes evaluation, `ConcreteSolver` described in Section 4.4.3 performs this task. Because `ConcreteSolver` performs its concrete operations and predicate for the *Constraint Graphs* through the same `Solver` interface used by the automata-based solvers, the model counts reported by the `ConcreteSolver` are an oracle of actual model counts for the *Constraint Graphs.* values returned by the concrete evaluation of actual string values provide the oracle for the automata models which symbolically represent this exact concrete process. Just as with the automata models, the oracle records the before branch, the *true* branch, and the *false* branch model count. The model count prior to the branching predicate is recorded as the number of concrete string values, which reach the predicate and serve as the target string for the terminal predicate. The *true* branch model count is the number of concrete strings that evaluate to *true.* Similarly, the *false* branch model count is the number of concrete strings that evaluate to *false.* These three model count numbers are used to measure the accuracy of the solvers.

5.3.2 Measurement

The model counting accuracy of an automata models is measured in two ways: first as the frequency with which the more probable branch determined by the automata model agrees with the actual branch more likely to be executed and second as the model count difference between the actual model count and the count reported by the automata model.

Agreement

When an automata model reports a branch as more probable to execute when the alternate branch is actually more likely to execute, this incorrect report of branch choice produces an invalid analysis as explained in Section 2.2.3. In this evaluation, the actual correct branch choice is provided by the oracle when the branch probabilities computed using Formula 2.1. To determine if the automata model agrees with the choice of more probable branch for a constraint, the *true* and *false* branch probabilities are compared using logical Formula 5.1 to determine agreement. This formula shows that agreement depends on either both the actual *true* branch probability P_t^A and the automata model *true* branch probability P_t being greater than or equal to their respective *false* branch probabilities, P_f^A and P_f, or both *false* branches being more probable than the *true* branches. The frequency of these agreement results is recorded across all constraints for each automata model to measure the probability of an invalid analysis when using the model.

$$(P_t^A >= P_f^A \wedge P_t >= P_f) \vee (P_t^A < P_f^A \wedge P_t < P_f) \implies Agreement \tag{5.1}$$

Percentage difference

Because automata models are representations of the concrete values satisfying a constraint, the actual numerical difference between the model count reported by the oracle and the model count reported by the automata model is not sufficient to measure accuracy. In PSE, model counts are used to determine the probabilities of taking one branch over another. For this reason, this evaluation uses the branch probabilities calculated using Formula 2.1 from the before branch and after branch model counts. The measurement of model counting accuracy (the branch percentage difference) is calculated from both the oracle and automata model branch probabilities using Formula 5.2. The formula calculates the percentage difference P_{diff} as the absolute value of the difference between the actual branch probability P_b^A and the automata model probability P_b. This percentage difference in branch probability measures the accuracy in the representation of concrete values by the symbolic string automata model.

$$P_{diff} = |P_b^A - P_b| \tag{5.2}$$

5.3.3 Effects of Independent Variables

The accuracy measurements of agreement and percentage difference are also applied to the results of the independent variable experiments. These additional experiments allow us to determine if there is a statistically significant effect for each independent variable on the model counting accuracy of each of the four automata models.

5.4 Performance

The second criteria used in evaluating the suitability of automata models in PSE is performance. Performance in this instance refers to the time required to perform the PSE analysis. A PSE analysis can be partitioned into the separate constraint solving and model counting phases for a constraint. While the time required for both of these parts of the analysis contribute to the overall required time, the time required by the constraint solving part has no influence on the time required by the model counting part and vice versa. Because of this separation, the data analysis can evaluate each performance metrics separately as well as their combination. This section is partitioned as follows: Section 5.4.1 explains the constraint solving performance metrics, Section 5.4.2 describes the model counting performance, and Section 5.4.3 details the combination of constraint solving and model counting performance.

5.4.1 Constraint Solving Performance

The constraint solving performance in a PSE analysis is determined by the time required to solve a string constraint for either the *true* or *false* branch of the constraint. This constraint solving time is the sum of the time required to initialize all the input string values to the constraint, the time required to model all of the intermediate operations and predicates for their symbolic string model arguments, and the time required to model either the *true* or *false* branch predicate for the two symbolic string model arguments.

Measurement

To measure the constraint solving time of a constraint, the following three result data columns are used: Cumulative Time, *true* Predicate Time, and *false* Predicate Time. The constraint solving time is calculated for either a *true* branch or a *false* branch and is the sum of the Cumulative Time and the respective branch predicate time. To compare the constraint solving time between the different automata models, the following is calculated from the constraint solving times of each model: average, median, variance, and standard deviation. These calculated values assess the expected relative constraint solving performance for each automata model while also determining the consistency of this performance.

5.4.2 Model Counting Performance

The model counting performance in a string variable PSE analysis is the time required to determine the number of string values represented by the symbolic string model. This model counting process is described in Sections 3.1.1, 3.2.1, 3.3.2, and 3.4.3 for models in this evaluation, the *Unbounded Automaton Model*, the *Bounded Automaton Model*, the *Aggregate Bounded Automata Model*, and the *Weighted Transition Aggregate Bounded Automata Model* respectively.

Measurement

The experimental results for each constraint graph record the time in microsecond required to determine the model count from the model representing the *true* and *false* branches of the constraint. From these model count times, we compute the average, median, variance, and standard deviation of these times for each automata model. These measurements allow a relative performance comparison to be made between the different automata models in terms of general performance and the consistency of such performances. Additionally, due to the almost identical model counting algorithms for each automata model, the relative performance between models will remain consistent for any alternative model counting algorithms.

5.4.3 Combined Constraint Solving and Model Counting Performance

The combined constraint solving and model counting performance is the time required to solve a constraint for either the *true* or *false* branch and to determine

Accuracy Measurement	**Performance Measurement**
Percentage Difference	Constraint Solving Time
Percentage Difference	Model Counting Time
Percentage Difference	Combined Constraint Solving and Model Counting Time

Table 5.5: Accuracy vs Performance Comparison Analyses

the model count of the resulting symbolic string model. The combined constraint solving and model counting time is measured as the sum of the constraint solving time and the model counting time for either the *true* or *false* branch of a constraint. This additional measurement of performance is included in an attempt to evaluate the general relative performance of the four automata models in a PSE analysis.

5.5 Comparison of Accuracy vs Performance

In addition to individual analyses of the accuracy and performance of automata models in PSE, three additional analyses are performed to compare the accuracy and performance criteria for each automata-based solver. Each of the analyses listed in Table 5.5 compares the measurement of accuracy to the measurement of performance for *true* and *false* branches of all constraint in the *Collected Solved Constraint Data*.

The three analyses are performed to determine what relationship if any exists between the accuracy and performance metrics for each solver. The comparisons will provide guidance on which solvers are best when accuracy and performance are equally important for a PSE analysis. Additionally, the same three comparisons are performed for each of the experiments isolating independent variables. The additional comparisons allow the combined effects on accuracy and performance to be determined in their respective proportions.

5.6 Evaluation Environment

The evaluation was conducted on a MacBook Pro$^{\text{TM}}$with a 2 Ghz Intel Core i7$^{\text{TM}}$processor and 8 GB of RAM. The version 1.8 JVM was used for the solving of constraints for each automata model and the oracle. The only change to the default JVM was a specification of maximum heap size which was set as 6 GB for all constraints solved.

CHAPTER 6

RESULTS AND CONCLUSIONS

In this chapter we review the results of the evaluation described in Chapter 5. This chapter begins with Section 6.1 which examines the results of the evaluation with a focus on accuracy (Section 6.1.1), performance (Section 6.1.2), and the balance between accuracy and performance (Section 6.1.1). Section 6.2 offers guidance on the strengths and weaknesses observed for the solvers. Finally, Section 6.3 discusses the known threats to the validity of these results. Section 6.4 covers the conclusions which can be reached from the analysis. Section 6.5 explores possible directions for future work based on this analysis. Finally, section 6.6 concludes this paper with some final thoughts regarding the conducted analysis.

6.1 Results

The results of the evaluation of automata-based symbolic string model suitability are divided into the following three subsections: Section 6.1.1 reviews automata model accuracy, Section 6.1.2 reviews automata model performance, and Section 6.1.1 reviews the comparison of automata model accuracy versus model performance.

6.1.1 Accuracy

Two different automata model accuracy measurements were recorded in the evaluation: the branch choice agreement described in Section 5.3.2 and the percentage difference described in Section 5.3.2. Section 6.1.1 reviews the branch choice agreement results. Section 6.1.1 reviews the percentage difference results.

Branch Choice Agreement

Selection	**Unbounded**	**Bounded**	**Aggregate**	**Weighted**
All	84.6%	99.3%	99.3%	99.8%

Table 6.1: Frequency of Branch Selection Agreement

Overall Table 6.1 displays the measured frequency of branch choice agreement for each of the automata models for all string constraints in the evaluated dataset. This table shows that the *weighted* automata model agrees with the constraint solving oracle more frequently than any other model. Unfortunately, this 99.8% branch choice agreement still leaves a 0.2% of string constraints where an invalid analysis still occurs. Later examination of the effects of the independent evaluation variables will demonstrate why these invalid analyses still occur for the *weighted* automata model.

