SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Inferring String Properties from Code Property Graphs

Severin Schmidmeier

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Inferring String Properties from Code Property Graphs

Herleitung von Eigenschaften von Strings aus Code Property Graphen

Author:	Severin Schmidmeier
Supervisor:	Prof. Dr. Claudia Eckert
Advisor(s):	Alexander Küchler, Florian Wendland
Submission:	15.03.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Bachelorarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15.03.2023

(Severin Schmidmeier)

Acknowledgments

Thanks everyone!

Abstract

In the last couple of years, I have supervized numerous bachelor's and master's thesis and various seminars. This led to a broad observation of typical questions and issues the students faced when writing their thesis or papers. Surprisingly, they are always quite similar. This template aims to give advise to future sstudents in order to answer the most frequent questions and avoid the most common mistakes. It provides the TUM template which has already been accepted many times, shows the most basic outline and some tips on the contents of each chapter. It further contains some tips on the style of scientific works. An evaluation on a small set of students showed that this guideline can assist in making progress faster. However, we found that we have to keep improving the tips to achieve better results.

Your abstract goes here. The typical structure is:

- Broad description of the current state
- Gap in the current state
- Your contribution

Contents

1 Introduction

1.1 Motivation

The increasing reliance on software applications in various aspects of modern life has led to a growing concern for the security of these applications. Among the many security threats that can affect software, injection vulnerabilities are among the most dangerous and prevalent. According to the Open Web Application Security Project (OWASP), injection attacks, which include SQL injection, LDAP injection, and command injection, are consistently listed as one of the top ten web application security risks [14].

Injection vulnerabilities occur when an attacker is able to insert malicious code or input into an application, often through input fields that accept user input such as search boxes or login forms. This can result in the attacker gaining unauthorized access to sensitive data, executing arbitrary code, or even taking control of the entire system.

To assist developers in spotting injection vulnerabilities in their code, they can use a variety of tools and techniques, including static analysis tools for string values. These tools analyze the source code of an application to identify potential vulnerabilities, including injection vulnerabilities. Static analysis tools are particularly useful because they can detect vulnerabilities that may not be apparent during testing or manual code review. Making developers aware of such issues during development enables them to fix the vulnerability early.

In order to detect injection vulnerabilities, such tools can try to analyze the possible values a string that is passed as a query to e.g. a database can take on.

From these inferred properties, a tool can then assess, whether the analyzed program contains any potential injection vulnerabilities and warn the programmer.

For example, consider a string variable, that is used as an SQL query and is determined to be described by the regular expression `DELETE \* FROM myTable WHERE id='.*'`. This information can be used to issue a warning during a static analysis, because the analyzed program allows for an arbitrary unchecked string to be inserted into the SQL query, which is a severe security vulnerability.

1.2 Contribution

In this thesis, we extend a Code Property Graph (CPG) implementation [20] to increase its capabilities in analyzing string values. We adapt the theoretical approach by Christensen et al. [2], which creates regular languages describing the values of a string, to the present CPG implementation. Christensen et al. work on a different representation of the analyzed code and use the result in a different way. For example they use a different definition of the data flow graph (DFG) and a novel data structure representing the complete analyzed code, from which they extract deterministic finite automatons (DFAs). We only analyze parts of the code that are required for the specific query. Therefore, using their approach requires adaption of the used techniques and solving of new problems like the handling of string operations for our use case. We also want to provide the information to an analyst in a human-readable format, in our case regular expression. To achieve this, we, first combine the mentioned approach with an algorithm by Nederhof [13] to transform the obtained results to automata. Further we use state elimination with a heuristic by Delgado and Morais [4] to convert them to regular expressions for users. We also provide a working proof of concept implementation covering a subset of the Java standard library.

After providing some theoretical background in Chapter 3, we describe the different steps of our approach in Chapter 4. In Chapter 5, we then evaluate the results and benchmark our implementation. There, we also highlight some limitations of our approach and include ideas for future continuation and improvement of the presented design. We present some related work in Chapter 6 before concluding the outcome of this thesis in Chapter 7.

2 Problem Description

The CPG implementation we extend currently has no means of providing information about the structure and contents of strings variables that go beyond propagating literals if they are not changed.

Describing such strings is not trivial, as often at least part of a given string stems from an unknown source, for example runtime user input.

We solve this issue by first describing a given string with a formal grammar, which conservatively approximates the values the string can take. This means, that for a string s, the language generated by the grammar we obtain to describe s, always contains all possible values s can have.

We want to use regular expressions as a final representation, because they are human-readable and therefore allow users to manually evaluate our results for a security analysis.

Since the generated grammars are context free, they generate context free languages, which are a superset of the regular languages accepted by regular expressions. Therefore we can't directly convert the obtained grammars to regular expressions, but rather need to approximate them into regular grammars first.

This approximation poses the challenge, that we need to decide, which information to retain and which parts to change. We also need to account for the effects of operations like a `replace` function on the analyzed strings. We also need to ensure that our approximation stays conservative.

Furthermore, we need to convert the obtained regular grammars to equivalent regular expressions. Since the grammars our approximation creates are not textbook regular grammars, but strongly regular grammars, we need to use algorithms suited to this type of grammar for this conversion.

Since we use automata as an intermediary step in the conversion from grammar to regular expression, we use the state elimination algorithm [1] to convert an nondeterministic finite automaton (NFA) to a regular expression. As we want our results to be human-readable, we need to minimize the length of the resulting regular expression by optimizing the state elimination algorithm.

3 Background

We first provide some background information and notation for formalisms used in the following chapters. This includes formal grammars in 3.1, a specific type of grammar in 3.2, automata in 3.3 and regular expressions in 3.4. In 3.5, we then describe the code property graph we use.

3.1 Formal Grammars

A formal grammar consists of a set of nonterminal symbols N, an alphabet Σ of terminal symbols, a set of production rules $(\Sigma \cup N)^*N(\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$, also called just "productions" and a start nonterminal.

In the context of this thesis, Σ is the set of characters making up the strings of the analyzed programming language.

To define grammars in examples, we use the notation $X \rightarrow Y$ to specify a production rule to transform some X to some Y. We use capital letters for nonterminals and lower case letters for terminals. We also don't specify the start symbol explicitly, but rather the nonterminal on the left hand side of the first given production is considered to be the start symbol.

context free grammars (CFGs) are grammars, where the left hand side of all productions consists of a single nonterminal. Regular grammars are CFGs, where the right hand side of each production consists either of a single terminal $a \in \Sigma$ or of exactly one nonterminal and one terminal. Additionally, the nonterminals on the right hand side are either always the first symbol or always the last symbol on the right hand side. A grammar containing both the productions $A \rightarrow aB$ and $B \rightarrow Ab$ with $A, B \in N \wedge a, b \in \Sigma$ therefore is not regular, as the nonterminal is the last symbol in the first production but the first symbol in the latter.

3.2 Strongly Regular Grammars

$\mathcal{R}$ is the equivalence relation defined on the set of nonterminals N of some grammar:

$$A\mathcal{R}B \Leftrightarrow (\exists \alpha, \beta \in V^* : A \xrightarrow{*} \alpha B \beta) \wedge (\exists \alpha, \beta \in V^* : B \xrightarrow{*} \alpha A \beta) \tag{3.1}$$

Here V is $\Sigma \cup N$, so the set of all symbols, terminal and nonterminal. $\xrightarrow{*}$ is the reflexive and transitive closure of the production relation $\rightarrow$ defined by the set of productions in the grammar. $A \xrightarrow{*} \alpha B \beta$ means, that there exists a sequence of productions starting at the symbol A to produce a set of symbols that contain B. Therefore $\mathcal{R}$ groups all nonterminals into disjoint equivalence classes, where each nonterminal in a class can be produced by each other nonterminal in the class. Those nonterminals are called mutually recursive.

A grammar is strongly regular if the production rules in each such equivalence class are either all right-linear or left-linear.

A production rule is right-linear if it is of the form $A \rightarrow w\alpha$, where w is a sequence of terminal symbols and α is empty or a single nonterminal symbol. Left-linear productions are defined accordingly but the nonterminal is on the left side of the production result. Strongly regular grammars are guaranteed to generate regular languages [11].

3.3 Automata

A deterministic finite automaton (DFA) consists of a set of states Q, an alphabet of input symbols Σ, a transition function $\delta : Q \times \Sigma \rightarrow Q$, an initial state $q_0 \in Q$ and a set $F \subseteq Q$ of accepting states. For a nondeterministic finite automaton (NFA), from a given state multiple states can be reached with the same input, so the transition function is $\delta : Q \times \Sigma \rightarrow 2^Q$, where 2^Q denotes the power set of Q.

We represent automata as graphs, where each state is a node and the transition function is represented by edges labeled with elements of Σ. The start state is marked with an incoming arrow and the accepting states are marked with double circles. An edge in this graph, henceforth also called transition, is denoted as (q_1, a, q_2), where $q_1 \in Q$ is the origin state, $a \in \Sigma$ is the label and $q_2 \in Q$ is the target state of the edge.

3.4 Regular Expressions

We use a regular expression syntax with the following metacharacters from the Java regular expression flavor:

- `*`: Kleene star. Matches the previous character zero or more times.
- `.`: Wildcard. Matches any character.

- `?`: Option. Matches the previous character zero or one time.
- `|`: Choice. Matches either the previous or the following expression.
- `[abc]`: Character class. Matches any of the contained characters (here `a`, `b` and `c`).
- `[^abc]`: Negative character class. Matches any character not contained (here anything except `a`, `b` and `c`).

Expressions are grouped using round brackets and meta characters escaped using single backslashes. In a character class consecutive characters can be abbreviated using `-`, e.g. `[0-9]` to match any digit.

3.5 Code Property Graph

The library[1] we extend in this thesis extracts a Code Property Graph (CPG) out of source code of a set of different programming languages.

A CPG is a directed multi graph, where the nodes represent syntactic elements like simple expressions or function declarations and the edges represent the relations between those elements. The nodes and edges have a list of key-value pairs called properties which contain general information for the element. For example, a node representing a statement in a source file contains the location of the underlying code and an edge representing evaluation order may contain whether the target statement is unreachable. The graph is initially created by language frontends, which create partially connected abstract syntax trees (ASTs), which are then enriched by additional information like the mentioned evaluation order by multiple passes [20].

Users of the library can extend this functionality by adding additional passes, which is how we implement the hotspot collection in this thesis.

While the CPG contains many different types of edges, the most relevant edge type for this thesis are data flow edges, which represent the data flow between different expressions.

```
String s = "xyz";
System.out.println(s);
```

Listing (3.1) Example code

Consider the short code example in Listing 3.1. Here, among others, the following nodes are part of the CPG:

- `Literal`, representing the string literal `"xyz"`

[1]https://github.com/Fraunhofer-AISEC/cpg

- `VariableDeclaration`, representing the declaration and initialization of the variable `s`
- `DeclaredReferenceExpression`, representing the reference to the variable `s` in line 2.

