

Grammar Recovery from Parse Trees and Metrics-Guided Grammar Refactoring

Nicholas A. Kraft, *Member, IEEE Computer Society*, Edward B. Duffy, and Brian A. Malloy

Abstract—

Many software development tools that assist with tasks such as testing and maintenance are specific to a particular development language and require a parser for that language. Because a grammar is required to develop a parser, construction of these software development tools is dependent upon the availability of a grammar for the development language. However, a grammar is not always available for a language, and in these cases, acquiring a grammar is the most difficult, costly, and time-consuming phase of tool construction. In this paper we describe a methodology for grammar recovery from a hard-coded parser. Our methodology comprises manual instrumentation of the parser, a technique for automatic grammar recovery from parse trees, and a semi-automatic metrics-guided approach to refactoring an iterative grammar to obtain a recursive grammar. We present the results of a case study in which we recover and refactor a grammar from version 4.0.0 of the GNU C++ parser and then refactor the recovered grammar using our metrics-guided approach. Additionally, we present an evaluation of the recovered and refactored by comparing it to the ISO C++98 grammar.

Index Terms—Grammar, grammar recovery, grammar refactoring, grammar metrics, parse tree

I. INTRODUCTION

TO facilitate the software development process, developers use tools that assist with implementation, testing, debugging, comprehension, and maintenance of the software under development. Many of these tools are necessarily specific to the development language and also require a parser for that language [1], [2], [3], [4], [5], [6]. For example, a source code editor that supports automated refactoring for Java can not be developed without a parser for Java. Because a grammar is required to develop a parser for a language, construction of many software development tools is dependent upon the availability of a grammar for the development language.

Unfortunately, a grammar does not exist for every language. When a grammar for a language is not available, acquiring a correct and complete grammar for that language is the most difficult, costly, and time-consuming phase of constructing a tool for use with the language [6], [7], [8], [9]. To address the problem of grammar acquisition, researchers have directed significant attention to the problem of *grammar recovery*, which comprises the procedures involved in the derivation of a grammar for a language from the available resource(s) [10]. Lämmel and Verhoef partition approaches to grammar recovery according to the resource(s) available, including: (1) code samples, (2) a language reference manual, or (3) compiler sources (either BNF or a hard-coded parser) [7]. Note that in a given situation one or more of these resources may or may not be available. As stated above, grammar recovery from

each of these resources is difficult, costly, and time-consuming; however, the issues associated with each resource are unique.

Recovering a grammar from only code samples is called *grammar inference*. Gold's theorem states that it is impossible to infer the grammar of an arbitrary unknown language from only positive (syntactically correct) code samples [11]; thus, Črepinšek, et al. require both positive and negative (syntactically incorrect) code samples to infer a grammar using a brute force approach [2]. Recovering a grammar from a language reference manual necessarily involves much manual effort, and reference manuals often contain errors [7]. Lämmel and Verhoef use a language reference manual (including code samples from that reference manual) to semi-automatically recover a grammar. Recovering a grammar from a BNF specification requires that such a BNF specification exist, whether in the form of input to a parser generator or source file comments. In the former case the BNF specification might be littered with tool-specific markup, including semantic actions, while in the latter case the BNF specification might not reflect the grammar actually encoded in the source code. Sellink and Verhoef automatically recover a grammar from BNF found in the source code of a compiler [6]. Finally, recovering a grammar from a hard-coded parser requires manual inspection of source code. Lämmel and Verhoef describe inspecting the source code of a hard-coded parser to recover, by hand, a grammar [7]. Duffy and Malloy describe a related, but distinct, approach to recovering a grammar from a hard-coded parser. They instrument the source code of a hard-coded parser to generate parse trees; using these parse trees, they automatically recover a grammar [4]. In this paper we extend their work.