The *unbounded* model chooses the incorrect branch with the most frequency where it disagrees for 15.4% of string constraints. This disagreement is 14.7% larger than the model with the second largest frequency of disagreement. This indicates that the use of an external maximum length, the only difference between the *unbounded* and *bounded* models, is a significant source of disagreement in *unbounded* automata models. The *bounded* and *aggregate* models both choose the correct predicate branch with 99.3% frequency which indicates that neither the *subtractive* collapse problems for *bounded* models nor the model count over-approximation for *aggregate* models are significant enough to cause a significant number of branch choice disagreements. The percentage difference measurements discussed in Section 6.1.1 will provide a more detailed comparison of the accuracy of these two models.

	Unbounded			**Bounded**			**Aggregate**			**Weighted**		
Selection	**Agree**	Δ	r	**Agree**	Δ	r	**Agree**	Δ	r	**Agree**	Δ	r
$\lvert\Sigma\rvert = 2$	82.6%	-2.0	0.02	98.9%	-0.4	0.01	98.8%	-0.5	0.02	99.6%	-0.2	0.01
$\lvert\Sigma\rvert = 3$	84.5%	-0.1	0.00	99.4%	0.1	0.00	99.3%	0.0	0.00	99.9%	0.1	0.01
$\lvert\Sigma\rvert = 4$	85.8%	1.2	0.01	99.4%	0.1	0.01	99.5%	0.2	0.01	99.8%	0.0	0.00
$\lvert\Sigma\rvert = 5$	85.7%	1.1	0.01	99.6%	0.3	0.02	99.6%	0.3	0.02	99.9%	0.1	0.01
$k = 1$	84.2%	-0.4	0.00	98.8%	-0.5	0.02	98.9%	-0.4	0.02	99.6%	-0.2	0.01
$k = 2$	84.7%	0.1	0.00	99.3%	0.0	0.00	99.3%	0.0	0.00	99.7%	-0.1	0.01
$k = 3$	84.7%	0.1	0.00	99.5%	0.2	0.01	99.5%	0.2	0.01	99.9%	0.1	0.01
$k = 4$	84.9%	0.3	0.00	99.7%	0.4	0.02	99.6%	0.3	0.02	100.0%	0.2	0.02
Literal	82.7%	-1.9	0.02	99.6%	0.3	0.02	99.6%	0.3	0.02	99.6%	-0.2	0.02
Simple	86.5%	1.9	0.02	99.0%	-0.3	0.01	99.0%	-0.3	0.01	99.9%	0.1	0.01
Complex	84.7%	0.1	0.00	99.4%	0.1	0.01	99.4%	0.1	0.01	99.8%	0.0	0.00

Table 6.2: Frequency of Branch Selection Agreement For Different Initial String Characteristics

Initial String Type Table 6.2 displays the measured frequency of branch choice agreement for each of the automata models for the string constraints in the evaluated dataset having the specified initial string characteristic. The Δ column for each automata model provides the difference between the overall agreement and this specific data subset. The r column for each automata model is the correlation coefficient which is a measure of the size of the effect of the independent variables. The guidelines for effect size of r are as follows: a small effect size is 0.1, a medium effect size is 0.3, and a large effect size is 0.5. This table focuses on the three following initial string characteristics: alphabet size, initial string length, and initial string type.

The effects of both the alphabet size and the initial string length on the branch choice agreement are similar for each of the four automata models. None of the effect sizes measured by the r correlation coefficient indicate a significant effect with the largest effect sizes measured at 0.02 which is well below the threshold for even a small effect size at 0.10. A trend seen from the Δ for each of the automata models is the increasing frequency of branch choice agreement as both alphabet size and initial string length increase. Extrapolating from this initial trend for each model, we expect the branch choice agreement frequency to increase for each larger alphabet size and initial string length.

Each of the initial string types has a minimal effect on the frequency of branch choice agreement where 0.02 is the largest measured effect size for a string type. This 0.02 effect size is the result of disagreement for a *literal* initial strings using either *unbounded* or *weighted* models. The only other string type with larger disagreement than overall is for the *simple* type where both the *bounded* and *aggregate* models measure a 0.01 effect size for the increased disagreement frequency.

String Operations Table 6.3 displays the measured frequency of branch choice agreement for each of the automata models for the string constraints in the evaluated dataset which use the specified string operation. The Δ column provides the difference between the overall and data subset percentage difference. The r column shows the effect size for the independent variable. The table shows results for the following four string operations: `concat`, `delete`, `replace`, and `reverse`. The $\exists$ prefix for an operation indicates a data subset which includes at least one instance of the operation in the constraint. The $\forall$ prefix for an operation indicates a data subset which includes only instances of the operation in the constraint.

The branch agreement frequency of the *unbounded* automata model is significantly

	Unbounded			**Bounded**			**Aggregate**			**Weighted**		
Selection	**Agree**	Δ	r	**Agree**	Δ	r	**Agree**	Δ	r	**Agree**	Δ	r
$\exists$ `concat`	75.5%	-9.1	0.11	99.6%	0.3	0.02	99.6%	0.3	0.02	99.9%	0.1	0.01
$\exists$ `concat`(L)	75.0%	-9.6	0.12	99.5%	0.2	0.01	99.5%	0.2	0.01	99.9%	0.1	0.01
$\exists$ `concat`(S)	76.2%	-8.4	0.11	99.7%	0.4	0.03	99.7%	0.4	0.03	99.9%	0.1	0.01
$\exists$ `concat`(C)	74.4%	-10.2	0.13	99.6%	0.3	0.02	99.7%	0.4	0.03	99.9%	0.1	0.01
$\forall$ `concat`	72.1%	-12.5	0.15	99.6%	0.3	0.02	99.9%	0.6	0.05	99.9%	0.1	0.01
$\forall$ `concat`(L)	70.7%	-13.9	0.17	100.0%	0.7	0.06	100.0%	0.7	0.06	100.0%	0.2	0.03
$\forall$ `concat`(S)	74.5%	-10.1	0.13	99.5%	0.2	0.01	99.5%	0.2	0.01	99.5%	-0.3	0.03
$\forall$ `concat`(C)	72.2%	-12.4	0.15	98.8%	-0.5	0.03	99.7%	0.4	0.03	99.7%	-0.1	0.01
$\exists$ `delete`	91.7%	7.1	0.11	99.3%	0.0	0.00	99.3%	0.0	0.00	99.7%	-0.1	0.01
$\exists$ `delete`(s)	88.9%	4.3	0.06	99.6%	0.3	0.02	99.6%	0.3	0.02	99.8%	0.0	0.00
$\exists$ `delete`(d)	95.5%	10.9	0.18	99.1%	-0.2	0.01	99.0%	-0.3	0.02	99.7%	-0.1	0.01
$\forall$ `delete`	99.5%	14.9	0.28	99.6%	0.3	0.02	99.5%	0.2	0.01	99.8%	0.0	0.00
$\forall$ `delete`(s)	99.5%	14.9	0.28	99.8%	0.5	0.04	99.8%	0.5	0.04	99.8%	0.0	0.00
$\forall$ `delete`(d)	99.5%	14.9	0.28	99.5%	0.2	0.01	99.5%	0.2	0.01	99.8%	0.0	0.00
$\exists$ `replace`	87.6%	3.0	0.04	98.9%	-0.4	0.02	98.9%	-0.4	0.02	99.7%	-0.1	0.01
$\exists$ `replace`(s)	89.0%	4.4	0.06	99.5%	0.2	0.01	99.5%	0.2	0.01	99.8%	0.0	0.00
$\exists$ `replace`(d)	87.5%	2.9	0.04	98.1%	-1.2	0.05	98.1%	-1.2	0.05	99.6%	-0.2	0.02
$\forall$ `replace`	97.6%	13.0	0.23	97.9%	-1.4	0.06	97.9%	-1.4	0.06	99.6%	-0.2	0.02
$\forall$ `replace`(s)	99.8%	15.2	0.28	99.8%	0.5	0.04	99.8%	0.5	0.04	99.8%	0.0	0.00
$\forall$ `replace`(d)	96.4%	11.8	0.20	96.9%	-2.4	0.09	96.9%	-2.4	0.09	99.4%	-0.4	0.03
$\exists$ `reverse`	89.0%	4.4	0.06	99.5%	0.2	0.01	99.4%	0.1	0.01	99.8%	0.0	0.00
$\forall$ `reverse`	99.9%	15.3	0.29	99.9%	0.6	0.05	99.9%	0.6	0.05	99.9%	0.1	0.01

Table 6.3: Frequency of Branch Selection Agreement For Different Operations

affected by the `concat` operation. The Δ for each `concat` data subset is negative and each has an effect size greater than 0.10 which is the small effect size threshold. Because each of the exclusive data subsets for the `delete`, `replace`, and `reverse` operations have both positive Δ values and effect sizes just short of the medium effect size threshold of 0.30, we can confirm that branch choice disagreement for the *unbounded* model is the primarily the result of *additive* string operations such as `concat`.

