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

Your introduction goes here

- Generic description of the broad field of research
- Current state of research
- What's the gap that you're trying to fill?
- Short motivation
- Summary of the most important results
- Your contribution
- Structure of the thesis

1-2.5 pages

This text is not too detailed. Start quite high-level, then narrow down until you reach your topic. After the introduction, the reader must want to read the rest of your thesis and understand the relevance. However, it doesn't have to be super technical.

2 Problem Description

The introduction is a bit like a teaser. Here, you dig more into details, also technical ones. After this chapter, the reader must understand why you do this work, why it's important, what makes it difficult and what you want to achieve.

- What's the problem that you're trying to solve?
- What is your goal?
- What is/are the research question(s)?
- What are special problems?

Probably 1-3 pages

3 Background

3.1 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.

The 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 [14].

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

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

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 data flow graph (DFG), from which we then extract the information on string values.

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.

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 4.4.1. Finally, in Section 4.4.2 we describe how to create a regular expression from this automaton.

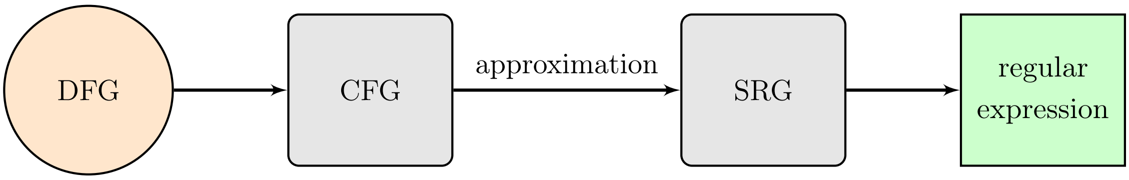

Fig. (4.1) The general approach for obtaining regular expressions

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 five 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
- `UnaryOpProduction`: $X \to op(Y)$ for unary operations on strings

- `BinaryOpProduction`: $X \rightarrow op(Y, Z)$ for binary 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.

`UnaryOpProductions` and `BinaryOpProductions` 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 `UnaryOpProduction` $X \rightarrow trim(Y)$, 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 and 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, it uses a `ValueEvaluator` which tries to obtain a value representing the node. For example for an integer created by usual arithmetic operations, the `ValueEvaluator` 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

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 production s[8].

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.

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

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 $A \rightarrow BC$ 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'} $\cup$ {'a', 'c'} = {'a', 'b', 'c'}.

Each operation defines a character set transformation - a function $T_{op} : 2^{\Sigma} \rightarrow 2^{\Sigma}$ - 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.

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

For comparison for a `replace` operation, where the newly inserted character is not known, if the replaced character is contained in S, the set is transformed to Σ, since the 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 $(\{\text{'a', 'b', 'c'}\} \setminus \{\text{'c'}\}) \cup \{\text{'x'}\} = \{\text{'a', 'b', 'x'}\}$ as the character set of S.

To determine the SCCs, we use Tarjan's algorithm [11]. This algorithm topologically sorts the returned components in reverse order, 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

To represent character sets easily, we have two different implementations, both conforming to a common `CharSet` interface that requires functions like `union : CharSet -> CharSet` and `intersect : CharSet -> CharSet`.

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 commonly occurring type of character sets, where only a few characters are removed from Σ. It also simplifies the creation of a regular expression from the character set, since the approach of using a character class containing all characters in the set produces very large character classes for sets with cardinality close to $|\Sigma|$. Using our approach, we can represent a `SigmaCharSet` using negated character classes. Since most character sets either contain a comparatively small amount of given chars, or all chars except a few this reduces the average length of the resulting regular expressions. For example the `SetCharSet` that represents the set {'a', 'b', 'c'} gives us 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 [8] describe an algorithm to transform a CFG into a SRG as defined in Section 3.2 that approximates the given CFG.

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 $A \rightarrow CX$ 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 transformation approach for productions with an arbitrary number of nonterminals on the left hand side [8]. 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 set of rules described by Christensen et al.[2].

For each nonterminal A in a given equivalence class M add a new nonterminal A'.

Replace all productions of A with the following new productions, where 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.

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

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.

Implementation

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

The notion of mutual "reachability", by which the relation $\mathcal{R}$ defined in 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 Transformation to Regular Expression

4.4.1 Strongly Regular Grammar to Automaton

Algorithm

Nederhof describes an algorithm to transform a SRG into an equivalent nondeterministic finite automaton (NFA) in [9]. More specifically, the algorithm creates an ϵ-NFA. 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.