Our work on recovering a grammar from a hard-coded parser is motivated by situations in which neither a language reference manual, nor a BNF specification is available. Further, we are focused on large languages such as C++ Java, and C#. Because existing approaches to inferring a grammar using brute force [2] and genetic algorithms [12] have not been shown to scale, we chose not to consider grammar inference. In particular, our work is motivated by *language dialects*, which are variations or extensions of a language. Language dialects are rarely accompanied by a grammar and sometimes are not even accompanied by a language reference manual or ad hoc language specification [7]. Consider the most recent versions of the GNU C++ compiler, g++. Prior to version 4.0.0, the GNU team implemented the g++ parser using Yacc. Thus, a BNF specification of the grammar was included in the g++ source code. However, beginning with version 4.0.0, the GNU team has implemented the g++ parser using a hand-written, backtracking recursive descent parser. This hard-coded

parser, which is located in the file `cp/parser.c`, is not accompanied by any form of language description; that is, the only representation of the grammar for the C++ dialect recognized by `g++` is the programmatic representation found in `cp/parser.c`. Similar situations exist for other well-known language dialects, including Eclipse C/C++ and IBM Jikes Java.

A significant issue associated with the recovery of a grammar from a hard-coded parser is that parser generation algorithms place restrictions on the form of a grammar. These restrictions make the grammar difficult to comprehend, meaning the grammar might not be useful in its recovered form. For example, left recursion often is used to introduce repetition in a grammar. However, a recursive descent parser can not recognize a grammar in which a nonterminal is expressed using left recursion. In some cases, such a nonterminal can be rewritten with right recursion; however, in other cases, such a nonterminal can not be easily rewritten with right recursion. In the latter cases, the nonterminal must be rewritten using iteration; that is, it must be rewritten with all possible production right hand sides expressed explicitly. We define a *recursive grammar* as one that contains a nonterminal expressed using left or right recursion and an *iterative grammar* as one that contains a nonterminal expressed using iteration. If a grammar contains both kinds of nonterminal, we refer to it as iterative. Unlike a recursive grammar, an iterative grammar is illegible to the average software developer [13]; further, an iterative grammar is more verbose than the corresponding recursive grammar. Finally, multiple implementations of an iterative grammar can result in distinct versions of the grammar, because iteration must be bounded and the bound is an implementation-defined value.

In this paper we extend the work of Duffy and Malloy [4] and present a case study in which we recover and refactor a grammar from a hard-coded parser. We use `hylian-c++`, which is based on version 4.0.0 of `g++`, to obtain parse trees for C++ programs. `hylian-c++` includes a modified version of `cp/parser.c` that was instrumented by Duffy and Malloy to produce parse trees in XML. While much manual effort was required to perform this instrumentation, we believe that this effort is no more significant than the manual effort required by other grammar recovery techniques. Further, once we obtain the parse trees, grammar recovery from those parse trees is automated. Because our technique for grammar recovery is based on parse trees, which are frequently utilized for testing and debugging of language-dependent software [3], [14], [15], [16], [17], we also believe that the technique can be applied successfully elsewhere. In addition to grammar recovery, we address refactoring of a recovered grammar. Our work on refactoring a recovered grammar is motivated by problems that result from recovering a grammar from a compiler source, such as a hard-coded parser. In particular, we present a metrics-guided approach to refactoring an iterative grammar to obtain a recursive grammar. In this approach we leverage the grammar metrics presented by Power and Malloy [18]. While our approach does require human input to identify candidate nonterminals for refactoring, computation of the metrics and refactoring of the identified candidate nonterminals are both

fully automated.

The primary contributions of this paper are:

- A description of a technique for automatic grammar recovery from parse trees.
- A description of a metrics-guided approach to grammar refactoring. In particular, the approach is aimed at rewriting an iterative grammar to obtain a recursive grammar.
- A case study in which we recover and refactor a grammar from a hard-coded parser. In particular, we recover a grammar for GNU C++ using our technique and 12 test cases that range from 2K lines of non-commented, non-preprocessed lines of code (NCLOC) to 320K NCLOC. We then refactor the recovered iterative grammar using our metrics-guided approach. The resulting recursive grammar is significantly smaller and easier to comprehend.
- An evaluation of our recovered and refactored GNU C++ grammar by comparison to the ISO C++98 grammar.

In the next section we review the research that relates to our work. In Section III we present our methodology for grammar recovery from parse trees and metrics-guided grammar refactoring. In Section IV we provide an overview of our system and in Section V we provide some results from our recovery of the GNU C++ dialect. In Section VI we list the threats to the validity of our work. Finally, in Section VII we draw conclusions.