The *bounded* and *aggregate* automata models have almost equivalent branch choice agreement for the different string operation data subsets. Both automata models have the largest disagreement due to `replace` operations where the string constraints which include only `replace` operations with *different* character arguments. The effect size for this data subset is 0.09, only slightly less than the threshold of small effect sizes at 0.10. The only significant difference between the agreement frequency for *unbounded* and *aggregate* models is seen for the exclusive `concat` operations with *complex* arguments where the *bounded* model has a negative Δ and the *aggregate* model has a positive delta.

The effect of string operations on the agreement frequency for *weighted* automata models is negligible where the maximum effect size is 0.03. While there are some negative Δ values for the different string operation data subsets, the smallest agreement percentage of 99.4% indicates the lack of significant effects on *weighted* automata model agreement frequency due to the operations in a string constraint.

	Unbounded			**Bounded**			**Aggregate**			**Weighted**		
Selection	**Agree**	Δ	r	**Agree**	Δ	r	**Agree**	Δ	r	**Agree**	Δ	r
`contains`	98.9%	14.3	0.22	99.2%	-0.1	0.01	99.2%	-0.1	0.01	99.6%	-0.2	0.02
`contains`(L)	97.6%	13.0	0.13	98.0%	-1.3	0.05	98.1%	-1.2	0.05	99.0%	-0.8	0.05
`contains`(S)	100.0%	15.4	0.14	100.0%	0.7	0.03	100.0%	0.7	0.03	100.0%	0.2	0.01
`contains`(C)	99.1%	14.5	0.17	99.6%	0.3	0.01	99.7%	0.4	0.02	99.9%	0.1	0.01
`equals`	70.4%	-14.2	0.17	99.5%	0.2	0.01	99.4%	0.1	0.01	100.0%	0.2	0.03
`equals`(L)	98.7%	14.1	0.14	98.6%	-0.7	0.03	98.5%	-0.8	0.03	100.0%	0.2	0.02
`equals`(S)	45.3%	-39.3	0.30	100.0%	0.7	0.03	100.0%	0.7	0.03	100.0%	0.2	0.01
`equals`(C)	63.0%	-21.6	0.21	99.8%	0.5	0.02	99.8%	0.5	0.02	100.0%	0.2	0.02

Table 6.4: Frequency of Branch Selection Agreement For `contains` and `equals` Predicates

String Predicates Table 6.4 displays the measured frequency of branch choice agreement for each of the automata models for the string constraints in the evaluated dataset which use the specified string predicate. The Δ column provides the difference between the overall and data subset percentage difference. The r column shows the effect size for the independent variable. The table shows results for the `contains` and `equals` string predicates.

The *unbounded* automata model shows very poor agreement accuracy for the `equals` predicates using both *simple* and *complex* types of string arguments. The 0.30 correlation coefficient for the `equals` predicate using *simple* arguments meets the threshold of a medium effect size ($r \geq 0.30$). This effect of disagreement along with the 0.21 effect size for disagreement of `equals` predicates using *complex* arguments account for nearly all of the branch choice disagreement for *unbounded* automata models.

Again the *bounded* and *aggregate* automata models have very similar frequencies of branch choice agreement for the different predicate data subsets. Both of these automata models have larger disagreement due toe predicates using a *literal* string type argument when compared to the positive Δ values for the other argument specific predicate data subsets. However, neither of these disagreements is as significant as the operation data subsets since the 0.05 and 0.03 effect sizes are significantly less than the 0.09 effect size maximum from the operation data subsets.

The predicate data subset for the *weighted* automata model is the most significant of all the data subsets for the model. Every variation of the `equals` predicate has 100% branch choice agreement, demonstrating that all branch choice disagreements arise from `contains` predicates. Specifically, the `contains` predicate for *literal* string type arguments is the only `contains` data subset with a negative Δ with an effect size of 0.05, the largest effect size of any data subset for the *weighted* automata model.

Model Count Percentage Difference

Selection	Unbounded	Bounded	Aggregate	Weighted
Average	9.93	1.02	0.95	0.40
Standard Deviation	27.12	5.54	5.57	4.76

Table 6.5: Model Counting Percentage Difference

Overall Table 6.5 displays the metrics for the difference between model and oracle branch probability for each of the automata models for all string constraints in the evaluated dataset. The model with the largest average percentage difference is the *unbounded* automata model at 9.93 which is nearly ten times the second largest percentage difference of the *bounded* model at 1.02. The *bounded* and *aggregate* are only separated by an average difference of 0.07. The *weighted* model is clearly the most accurate automata model with an average percentage difference of only 0.40, less than half the difference of the *aggregate* model. While each of the *bounded*, *aggregate*, and *weighted* automata models have average differences of 1.02 or less, the standard deviations for each of these models is significantly higher ranging from 5.57 for the *aggregate* model to 4.76 for the *weighted* model. These larger standard deviations indicate that the majority of constraints had no or very small percentage differences. The standard deviation of the *unbounded* automata model at 27.12 indicate the magnitude of the problem with inaccuracy for the model.

	Unbounded			**Bounded**			**Aggregate**			**Weighted**		
Selection	**Avg**	Δ	d	**Avg**	Δ	d	**Avg**	Δ	d	**Avg**	Δ	d
true Branches	16.28	6.35	0.19	0.99	-0.03	0.01	0.88	-0.07	0.02	0.05	-0.35	0.41
false Branches	3.57	-6.36	0.40	1.06	0.04	0.01	1.02	0.07	0.01	0.75	0.35	0.05

Table 6.6: Model Counting Percentage Difference For *true* and *false* Branches

***true* or *false* Predicate Branch** Table 6.6 shows the metrics for the difference between model and oracle branch probability for each of the automata models for the string constraints in the evaluated dataset for either only the *true* branch or only the *false* branch. The Δ column provides the difference between the overall percentage difference and the percentage difference of the specific data subset. The d column uses Cohen's d measurment of effect size which has the following guidelines: a very small effect is 0.01, a small effect is 0.2, a medium effect is 0.5, a large effect is 0.8, a very large effect is 1.2, and a huge effect is 2.0.

The *unbounded* automata model has a clear accuracy deficiency when calculating the probability of the *true* constraint branch shown by the effect size of 0.19 which is just short of the small effect size threshold of 0.20. This increased difference in *true* predicate branches is explained by the inaccuracy of *full match* predicates for *unbounded* models seen earlier in agreement frequency as well as shown later when examining percentage differences for predicated data subsets. Each of the *bounded*, *aggregate*, and *weighted* automata models average larger percentage differences for *false* branches than for *true* branches. While the *bounded* and *aggregate* models have very small effect sizes of 0.01 due to the *false* branchs, the *weighted* model has a larger effect size of 0.05 although that still falls well short of the small effect size threshold of 0.20.

Initial String Type Table 6.7 shows the metrics for the difference between model and oracle branch probability for each of the automata models for the string constraints in the evaluated dataset having the specified initial string characteristic. The Δ column provides the difference between the overall and data subset percentage difference. The d column shows the effect size for the independent variable. This table focuses on the three following initial string characteristics: alphabet size, initial

	Unbounded			**Bounded**			**Aggregate**			**Weighted**		
Selection	**Avg**	Δ	d	**Avg**	Δ	d	**Avg**	Δ	d	**Avg**	Δ	d
$\|\Sigma\| = 2$	10.80	0.87	0.03	1.82	0.80	0.09	1.72	0.77	0.09	0.97	0.57	0.07
$\|\Sigma\| = 3$	10.01	0.08	0.00	0.87	-0.15	0.03	0.82	-0.13	0.03	0.27	-0.13	0.04
$\|\Sigma\| = 4$	9.55	-0.38	0.01	0.79	-0.23	0.05	0.71	-0.24	0.06	0.25	-0.15	0.04
$\|\Sigma\| = 5$	9.34	-0.59	0.02	0.59	-0.43	0.13	0.56	-0.39	0.11	0.12	-0.28	0.14
$k = 1$	12.95	3.02	0.10	1.41	0.39	0.05	1.26	0.31	0.04	0.91	0.51	0.06
$k = 2$	8.70	-1.23	0.05	1.18	0.16	0.03	1.10	0.15	0.03	0.44	0.04	0.01
$k = 3$	8.93	-1.00	0.04	0.83	-0.19	0.05	0.79	-0.16	0.04	0.14	-0.26	0.12
$k = 4$	9.11	-0.82	0.03	0.67	-0.35	0.10	0.66	-0.29	0.08	0.11	-0.29	0.15
Literal	10.12	0.19	0.01	0.42	-0.60	0.11	0.41	-0.54	0.10	0.37	-0.03	0.01
Simple	9.30	-0.63	0.03	1.50	0.48	0.10	1.31	0.36	0.07	0.40	0.00	0.00
Complex	10.33	0.40	0.01	1.14	0.12	0.02	1.12	0.17	0.03	0.42	0.02	0.00

Table 6.7: Model Counting Percentage Difference For Different Initial String Characteristics

string length, and initial string type.