The `create_state` function used in the pseudo code just creates a new state object which can then be added to the automaton.

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 $C \to op(DX_1)$ with C and D in the same SCC can no longer occur. How the effects of the operation productions are resolved is described in the following section.

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. Nederhofs definition therefore has $a \in \Sigma$ as a condition for this case instead of our $a \in \Sigma^*$. This also changes the type of the generated NFA because it contains edges labeled with multi-character strings. Due to the fact that the only thing we use the generated automaton for is the algorithm described in Section 4.4.2, this difference in definition is no problem.

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 show why this is the case, consider the grammar in Figure 4.3 creating just the two words "b" and "c". Here A, B and C are non-recursive nonterminals, so in the initial procedure call with arguments (q_0, A, q_1) there are just the two recursive calls

$$\begin{aligned} A &\rightarrow B \\ A &\rightarrow C \\ B &\rightarrow b \\ C &\rightarrow c \end{aligned}$$

Fig. (4.3) Example grammar with no recursion

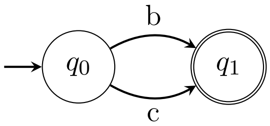

Fig. (4.4) Resulting automaton for the grammar in Figure 4.3

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 4.4 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. To explain the differences between the recursive calls in the different cases consider the grammars in Figures 4.5 and 4.7.

Nederhof only defines the case for left recursion in his publication and states that the else part is "the converse of the then part" [9]. This suggests, that besides switching the condition for the second loop from $C \rightarrow DX_1 \dots X_m$ to $C \rightarrow X_1 \dots X_m D$, 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.

The inverting of the states in the recursive calls leads to the edges between states q_2 and q_3 of the automata in Figures 4.6 and 4.8 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. (4.5) Example grammar with left recursion

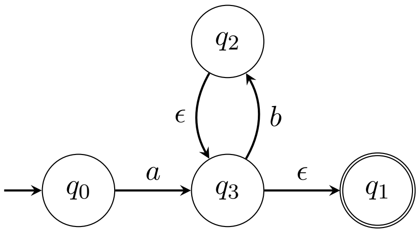

Fig. (4.6) Resulting automaton for the grammar in Figure 4.5

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

Fig. (4.7) Example grammar with right recursion

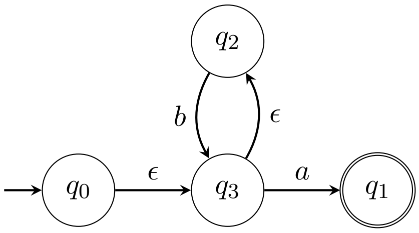

Fig. (4.8) Resulting automaton for the grammar in Figure 4.7

Operation Productions

As described above, operation productions of form $C \to op(X)$ are treated like a normal production $C \to X$.

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 $C \to op_1(X)$, 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, to which the transformation of the corresponding operation can then be applied.

Consider the grammar in Figure 4.9 and the corresponding automaton in Figure 4.10. Here the production $A \to F$ leads to the creation of the left path including state q_2, while for the operation production $A \to replace[f, x](F)$ the subsequent algorithm calls create the colored path. All colored edges and states are tainted with the *replace* operation. The created NFA has two identical paths, since $A \to replace[f, x](F)$ is treated like a second $A \to F$ production, 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 4.11 for the given example.

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

Fig. (4.9) Example grammar with operation production

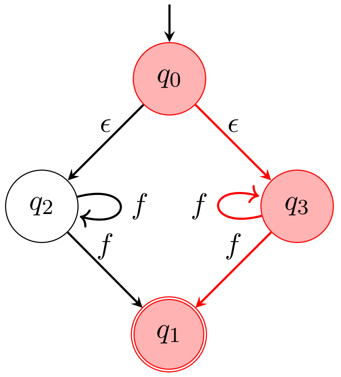

Fig. (4.10) Resulting automaton for the grammar in Figure 4.9

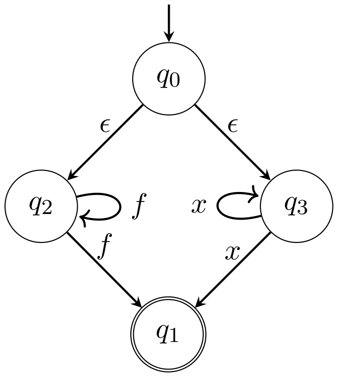

Fig. (4.11) Automaton in figure 4.10 after applying operation transformation

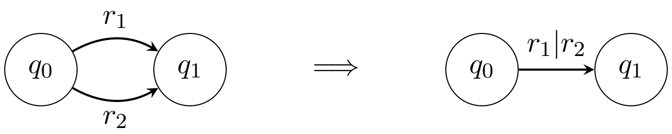

Fig. (4.12) Replacement of edge pairs

.