II. RELATED WORK

In this section we review the research that relates to our work in grammar recovery from parse trees and grammar refactoring to replace iteration with recursion.

Lämmel and Verhoef describe a sequence of cases that cover virtually all of the approaches for recovering a grammar [7]. They refer to the grammar recovery process as *grammar stealing*, since the language already exists and the goal of grammar recovery is to leverage existing language artifacts to “steal” the grammar. The sequence of cases that they enumerate are distinguished by the language artifacts that are available for the recovery process. The first case in the sequence distinguishes those artifacts that include compiler sources from those that include only a language reference manual. The case where compiler sources are available can be distinguished further by the availability of a hard-coded parser or by the availability of a parser generator, which includes a BNF specification of the grammar in the compiler source code. The case of an available hard-coded parser is addressed in this paper. In the case of a parser generator, the available BNF can itself be parsed and extracted from the parser generator. In the case where a language reference manual is the only available artifact, the approaches to grammar recovery can be distinguished by those that use general rules from those that use test cases or program samples. Sometimes the language reference manual includes useful code samples and, in other cases, the language manual is not accompanied by code samples.

In the next section we summarize research that uses a hard-coded parser to recover a grammar. In Section II-B we summarize research that uses a language reference manual, and

possibly code samples, to recover a grammar. In Section II-C, we describe research about grammar refactoring to introduce recursion.

A. Grammar Recovery from a Hard-Coded Parser

Sellink and Verhoef present a completely automated approach to grammar recovery using the source code of a compiler [6]. Their approach leverages a parser for which the grammar is encoded in a dialect of BNF. They translated the extracted production rules into the modular syntax definition formalism (SDF) and used the recovered grammar in the development of a Software Renovation Factory [6], [19]. They applied their approach to a switching system language (SSL), which consists of approximately 20 sublanguages, and were able to generate approximately 3,000 production rules in half a day. There are several advantages to the approach of Sellink and Verhoef. First, their approach is applicable when the language reference manual is unusable. Second, they require neither code samples nor parse trees in their grammar recovery. A disadvantage of their approach is that they do require that the BNF of the grammar be included in the compiler source code.

Duffy and Malloy describe an approach to grammar recovery for a dialect of the C++ language [4]; the current paper is an extension of this previous research. An advantage of their approach is that the grammar need not be hard-coded in the parser and, given parse trees for the language, their approach is fully automated. However, their approach does require an existing parser and, to generate the parse trees, they may have to modify the parser by inserting probes into the semantic actions of the parser. Furthermore, their approach to introduction of left recursion in place of iteration requires a priori knowledge of the grammar.

There are many examples of reuse of parser output in the literature [3], [20]. For example, Devanbu describes a language framework, *Genoa*, a portable, language-independent analysis tool for abstract syntax graphs (ASGs) generated from an existing parser [3]. An implementation of *Genoa*, *Gen++*, leverages the parser output of CFront to produce a simplified view of an ASG representation of the C++ source code. *Gen++* permits analysis of the C++ source code, but does not recover the CFront dialect of the C++ grammar.

B. Grammar Recovery from a Language Reference Manual

In this section, we review the research that deals with grammar recovery from a language reference manual and, possibly, code samples. Lämmel and Verhoef observe that the manual can be either a compiler vendor manual or an official language standard [7]. Moreover, the language is explicated either through code samples, through general rules, or through a combination of both samples and rules. For example, the C++ language standard includes both rules and code samples, and the code samples have been extracted from the standard to form a test suite that has been used to gauge C++ compiler conformance to the language standard [21], [22], [23]. For each published work that we review, we list the particular language artifact used in the grammar recovery process.

Lämmel and Verhoef present a semi-automated approach to grammar recovery that uses a language manual and a test suite [10]. They use the manual to construct syntax diagrams for the language, correct the diagrams, write transformations to correct connectivity errors, and then use the test cases to further correct the generated grammar. There are several advantages to their approach. The first advantage is that the grammar can be recovered quickly; for example, recovery of a Cobol grammar required only two weeks effort [7]. A second advantage of their approach is that their grammar recovery technique is not connected to a specific parser implementation. A disadvantage of their approach is that many phases of the recovery process are manual.