Each of the four automata models demonstrate increasing branch probability accuracy as either the alphabet size or the initial string length increase. This trend of increasing branch probability accuracy is a stronger indication than the branch agreement frequency results for determining that larger alphabet sizes and initial string lengths will produce more accurate automata models.

The *literal* initial string type only increased percentage difference for the *unbounded* automata model and only with the very small effect size of 0.01. The *simple* initial string type increased the percentage difference for both the *bounded* and *aggregate* automata models with 0.10 and 0.07 effect sizes respectively. While this is half and less than half of the small effect size threshold, these effect sizes are some of the largest out of the independent variable data subsets for both of these automata models. The *complex* initial string type has an increased percentage difference for each of the four automata models, but not with any significant effect sizes.

	Unbounded			**Bounded**			**Aggregate**			**Weighted**		
Selection	**Avg**	Δ	d	**Avg**	Δ	d	**Avg**	Δ	d	**Avg**	Δ	d
$\exists$ `concat`	14.94	5.01	0.15	0.94	-0.08	0.02	0.81	-0.14	0.03	0.39	-0.01	0.00
$\exists$ `concat`(L)	14.38	4.45	0.14	0.82	-0.20	0.03	0.85	-0.10	0.02	0.41	0.01	0.00
$\exists$ `concat`(S)	15.51	5.58	0.17	1.24	0.22	0.04	0.86	-0.09	0.02	0.48	0.08	0.02
$\exists$ `concat`(C)	16.04	6.11	0.18	0.82	-0.20	0.04	0.72	-0.23	0.04	0.41	0.01	0.00
$\forall$ `concat`	17.98	8.05	0.22	0.79	-0.23	0.04	0.66	-0.29	0.05	0.60	0.20	0.03
$\forall$ `concat`(L)	16.94	7.01	0.19	0.52	-0.50	0.07	0.52	-0.43	0.06	0.52	0.12	0.02
$\forall$ `concat`(S)	17.53	7.60	0.22	1.35	0.33	0.06	0.79	-0.16	0.03	0.73	0.33	0.06
$\forall$ `concat`(C)	17.92	7.99	0.22	0.67	-0.35	0.07	0.48	-0.47	0.09	0.44	0.04	0.01
$\exists$ `delete`	5.66	-4.27	0.21	0.95	-0.07	0.01	0.91	-0.04	0.01	0.42	0.02	0.00
$\exists$ `delete`(s)	7.23	-2.70	0.11	0.67	-0.35	0.07	0.59	-0.36	0.08	0.33	-0.07	0.02
$\exists$ `delete`(d)	3.46	-6.47	0.45	1.22	0.20	0.03	1.24	0.29	0.05	0.51	0.11	0.02
$\forall$ `delete`	0.95	-8.98	1.35	0.70	-0.32	0.07	0.71	-0.24	0.05	0.35	-0.05	0.01
$\forall$ `delete`(s)	1.27	-8.66	0.90	0.29	-0.73	0.21	0.29	-0.66	0.19	0.29	-0.11	0.03
$\forall$ `delete`(d)	0.73	-9.20	1.88	0.73	-0.29	0.06	0.75	-0.20	0.04	0.36	-0.04	0.01
$\exists$ `replace`	8.27	-1.66	0.07	1.37	0.35	0.06	1.36	0.41	0.07	0.36	-0.04	0.01
$\exists$ `replace`(s)	7.54	-2.39	0.10	0.80	-0.22	0.04	0.71	-0.24	0.05	0.32	-0.08	0.02
$\exists$ `replace`(d)	8.48	-1.45	0.06	2.07	1.05	0.15	2.14	1.19	0.17	0.40	0.00	0.00
$\forall$ `replace`	3.82	-6.11	0.41	1.88	0.86	0.12	1.88	0.93	0.13	0.41	0.01	0.00
$\forall$ `replace`(s)	2.17	-7.76	0.58	0.25	-0.77	0.24	0.25	-0.70	0.21	0.25	-0.15	0.05
$\forall$ `replace`(d)	4.64	-5.29	0.34	2.66	1.64	0.19	2.66	1.71	0.20	0.51	0.11	0.02
$\exists$ `reverse`	7.51	-2.42	0.10	0.81	-0.21	0.04	0.72	-0.23	0.05	0.35	-0.05	0.01
$\forall$ `reverse`	2.25	-7.68	0.56	0.24	-0.78	0.26	0.24	-0.71	0.23	0.24	-0.16	0.05

Table 6.8: Model Counting Percentage Difference For Different String Operations

String Operations Table 6.8 shows the metrics for the difference between model and oracle branch probability for each of the automata models for the string constraints in the evaluated dataset which uses the specified string operation. The Δ column provides the difference between the overall and data subset percentage difference. The d column shows the effect size for the independent variable. The table shows results for the following four string operations: `concat`, `delete`, `replace`, and `reverse`.

The *unbounded* automata model is significantly affected by *additive* string operations as demonstrated by the `concat` data subsets where the effect sizes range from 0.14 to 0.22. These results indicate that over-approximation of the solution sets seen in *unbounded* automata models produces greater inaccuracy than the different collapse problems due to *subtractive* and *substitutive* operations. From these *unbouned* percentage difference results, we can see that the percentage difference due to *subtractive* collapse is significantly less than the difference due to *substitutive* collapse. Specifically, the 0.73 percentage difference of `delete` operation with *different* index arguments provides a positive 1.88 effect size which is closer to the huge effect size threshold at 2.0 than the large effect size threshold at 1.2. Comparing this to the `replace` operation with *different* character arguments which has a 4.64 percentage difference and only a 0.34 positive effect size. This comparison clearly shows that collapses due to *substitutive* operations such as `replace` are a more signifigant source of inaccuracy for automata-based symbolic string models.

The *bounded* and *aggregate* automata models both have the largest difference in branch probability due to the `replace` string operation. The two data subsets for `replace` operations using *different* character arguments produce effect sizes of 0.19 and 0.20 for the *bounded* and *aggregate* models respectively. These results indicate the small but significant effect that *substitutive* collapses have on the branch probability and therefore the model accuracy. As with the agreement frequency, this measurement of percentage difference also indicates less accuracy for a *bounded* model for `concat` operations using *simple* string type arguments.

The *weighted* automata model has only insignificant effect sizes ranging from 0.00 to 0.06 for each of the operation data subsets. While the 0.06 effect size falls well below the small effect size threshold of 0.20, it is the largest effect size of all the operation data subsets indicating minor inaccuracies in branch probabilities due to *additive* operations for *weighted* automata models.

	Unbounded			**Bounded**			**Aggregate**			**Weighted**		
Selection	**Avg**	Δ	d	**Avg**	Δ	d	**Avg**	Δ	d	**Avg**	Δ	d
`contains`	3.27	-6.66	0.46	1.26	0.24	0.03	1.19	0.24	0.03	0.75	0.35	0.05
`contains`(L)	2.19	-7.74	0.86	1.92	0.90	0.10	1.81	0.86	0.10	1.18	0.78	0.09
`contains`(S)	1.04	-8.89	1.04	1.04	0.02	0.00	1.04	0.09	0.01	1.04	0.64	0.08
`contains`(C)	5.49	-4.44	0.22	0.87	-0.15	0.04	0.79	-0.16	0.04	0.22	-0.18	0.07
`equals`	16.58	6.65	0.19	0.78	-0.24	0.07	0.71	-0.24	0.07	0.05	-0.35	0.41
`equals`(L)	1.04	-8.89	2.23	0.91	-0.11	0.03	0.93	-0.02	0.00	0.09	-0.31	0.27
`equals`(S)	24.52	14.59	0.36	0.61	-0.41	0.14	0.45	-0.50	0.18	0.03	-0.37	0.55
`equals`(C)	24.25	14.32	0.37	0.78	-0.24	0.07	0.69	-0.26	0.08	0.04	-0.36	0.57

Table 6.9: Model Counting Percentage Difference For `contains` and `equals` Predicates

String Predicates Table 6.9 shows the metrics for the difference between model and oracle branch probability for each of the automata models for the string constraints in the evaluated dataset which uses the specified string predicates. The Δ column provides the difference between the overall and data subset percentage difference. The d column shows the effect size for the independent variable. The r column shows the effect size for the independent variable. The table shows results for the `contains` and `equals` string predicates.

The *unbounded* automata model has the most significant difference to branch probability due to `equals` predicates for both the *simple* and *complex* types of arguments. The effect sizes of 0.36 and 0.37 are the largest effect sizes for an *unbounded* model due to an increase in percentage difference, demonstrating that *full match* predicates are the largest source of inaccuracy for *unbounded* models. This is consistent with the results for the frequency of branch choice agreement.

Each of the *bounded*, *aggregate*, and *weighted* automata models have increased percentage differences for the `contains` predicates using *literal* and *simple* arguments.