4.4.2 Automaton to Regular Expression

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 [7].

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 [4].

Figures 4.12 and 4.13 contain examples adapted from Esparza [4] 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.

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 [5]. If there were an efficient

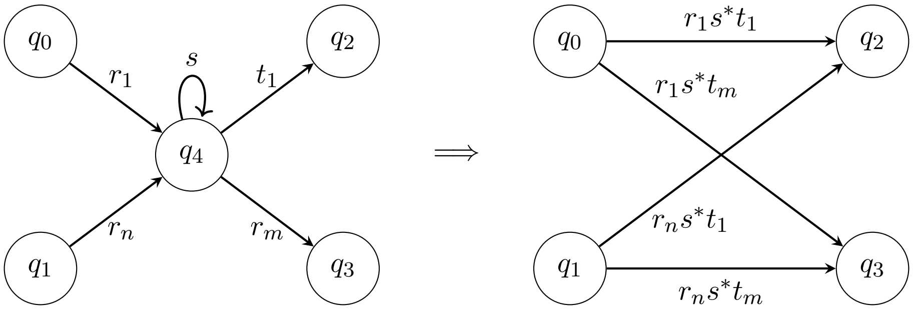

Fig. (4.13) Elimination of state q_4

.

algorithm to obtain the minimal regular expression for a given NFA, one could first apply Thompson's algorithm[12] 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.

Therefore there exists no efficient algorithm to find the minimal regular expression for a given NFA.

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 [3]. 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 [3]. Gruber et al. also show it outperforms almost all other heuristics they considered [6]. 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 [3].

Another reduction of the regular expression size can often be obtained by first converting the automaton to a deterministic finite automaton (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 most 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.

4.5 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. The grammar creation is completely independent of this collection however 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 $(K, \Sigma, \Delta, s, F)$

```
let $\Delta = \emptyset; s =$ create_state()$; f =$ create_state()$; F = \{f\}; K = \{s, f\}$
MAKE_FA$(s, S, f)$
procedure MAKE_FA$(q_0, \alpha, q_1)$
    if $\alpha = \epsilon$ then
        let $\Delta = \Delta \cup (q_0, \epsilon, q_1)$                ▷ add $\epsilon$ transition from state $q_0$ to state $q_1$
    else if $\alpha = a,$ some $a \in \Sigma^*$ then
        let $\Delta = \Delta \cup (q_0, \alpha, q_1)$
    else if $\alpha = X\beta,$ some $X \in V, \beta \in V^*$ such that $|\beta| > 0$ then
        let $q =$ create_state();
            $K = K \cup \{q\}$        ▷ create some new state $q$ and add it to the automaton
        MAKE_FA$(q_0, X, q)$
        MAKE_FA$(q, X, q_1)$
    else
        let $A = \alpha$                                  ▷ $\alpha$ must be a single nonterminal
        if $A \in N_i$ some $i$ then
            for $B \in N_i$ do
                let $q_B =$ create_state()$;\ K = K \cup \{q_B\}$
            end for
            if $recursive(N_i) = left$ then
                for $(C \rightarrow X_1...X_m) \in P$ such that $C \in N_i \wedge X_1, ..., X_m \notin N_i$ do
                    MAKE_FA$(q_0, X_1...X_m, q_C)$
                end for
                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
                    MAKE_FA$(q_D, X_1...X_m, q_C)$
                end for
                let $\Delta = \Delta \cup (q_A, \epsilon, q_1)$
            else
                for $(C \rightarrow X_1...X_m) \in P$ such that $C \in N_i \wedge X_1, ..., X_m \notin N_i$ do
                    MAKE_FA$(q_C, X_1...X_m, q_1)$
                end for
                for $(C \rightarrow X_1...X_mD) \in P$ such that $C, D \in N_i \wedge X_1, ..., X_m \notin N_i$ do
                    MAKE_FA$(q_C, X_1...X_m, q_D)$
                end for
                let $\Delta = \Delta \cup (q_0, \epsilon, q_A)$
            end if
        else
            for $(A \rightarrow \beta)$ do                              ▷ A is not recursive
                MAKE_FA$(q_0, \beta, q_1)$
            end for
        end if
    end if
end procedure
```

5 Evaluation and Discussion

5.1 Evaluation and Benchmarking

5.1.1 Correctness

Tricky

We adapted the `Tricky` example code Christensen et al. created for their implementation, which can be seen in Listing 5.1.