In this example, the data flows from the `Literal` node to the `VariableDeclaration` and from there to the `DeclaredReferenceExpression`.

The nodes connected by those egdes effectively form a subgraph of the CPG, the DFG, from which we then extract the information on string values.

4 Approach and Implementation

4.1 General Approach

The general approach for our implementation is adapted from the one described by Christensen et al. [2]. Conceptually, we first create a context free grammar (CFG) from the DFG in a process described in Section 4.2. The created CFG is then approximated to a strongly regular grammar (SRG) using the Character Set Approximation described in Section 4.3.1 and the Mohri-Nederhof algorithm described in Section 4.3.2. We then transform this SRG into an automaton using Nederhof's algorithm described in Section 4.4. Finally, in Section 4.5 we describe how to create a regular expression from this automaton using the state elimination strategy. Figure 4.1 visualizes this process and the different steps from a graph to different types of formal grammars to a regular expression.

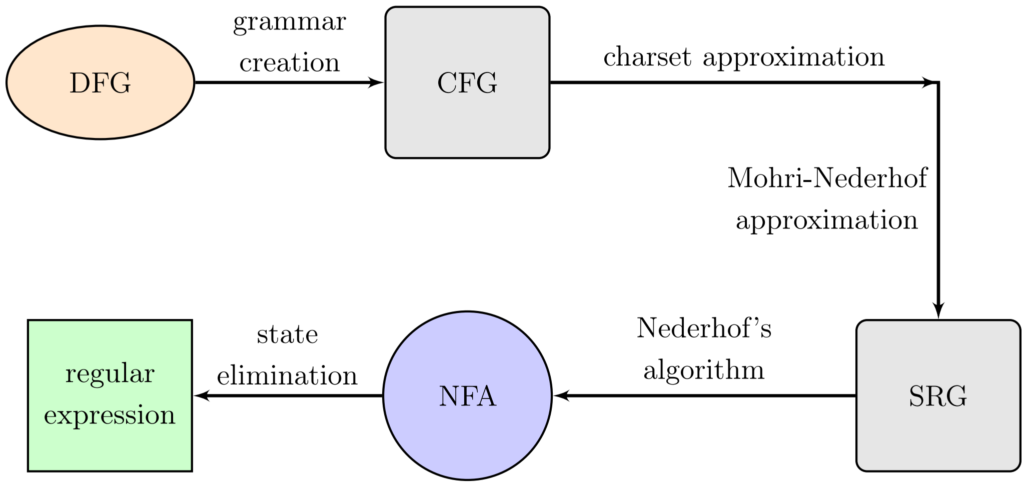

Fig. (4.1) The general approach for obtaining regular expressions

4.2 Grammar Creation

To create the grammar for a given CPG node, we traverse the DFG backwards, starting at the given node. For each visited node, we add a `Nonterminal` and the fitting productions to our grammar.

Our grammar contains the following types of productions:

- `UnitProduction`: $X \to Y$ for references between nodes
- `ConcatProduction`: $X \to Y\ Z$ for concatenation of two nodes
- `TerminalProduction`: $X \to$ `<terminal>` for literal string values and other terminal symbols
- `OperationProduction`: $X \to op(Y)$ for operations on strings

Here, `<terminal>` represents a terminal symbol containing a regular expression that describes a string value and "*op*" is a placeholder for a string operation that is applied to some arguments.

```
String s$^1$ = "␣foo";          1
s$^2$ = s$^3$ + "bar";           2
s$^4$ = s$^5$.trim();            3
```

Listing (4.1) Example code

Consider the code example in Listing 4.1 for the following explanations of the different productions.

`UnitProductions` mostly represent references between nodes where the underlying string is not changed. In Listing 4.1 this would be the case for the reference from `s`^{3} to the variable declaration in line 1.

`ConcatProductions` are created for `BinaryOperator` nodes that represent a string concatenation using the `+` operator. For the example in Listing 4.1 the nonterminal corresponding to the `BinaryOperator` node for the `+` in line 2 would have a `ConcatProduction` with the right hand side nonterminals corresponding to the nodes for `s`^{3} and the string literal respectively.

`TerminalProductions` point to a `Terminal` that represents a fixed regular expression. For example, for the `Literal` CPG node representing the `"bar"` string literal, the corresponding nonterminal has a `TerminalProduction` where the `Terminal` contains a regular expression that matches only the string "abc". `TerminalProductions` also occur at CPG nodes without incoming DFG edges where the value is not known. Those nodes could represent any string value and therefore the corresponding `Terminal` contains the regular lanuage `.*`, matching all strings.

`OperationProductions` represent function calls or other operators. The CPG for 4.1 contains a `CallExpression` representing the function call of the library function `trim`. We then create a `Trim` object representing this operation and the `OperationProduction` $X \to trim(Y)$, where X is the nonterminal corresponding to the node representing `s`^{4} and Y to the one representing `s`^{5}. All operation objects like `Trim` also contain information about possible arguments and implement a character set transformation and an automaton transformation. These transformations describe the effect of the operation on the set of characters making up the words the operation is applied on or automata accepting those words respectively. Examples and how these transformations are used for the approximation are described in Section 4.3. This language agnostic representation of string operation allows developers of the CPG library to add support for functions and operators in other languages with different semantics compared to the corresponding Java functions, without needing to change the grammar approximation. For example for the Python expression `"abc" * 5` the `*` operator can be represented using a generic `Repeat` operation object. For all further steps it is not relevant whether this repeat operation is created from the mentioned Python operator `s * n` or from the corresponding Java function `s.repeat(n)`.

Improvements

Unlike Christensen et al. [2], we do not consider the total DFG when extracting the grammar. They parse the whole graph into a grammar describing all nodes, while we create the grammar starting from a single node and ignore all parts of the graph not connected via DFG edges to this node.

Since often the majority of a large program is not relevant for a specific node, this reduces the amount of nodes we need to handle during analysis. Consequentially, this reduces the size of the resulting grammar, therefore leading to performance improvements.

Additionally, we can traverse the DFG conditionally, stopping at nodes representing numbers. If the traversal reaches such a node,we use an existing analysis that tries to compute the precise value. For example for an integer created by usual arithmetic operations, this analysis can obtain the resulting value that is added to a string. In this case, we can add a `TerminalProduction` with the `Terminal` representing the value literal and otherwise, if the value is not known, the `Terminal` contains a regular expression matching all numbers of the present type, e.g. `"0|(-?[1-9][0-9]*)"` for integrals.

4.3 Regular Approximation

To transform the created grammar into a regular expression, we need to approximate the CFG we obtained like described in the previous section. Since basic regular expressions accept regular languages, the result of this approximation has to be a type of grammar that produces only regular languages to allow direct conversion to regular expressions without loosing information. Using the two different approximation steps described in the following section, the CFG is approximated into a strongly regular grammar.

4.3.1 Character Set Approximation

To use the Mohri-Nederhof approximation algorithm described in Section 4.3.2, we need to eliminate all cycles in our grammar that contain operation productions [11].

First, we view the grammar as a graph in which each symbol of the grammar corresponds to one graph node and for each production there are edges from the nonterminal on the left hand side to all symbols on the right hand side. For two nonterminals A and B there is an edge from the node corresponding to A to the one corresponding to B iff there exists a production of form $A \to \alpha B \beta$ with α and β sequences of arbitrary symbols. This graph allows us to group terminals that are reachable from each other by finding the strongly connected components (SCCs) of the graph.

All nonterminals are assigned a character set, containing all characters that make up the words in the language of the corresponding nonterminal.

$$\begin{aligned}
&S \to replace[c, x](A)\\
&A \to BC|CB\\
&B \to \text{"}ba\text{"}\\
&C \to \text{"}ca\text{"}
\end{aligned}$$

Fig. (4.2) Example grammar

We assign a character set to each nonterminal N using a fixpoint iteration inside the graph component of N that constructs a character set $C(N)$ for N from the character sets of the nonterminals on the right hand side of N's productions.

For productions with terminals on the right hand side, the character set is just the set of all characters occurring in the terminal. For the terminals that are regular expressions, we create the character set when we construct the regular expression. For example for the regular expression representing integers mentioned above, the character set contains all digits and the minus sign. For concatenation productions like $A \to BC$ in the example above, we take the union of the two character sets of the two nonterminals on the right hand side.

For the example grammar in Figure 4.2, B represents the word *ba* and and C represents *ca*, therefore the corresponding sets of characters are {'a', 'b'} and {'a', 'c'} respectively. The words that can be generated from A are combinations of B and C and therefore always contain all characters in the character sets of B and C. Thus, the character set for A is {'a', 'b'} ∪ {'a', 'c'} = {'a', 'b', 'c'}.

Each operation defines a character set transformation - a function $T_{op} : 2^{\Sigma} \to 2^{\Sigma}$ - that approximates how the application of the given operation changes the character set. Here, Σ

represents the set of all possible characters. For example the character set transformation for a `replace` operation, where a known character `o` is replaced by a known character `n` has the character set transformation described in Formula 4.1.

$$T_{replace[o,n]}(S) = \begin{cases} (S \setminus \{o\}) \cup \{n\}, & \text{if } o \in S \\ S, & \text{if } o \notin S \end{cases} \tag{4.1}$$

In comparison, for a `replace` operation, where the newly inserted character is not known, the transformation is defined as shown in Formula 4.2. Here, if the replaced character is contained in S, the set is transformed to Σ, since the newly inserted character could be any element of Σ.

$$T_{replace[o,?]}(S) = \begin{cases} \Sigma & \text{if } o \in S \\ S, & \text{if } o \notin S \end{cases} \tag{4.2}$$

These approximations are used in the fixpoint computation to assign character sets. As mentioned above, in the example in Figure 4.2 the character set for A is {'a', 'b', 'c'}. To obtain the character set of S, we apply the transformation defined by the $replace[c, x]$ operation to set of A, which gives us ({'a', 'b', 'c'} \ {'c'}) ∪ {'x'} = {'a', 'b', 'x'} as the character set of S.

To determine the SCCs, we use Tarjan's algorithm [17]. The SCCs of a graph form a directed acyclic graph (DAG), because if the graph of SCCs would contain a cycle, all contained components would be strongly connected and therefore joined into one component. This DAG implies that there exists a topological ordering of the SCCs [3]. Tarjan's algorithm topologically sorts the returned components in reverse order as a byproduct, which is necessary for the fixpoint computation to terminate. During the computation, for a given nonterminal N, its charset is updated using the charsets of its successors. The reverse topological ordering of the components ensures, that the first handled component is the root in the graph formed by the SCCs, while leafs in this graph are handled last. This ensures that the successors of each nonterminal are either in the same component or in a component that has already been handled earlier.

To break up the cycles containing operation productions, we replace one operation production $X \to op(Y)$ in each cycle with a production $X \to r$, where r is the regular expression that matches the language $C(X)^*$.