Črepinšek, et al. use a brute force approach to address the problem of grammar recovery [2]. Using both positive and negative code samples, they systematically search for valid parse trees, stopping when either the parse tree produces a context-free grammar that accepts all positive code samples and rejects all negative code samples, or when the search is exhausted. Using their implementation, GIE-BF, they were able to recover a grammar provided that it consisted of no more than five productions. Their promising work was extended using a genetic algorithm and an interesting fitness function that assigns a fitness value between 0 and 1. Their evolutionary approach enabled them to recover grammars for small domain-specific languages such as DESK and a simplified version of FDL [24], [25]. Their approach does not scale to grammar recovery for languages consisting of hundreds of productions, such as C# and C++.

Dubey et al. describe an iterative approach with backtracking that uses positive code samples and a grammar approximation to reverse engineer a grammar for a language dialect [26]. A set of possible rules is constructed during each iteration and one of them is added to the grammar. After a certain number of iterations, the extended grammar is evaluated to determine if it parses all of the code samples; if the grammar does not parse all of the code samples, the algorithm backtracks to the previous iteration and selects another rule.

Sakakibara describes a useful application of grammar inference to bioinformatics [27]. DNA, RNA and protein sequences are viewed as strings of a formal language on alphabets of four nucleotides A, C, G, and T; these strings can be described by a formal grammar. Grammar inference methodology can be applied to these sequences for biological analysis. Sakakibara demonstrates the utility and import of applying grammatical representations, especially stochastic grammars, to the problem of biological sequence analysis.

C. Research on Replacing Iteration with Recursion

We found no reported research dealing with grammar refactoring to replace iteration with recursion; thus, the research that we present is unique in this regard. However, Lohmann et al. present a technique for preserving semantic rules during left recursion removal in attribute grammars [13]. They observe that even though many syntactic structures are expressed naturally using recursion, removal of left recursion is required for recursive descent parsers and many grammar engineering

tools such as ANTLR, JavaCC, and TXL. The GNU C++ grammar that we recover is derived from parse trees generated from a recursive descent parser; thus, the recovered grammar is devoid of recursion. To produce a more readable grammar, we present a semi-automatic technique to refactor recursion into the recovered grammar.

III. METHODOLOGY FOR GRAMMAR RECOVERY AND REFACTORING

In this section we describe a technique for automatic grammar recovery from parse trees and a metrics-guided approach to semi-automatic grammar refactoring. Our approach to grammar refactoring comprises three steps: computation of grammar metrics, analysis of the metrics to identify candidate nonterminals to be transformed by replacing iteration with left recursion, and transformation of the candidate nonterminals. The first and third steps are fully automated; the second step is manual.

By definition, a parse tree captures the derivation of a sentence from a language; that is, a parse tree encodes the productions of a grammar that are exercised by the sentence in the language from which the parse tree is derived. Thus, a parse tree encodes an instance of the grammar, a *partial grammar*, for the language, and we can recover a partial grammar for a language from a parse tree. By taking the union of two partial grammars, that is, the union of the productions in the two partial grammars, we can recover a grammar that captures the productions encoded in each of the corresponding parse trees. It follows that by taking the union of all partial grammars recovered for a test suite we can forge a grammar that generates the sentences in the test suite. This observation serves as the basis for the technique for automatic grammar recovery from parse trees that we describe in Section III-A.

Depending on the parsing technology used by the parser that generates the parse trees, we might recover an iterative grammar rather than a recursive grammar. Because all possible production right hand sides are expressed explicitly for a non-terminal written using iteration, a recovered iterative grammar will generate only a subset of the intended language unless an exhaustive test suite is used in conjunction with the grammar recovery technique described in the previous paragraph. This is in stark contrast to the model typically used when developing language-based software, where the grammar generates a superset of intended language and semantic rules are used to eliminate invalid sentences. Furthermore, as described in Section I, an iterative grammar is likely an artifact of a parser, and a parser can force artificial restrictions on the form of a grammar. The introduction of left recursion in place of iteration is an example of a transformation that makes a grammar more useful to another application [5], as well as more useful to a human [13]. These observations motivate the metrics-guided approach to semi-automatic grammar refactoring that we describe in Section III-B.