For each of these three models, the `contains` predicate with *literal* arguments is a significant source of percentage difference for the model where it is the second largest effect size for both the *bounded* and *aggregate* models and is the largest effect size of any of the independent variable data subsets for the *weighted* model. The *weighted* model is also significantly affected by the `contains` predicate with *simple* arguments with the second largest effect size of *weighted* models.

6.1.2 Performance

The findings of the evaluation for solver suitability based upon performance criteria appear in the following three subsections: cumulative constraint solving time in Section 6.1.2, model counting time in Section 6.1.2, and combined cumulative constraint solving and model counting time in Section 6.1.2.

Cumulative Constraint Solving Time

Figure 6.1 shows the boxplot for the constraint solving time of each automata model where the y-axis is in microseconds and is on a logarithmic scale. The figure shows that the constraint solving times for the *unbounded* and *bounded* automata models are very similar where the solving time for *bounded* models varies slightly more than *unbounded* models. The *aggregate* automata model requires more time to solve string constraints than either the *unbounded* or the *bounded* models which is to be expected due to the *aggregate* model containing a sequence of FSAs instead of a single FSA. Finally, the *weighted* automata model has significantly worse constraint solving performance than any of the other three automata models. This performance difference is due to the complexity added by weighted transitions when simulating string operations and predicates.

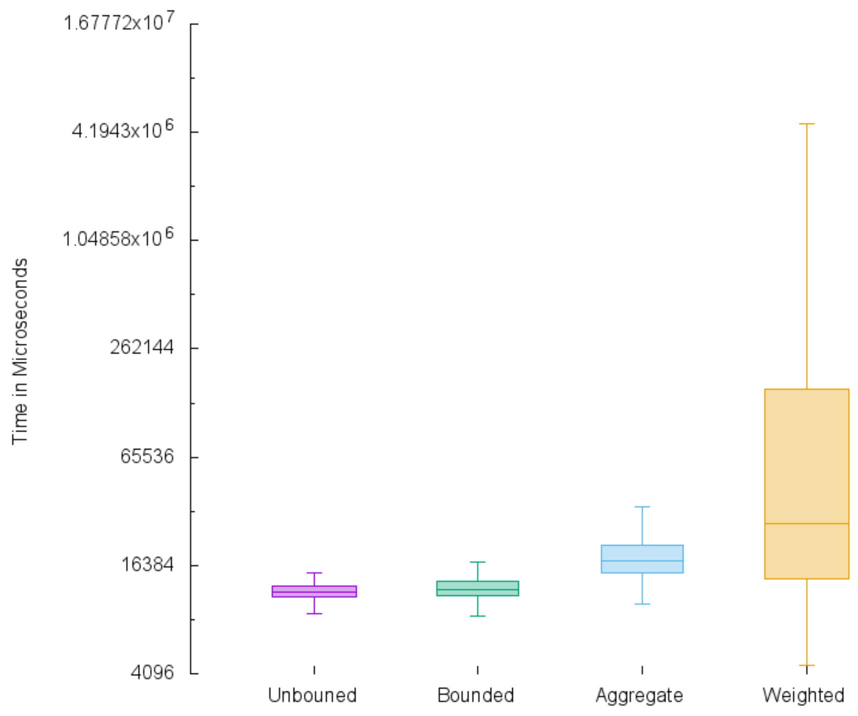


Figure 6.1: Boxplots of Constraint Solving Times

Model Counting Time

Figure 6.2 shows the boxplots for both the model counting times and the constraint solving times of each automata model where the y-axis is in microseconds and is on a logarithmic scale. The boxplots for the model counting times (labeled MC) of each of the four automata models shows that model counting performance is nearly equivalent between models. The box plots for constraint solving time (labeled S) in the same plot shows the extreme difference between model counting and constraint solving times, demonstrating the practical irrelevance of model counting time to the overall analysis time in a quantitative string analysis.

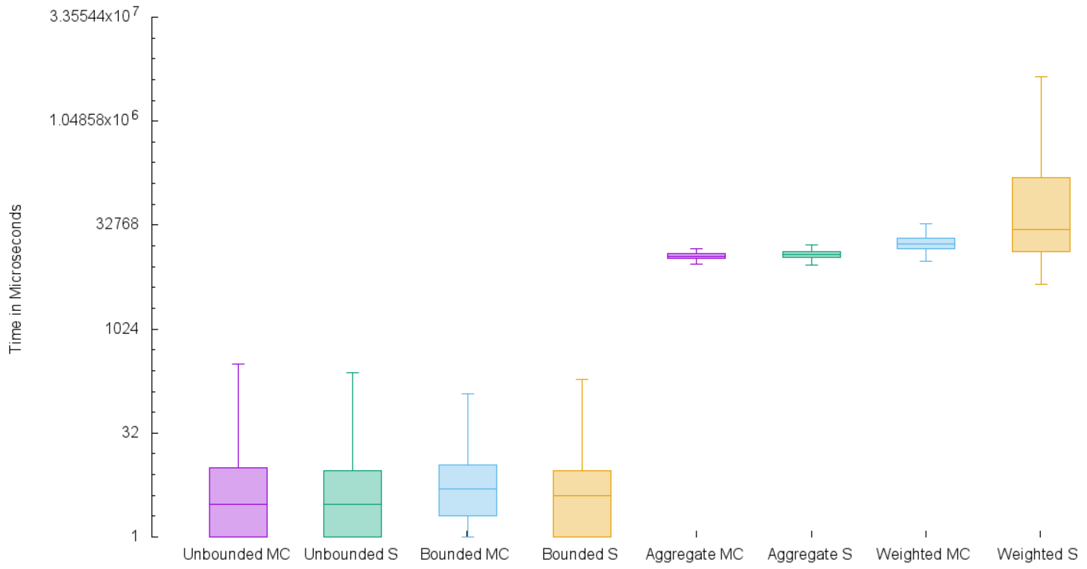


Figure 6.2: Boxplots of Model Counting Times and Constraint Solving Times

Combined Model Counting and Constraint Solving Time

Figure 6.3 shows the boxplot for the combined constraint solving and model counting time of each automata model where the y-axis is in microseconds and is on a logarithmic scale. This boxplot is nearly identical to the boxplot for the constraint solving times for the automata models due to the practical irrelevance of the model counting time when combining it with the constraint solving time.

6.1.3 Accuracy vs Performance Comparisons

The comparisons of the accuracy and performance metrics reported by the evaluation are summarized in the following three subsections: model counting percentages difference versus cumulative constraint solving time in Section 6.1.3, model counting percentages difference versus model counting time in Section 6.1.3, and model counting percentages difference versus combined cumulative constraint solving and model counting time in Section 6.1.3.

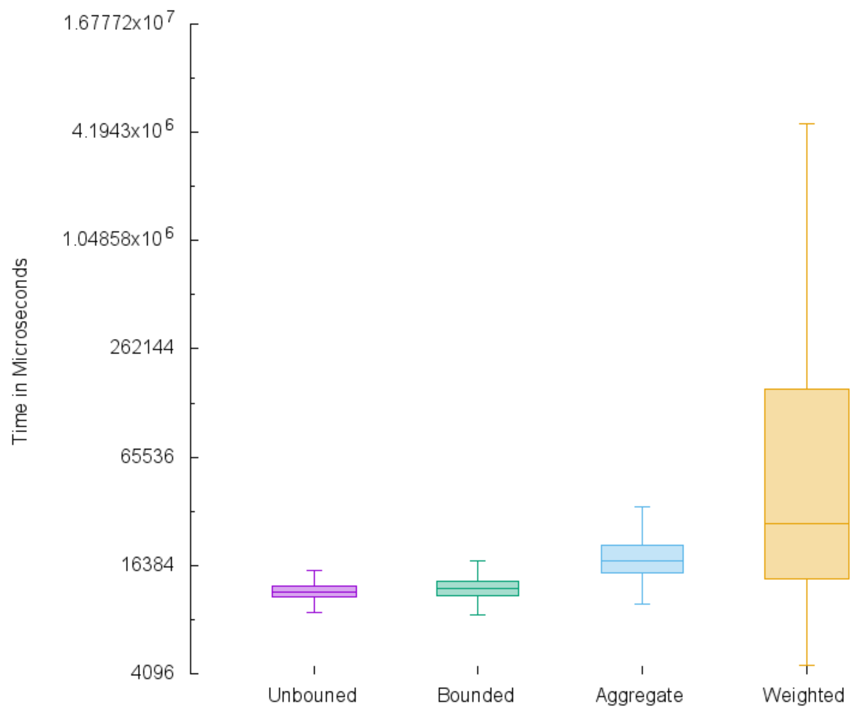


Figure 6.3: Boxplots of Combined Model Counting and Constraint Solving Times

Accuracy vs Constraint Solving Performance

Figure 6.4 shows a scatter plot for all four automata models where the x-axis is the constraint solving time and the y-axis is the branch probability percentage difference, the x-axis is on a logarithmic scale. The scatter plot shows the *bounded* automata model has the best balance of performance and accuracy where the *bounded* model is clustered just left of the *aggregate* model on the log scaled x-axis. This scatter plot also helps to illustrate just how many of the 542592 string constraints for each automata model result in large percentage differences. This was captured by the standard deviation measurements for each model, but the visualization in Figure 6.4 shows that achieving 100% model counting accuracy will remain a difficult problem.