To find the cycles in the grammar, we check for each nonterminal N in a given component C, whether it has an operation production, and if yes, whether one of the nonterminals on its right-hand side is also part of C. If this is the case, by definition of SCCs, N is reachable from this nonterminal and therefore the operation production is part of a cycle.

Character Set Implementation

In real world applications, the occurring character sets usually either contain only a few characters, for example the alphanumericals in a string literal, or almost all characters, for example when a single character is removed from an unknown variable represented by Σ.

We also need to efficiently convert both of these types of sets into short regular expressions. A naive implementation like joining all contained characters using the regular expression choice operator would lead to extremely long expressions for very large sets.

To solve this problem and easily represent these two extremes, we have two different implementations, both conforming to a common `CharSet` interface that requires functions like `union : CharSet -> CharSet`, `intersect : CharSet -> CharSet` and functionality for adding and removing characters.

The first, `SetCharSet`, is mostly a simple wrapper around a `Set<Char>` containing the characters. The second, `SigmaCharSet`, is used to easily represent sets like $\Sigma \setminus \{a, b, c\}$ by storing a `Set<Char>` containing the characters *not* contained in the set, while all other characters are assumed to be members.

The behavior of the the set operations `union` and `intersect` can be described using the following set operations:

$$\begin{aligned}
&\texttt{SigmaCharSet union SigmaCharSet} &&\hat{=}(\Sigma \setminus A) \cup (\Sigma \setminus B) &&= \Sigma \setminus (A \cap B)\\
&\texttt{SigmaCharSet union SetCharSet} &&\hat{=}(\Sigma \setminus A) \cup S &&= \Sigma \setminus (A \setminus S)\\
&\texttt{SetCharSet union SetCharSet} &&\hat{=} && S_1 \cup S_2\\
&\texttt{SigmaCharSet intersect SigmaCharSet} &&\hat{=}(\Sigma \setminus A) \cap (\Sigma \setminus B) &&= \Sigma \setminus (A \cup B)\\
&\texttt{SigmaCharSet intersect SetCharSet} &&\hat{=}(\Sigma \setminus A) \cap S &&= S \setminus A\\
&\texttt{SetCharSet intersect SetCharSet} &&\hat{=} && S_1 \cap S_2
\end{aligned}$$

This approach reduces the storage needed to represent the type of character set, where only a few characters are removed from Σ compared to always storing all contained characters. It also simplifies the creation of regular expression from the character set. As mentioned above, always using all contained characters in the regular expression produces large regular expressions for sets with cardinality close to $|\Sigma|$. Using our approach, we can represent the regular expression created from a `SigmaCharSet` using negated character classes and a `SetCharSet` using normal character classes. This reduces the average length of the resulting expressions, compared to always using the same approach. For example the `SetCharSet` that represents the set {’a’, ’b’, ’c’} is used to create the regular expression `[abc]*`, while the `SigmaCharSet` representing $\Sigma \setminus \{'0', '1', '2'\}$ corresponds to `[^012]*`.

4.3.2 Mohri-Nederhof Approximation

Mohri and Nederhof [11] describe an algorithm to approximate a CFG with a SRG.

Recall from the definition of SRGs in Section 3.2, that we can partition the nonterminals of a grammar into equivalence classes based on whether they are reachable from each other. Also recall, that a grammar is strongly regular, if for each such equivalence class, all recursive productions of nonterminals contained in it are either left-linear or right-linear.

For determining if a production rule of a given equivalence class is right- or left-linear all nonterminals that are not part of the class can be considered as terminals. For example a production $A \to CX$ where A and C are nonterminals in the same equivalence class and X is a nonterminal in another class is left linear because X can be viewed as a terminal.

To transform a CFG into a SRG, we only need to transform the sets of mutually recursive nonterminals, where not all productions are either left-linear or right-linear.

Transformation

Mohri and Nederhof describe a more general approach for transforming the required equivalence classes, that accounts for productions with an arbitrary number of nonterminals on the right hand side [11]. Since all productions we use have either one or two nonterminals or exactly one terminal on the right hand side, we can reduce this more general approach to the following algorithm described by Christensen et al.[2].

The approach to transform a given equivalence class M consists of the following two steps:

First, for each nonterminal A in M add a new nonterminal A'.

Second, replace all productions of A with the corresponding new production shown in Figure 4.3. Here B and C are nonterminals in M, X and Y are any nonterminals in a different equivalence class and R is a newly created nonterminal.

Since all newly created productions are right-linear, after applying this transformation to all components where it is required, all components in the grammar either contain only left- or only right-linear productions. Therefore the resulting grammar is strongly regular.

For example, consider a nonterminal A that is part of the currently transformed component and has two productions $A \to XB$ and $A \to BX$, again with B being in the same component as A and X in a different component. Here the first production is right-linear and the second production is left-linear, so the grammar is not strongly regular. Now these productions are replaced according to the rules in Figure 4.3, which gives us the following productions for A: $A \to XB$, $B' \to A'$, $A \to B$ and $B' \to XA'$.

$$\begin{array}{llllll}
A \to X & \leadsto & A \to X\ A' & & \\
A \to B & \leadsto & A \to B, & B' \to A' & \\
A \to X\ Y & \leadsto & A \to R\ A', & R \to X\ Y & \\
A \to X\ B & \leadsto & A \to X\ B, & B' \to A' & \\
A \to B\ X & \leadsto & A \to B, & B' \to X\ A' & \\
A \to B\ C & \leadsto & A \to B, & B' \to C, & C' \to A' \\
A \to \texttt{terminal} & \leadsto & A \to R\ A', & R \to \texttt{terminal} & \\
A \to op(X) & \leadsto & A \to R\ A', & R \to op(X) &
\end{array}$$

Fig. (4.3) Production replacement rules for regular approximation

The new productions of A are all right linear and therefore the grammar can be strongly regular.

Implementation

We can again view a grammar as a directed graph as described in Section 4.3.1.

The notion of mutual "reachability", by which the relation $\mathcal{R}$ defined in Section 3.2 groups the nonterminals, corresponds to SCCs in this graph view of the grammar.

If two nonterminals A and B are mutually reachable in the graph and therefore part of the same SCC, there is a sequence of productions to produce B from A and vice versa, which, by definition of $\mathcal{R}$, means they are in the same equivalence class of $\mathcal{R}$.

Thus, to approximate a grammar we view it as a directed graph and find its SCCs, determine the components, where not all productions are of the same linearity and apply the transformation mentioned above to those components.

4.4 Strongly Regular Grammar to Automaton

In the previous section, we obtained a strongly regular grammar. This grammar is now converted into a regular expression, by first transforming into an automaton like described in this Section. Section 4.5 then describes how this automaton is converted to a regular expression.

4.4.1 Algorithm

Nederhof describes an algorithm to transform a SRG into an equivalent ϵ-NFA [13]. The generated automaton always accepts the same language as the given grammar.

The full algorithm can be seen in Algorithm 1. It creates an NFA $(K, \Sigma, \Delta, s, F)$ with states K, alphabet Σ, transitions Δ, initial state s and accepting states F from a given SRG (Σ, N, P, S) with alphabet Σ, nonterminals N, productions P and a start nonterminal S.

Note that, for the general algorithm an operation production of form $A \to op(X)$ is treated like an unary production of form $A \to X$. The operation productions are always handled by one of the loops in lines 21, 29 or 37, because for any operation production initially contained in a cycle, the cycle is broken up by the character set approximation described in Section 4.3.1. Therefore, e.g. an operation production $C \to op(D)$ with C and D in the same SCC can no longer occur. We provide a detailed description of resolving the effects of the operations in Section 4.4.2.

The MAKE_FA procedure takes two states q_0 and q_1 and a sequence α of symbols - terminals and nonterminals - and creates an automaton equivalent to the grammar starting at α between those two states.

This recursive process is started in line 2 with the start nonterminal S, a newly created initial state s as q_0 and a newly created accepting state f as q_1.

For single terminals and ϵ the algorithm adds an according edge between the two nodes q_0 and q_1 in lines 4 to 7. Here our implementation differs from the original definition because we allow strings and regular expressions as terminals, whereas usually terminals are single characters. We generalize Nederhof's definition by allowing $a \in \Sigma^*$ instead of just $a \in \Sigma$. This changes the type of the generated NFA because it contains edges labeled with multi-character strings instead of just single symbols. This generalization is possible, because we use the resulting automaton as an input to the state elimination algorithm to create a regular expression in Section 4.5. This algorithm uses a generalized NFA definition with regular expressions - and therefore also strings - as edge labels. If we want to use the NFA directly, we can convert it to a usual NFA by replacing an edge labeled with a string of length n with n edges chained together using new intermediate states.

When α contains multiple symbols, a new state q is created and the automaton for the first symbol in α is inserted between q_0 and q and the one for the rest of α between q and q_1. Note that for our use case the rest of alpha always contains at most 1 nonterminal since our productions have at most 2 nonterminals on their right hand side.

If α consists of just a single terminal A, that is not part of any set of mutually recursive nonterminals, so from A there is no sequence of productions to reach A again, we just continue the recursion with the right hand sides of A's productions. The created automaton does not need edges or states corresponding to those single non-recursive nonterminals.

To explain why this is the case, consider the grammar in Figure 4.4 creating just the two words "b" and "c". Here A, B and C are non-recursive nonterminals, so in the

$$A \to B$$
$$A \to C$$
$$B \to b$$
$$C \to c$$

Fig. (4.4) Example grammar with no recursion

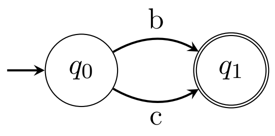

Fig. (4.5) Resulting automaton for the grammar in Figure 4.4

initial procedure call with arguments (q_0, A, q_1) there are just the two recursive calls MAKE_FA(q_0, B, q_1) and MAKE_FA(q_0, C, q_1) in line 38. For those calls again the non-recursive case is chosen, such that for the next recursions α equals b or c respectively, which leads the corresponding edges being created in line 7. As demonstrated no edges or states are created for any of the 3 nonterminals, only for the two terminals a and b and the resulting automaton in Figure 4.5 accepts the correct language.

For the last remaining case, where α consists of a single nonterminal A that is part of some set of mutually recursive nonterminals N_i, the algorithm first adds a new state for each nonterminal in N_i to the graph.

Then we differentiate according to the recursion type of N_i, which is obtained by the call to $recursive(N_i)$.

Note that sets with neither left nor right recursion can be handled by either case.

Now for all productions where the left hand side is a nonterminal in N_i, a recursive call depending on the right hand side of the production is performed.

Definition of the case for right recursive components

Nederhof only defines the case for left recursion in his publication and states that the else part is "the converse of the then part" [13]. This suggests, that besides switching the condition for the second loop from $C \to DX_1 \dots X_m$ to $C \to X_1 \dots X_m D$, switching the order of the states passed to the recursive calls suffices for handling the right recursive case.