A. Grammar Recovery from Parse Trees

Our methodology for automatic grammar recovery minimally requires as input a single parse tree but can accept

multiple parse trees with no modification. For simplicity, in this section we describe the recovery of a (partial) grammar from a single parse tree. Figures 1 and 2 illustrate our approach to recovering a grammar from a parse tree. In Figure 1 we illustrate three example artifacts for the expression $a + 1$, and in Figure 2 we illustrate our grammar recovery algorithm.

Figure 1a shows a graphical representation of an example parse tree that encodes the expression $a + 1$ for a C++-like language. By definition, the interior nodes of the parse tree are labeled by nonterminals from the grammar and the leaf nodes are labeled by terminals from the grammar. Further, the parse tree is hierarchical, and each pair of levels encodes a production rule. Given hierarchical data and an accompanying schema one can express that data as an XML document. While our technique for grammar recovery from parse trees does not require that those parse trees be encoded as XML documents, such an encoding further elucidates the relationship between the parse tree and the partial grammar. Thus, with the grammar serving as the schema, we encode the example parse tree from Figure 1a as the XML illustrated in Figure 1b.

The XML-encoded parse tree shown in Figure 1b contains two categories of tags. The first category contains only one tag, `token`, which is used to represent terminals. The `token` tag uses the `type` attribute to differentiate terminal types, and where appropriate, the `value` or `expr_value` attribute to hold an instance value. The second category of tags contains all tags other than `token` and is used to represent nonterminals. The names of the tags in this category differentiate the nonterminals and none of these tags have attributes. Note lambda productions are encoded as singleton tags; for example, the lambda production shown at the bottom left of Figure 1a corresponds to the singleton tag `nested_name_specifieropt` from Figure 1b.

```

1  grammar = { }
2  recover_production(node)
3      production = [ node ]
4      foreach n in node.children
5          production.append(n)
6      return production
7
8  recover_grammar(root)
9      nodes = { root }
10     foreach n in nodes
11         if n is interior node
12             grammar.add(recover_production(n))
13             nodes.add(n.children)
14     nodes.remove(n)

```

Fig. 2. Grammar Recovery Algorithm. *The important steps in our algorithm to recover a grammar from a parse tree.*

Figure 1c illustrates the grammar that the algorithm shown in Figure 2 recovers from the parse tree illustrated graphically in Figure 1a and as an XML document in Figure 1b. For example, the second production listed in Figure 1c

$$\begin{aligned} & \text{binary_expression} \\ & \rightarrow \text{postfix_expression PLUS binary_expression} \end{aligned}$$

is recovered from the root of the parse tree and its three immediate children (which can be encoded as the outermost