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. The difference to the corresponding grammar Christensen et al. created 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 to large to display in this thesis. The NFAs created using Nederhofs algorithm in general often have unnecessary states and transitions, like chains of states only connected with ϵ transitions. With a length of 622 characters, the regular expression we obtain is more complex than necessary. However it accepts the correct language described with the expression `\(*<int>([+*]<int>\))*` by Christensen et al. [2], where `<int>` abbreviates the expression `0|(-?[1-9][0-9]*)`. 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.

Also note that our 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 4.4.2, 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>)
```

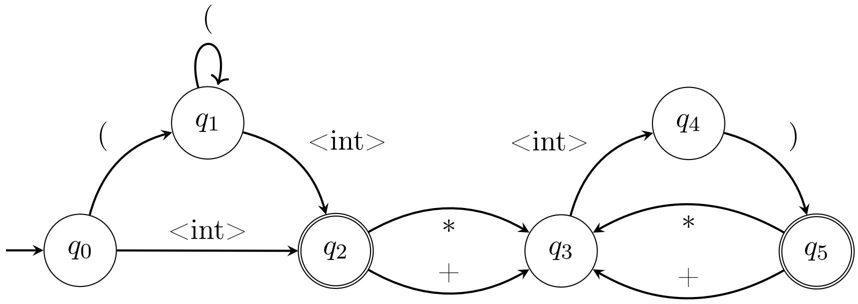

Fig. (5.1) DFA for `Tricky` example

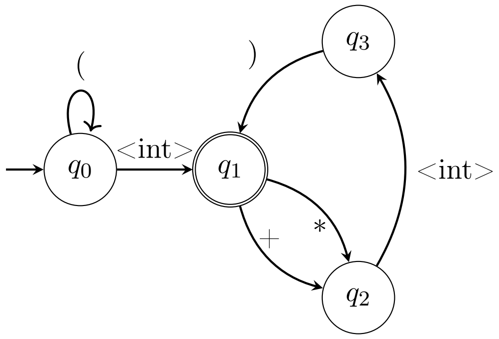

Fig. (5.2) Minimal DFA for `Tricky` example

Minimizing the created DFA gives the automaton in Figure 5.2, which is transformed to the regular expression `(\()*<int>((\*|\+)<int>\))*`.

As Christensen et al. mention, this expression is a good result, but by distinguishing the two calls to the `bar` method, an even more precise expression could be obtained [2]. Due to the given properties of the CPG creation, this is currently not an option for our implementation.

5.1.2 Performance

5.2 Discussion and Future work

5.2.1 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 5.2 where `getSomeKnownValue()` returns some value we can analyze, which is henceforth abbreviated with the generic `<val>`.

In our CPG the only incoming DFG edge of $\texttt{s}^3$ in line 3 is an edge from $\texttt{s}^1$ in line 1. However, there is an implicit information flow from $\texttt{s}^2$ in line 2 to $\texttt{s}^3$, as the result of

```java
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

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

Listing (5.2) Assertion Example

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 5.2, 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`)` as our current result. 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.

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.

5.3 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, determine, whether a given string matches the expression. With a sufficiently advanced automaton implementation more advanced checks can be made. 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 string 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.

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. They use the same approximation methods for obtaining regular languages from the generated context free grammars, but instead of making the regular languages available as a regular expression they generate automata. Furthermore they introduce a novel formalism, the multi-level automaton (MLFA) which allows easy extraction of these automata for different hotspots. Due to the aforementioned on-demand generation of the grammar, we don't need this extraction for single hotspots the MLFA provides in our implementation. The authors provide a feature rich implementation[1] of their approach and show that it efficiently produces useful results.

Tabuchi et al. [10] describe a type system for a minimal functional calculus, where strings have a regular expression as their 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.

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 [13] present an approach comparable to ours, where they also

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

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

Summarize your main contributions and observations. Further research directions?

$\leq$ 1 page

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] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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`.

[11] 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`.

[12] 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`.

[13] 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.

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