However, only changing e.g. MAKE_FA$(q_0, X_1 \dots X_m, q_C)$ to MAKE_FA$(q_C, X_1 \dots X_m, q_0)$ leads to incorrect results. Applying this version of the algorithm to a fully right recursive grammar returns a graph with correct states and correct edges, with the only difference to a correct solution being that the start and the end state are switched. To get correct results, besides switching the argument order, all occurrences of q_0 as an argument to recursive calls need to be replaced with q_1 and vice-versa q_1 with q_0. Algorithm 1 shows this corrected version of Nederhof's algorithm.

To explain the differences between the recursive calls in the different cases consider the grammars in Figures 4.6 and 4.8 and the corresponding automata in Figures 4.7 and 4.9.

The inverting of the states in the recursive calls leads to the edges between states q_2 and q_3 of the automata being inverted, which has no influence on the accepted language.

Switching q_0 and q_1 in the recursive calls is what leads to the needed difference in the resulting automata. In the case of the left recursive grammar, any production sequence of n applications of $B \to Ab$ has to end with replacing the A on the left hand side of the resulting word with the terminal a to finalize the production rule application. This means that each word has to start with a, which is realized in the automaton by adding an edge labeled with a from q_0 to q_3 due to the recursive call in line 21. For the right recursive grammar conversely, each word has to end with an a due to the bs being generated on the left hand side of the A in $B \to bA$. Therefore an edge from q_3 to the finale state q_1 is being added by the recursive call in line 29. Accordingly the corresponding ϵ-edges are added in lines 26 and 34.

$$\begin{aligned} A &\to a \\ A &\to B \\ B &\to Ab \end{aligned}$$

Fig. (4.6) Example grammar with left recursion

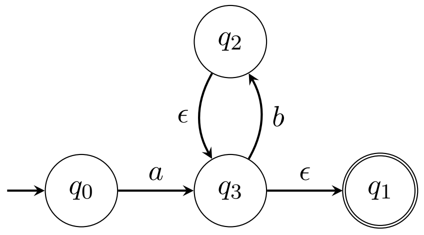

Fig. (4.7) Resulting automaton for the grammar in Figure 4.6

$$\begin{aligned} A &\to a \\ A &\to B \\ B &\to bA \end{aligned}$$

Fig. (4.8) Example grammar with right recursion

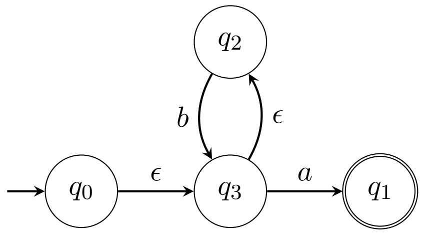

Fig. (4.9) Resulting automaton for the grammar in Figure 4.8

4.4.2 Operation Productions

In the following we use the two Java operations `reverse` and `replace` as examples for operations with different complexities. Note however, that we did not implement the complete list of operations on strings the Java standard library contains. To fully support the standard library, one has to define the transformation described in this section for each operation.

As described above, for the Nederhof algorithm, operation productions of form $C \rightarrow op(X)$ are treated like a normal unary production $C \rightarrow X$.

Each operation defines an automaton transformation that changes a given automaton. The new automaton accepts the language obtained by applying the operation to each word in the language of the input automaton. Consider an operation $replace[old, new]$ corresponding to the Java call `s.replace(old, new)`, that returns a copy of the `String s`, where each occurrence of the `char old` is replaced with the `char new`. The automaton transformation for $replace[old, new]$ traverses the automaton and replaces each occurence of *old* on any edge with *new*.

To apply the effect of the different operations onto the created automaton, we first need to find the sub-automata affected by each operation.

To obtain these sub-automata, we taint all nodes and edges if they are created in recursion calls originating from an operation production. If a recursive call of the Nederhof algorithm MAKE_FA(q_0, X, q_C) in line 21 is caused by an operation production $C \rightarrow op_1(X)$, we pass op_1 as a taint to the recursive call. All edges and states created further down this recursion path will be tainted with op_1. In the resulting NFA, for each operation that is part of the given grammar, there's a set of tainted nodes and edges representing the parts of the automaton affected by this operation. These sets form a sub-automaton of the NFA, onto which the transformation of the corresponding operation can then be applied.

Consider the grammar in Figure 4.10 and the corresponding automaton in Figure 4.11. Here the production $A \rightarrow E$ leads to the creation of the left path including state q_2, while for the operation production $A \rightarrow replace[f, x](F)$ the subsequent algorithm calls create the colored path. All colored edges and states are tainted with the *replace* operation. The created NFA has two similar paths, since $A \rightarrow replace[f, x](F)$ is treated like a $A \rightarrow F$ production analogous to $A \rightarrow E$, just that the resulting edges and adjacent states are tainted.

After completing the NFA creation, we can collect all tainted nodes and apply the automaton transformation defined by $replace[f, x]$ to this sub-automaton consisting of the states q_0, q_3 and q_1.

As mentioned above, for the $replace[f, x]$ operation this transformation consists of replacing all occurrences of f on tainted edges with x, which gives us the automaton in Figure 4.12 for the given example.

For regular expressions as edge labels the replace operations is more complex.

For example `.*`, which is created when we encounter an unknown value like user input, matches all strings, and therefore also strings containing f. After applying the $replace[f, x]$ operation to these strings, they can never contain an f, so therefore the edge label should match all strings that contain no f. This can be implemented using a negative character class, so `.*` is transformed to `[^f]` by the *replace*$[f, x]$ operation. Similarly, existing character classes need to be transformed, for example `[abf]` to `[abx]` and `[^ab]` to `[^abf]`.

$$\begin{aligned}
&A \to E\\
&A \to replace[f, x](F)\\
&E \to eE\\
&E \to e\\
&F \to fF\\
&F \to f
\end{aligned}$$

Fig. (4.10) Example grammar with operation production

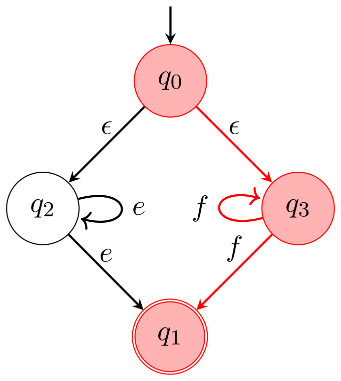

Fig. (4.11) Resulting automaton for the grammar in Figure 4.10

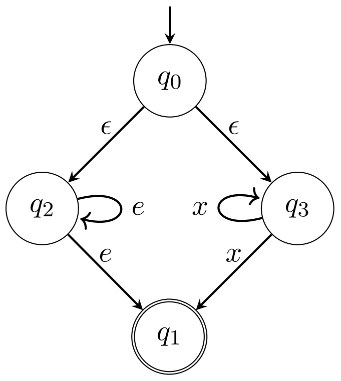

Fig. (4.12) Automaton in figure 4.11 after applying operation transformation

For more complex operation transformations like a *reverse* operation, the operation transformation also includes adding and removing states and edges of the automaton.

Consider the automaton in Figure 4.14, where the colored parts are tainted with the *reverse* operation.

To apply the reverse operation we first duplicate the tainted subautomaton and create a new state for each contained state. Here q_4 is created for q_0, q_5 is created for q_1 and q_6 is created for q_3. We also duplicate all tainted edges alongside the states. The resulting automaton can be seen in Figure 4.15. Now we reverse the direction of all edges in the duplicated subautomaton. For example, after duplication there was the edge (q_6, c, q_5) because the original automaton had an edge (q_3, c, q_1). This edge is now reversed to give us the edge (q_5, c, q_6) that is present in the final automaton.

After reversing all edges, the subautomaton is connected back to the rest using new ϵ edges. Finally all tainted edges between the original states are removed together with all states that are not connected to the automaton anymore after this step. In the example automata, after removing the tainted edges, q_3 is disconnected from all other states and therefore removed.

4.5 Automaton to Regular Expression

To get a humand-readable format for the information we obtained, we transform the automaton we created to a regular expression. Section 4.5.1 describes this conversion and 4.5.2 presents an optimization we used to improve the results.

$S \to A$
$S \to reverse(B)$
$A \to aA$
$A \to c$
$B \to bB$
$B \to c$

Fig. (4.13) Example grammar with operation production

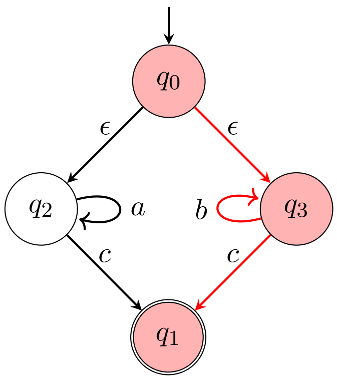

Fig. (4.14) Resulting automaton for the grammar in figure 4.13

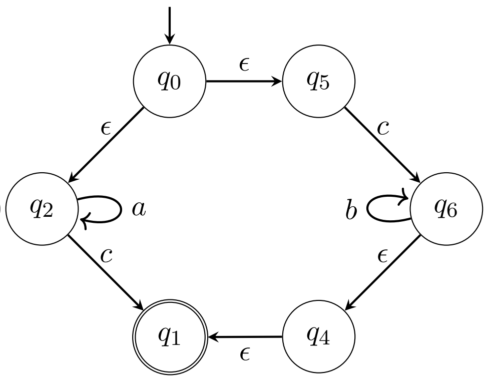

Fig. (4.15) Automaton in figure 4.14 after applying operation transformation

4.5.1 State elimination

To transform the automaton we created from a SRG, we use the state elimination strategy, also known as the Brzozowski-McCluskey procedure [1].

To apply the procedure, an automaton is first transformed into a generalized nondeterministic finite automaton (GNFA). A GNFA is an NFA where the edges are labeled with regular expressions instead of single symbols. Also a GNFA must only have a single start state and a single end state [9].

To achieve this characteristic, one can add a new start state with a single ϵ transition to the old start state and a new finale state with incoming ϵ edges from all previously accepting states.

However, due to the automaton construction using the Nederhof algorithm described above, the automata we obtain already fulfill this property without any need for further modification. We also already use regular expressions as edge labels from the start.

We first replace each pair of edges $(q_0, r_1, q1), (q_0, r_2, q1)$ between two states with a single edge $(q_0, r_1|r_2, q_1)$. After applying this replacement rule exhaustively, there are no two states q_0 and q_1 with more than one direct edge between them.

To eliminate a state q, we "shortcut" the state by replacing all pairs of transitions $(q', r, q), (q, t, q'')$ with a new transition from q' to q''. The new transition is (q', rt, q'') if q has no loop edge to itself and (q', rs^*t, q'') if it has one with label s [6].

Figures 4.16 and 4.17 contain examples adapted from Esparza [6] that visualize those rules.

After repeatedly applying the two rules and eliminating all other states, the resulting automaton contains only the start and the end state. The single edge between those two states then has the resulting regular expression as a label.