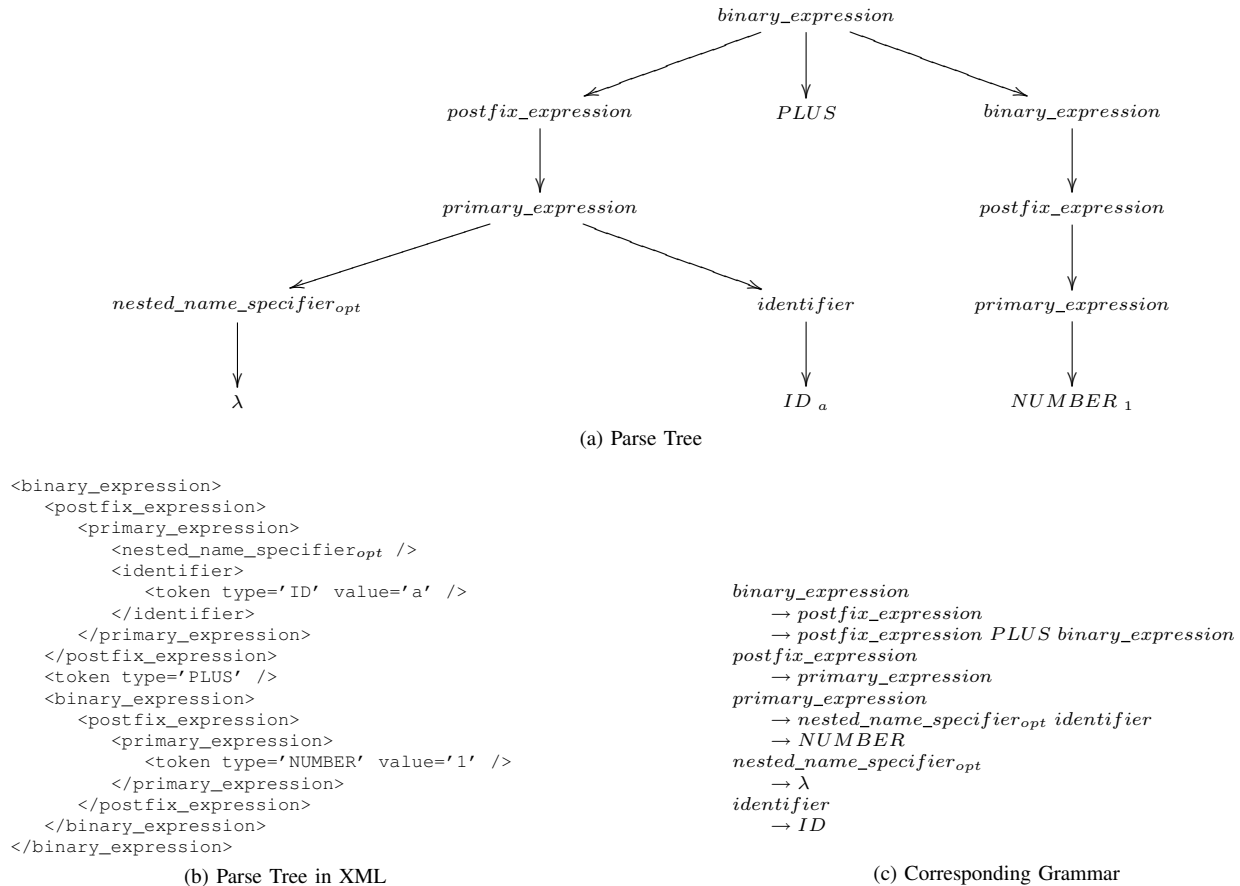


Fig. 1. Parse Trees and Corresponding Grammar. Two representations of an example parse tree for the expression $a + 1$ and the corresponding grammar.

XML tag and its three immediate children tags).

Figure 2 lists our algorithm for recovering a grammar from a parse tree. Line 1 of the figure lists the declaration for the global set `grammar`, which holds the recovered grammar. The grammar we recover using our algorithm is represented as a set of lists, where each list represents a production. Further, each the first item in each list is a nonterminal that represents the left hand side of the production and the subsequent items are terminals or nonterminals that represent the right hand side of the production. The algorithm consists of two subroutines and commences when the root of the parse tree is passed to `recover_grammar`, which is listed on Line 8 of Figure 2. On line 9 of the figure a set `nodes` is initialized to include only the root node. The loop that begins on line 10 adds the children of the current node, `n`, to `nodes` each time an interior node of the parse tree is encountered. Further, as an interior node corresponds to the left hand side of a production, each such node is passed to the `recover_production` subroutine on line 12 of Figure 2. The result of the subroutine call, a production in list form, is added to `grammar`. Upon termination of the `recover_grammar` subroutine, the grammar encoded by the parse tree is stored in `grammar`.

B. Metrics-Guided Grammar Refactoring

Our technique for automated grammar recovery might recover an iterative grammar. In this paper, we have listed

several disadvantages of an iterative grammar, including: it (most likely) represents a subset of the intended language and it is illegible to the average software developer [13]. Therefore, we have developed an approach to refactor a grammar by replacing iteration with left recursion. A key issue is the identification of nonterminals that are expressed using iteration, because the identified nonterminals are candidates to be rewritten with left recursion. We found three of the six grammar size metrics described by Power and Malloy [18] to be useful to guide the identification of these candidates, and we describe these size metrics in the following paragraph.

