Towards the Automated Generation of Island Grammars

Rosetta Roberts and Isaac Griffith
Informatics and Computer Science
Idaho State University
Pocatello, Idaho, 83208
Email: {robepea2@isu.edu, grifisaa@isu.edu}

***Abstract*—Background: Since the introduction of island grammars, they've been successfully used for a variety of tasks. This includes impact analysis, multilingual parsing, source code identification, and other tasks. However, there has been no attempt to automate the generation of island grammars.**

Objective: In this paper, we presented a tool to automate the generation of island grammars. It functions by receiving as input grammars from different programming languages. It then outputs and island grammars for each input grammar that describes structure and content not described by other input grammars. We perform this by converting the input grammars into equivalent graph structures, identifying maximum common subgraphs between the input grammars, removing these subgraphs from the input graphs, and then converting the remains into grammars.

Methods: What is the statistical context and methods applied?

Results: What are the main findings? Practical implications?

Limitations: What are the weaknesses of this research?

Conclusions: What is the conclusion?

***Index Terms*—**

1. Introduction

Code analysis tools are an essential part of modern coding. Using them, software engineers find bugs, identify security flaws, increase future maintainability, and comply with business rules and regulations. Modern development workflows use code analysis tools to provide coding assistance, thereby increasing productivity. Software build processes integrate them into their pipelines for quality assurance and other services. Improvements in the technologies that these tools use allow them to solve more problems.

1.1. Problem Statement

Codebases in modern work environments are becoming increasingly multilingual. For example, a stack for a modern web application might use 5 different languages from the start (e.g. SQL, Java, TypeScript, HTML, CSS). Multilingual codebases challenge existing code analysis tools. Especially of note is the difficulty of building static analysis tools that work across multiple programming languages 1. Current approaches focus on handwriting a grammar for each language that a tool wants to support. Our goal is to find and evaluate a way to automate this process.

1.2. Research Objectives

Many code analysis tools transform source code into alternative forms that are easier to process before performing their analysis. One of these forms is an abstract syntax tree. The abstract syntax tree of code preserves the meaning and structure of the code, while removing details that are usually unimportant, such as spaces. The abstract syntax tree produced for code is specific to the grammar used. The focus of this research is to develop an automated method to combine grammars into an island grammar. This island grammar should describe similar constructs in the original grammars the same so that tools designed to work on this grammar will function for any programming language in the initial grammars. We anticipate that this will allow tool designers to easily support multiple programming languages in a much more maintainable way. This is especially relevant as many codebases are becoming multilingual. There is currently extremely little research on the development of multilingual parsers, which has made static analysis of these multilingual codebases difficult 1.

1.3. Context

Code analysis tools are an essential part of modern coding. Using them, software engineers find bugs, identify security flaws, increase future maintainability, and comply with business rules and regulations. Modern development workflows use code analysis tools to provide coding assistance, thereby increasing productivity. Software build processes integrate them into their pipelines for quality assurance and other services. Improvements in the technologies that these tools use allow them to solve more problems.

1.4. Organization

2. Background and Related Work

2.1. Context Free Grammars

A context free grammar can be described as $G = (V, \Sigma, P, S)$ [2]. V is the set of non-terminal symbols. Σ is the set of terminal symbols. $P \subseteq V \times (V \cup \Sigma)^*$ is the set of productions describing how the symbols of V can be substituted for other symbols. These productions are written as $a \to b$ with $a \in V$ and $b \in (V \cup \Sigma)^*$. When b is empty, the production is denoted by $a \to \varepsilon$. $S \in V$ is the starting symbol. A valid string is represented by a grammar if it can created by repeatedly applying productions [3]. $L(G)$ is used to denote the set of all valid sentences, or language, of grammar G.

2.2. Abstract Syntax Trees

An abstract syntax tree is an ordered tree describing the symbols and productions of a grammar that are applied for a given sentence in the grammar. An ordered tree is a tree for which the children of each node are numbered. Given a grammar $G = (V, \Sigma, P, S)$, the leaf nodes of a tree must be elements from $\Sigma \cup \{\varepsilon\}$ (ε represents the empty string). The concatenation from left to right of the leaf nodes should equal the sentence for which the parse tree is for. Each internal node must be a member of V. For each internal node v with children $c_1, c_2, ...$, the production $v \to c_1 c_2 ...$ must be a member of P. The root node of the tree must be S. [4]

2.3. Island Grammars

Island grammars are specialized grammars designed to match certain constructs of interest, called islands [3]. They are also designed to blindly match surrounding content that is not of interest as water. They offer several advantages over regular grammars for many applications including faster development time, lower complexity, and better error tolerance. Island grammars have been used for many applications including documentation extraction and processing [5], impact analysis [6], and extracting code embedded in natural language documents [7] [8]. Of particular interest to our research is their use for creating multilingual parsers [9], which inspired this research, and the development of tolerant grammars [10] [11] [12].