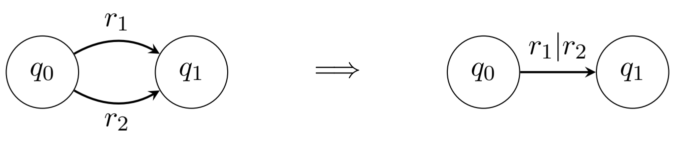

Fig. (4.16) Replacement of edge pairs

.

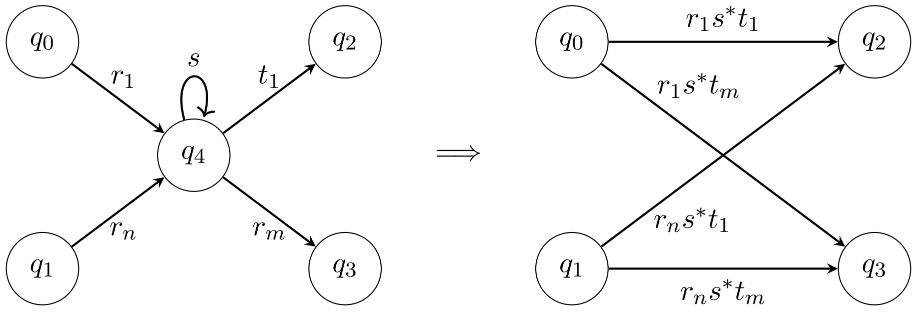

Fig. (4.17) Elimination of state q_4

.

4.5.2 Delgado heuristic

The resulting regular expressions often do not have minimal length, but there exists no efficient algorithm that always produces optimal regular expressions.

Minimizing regular expressions is PSPACE-complete [7]. If there were an efficient algorithm to obtain the minimal regular expression for a given NFA, one could first apply Thompson's algorithm[18] for turning a regular expression into an equivalent NFA and then this algorithm. This chaining would then be an efficient regular expression minimization algorithm, which is not possible as mentioned above.

However, we can still improve the result we obtain using the state elimination method. The order in which states are eliminated affects the size of the resulting regular expression. There exist different heuristics for choosing an elimination order to reduce the expression size.

We chose a heuristic described by Delgado and Morais [4]. For each state a weight is calculated using the following formula, where In_q is the set of incoming edges of q, Out_q the set of outgoing edges, W_e the size of the label on any edge e. Out_q and In_q both do not contain a potential loop on q. W_{loop} is the size of the loop around q if it exists and 0 otherwise.

$$weight(q) = \sum_{e \in In_q} (W_e \times (|Out_q| - 1)) + \sum_{e \in Out_q} (W_e \times (|In_q| - 1)) + W_{loop} \times (|In_q| \times |Out_q| - 1)$$

The weight represents the length of the expression added to the result by removing this state. Therefore, in each algorithm run, the state with the smallest weight is chosen for elimination.

Delgado and Morais show that using this heuristic produces significantly shorter expressions compared to the naive state elimination [4]. Gruber et al. also show it outperforms almost all other heuristics they considered [8]. Improving the algorithm further by implementing a look-ahead additionally to the heuristic also improves the results, but adds more complexity and impairs the algorithms performance [4].

Another reduction of the regular expression size can often be obtained by first converting the automaton to a DFA, for example using the powerset construction. For a given NFA with n states, the DFA obtained by using the powerset construction to convert the NFA can have up to 2^n states. However, we observed that for many NFAs generated using Nederhof's algorithm, the resulting DFAs is significantly smaller than the input NFA or even minimal. This DFA can be minimized using common algorithms like Hopcroft's or Brzozowski's algorithm for even better results.

Since it is unclear, whether conversion to DFAs always leads to better results, we can use both approaches and return the shorter expression.

4.6 Hotspot Collection

We also implemented a new pass that traverses the CPG and collects nodes representing string values which might be of interest for further analysis. This hotspot collection provides common starting points for our grammar creation to the user. However, the grammar creation is completely independent of this collection and a grammar can be created for any string node, independent of whether it is part of the collection. We consider all strings that are passed as a query to the Standard Java SQL API and all strings in return statements as hotspots.

Algorithm 1 Nederhof Algorithm: SRG $(\Sigma, N, P, S) \rightarrow$ NFA $(K, \Sigma, \Delta, s, F)$

```
 1: let $\Delta = \emptyset; s = \texttt{create\_state()}; f = \texttt{create\_state()}; F = \{f\}; K = \{s, f\}$
 2: MAKE_FA$(s, S, f)$
 3: procedure MAKE_FA$(q_0, \alpha, q_1)$
 4:     if $\alpha = \epsilon$ then
 5:         let $\Delta = \Delta \cup (q_0, \epsilon, q_1)$        ▷ add $\epsilon$ transition from state $q_0$ to state $q_1$
 6:     else if $\alpha = a,$ some $a \in \Sigma^*$ then
 7:         let $\Delta = \Delta \cup (q_0, \alpha, q_1)$
 8:     else if $\alpha = X\beta,$ some $X \in V, \beta \in V^*$ such that $|\beta| > 0$ then
 9:         let $q = \texttt{create\_state()};$
10:             $K = K \cup \{q\}$        ▷ create some new state $q$ and add it to the automaton
11:         MAKE_FA$(q_0, X, q)$
12:         MAKE_FA$(q, X, q_1)$
13:     else
14:         let $A = \alpha$        ▷ $\alpha$ must be a single nonterminal
15:         if $A \in N_i$ some $i$ then
16:             for $B \in N_i$ do
17:                 let $q_B = \texttt{create\_state()};\ K = K \cup \{q_B\}$
18:             end for
19:             if $recursive(N_i) = left$ then
20:                 for $(C \rightarrow X_1 ... X_m) \in P$ such that $C \in N_i \wedge X_1, ..., X_m \notin N_i$ do
21:                     MAKE_FA$(q_0, X_1 ... X_m, q_C)$
22:                 end for
23:                 for $(C \rightarrow DX_1 ... X_m) \in P$ such that $C, D \in N_i \wedge X_1, ..., X_m \notin N_i$ do
24:                     MAKE_FA$(q_D, X_1 ... X_m, q_C)$
25:                 end for
26:                 let $\Delta = \Delta \cup (q_A, \epsilon, q_1)$
27:             else
28:                 for $(C \rightarrow X_1 ... X_m) \in P$ such that $C \in N_i \wedge X_1, ..., X_m \notin N_i$ do
29:                     MAKE_FA$(q_C, X_1 ... X_m, q_1)$
30:                 end for
31:                 for $(C \rightarrow X_1 ... X_m D) \in P$ such that $C, D \in N_i \wedge X_1, ..., X_m \notin N_i$ do
32:                     MAKE_FA$(q_C, X_1 ... X_m, q_D)$
33:                 end for
34:                 let $\Delta = \Delta \cup (q_0, \epsilon, q_A)$
35:             end if
36:         else
37:             for $(A \rightarrow \beta)$ do        ▷ A is not recursive
38:                 MAKE_FA$(q_0, \beta, q_1)$
39:             end for
40:         end if
41:     end if
42: end procedure
```

5 Evaluation and Discussion

We first evaluate our approach regarding the obtained results and its performance. Afterwards, we discuss its limitations and potential future work to improve and extend our current implementation.

5.1 Evaluation and Benchmarking

In this section, we first analyze the quality of the results of our approach, e.g. concerning the length of the regular expression and whether it describes a correct language that contains all possible values of the analyzed variable in Section 5.1.1. As a metric for the quality of the result we use the length of the regular expression, because shorter and more concise expression are easier to read and understand for a human user.

Afterwards, we measure execution times of the different steps, including grammar creation, character set and Mohri-Nederhof approximation and automaton and regex creation, in Section 5.1.2. We analyze the effects of different inputs and variations of our approach.

5.1.1 Correctness

We analyze the resulting regular expressions of two synthetic code examples, namely the "Tricky" example from Christensen et al. [2] and one custom example containing simple sanitization logic. We also analyze the results for the SQL Injection test cases of the Juliet test suite[1]. We choose these examples due to the wide variety of complexity they offer. The Tricky example is specifically crafted to be complex, while the Juliet test cases are comparatively simple. The first therefore shows theoretical limitations of our approach, while the latter and the second example are closer to real word applications.

Tricky

We adapted the `Tricky` example code Christensen et al. [2] created for their implementation, which can be seen in Listing 5.1. It creates strings of the form

[1]https://samate.nist.gov/SARD/test-suites/111

`((((((((8*7)*6)*5)+4)+3)+2)+1)+0)`. Since regular languages can not count occurrences, a normal regular expression describing those strings can not guarantee for example an equal amount of opening and closing brackets. A good description using regular languages would for example be `\(*<int>(\*<int>\))*(\+<int>\))*`, where `<int>` abbreviates the expression `0|(-?[1-9][0-9]*)`.

We create the grammar starting at the node representing the variable reference `res` in line 23. After the regular approximation the resulting grammar contains 51 nonterminals and 59 productions. Christensen et al. obtain a different grammar for this example. This difference stems from differences in the definition and implementation of the data flow graph.

The NFA created from the grammar contains 28 states and 40 transitions, of which 27 are ϵ transitions, and is too large to usefully be displayed in this thesis. The NFAs created using Nederhof's algorithm in general often have unnecessary states and transitions, like chains of states only connected with ϵ transitions.

Christensen et al. describe the language they obtained with the expression `\(*<int>([+*]<int>\))*` [2]. Note that Christensen et al. only create automata and not regular expressions, so this expression is just to describe their created automaton. How their automaton compares to ours is unclear, as they only share this description of the language.

With a length of 622 characters, the regular expression we obtain is more complex than necessary, but it accepts the same language as the one given by Christensen et al.. The high complexity stems from many optional cases in the expression, which could either be combined into one case or which accept a subset of another option already present in the expression. However, this length is for a sub-optimal scenario, and can be improved like described in the following.

Note that our implementation uses the built in Kotlin functionality to escape strings. This implementation escapes literals by surrounding them with the special characters `\Q` and `\E`, which adds 120 characters compared to escaping using a backslash. To increase readability we use the `<int>` abbreviation and replace the `\Q\E` esape characters with single backslashes in the following regular expressions.

Like mentioned in Section 4.5, converting the NFA into an equivalent DFA significantly improves the result. The corresponding regular expression for this DFA, which can be seen in Figure 5.1, is

```
(((\((\()*(<int>)|<int>)(\*|\+))(((<int>)\))(\*|\+))*((<int>)\)))|(\((\(⌋
↪  )*(<int>)|<int>)
```

Minimizing the created DFA gives the automaton in Figure 5.2, which our implementation transforms to the even shorter regular expression `(\()*<int>((\*|\+)<int>\))*`.