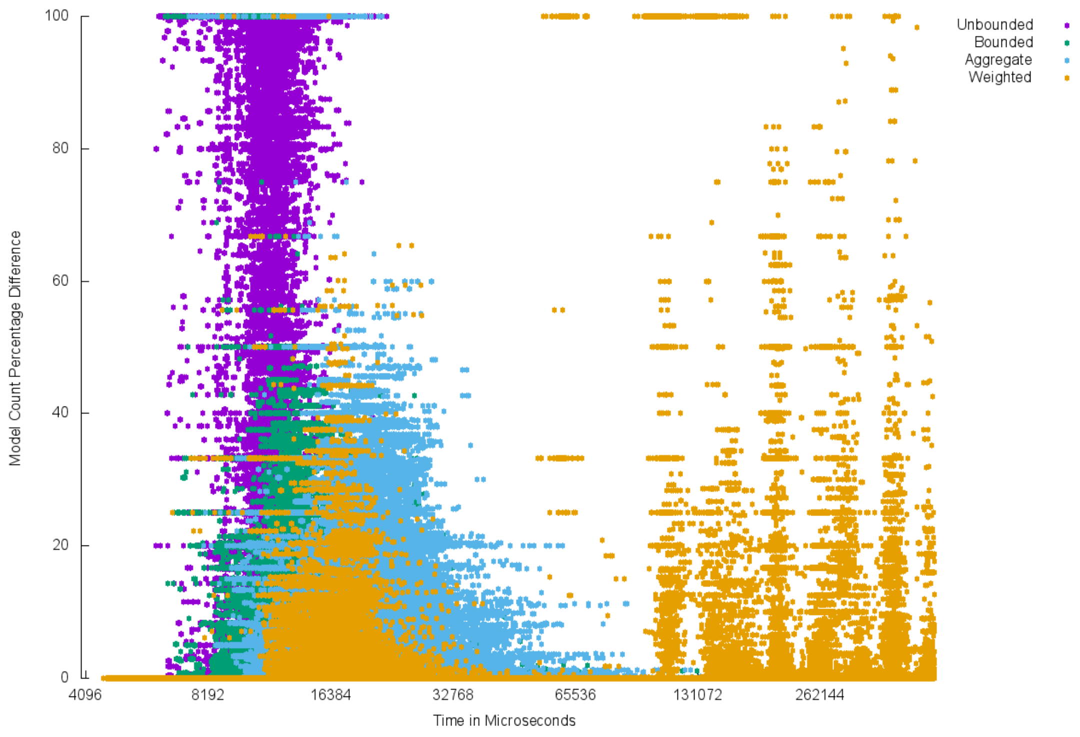


Figure 6.4: Scatterplot of Percentage Difference vs Constraint Solving Times

Accuracy vs Model Counting Performance

Figure 6.5 shows a scatter plot for all four automata models where the x-axis is the model counting time and the y-axis is the branch probability percentage difference. Because the model counting performance of each of the four automata models is practically equivalent, the plot of percentage difference vs model counting time mostly serves as as a visualization of the percentage differences between the automata models.

Accuracy vs Combined Model Counting and Constraint Solving Performance

Figure 6.6 shows a scatter plot for all four automata models where the x-axis is the combined constraint solving and model counting time and the y-axis is the branch

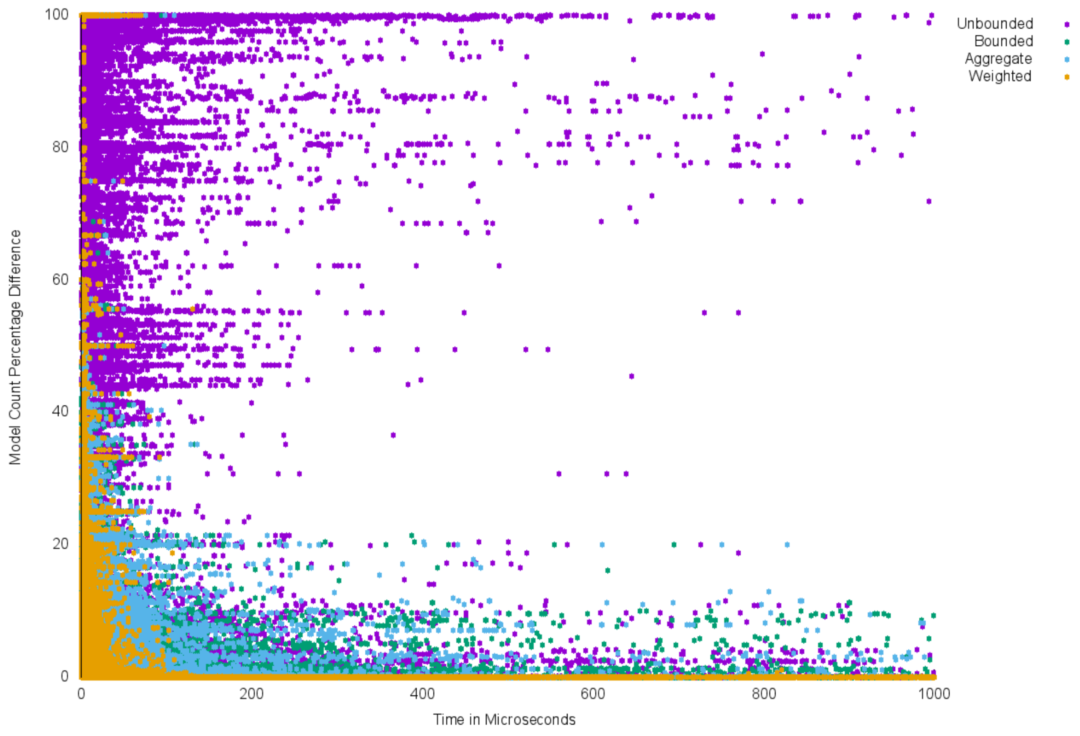


Figure 6.5: Scatterplot of Percentage Difference vs Model Counting Times

probability percentage difference, the x-axis is on a logarithmic scale. Again, due to the practical irrelevance of model counting time when combined with constraint solving time, the scatter plot of percentage difference and combined time is nearly identical to the scatter plot of percentage difference and constraint solving time.

6.2 Recommendations

Criteria	**Model Comparison Relationship**
Accuracy	$Weighted > Aggregate \approx Bounded > Unbounded$
Combined Performance	$Unbounded \approx Bounded > Aggregate > Weighted$
Balance of Accuracy and Performance	$Bounded > Aggregate > Weighted > Unbounded$

Table 6.10: Solver Recommendations Based on Suitability

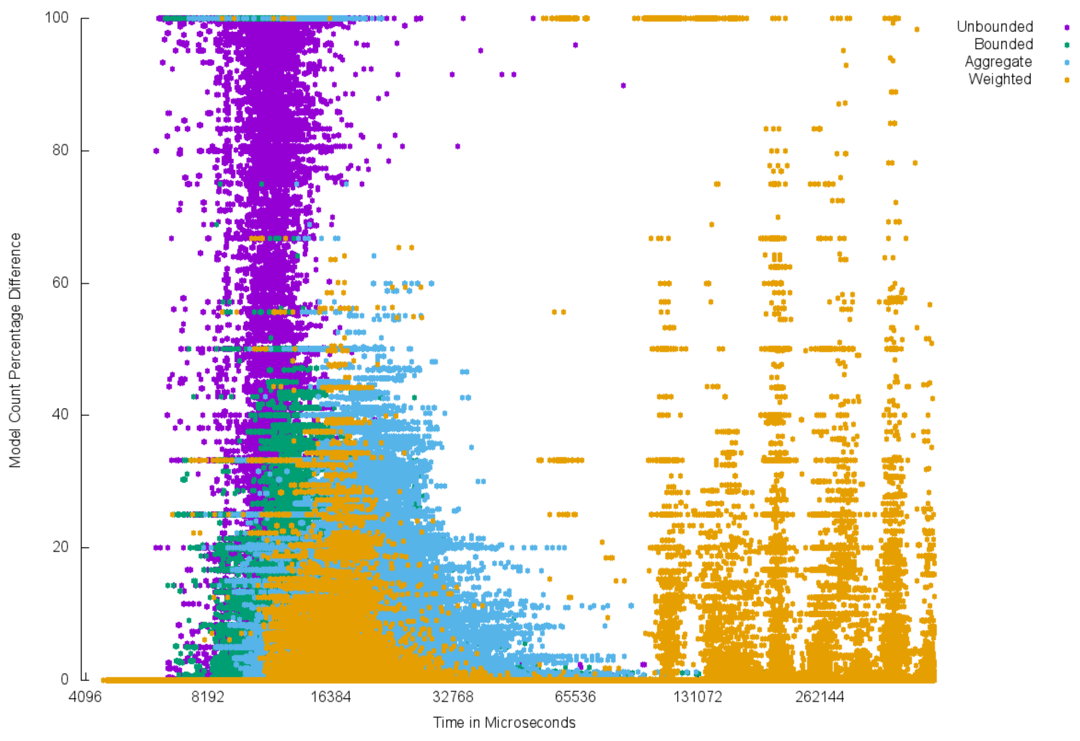


Figure 6.6: Scatterplot of Percentage Difference vs Combined Model Counting and Constraint Solving Times