2.4. Tolerant Grammars

Tolerant grammars were initially designed as a way of making island grammars more robust. To do this, Klusener and Lammel 10 first identified possible problems with island grammars. False positives are content identified by the island grammar as islands even though they are not constructs of interest. False negatives are constructs of interest that the island grammar fails to recognize as islands. To minimize the number of false positives and false negatives, 10 came up with the idea to share productions between the island grammar and the full grammar it's related to. The grammars created in this process minimize false negatives and false positives. The shared portions between the island grammar and the full grammar must start at their start nodes and extend to a certain depth.

2.5. Maximum Common Subgraph (MCS)

The maximum common subgraph problem is the problem where you try to find maximal isomorphic subgraphs in two labeled graphs. It's commonly used in bioinformatics for finding similarities between chemical structures [13]. This problem is usually broken down into different variations depending on the connectedness of the subgraph, the maximality measure used, and whether approximate or exact solutions are desired. These variations can usually be converted between one another with little work (with exception between approximate and exact variations). For our research, we depend on the connected maximum common induced subgraph variation (connected MCIS). An induced subgraph S of a graph J has an edge between two vertices if and only if J has the same edge between the same two vertices. In connected MCIS, the subgraphs found must be connected and must be induced. The maximality measure used is the number of vertices in the isomorphic subgraphs.

The maximum common subgraph problem is an NP-Complete problem 14. Because of the difficulty of the problem, several algorithms have been designed for solving the different variations. The algorithms for exact solutions tend to fall into two different kinds of algorithms. The first kind are clique-based algorithms that depend on reducing the MCS problem to the maximal-clique (MC) problem. This is done by calculating the modular product between two graphs. The second kind of algorithms used are backtracking algorithms. These algorithms function by modeling possible solutions with a search tree and then pruning away non-promising solutions.

For more than two graphs, this problem reduces to the maximal frequent subgraph problem. While these problems might seem quite similar, the algorithms for solving them are quite different.

3. Experimental Design

3.1. Goals, Hypotheses and Variables

We hypothesize that our method for creating multilingual grammars has the following properties:

- The grammar should accept exactly the sentences that are valid in any of the languages used to build the grammar. $s \in L(G_1) \cup L(G_2) \leftrightarrow s \in L(G_{12})$.

3.2. Experimental Design

3.3. Experimental Subjects

3.4. Experimental Objects

3.5. Instrumentation

3.6. Data Collection Procedures

3.7. Analysis Procedures

3.8. Evaluation of Validity

4. Execution

4.1. Sample

4.2. Preparation

4.3. Data Collection Performed

4.4. Validity Procedures

5. Analysis

5.1. Descriptive Statistics

5.2. Data Set Reduction

5.3. Hypothesis Testing

6. Interpretation

6.1. Evaluation of Results

6.2. Limitations of Study

6.3. Inferences

6.4. Lessons Learned

7. Conclusions and Future Work

7.1. Summary of Findings

7.2. Relation to Existing Evidence

7.3. Impact

7.4. Limitations

7.5. Future Work

Appendix

Appendices

[1] Z. Mushtaq, G. Rasool, and B. Shehzad, "Multilingual Source Code Analysis: A Systematic Literature Review," *IEEE Access*, vol. 5, pp. 11307–11336, 2017.

[2] M. Haoxiang, *Languages and Machines An Introduction to the Theory of Computer Science*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 1988.

[3] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 13–22.

[4] D. M. Reinhard Wilhelm, *Compiler Design*. Boston, United States: Addison-Wesley, 1995.

[5] A. V. Deursen and T. Kuipers, "Building documentation generators," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). "Software Maintenance for Business Change" (Cat. No.99CB36360)*, 1999, pp. 40–49.

[6] L. Moonen, "Lightweight Impact Analysis using Island Grammars." in *IWPC*, 2002, pp. 219–228.

[7] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 308–318.

[8] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci, "Extracting structured data from natural language documents with island parsing," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011, pp. 476–479.

[9] N. Synytskyy, J. R. Cordy, and T. R. Dean, "Robust multilingual parsing using island grammars," in *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, 2003, pp. 266–278.

[10] S. Klusener and R. Lammel, "Deriving tolerant grammars from a base-line grammar," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, 2003, pp. 179–188.

[11] A. V. Goloveshkin and S. S. Mikhalkovich, "Tolerant parsing with a special kind of ńAnyż symbol: the algorithm and practical application," *Proceedings of the Institute for System Programming of the RAS*, vol. 30, no. 4, pp. 7–28, 2018.

[12] J. Kur, M. Lungu, R. Iyadurai, and O. Nierstrasz, "Bounded seas," *Computer languages, systems & structures*, vol. 44, pp. 114–140, 2015.

[13] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," p. 13.

[14] M. R. Garey and D. S. Johnson, *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman, 1979.