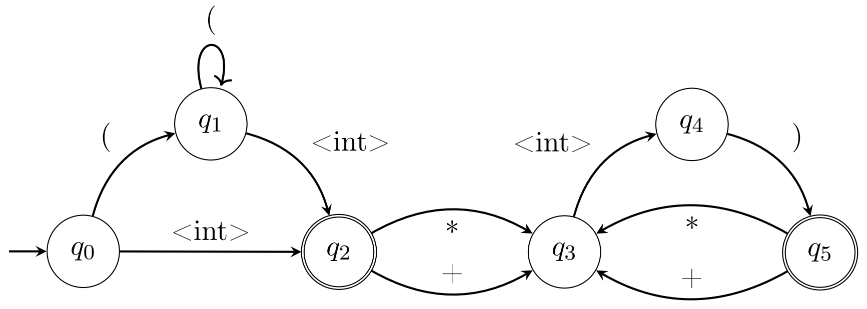

Fig. (5.1) DFA for `Tricky` example

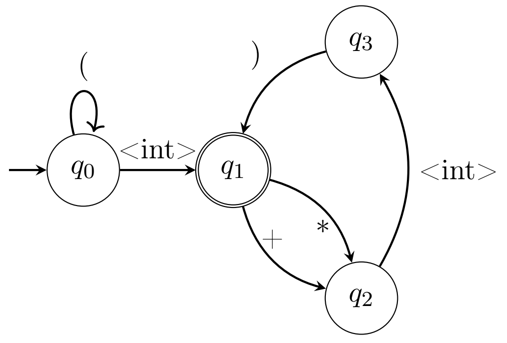

Fig. (5.2) Minimal DFA for `Tricky` example

In essence, this is equivalent to the description by Christensen et al. we mentioned earlier. Therefore, when using some optimizations, our approach produces very short, human-readable regular expressions even for complex inputs.

The regular expressions mentioned here all accept the same language and just differ in their length and complexity.

Note, that neither our nor Christensen et al.'s result does account for the fact, that in the strings generated in the Tricky example all occurrences of `*` are before the first occurrence of `+`. This is a shortcoming compared to the manually created regular expression mentioned earlier. As Christensen et al. mention, this improvement could be achieved by distinguishing the two calls to the `bar` method using a polyvariant analysis [2], which is explained further in Section 5.2.3.

SQL query sanitization

Since the Tricky example is artificially complex, we created the example in Listing 5.2, which resembles a possible real attempt to escape an SQL input.

Note that the given code does not compile as `Connection` is an interface and cannot be instantiated, but as this is not relevant for our analysis we ignore it for the sake of brevity.

```
public class Tricky{
    String bar(int n, int k, String op) {
        if (k==0) {
            return "";
        }
        return op+n+"]"+bar(n-1,k-1,op)+"";
    }
    String foo(int n) {
        String b = "";
        if (n<2) {
            b = b + "(";
        }
        for (int i=0; i<n; i++){
            b = b + "(";
        }
        String s = bar(n-1,n/2-1,"*");
        String t = bar(n-n/2,n-(n/2-1),"+");
        return b+n+(s+t).replace(']',')');
    }
    public static void main(String args[]) {
        int n = new Random().nextInt();
        String res = new Tricky().foo(n);
        System.out.println(res);
    }
}
```

Listing (5.1) Tricky example

```java
import java.sql.*;
public class DatabaseSanitization{
    public static void main(String[] args) throws SQLException {
        String input = args[1];

        String param = (args[2] == "id") ? "id" : "name";
        String sanitized = sanitize(input);

        Statement stmnt = (new Connection()).createStatement();
        stmnt.executeQuery(
            "DELETE␣*␣FROM␣users␣WHERE␣" +
            param + "␣=␣\'" + sanitized + "\'"
        );
    }

    public static String sanitize(String input){
        return input.replace('\'', '␣').replace('-', '␣');
    }
}
```

Listing (5.2) SQL query sanitization code example

Our approach returns `(DELETE \* FROM users WHERE ((id|name) = '[^'\-]*'))` as a result. This result correctly displays the two possible options `id` and `name` for the parameter.

The transformation of the two `replace` operations also correctly transformed the initial wildcard `.*` inserted for `input` to an expression matching any strings that do not contain one of the SQL special characters `'` or `-`. Since for a successful SQL-injection an attacker would need to close the opened quote using a single quote character, a further evaluation of this result could show that this sanitization reduces the risk compared to unfiltered input.

Juliet

The Juliet Test Suite is created by the National Security Agency's (NSA) Center for Assured Software (CAS) and specifically designed to assess the capabilities of static analysis tools.

The test cases each target one type of flaw corresponding to a specific CWE[2] entry. All 2224 test cases we analyzed target SQL-Injection vulnerabilities described in CWE-89, as these are flaws, where strings and string operations are the relevant points, while also being relevant risks in real applications.

[2]https://cwe.mitre.org

The Juliet test cases can generally be grouped into ones using bad sinks, where a query string is vulnerable to an SQL-Injection and ones using good sinks, where prepared statements are correctly used. For the latter, the queries are just literal strings like `"select␣*␣from␣users␣where␣name=?"`, where the replacement of the question mark with the desired parameter is internally handled by the SQL library.

The vulnerable test cases in the Juliet test suite build an SQL query similar to `"insert␣into␣users␣(status)␣values␣('updated')␣where␣name='"+data+"'"`, where `data` is an unsanitized string. The actual semantics of the statements differ, but they all have the same structure where the value is injected as a parameter at the end of the query.

The cases also differ in the data flow from the source of the `data` string and the sink, where it is passed to a database library. Sometimes `data` is unknown and sometimes a string constant. We are interested in the case, where the value of `data` is unknown, as these cases pose security vulnerabilities.

For these cases, we obtain regular expressions similar to the example `(insert into users (status) values ('updated') where name='.*')` as a result.

While the interpretation and analysis of the obtained expressions is out of scope for this thesis, it is clear that this result exposes a severe security risk, because any injection string could be inserted as a value for `data`, signified by the wildcard pattern `.*`.

5.1.2 Performance

Further, we measured the execution time of the different steps in our analysis approach. We measure the steps separately to observe the relative differences between the steps and the effects of the optimizations we described. The total execution times naturally depend on the machine used to run the analysis. We used a common desktop computer[3] for the benchmarks. As in the previous section we considered the two synthetic examples and the test cases of the Juliet test suite.

In the following, all mentioned averages are truncated means, which means that before calculating the mean, first the highest $n\%$ and the lowest $n\%$ of values are discarded. The truncated mean is a common method to make the mean more robust, usually with values for n ranging from 10 to 20% [10]. In our benchmarks we use 20% for this cutoff value. This is done to eliminate the effect of statistical outliers, which do not represent an actual measurement but e.g. are influenced by an external factor like CPU scheduling or caching. For example when running our benchmarks using JUnit tests, the first test case had the highest durations for all steps including grammar creation, approximation and automaton creation by a factor of 10-20 for some inputs. This effect occurred even when

[3]Intel i7-3770K (8 cores) @ 4.100GHz with 16 GB RAM running Arch Linux 6.1

repeatedly testing the same input.

Tricky

Consider the Figures 5.3 and 5.4. The plots show the durations of each step in the process from CPG to regular expression. In the second plot, we additionally transform the created NFA to a DFA before we create the regular expression, whereas in the first plot, the NFA is converted directly. As visualized by this plot, the DFA creation adds an additional 300 μs to transform the automaton, but drastically reduces the time the state elimination algorithm takes to create a regular expression.

For both plots we averaged the measurements of 100 runs after we trimmed the highest and lowest 20% of values.

SQL query sanitization

Figure 5.5 shows the execution time of analyzing the SQL query sanitization example from the previous section. Similar to the Tricky example, the plot shows the average of 100 measurements trimmed by 20%.

Note the different scale of the x axis compared to the previous examples. The difference compared to the other plots is explained with the different complexity of the analyzed code. Since the SQL query sanitization example is considerably less complex than the Tricky example, naturally the analysis execution time is lower.

Figure 5.6 again shows the execution times of the query sanitization example, but with the additional step of creating a DFA before performing the state elimination algorithm.

Due to the characteristics of this example, like a simpler data flow, the resulting NFA already is a DFA. Therefore, the - in this case - unnecessary DFA creation just adds additional time, without significantly changing the execution time of the state elimination algorithm. This shows, that whether converting the automaton to a DFA improves the execution time is dependent on properties of the input.

Juliet

We analyzed the execution time of each database related hotspot of all 2224 test cases in the Juliet test suite targeting SQL Injection vulnerabilities.

Figure 5.7 shows the averaged execution times of each step, again trimmed by 20%. We can see, that the percentage of the total time spent on regex creation is lower compared to the plot for the Tricky example in Figure 5.3, even without the intermediary DFA step. This is due to the fact that the resulting automata for the Juliet test cases are very small, ranging from 2 to 4 states, compared to the 28 nodes of the Tricky NFA.