Table 6.10 summarizes the accuracy and performance results discussed in Sections 6.1.1 and 6.1.2. The model with the best accuracy was the *weighted* automata model which maintained at least a 99% branch agreement frequency across all independent variable data subsets and had a significantly smaller average percentage difference than the other automata models. The *aggregate* automata model had slightly smaller percentage difference metrics than the *bounded* model despite both models having practically equivalent branch choice frequencies. The *unbounded* automata model had significantly worse branch choice agreement frequency and significantly larger percentage differences than the other automata models. Both the *unbounded* and the *bounded* automata models have similar combined constraint solving and model counting times with the *unbounded* model performing slightly better than the *bounded* model. The *aggregate* model has a consistently longer combined times than either the *unbounded* or *bounded* models while the *weighted* model has significantly worse performance than any of the other three models.

When attempting to balance accuracy and performance, the *bounded* automata model has very good combined time and sufficient accuracy for most analyses with 99.3% branch agreement frequency and only 1.02 average percentage difference. The aggregate model provides the second best balance between accuracy and performance with equivalent agreement frequency and slightly better percentage difference but the combined time of the *aggregate* model is significant enough compared to the *bounded* model that the *bounded* is clearly better balanced of the two models. While the *weighted* model is the most accurate of all the automata models, it is not enough of an accuracy improvement to overcome the significant combined time performance cost of the model compared to either the *bounded* or *aggregate* model. The *unbounded* model is has extremely poor accuracy when compared to each of the other automata models making it a worse choice than even the extremely slow *weighted* model.

6.3 Threats to Evaluation Validity

We have identified nine threats to the validity of the evaluation of the suitability of automata-based symbolic string models. These threats include two internal threats to validity and sLiteral external threats to validity.

6.3.1 Internal Threats to Validity

The identified internal threats to the validity of this evaluation include: the small size of the *string alphabet* and the small values for *initial maximum string length.*

Small Size of the *String Alphabet* The small sizes of the *string alphabet* values in this evaluation appear to be a source of selection bias. This is a necessary choice in the evaluation because the time required to complete the evaluation grows quadratically as the alphabet size increases. This makes evaluation of larger alphabet sizes infeasible. To mitigate this risk, four alphabet sizes were chosen so that the effects on accuracy and performance for constraints with larger alphabets can can be extrapolated from the curve produced. Both the branch choice agreement and the percentage difference results indicate that model accuracy increases as the size of the *string alphabet* increases.

Small Values for the *Initial String Length* The small lengths chosen as *initial maximum string length* values in this evaluation appear to be another source of selection bias. This is also a necessary choice due to the evaluation time growing as the initial maximum length increases. As a result, small values are needed for the *initial maximum string length* if an evaluation to be feasible. As with the size of the *string alphabet*, the accuracy and performance suitability of the solvers for larger *initial maximum string length* values can be extrapolated from the curve produced for the four values evaluated. Both the branch choice agreement and the percentage difference results indicate that model accuracy increases as *initial string length* increases.

6.3.2 External Threats to Validity

The identified external threats to the validity of this evaluation include: the synthetic data set used in the evaluation, the evaluation of only `reverse`, `concat`, `delete`, and `replace` operations, using the `char` argument `replace` operation instead of `String` arguments, the use of only `contains` and `equals` predicates, the creation of *Complex* string types using only the `contains` predicate, the possible sub-optimal string operation modeling algorithms, the sub-optimal model counting algorithms, and the automaton models chosen for the evaluation.

Only `reverse`, `concat`, `delete`, and `replace` Operations Because only the `reverse`, `concat`, `delete`, and `replace` operations appear in the the string constraint graphs, the results of this evaluation may not apply for constraints with other string operations. However, observations from the SCSF test suite and made during early experimental testing show that each operation in a category has similar effects on the automata model count. Additionally, the string operation modeling algorithms for each category of operation have similar performance cost complexity. These two observed similarities indicate that the relative differences the evaluation reports between automata-based solvers are likely to hold for those operations not evaluated from evaluated categories.

`replace` with `char` Arguments Similar to the previous threat, using the `char` argument version of the `replace` operation appearing in the string constraint graphs without the `String` argument version also being included can mean the evaluation finding are not applicable to constraints which include the `String` argument version of the operation. The reason this threat to validity differs from the previous threat is the length alteration that can result from a `String` argument `replace` operation. Since this alteration can not occur for the `char` argument version, it would appear capable of producing different accuracy and performance results for subsequent constraints. However, exploratory testing shows that this does not occur for known string argument values. The use of unknown string argument values as used in the `concat` operation, the `contains` predicate, and the `equals` predicates is a topic for future exploration and is detailed in the *Future Work* section (6.5.2).

Only `contains` and `equals` predicates Because the `contains` and `equals` predicates are the only predicates included in the the *Constraint Graphs*, the results of the evaluation may not be generalizable to constraints with predicates which were not included. However, as with the string operations in the evaluation, the effects observed for automata model count using different predicates within both *partial match* and *full match* categories are similar enough that evaluating more than one predicate from either category was deemed redundant. This also applies to the performance effects of the different predicates where the algorithm complexity of the predicates within either category was similar enough that more than one predicate from either category was again deemed redundant.

Creating *Complex Unknown* String Type Using Only `contains` As a consequence of only using the `contains` operation to create *complex unknown* string types in the string constraint graphs, the evaluation findings may not apply to string constraints where *complex unknown* strings types may occur with different origins. The reason a *complex unknown* string type is needed as an initial string in the evaluation is due to the observed added vulnerability to such string types to over-approximation as well as collapses due to both *subtractive* and *substitutive* string operations. However, since automata models produced from `contains` operations have been proven to experience both the over-approximation and collapse issues, we feel *complex unknown* string types created using only the `contains` predicate is sufficient to characterize the accuracy and performance of automata-based symbolic string models.

Sub-optimal Operation and Predicate Simulation Algorithms The use of string operation and predicate simulation algorithms which are likely to be sub-optimal can result in the evaluation results not being relevant for automata-based solvers using optimal operation modeling algorithms. However, because the SCSF tool implemented the same simulation algorithms for FSAs contained within *unbounded*, *bounded*, and *aggregate* automata models, the comparative accuracy and performance of these three automata models would be consistent with an evaluation using more optimized FSA algorithms. Additionally, because the *weighted* automata model uses string operation and predicate simulation algorithms which use WFSAs similarly to the corresponding FSA algorithms, the similar complexity of these WFSA algorithms allow comparisons to be made between *weighted* automata model accuracy and performance and the other automata models which would be consistent with an evaluation with more optimized WFSA algorithms.

Sub-optimal Model Counting Algorithms Similar to the previous threat, the use of sub-optimal model counting algorithms may result in the evaluation results not applying to solvers which use optimal model counting algorithms. However, the nearly identical computational complexity of the three utilized model counting algorithms MCUnbounded, MCBounded, and MCWeighted allows the relationship between the four automata models regarding accuracy and performance to be consistent with evaluations using more optimal model counting algorithms.

Chosen Automata Models The choices of automaton models is the last identified threat to evaluation validity. It is possible that the results of this evaluation would not be applicable to either significantly modified versions of the four automaton models or entirely new automaton models. However, because the evaluation of automata models is intended to explore and illuminate the different accuracy problems suffered by automata models and how performance is affected by the presence of such problems, the four chosen automata models are sufficient to satisfy this intention. Since these accuracy issues arise from the *unbounded* automata model which is the automata model of choice in string analysis research, any other new or heavily modified automata models would need to address these same concerns as well.

6.4 Conclusions

Our analysis of the suitability of automata-based symbolic string models determined that none of the four proposed automata models were suitable to model string constraints without some amount inaccuracy. Whether or not a particular automata model is suitable for a particular analysis will largely depend on the accuracy and performance requirements of the analysis itself and we hope the results of our evaluation provide enough data to help others in making this determination. While this evaluation of suitability is dependent upon the particular analysis, the significant inaccuracies of the *unbounded* automata model makes it unsuitable for all but the most forgiving string analyses. Because the *unbounded* automata model is essentially the same model used in most ongoing quantitative string analysis research, it is important that unsuitability of the model in any practical quantitative analysis is known.

