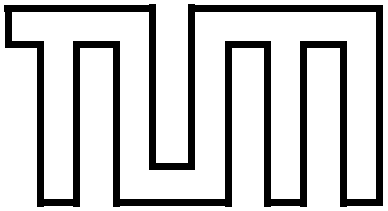


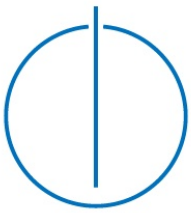
DEPARTMENT OF INFORMATICS

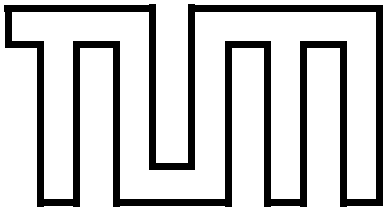
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Inferring String Properties from Code Property Graphs

Severin Schmidmeier





DEPARTMENT OF INFORMATICS

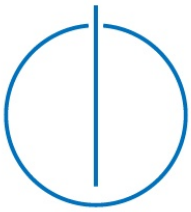
TECHNISCHE UNIVERSITÄT MÜNCHEN

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

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 [5].

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, amongst 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.

4 Approach and Implementation

The general approach for our implementation is adapted from the one described by Christensen et al. [1]. Conceptually, we first extract a context free grammar (CFG) from the DFG. In multiple steps using different methods we then approximate this grammar into a regular grammar. From this regular grammar we can create a regular expression object to provide to the user for further analysis.

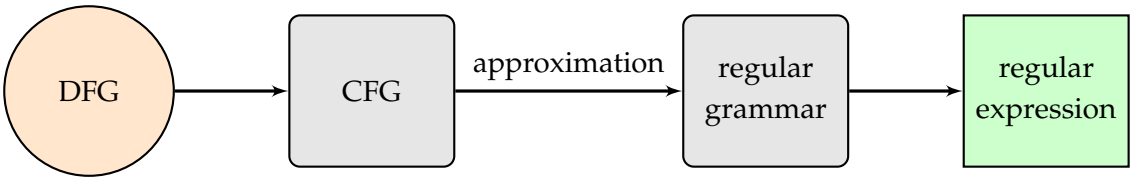


Figure 4.1: The general approach for obtaining regular expressions

4.1 Hotspot Collection

- collect hotspots for grammar creation - set of interesting locations - start grammar creation from there

4.2 Grammar Creation

Our Grammar contains five types of productions:

- `UnitProduction`: $X \rightarrow Y$
- `ConcatProduction`: $X \rightarrow Y\ Z$
- `TerminalProduction`: $X \rightarrow$ <terminal>
- `UnaryOpProduction`: $X \rightarrow op(Y)$
- `BinaryOpProduction`: $X \rightarrow op(Y, Z)$

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

```
String s1 = "␣foo";                                    1
s2 = s3 + "bar";                                       2
s4 = s5.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 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 string concatenation using the + operator. For the example in 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 an `Operation` 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. The `Operation` objects also contain information about possible arguments and implement the character set transformation and regular approximation needed for the approximation of the grammar described in 4.3. This language agnostic representation of string operation allows developers of the Code Property Graph (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`.

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

Unlike Christensen et al. [1], we do not consider the total DFG when extracting the grammar. While they parse the whole graph into a data structure, to later extract automata for specific Nodes, 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 data flow graph (DFG) conditionally, stopping at nodes representing numbers. If the traversal reaches such a node, it uses a `ValueEvaluator` to try, whether the value the node represents is known. 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, we need to eliminate all cycles in our grammar that contain operation productions. All nonterminals are assigned a character set, containing all characters that make up the words in the language of the corresponding nonterminal. 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 char `o` is replaced by a known char `n` has the following character set transformation,

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

whereas for a `replace` operation, where the newly inserted char is not known, S is transformed to Σ if the replaced char is contained in S.

These approximations, together with the terminals where the character set is known, for example a string literal, can be used in a fixed point computation to assign a

character set $C(X)$ to each nonterminal X.

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 $C(X)^*$.

We find those operation cycles by viewing the grammar as a graph and determining the strongly connected components (SCCs) of this graph. Now for each nonterminal N in a given component C, we check, whether it has an operation productions, and if yes, whether one of the nonterminals 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.

To determine the SCCs, we use Tarjan's algorithm [3]. This algorithm topologically sorts the returned components in reverse order, which is necessary for the fixpoint computation used to find the charsets. 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 been handled earlier.

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:

`SigmaCharSet union SigmaCharSet`	$\hat{=}(\Sigma \setminus A) \cup (\Sigma \setminus B)$	$= \Sigma \setminus (A \cap B)$
`SigmaCharSet union SetCharSet`	$\hat{=}(\Sigma \setminus A) \cup S$	$= \Sigma \setminus (A \setminus S)$
`SetCharSet union SetCharSet`	$\hat{=}$	$S_1 \cup S_2$
`SigmaCharSet intersect SigmaCharSet`	$\hat{=}(\Sigma \setminus A) \cap (\Sigma \setminus B)$	$= \Sigma \setminus (A \cup B)$
`SigmaCharSet intersect SetCharSet`	$\hat{=}(\Sigma \setminus A) \cap S$	$= S \setminus A$
`SetCharSet intersect SetCharSet`	$\hat{=}$	$S_1 \cap S_2$

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

Strongly Regular Grammars

Mohri and Nederhof describe an algorithm to transform a context free grammar (CFG) into a strongly regular grammar that approximates the given CFG.

They define strongly regular grammars as follows:

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

$$A\mathcal{R}B \Leftrightarrow (\exists \alpha, \beta \in V^* : A \xrightarrow{*} \alpha B \beta) \wedge (\exists \alpha, \beta \in V^* : B \xrightarrow{*} \alpha A \beta) \tag{4.2}$$

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 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 nonterminal is on the left side of the production result.

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.

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

Transformation

TODO:

- Transformation verstehen
- Transformation hier erklären
- haha joke als würd ich die verstehen können

Implementation

We can view a grammar as a directed graph, with the nonterminals as nodes and an edge from a node A to a node B iff there is a production with A on its left-hand side and B contained in its right-hand side, so a production of form $A \to \alpha A \beta$.

The aforementioned notion of mutual "reachability", by which $\mathcal{R}$ 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 whether all productions are of the same linearity and apply the transformation described by Mohri and Nederhof to those components where the productions are not either all left or all right-linear.

In our implementation we use

5 Evaluation and Discussion

- How did you test/evaluate your PoC?
  - E.g. case studies, large-scale studies, test bench, etc.
  - What did you do to verify results (if applicable)
- What did you learn from these tests? Depends on your work. E.g.
  - TP/TN/FP/FN rates
  - Performance
  - Results of your studies
  - Interpretation of the results, lessons learned
- Limitations of the approach and your implementation. Any ideas on how to fix them?

Probably 5-15 pages

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. [1]. 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. [2] 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 [4] 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?
≤ 1 page

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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