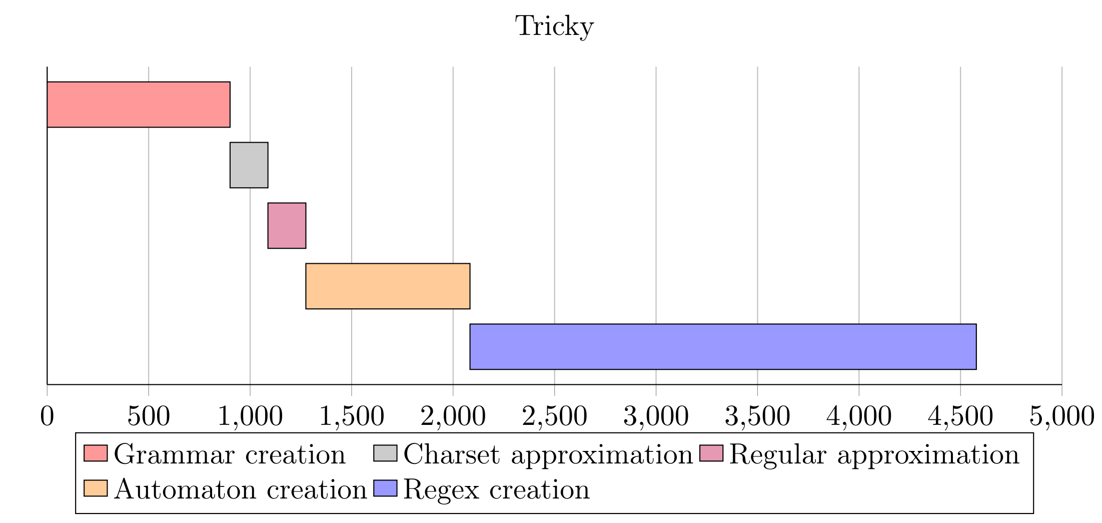

Fig. (5.3) Durations of Tricky example in μs

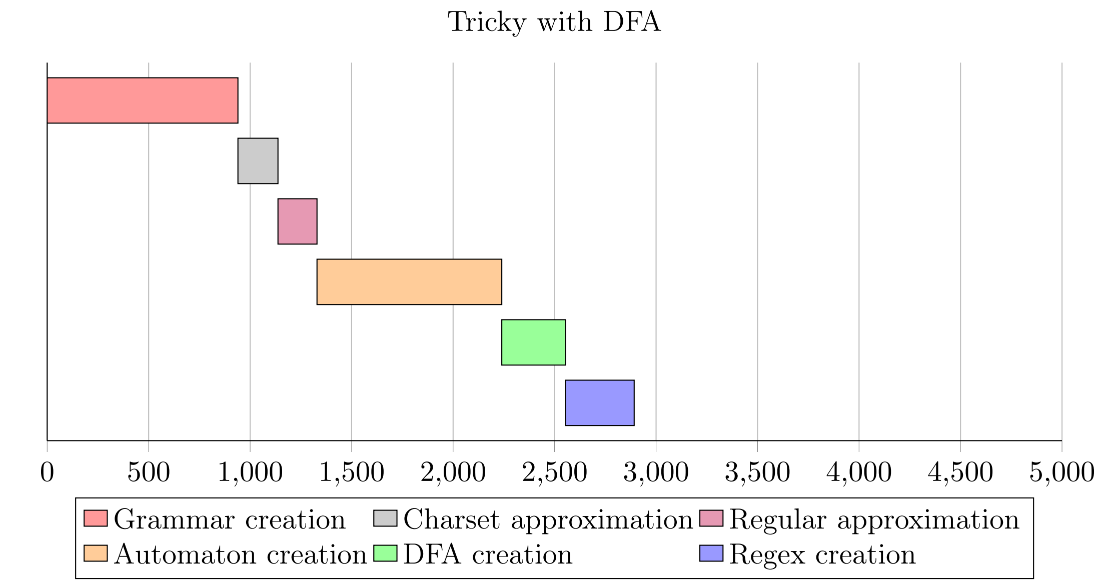

Fig. (5.4) Durations of Tricky example with DFA creation in μs

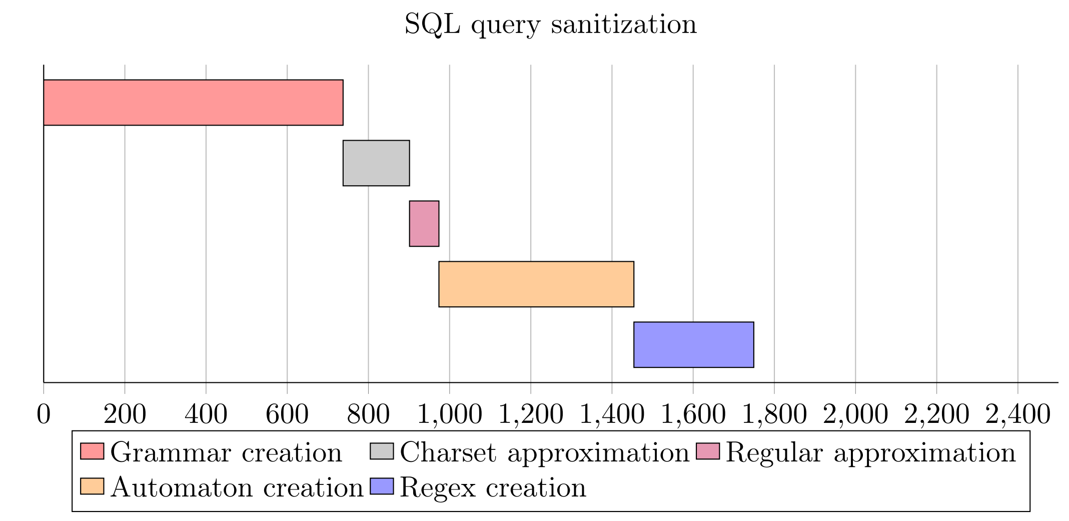

Fig. (5.5) Durations of SQL query sanitization test case in μs

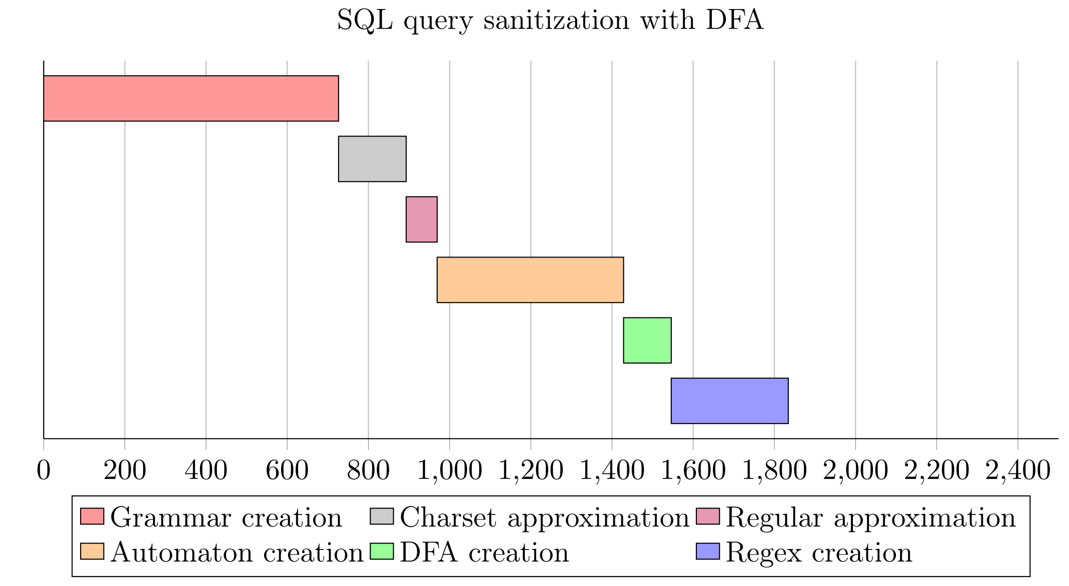

Fig. (5.6) Durations of SQL query sanitization test case with DFA creation in μs

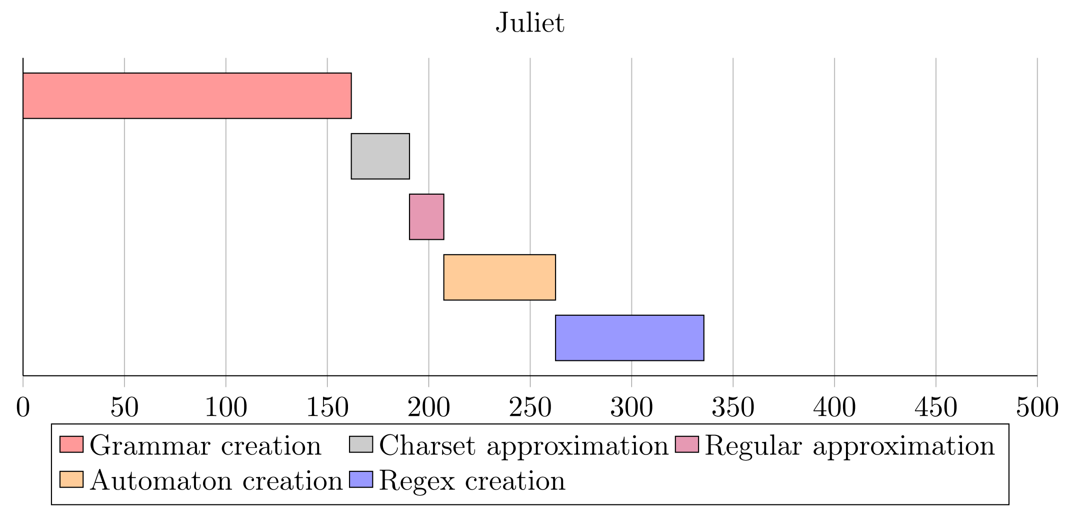

Fig. (5.7) Durations of Juliet test cases in μs

Again, note the difference in the scale of the x-axis of factor 10, which shows that the Juliet test cases are analyzed significantly faster due to their lower complexity.

Also note that the relative execution times of the different steps are fairly consistent among all the different examples.

5.2 Discussion and Future work

We discuss several limitations of our implementation and present potential approaches, how future work can mitigate these limitations and extend our implementation.

5.2.1 Examples

We have to note, that all examples we used to test our implementation are synthetic examples. This is due to the fact, that we currently only support a small subset of the Java language, which is not enough to obtain meaningful results from real code. It is unclear how closely the used examples resemble most actual application code and to what extent the observations we made can be transferred to real world use.

5.2.2 Assertions

Our implementation currently does not try to evaluate assertions like `s.isEmpty()` due to limitations in the creation of the CPG we use.

Consider the example in Listing 5.3 where `getSomeKnownValue()` returns some value we can analyze, which is henceforth abbreviated with the generic `<val>`.

```
String s^1 = getSomeKnownValue();     1
if(s^2.isEmpty()){                    2
    s^4 = s^3 + "empty";              3
}                                     4
System.out.println(s^5);              5
```

Listing (5.3) Assertion Example

In our CPG the only incoming DFG edge of `s`^{3} in line 3 is an edge from `s`^{1} in line 1. However, there is an implicit information flow from `s`^{2} in line 2 to `s`^{3}, as the result of applying the `isEmpty` operation on `s`^{2} influences the information we can get about `s`^{3}. If there was a DFG edge to the `s`^{2}`.isEmpty()` call from `s`^{3} instead of the edge from `s`^{1}, we could include the operation in our analysis.

For example, for such an edge, we could add a new type of production comparable to the existing operation productions, from the nonterminal representing `s`^{3} to the one representing `s`^{2}.

To resolve such an assertion production $A \rightarrow \mathit{assertion}(B)$ we could implement transformations similar to the existing operation productions. For this example, the transformation of the *isEmpty* assertion would always return just the empty string. In Listing 5.3, we could always infer that `s`^{3} is empty, which is clear from the code.

For this example we currently get the regular expression `(<val>empty)|(<val>)` as a result. Consider `<val>` to be `abc|`ϵ, which gives us `((abc|`ϵ`)empty)|(abc|`ϵ`)`. Here the first part `((abc|`ϵ`)empty)` corresponds to the value of `s`^{4}, which is a possible value of the analyzed `s`^{5} and the second part `(abc|`ϵ`)` to the value of `s`^{1}, which is the result if the condition evaluates to false.

With the mentioned additional DFG edges and the described logic, we could sharpen this result. As mentioned above, the value of `s`^{3} would be ϵ due to the `isEmpty` assertion transformation and therefore `s`^{4} would be a concatenation of ϵ and the string `"empty"`, so just `empty`.

This gives us `empty|(abc|`ϵ`)` as a result, which is more precise, as for example the unobtainable `abcempty` is not part of the language of this regular expression.

Similar transformations could also be defined for more complex assertions like `s.length() == 1`.

However, as mentioned above, this is currently not possible because the CPG is missing the required DFG edges representing this implicit information flow from an assertion to subsequent variable usages.