Some insights into the factors which affect automata model accuracy were observed in the evaluation of the four automata models. Both the branch choice agreement and percentage difference results demonstrate that the characteristics of the initial string do not have any significant effect upon the resulting accuracy metric. In fact, the trend of ever increasing accuracy as both the size of the string alphabet and the initial string length increases indicates that for much larger alphabet sizes and initial string lengths, such as those appearing in analyses of real world software programs, The inaccuracies seen by each of the automata models would be less likely to occur. The main source of inaccuracy for each of the automata models is the string operations and predicates, in particular the *substitutive* operations and the *partial match* predicates. Only the *weighted* automata model did not suffer from significant inaccuracy due to *substitutive* string operations in our evaluation because it was created to prevent *substitutive* string collapses. However, even the *weighted* model had significant inaccuracy due to *partial match* predicates, although the magnitude of these inaccuracies were not as large as those seen for the other three automata models for *partial match* predicates.

From the results of this evaluation, we propose that a greater focus should be applied to three particular areas for automata-based symbolic string model research: improvements in multiset models, optimization of string operation simulation algorithms, and optimization of string predicate simulation algorithms.

6.5 Possible Future Work

While our analysis of the suitability of automata-based symbolic string models provides answers about what is required of a sufficiently suitable automata model, this analysis has raised many possibilities for future testing, additional evaluation enhancements, and other possible techniques to provide symbolic string models.

6.5.1 Additional testing

We would like to expand the variety of string operations and predicates used in this evaluation to include addition *injective*, *additive*, *subtractive*, and *substitutive* operations as well as additional *full match* and *partial match* predicates. However, such an addition would require each additional operation and predicate to be simulated by an algorithm which would need to be created for the *weighted* automata model and possibly for the other three models if no such algorithm currently exists in the Java String Analyzer library.

We would also like to evaluate the string constraint graphs of real world software programs such as the open source software dataset evaluated by Kausler [21]. Unfortunately, such an evaluation would require either significant optimization of the string constraint solver oracle or the use of a super computer if not both. This would also require a significant rewrite of the solver oracle to process very large collections of strings without encountering memory errors as the current version would encounter for such large collections.

6.5.2 Enhancements

We would like add different automata models to the SCSF and evaluation to determine if some other choices about the design of the automata models could result in better accuracy or performance. One such change would be the use of factors and the strategy of restructuring *aggregate* automata models, we would like to explore other possible strategies to handle this problem without introducing the model count over-approximation discussed in Section 3.3.3. Another such change would be to test a version of the *weighted* automata model which does not utilize a sequence of

WFSAs but instead uses only a single WFSA since the weighted-transitions should be sufficient to prevent *subtractive* collapses. While these are the only two known additional models we would like to evaluate, it would be very useful to evaluate other currently unknown automata models in the same way as the four currently implemented models.

We would also like to improve the algorithms required to solve and count the string constraints. While our evaluation ensured that the relative comparisons between each automata model was consistent due to the similar operation and predicate simulation algorithms, we would like to explore the accuracy and performance improvements that could be possible by optimizing these algorithms. One idea in particular would be to utilize concurrent processing for many of these algorithms where a process is applied for each transition or state in a collection. While no amount of this kind of optimization would allow constraint solving time to be similar to model counting time, such optimization would allow for much quicker quantitative string analyses.

Another area we would like to explore is the impacts upon accuracy of string operations and predicates with different combinations of arguments. While we explored some different operation and predicate argument configurations in our evaluation, we did not explore the full depth of such configurations. An example of such a set of argument configurations is seen for the SUBSTRING operation which can have four possible argument configurations: known *start* and *end* indices, a known *start* index but unknown *end* index, an unknown *start* index and unknown *end* index, and unknown *start* and *end* indices. Similar configurations exist for most of the different string operations and predicates and we would like to explore how these different configurations impact the accuracy and performance of a string analysis.

Finally, we would like to enhance our evaluation by adding support for *mixed* *constraint* string operations and predicates. This would allow us to evaluate the impacts upon accuracy and performance of *mixed constraint* operations and predicates. This evaluation could also include comparisons between different implementations of the symbolic models of the other data constraints in an attempt to find the best combination of symbolic string models and other data type symbolic models.

6.5.3 Other

One area of future work that we have shown to be of vital importance to quantitative string analysis is the need for multiset symbolic models. While we have attempted to create one such model with the *weighted* automata model, it is important to evaluate the advantages and disadvantages for creating and using multiset models for other commonly used symbolic string models such as bit-vectors and axiom-based models. Obviously a bit-vector can not be used since the use of only a single bit does not allow the representation of more than a single instance of an element in a set, but other vector based symbolic models could be created in the same manner that we created the weighted-transition finite state automaton from the finite state automaton. Also, the creation of multiset models should not be limited to symbolic string models either, multiset models are needed for all datatypes in quantitative analyses.

Another possible area of future research we can speculate about based on our evaluation is the use of machine learning models to predict both branch choice and global execution probability. By utilizing either our concrete string solver oracle or even an accuracy automata model such as the *weighted* model, a machine learning model could be trained to perform quantitative string analyses with the performance advantages enjoyed by most machine learning models.

6.6 Final Thoughts

We hope that our evaluation of the suitability of finite-state automata to model string constraints in quantitative string analyses has provided useful information to guide both the selection of automata models in analyses and the direction of future string analysis research. We believe we proved the unsuitability of the commonly used *unbounded* automata-based symbolic string model for quantitative analyses to to it being significantly susceptible to accuracy errors. We believe we demonstrated the need to use multiset symbolic models such as the *weighted* automata-based symbolic string model in quantitative analyses. Finally, we believe we demonstrated the need for more research and optimization of the algorithms which simulate string operations and predicates for symbolic string models in quantitative analyses instead of research and optimization of model counting symbolic string models.

REFERENCES

[1] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*, pages 255–272. Springer, 2015.

[2] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[3] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 307–321. Springer, 2009.

[4] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S Păsăreanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In *ACM SIGPLAN Notices*, volume 49(6), pages 123–132. ACM, 2014.

[5] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[6] Janusz A Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12(6):529–561, 1962.

[7] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. Precise analysis of string expressions. In *International Static Analysis Symposium*, pages 1–18. Springer, 2003.

[8] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[9] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Conference on Computer Aided Verification*, pages 365–373. Springer, 1989.

[10] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *ACM SIGPLAN Notices*, volume 46(6), pages 567–577. ACM, 2011.

[11] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162. ACM, 2007.

[12] Asger Feldthaus and Anders Møller. *The Big Manual for the Java String Analyzer*. Department of Computer Science, Aarhus University, November 2009. Available from `http://www.brics.dk/JSA/`.

[13] Antonio Filieri, Corina S Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 622–631. IEEE Press, 2013.

[14] Martin Fowler. Givenwhenthen, August 2013.

[15] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[16] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176. ACM, 2012.

[17] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. Jst: An automatic test generation tool for industrial java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 992–1001. IEEE Press, 2013.

[18] Pieter Hooimeijer and Margus Veanes. An evaluation of automata algorithms for string analysis. In *Verification, Model Checking, and Abstract Interpretation*, pages 248–262. Springer, 2011.

[19] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *ACM Sigplan Notices*, volume 44(6), pages 188–198. ACM, 2009.

[20] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

[21] Scott Kausler. *Evaluation of string constraint solvers using dynamic symbolic execution*. PhD thesis, Boise State University, 2014.

[22] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[23] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. Hampi: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116. ACM, 2009.

[24] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[25] Daniel Kröning, Philipp Rümmer, and Georg Weissenbacher. A proposal for a theory of finite sets, lists, and maps for the smt-lib standard. In *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE*, volume 22, page 24, 2009.

[26] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14*, pages 11–11. USENIX Association, 2005.

[27] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *ACM SIGPLAN Notices*, volume 49(6), pages 565–576. ACM, 2014.

[28] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[29] Quoc-Sang Phan, Pasquale Malacaria, Corina S Păsăreanu, and Marcelo d'Amorim. Quantifying information leaks using reliability analysis. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, pages 105–108. ACM, 2014.

[30] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S Păsăreanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[31] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[32] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. Symbolic execution of programs with strings. In *Proceedings of the South African Institute for*

Computer Scientists and Information Technologists Conference, pages 139–148. ACM, 2012.

[33] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30(5), pages 263–272. ACM, 2005.

[34] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, pages 13–22. IEEE Computer Society, 2007.

[35] Geoffrey Smith. On the foundations of quantitative information flow. In *International Conference on Foundations of Software Science and Computational Structures*, pages 288–302. Springer, 2009.

[36] Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path-and index-sensitive string analysis based on monadic second-order logic. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):33, 2013.

[37] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157. Springer, 2010.

[38] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.

[39] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H Ibarra. Symbolic string verification: An automata-based approach. In *Model Checking Software*, pages 306–324. Springer, 2008.

[40] Fang Yu, Tevfik Bultan, and Oscar H Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 322–336. Springer, 2009.

[41] Fang Yu, Tevfik Bultan, and Oscar H Ibarra. Relational string verification using multi-track automata. *International Journal of Foundations of Computer Science*, 22(08):1909–1924, 2011.

[42] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124. ACM, 2013.