5.2.3 Polyvariance

Polyvariance is an analysis strategy, where functions are analyzed more than once, usually once for every call site [15]. The best result we currently obtain for the Tricky example is the regular expression `(\()*<int>((\*|\+)<int>\))*`, that does not differentiate between `*` and `+`. Using a polyvariant analysis could improve this, because the two calls to the `bar` function would be analyzed separately with respect to their corresponding arguments. Currently, we only follow the DFG edges, where the parameter of the function has one to each variable passed as this parameter, here `+` and `*`. These two edges are not differentiated and the same result is used for both calls to the function. By differentiating between the corresponding arguments for the analysis of each call, we could sharpen the result to `\(*<int>(\*<int>\))*(\+<int>\))*`.

5.2.4 More extensive implementation

Currently, the CPG does not contain DFG edges to differentiate which field of an array is accessed in an array subscription expression like `myArray[5]`.

Therefore, we currently do not further analyze values stored in arrays but rather just insert a regular expression generally describing the type stored in the array.

We mostly focused on simple strings, but in general everything we described can be applied to string builders. For example, during grammar creation, a `ConcatProduction` could not only be created for `s1 + s2` with two strings, but also for `sb.append(s)` with a `StringBuilder sb` and some other string.

As this is just a proof of concept, we also implemented only a few operations on strings to showcase the approach.

5.2.5 State Elimination Heuristics

Moreira et al. analyzed and evaluated different existing and proposed new heuristics for choosing a good order to eliminate states during the state elimination algorithm [12].

While they conclude that the method by Delgado and Morais [4] we use is similar to their newly proposed heuristics, they also note that the new strategies outperform Delgado's for small automata. Since the automata we handle are comparatively small, adapting our implementation to use one of the proposed new heuristic could improve the quality of the results.

Moreira et al. also observe that the strategy of taking the best result of using all three heuristics as a final result leads to a gain of 25%. This approach of trying multiple strategies is also used by the powerful Vcsn platform for computations on finite state machines [5].

5.2.6 Automata Centric Approach

We chose to provide the information we extracted only as regular expressions due to regular expressions being widely used and supported. However, representing the information as DFAs instead of converting the automata to regular expressions has some advantages due to theoretical properties of DFAs.

Regular expression objects in most programming languages, for example Kotlin's `Regex` object, can determine, whether a given string matches the expression. With a sufficiently advanced automaton implementation more advanced checks can be performed. After analyzing a given hotspot and obtaining an automaton M, instead of just matching a given string as a query, the input can be a regular expression. This regular expression can then be turned into another DFA N. Since DFAs are closed under intersection and complement, we can build the DFA $R = M \cap \overline{N}$, which accepts words that are in the language of the analysis result, but not in the language of the query expression. Now we can check, whether R's language is empty to determine, whether all strings of the query are possible values of the analyzed node. Furthermore, if R's language is not empty, we can generate a word from this language as an example for a string that is a possible value of the analyzed node, but not part of the query language. Additionally, we can check whether M and N are equivalent to determine, whether the query expression matches the computed result.

Since we use NFAs or optionally already DFAs as an intermediate representation, future work could increase the capabilities of our automata implementation and implement the mentioned advanced query possibilities.

6 Related Work

The challenge of statically obtaining information about the values of strings is not new and over the years there have been different approaches to it.

We follow the approach by Christensen et al. [2]. The authors construct a context free grammar from a flow graph, but instead of creating it on-demand, starting at the chosen hotspot node like we do, they consider the total flow graph for grammar creation. As the total amount of potential hotspots can be far greater than the number of actual points of interest, this total grammar is potentially significantly larger than needed. This increased size reduces the performance of the subsequent steps. Christensen et al. use the same approximation methods for obtaining regular languages from the generated context free grammars, but instead of directly transforming these grammars into automata like we do, they produce a novel formalism, the multi-level automaton (MLFA). This automaton allows the extraction of the respective automata for different hotspots. Due to the aforementioned on-demand generation of the grammar, we don't need the extraction capability for single hotspots that the MLFA provides. Christensen et al. also don't transform the obtained automata to regular expressions, but rather provide query options using automata. The authors provide a feature rich implementation[1] of their approach and show that it efficiently produces useful results.

Tabuchi et al. [16] describe a type system for a minimal functional calculus, where strings have a regular expression as their type. Using typeinference and reconstruction algorithms, they assign such a type to each string variable. Properties about a variable can then be obtained from its type. They show that their proposed type system can produce good results when applied to their minimal calculus. While we considered implementing this approach for the analysis, there are some problems, especially due to our different requirements and prerequisites.

To use the presented approach in practice an (efficient) algorithm for type checking and type reconstruction is needed. The given paper does not include those, but rather indicates several problems in constructing such algorithms for the given situation without losing some of the desired preciseness. The authors mention that using standard type reconstruction by constraint solving for the proposed type system even is impossible due to limitations of regular languages.

[1]https://www.brics.dk/JSA/

Additionally this approach is tailored to the mentioned calculus and utilizes specific features like pattern matching, which would make adapting it to our use case more difficult.

The additional layer of abstraction introduced by the DFG used in the approach we chose eliminates this problem and makes adaption easier.

Wassermann and Su [19] present an approach comparable to ours, where they also characterize values of string variables using context free grammars. They specifically target SQL injection vulnerabilities by using the generated CFGs to check whether user input can change the syntactic structure of a query. While this approach is successful in detecting those vulnerabilities, our approach is more general and not focused on detecting one specific type of problem, but rather on providing general information for unspecified further use.

7 Conclusion

In this thesis we implemented a method to obtain information about the values of strings from the data flow graph of an analyzed program. We provide a proof of concept implementation as an extension of a Code Property Graph implementation used in the static analysis tool Codyze.

We adapted part of an existing approach to obtain a strongly regular grammar from the graph. We then convert this grammar into an automaton using an algorithm we adapted and extended for our use case. Further, this automaton is transformed into a regular expression, which describes the analyzed string. We use approximations of different precision to model the effects of concatenation and other operations on strings.

Additionally we described different methods, like intermediate conversion to a DFA and a heuristic for state elimination, to potentially increase the performance of our implementation.

We also showed, that even for complex examples our implementation provides useful results which could be used to detect security vulnerabilities like SQL injections. Furthermore, we tested and benchmarked our implementation using different examples and the well known Juliet test suite, which showed the viability of our performance optimizations and general approach. Moreover, we summarized limitations of our implementation and provide starting points for potential further research.

We think that, especially with further enhancements to our implementation, the information our approach provides can be beneficial for static analysis, especially in preventing common security vulnerabilities.

Bibliography

[1] J. A. Brzozowski and E. J. McCluskey. "Signal Flow Graph Techniques for Sequential Circuit State Diagrams." In: *IEEE Transactions on Electronic Computers* EC-12.2 (1963), pp. 67–76. DOI: 10.1109/PGEC.1963.263416.

[2] A. S. Christensen, A. Møller, and M. I. Schwartzbach. "Precise Analysis of String Expressions." In: *Proc. 10th International Static Analysis Symposium (SAS)*. Vol. 2694. LNCS. Available from http://www.brics.dk/JSA/. Springer-Verlag, June 2003, pp. 1–18.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT press, 2022.

[4] M. Delgado and J. Morais. "Approximation to the smallest regular expression for a given regular language." In: *Implementation and Application of Automata: 9th International Conference, CIAA 2004, Kingston, Canada, July 22-24, 2004, Revised Selected Papers 9*. Springer. 2005, pp. 312–314.

[5] A. Demaille, A. Duret-Lutz, S. Lombardy, and J. Sakarovitch. "Implementation Concepts in Vaucanson 2." In: *Proceedings of Implementation and Application of Automata, 18th International Conference (CIAA'13)*. Ed. by S. Konstantinidis. Vol. 7982. Lecture Notes in Computer Science. Halifax, NS, Canada: Springer, July 2013, pp. 122–133. ISBN: 978-3-642-39274-0. DOI: 10.1007/978-3-642-39274-0_12.

[6] J. Esparza. "Automata theory – An algorithmic approach." Lecture Notes, https://www7.in.tum.de/~esparza/autoskript.pdf. Aug. 2017.

[7] G. Gramlich and G. Schnitger. "Minimizing nfa's and regular expressions." In: *Journal of Computer and System Sciences* 73.6 (2007), pp. 908–923.

[8] H. Gruber, M. Holzer, and M. Tautschnig. "Short regular expressions from finite automata: Empirical results." In: *Implementation and Application of Automata: 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings 14*. Springer. 2009, pp. 188–197.

[9] Y.-S. Han and D. Wood. "The generalization of generalized automata: Expression automata." In: *International Journal of Foundations of Computer Science* 16.03 (2005), pp. 499–510.

[10] U. Krenel. *Einführung in die Wahrscheinlichkeitstheorie und Statistik.* Vol. 8. Wiesbaden: Vieweg, 2005, p. 171. ISBN: 3-8348-0063-5. DOI: 10.1007/978-3-663-09885-0.

[11] M. Mohri and M.-J. Nederhof. “Regular approximation of context-free grammars through transformation.” In: *Robustness in language and speech technology.* Springer, 2001, pp. 153–163.

[12] N. Moreira, D. Nabais, and R. Reis. “State elimination ordering strategies: Some experimental results.” In: *arXiv preprint arXiv:1008.1656* (2010).

[13] M.-J. Nederhof. “Regular approximation of CFLs: a grammatical view.” In: *Advances in Probabilistic and other Parsing Technologies.* Springer, 2000, pp. 221–241.

[14] Open Worldwide Application Security Project (OWASP). *OWASP Top 10.* 2021. URL: `https://owasp.org/Top10/` (visited on 02/25/2023).

[15] J. PALSBERG and C. PAVLOPOULOU. “From Polyvariant flow information to intersection and union types.” In: *Journal of Functional Programming* 11.3 (2001), pp. 263–317. DOI: `10.1017/S095679680100394X`.

[16] N. Tabuchi, E. Sumii, and A. Yonezawa. “Regular Expression Types for Strings in a Text Processing Language.” In: *Electronic Notes in Theoretical Computer Science* 75 (2003). TIP’02, International Workshop in Types in Programming, pp. 95–113. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/S1571-0661(04)80781-3`.

[17] R. Tarjan. “Depth-First Search and Linear Graph Algorithms.” In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: `10.1137/0201010`. eprint: `https://doi.org/10.1137/0201010`.

[18] K. Thompson. “Programming Techniques: Regular Expression Search Algorithm.” In: *Commun. ACM* 11.6 (June 1968), pp. 419–422. ISSN: 0001-0782. DOI: `10.1145/363347.363387`.

[19] G. Wassermann and Z. Su. “Sound and precise analysis of web applications for injection vulnerabilities.” In: *ACM-SIGPLAN Symposium on Programming Language Design and Implementation.* 2007.

[20] K. Weiss and C. Banse. *A Language-Independent Analysis Platform for Source Code.* 2022. DOI: `10.48550/ARXIV.2203.08424`.