Number of Productions (PROD), *Average RHS Size (AVS)*, and *Halstead Effort (HAL)* are grammar size metrics that can be computed at the granularity of a grammar or a nonterminal. A large value of PROD for a nonterminal might indicate the use of iteration; however, it might indicate a more general need for refactoring. When computed over the set of productions for a nonterminal, AVS is the average number of symbols on the right-hand side of a production for that nonterminal. A large value for AVS can result from the use of iteration in a grammar. For example, if a grammar includes a nonterminal with a set of productions that generates a parenthesized list of parameters, and if that set of productions is expressed using iteration rather than left recursion, a production that expresses a list of each possible length must be included, which results in a large value for AVS. HAL is a relativization

```

postfix_expression
  → primary_expression
  → postfix_expression . id_expression
  → postfix_expression ( expression_list_opt )

```

Fig. 3. Left Recursive Productions. A left recursive expression of the productions from Figure 4.

of McCabe cyclomatic complexity that weights the counts of unique symbols by the numbers of occurrences of each symbol [18]. The divisor in the equation expressing HAL is the number of unique operands times a constant; thus, minimizing this value will maximize the value of HAL. Similarly to AVS, a large value for HAL can result from the use of iteration in a grammar. Continuing the previous example, the repetition of symbols in the lists of each possible length keeps the number of unique operands low, which increases the value of HAL.

Given the definitions of PROD, AVS, and HAL we expect nonterminals expressed using iteration to exhibit unusually large values for these three size metrics. However, the definition of unusually large is dependent on context and in some cases is subjective. Thus, while we can automatically compute these metrics for a grammar, human intervention is required to use these metrics for identification of candidate nonterminals. We demonstrate the use of these metrics to guide identification of candidate nonterminals in Section V. In the remainder of this section we describe our algorithm to replace iteration with left recursion once a candidate nonterminal is identified.

Figure 4 illustrates five productions and corresponding example sentences. The productions are expressed using iteration and generate a subset of the language that is generated by the productions illustrated in Figure 3. For example, the productions in Figure 4 do not generate the sentence `o.foo(p).bar(q).foobar(r)` but the productions from Figure 3 do. We could add a sixth production to Figure 4 to generate that sentence; however, it would be easy to then identify another sentence generated by the productions from Figure 3 but not by the resulting six productions. Thus, the approach of repeatedly adding iterative productions to a grammar is neither scalable nor elegant. Furthermore, the grammar that results is unwieldy, making it difficult for a human to comprehend, and specific to a particular parsing algorithm, rendering it not useful to other applications.

Figure 5 illustrates our algorithm, which takes as input a nonterminal and its set of right-hand sides expressed using iteration and gives as output the set of right-hand sides expressed using left recursion. The input nonterminal is identified as a candidate for refactoring using the metrics-guided approach described in the previous section. The algorithm considers a right-hand side to be a string of symbols and assumes the following definitions:

- prefix** string a is a prefix of string b if the symbols in a match the first $\text{len}(a)$ symbols of b
- suffix** string a is a suffix of string b if the symbols in a match the last $\text{len}(a)$ symbols of b

Furthermore, the algorithm contains two stages: (1) *Replace Prefixes* and (2) *Remove Suffixes*. In the first stage, we introduce left recursion wherever possible, and in the second stage

```

1  replace_iteration(lhs,rhsSet)
2  # PHASE 1: Replace Prefixes
3  max_len = len(longest rhs in rhsSet)
4  do
5    foreach j in 1..max_len
6      partition rhsSet into three sets: len_j, len_lt_j, len_gt_j
7      foreach x in len_j
8        foreach y in len_gt_j
9          if x is a prefix of y
10             remove first j symbols of y
11             add lhs to front of y
12             if y is not unique in rhsSet
13                 remove y from rhsSet
14  while(rhsSet changes)
15
16  # PHASE 2: Remove Suffixes
17  partition rhsSet into two sets: left_recursive, not_left_recursive
18  foreach lr in left_recursive
19     lr_part = lr.pop_front() # lr without the recursive "call"
20     for n in not_left_recursive where len(n) >= len(lr_part)
21       if lr_part is a suffix of n
22         remove last len(lr_part) symbols of n
23       if n is not unique in rhsSet
24         remove n from rhsSet

```

Fig. 5. Grammar Refactoring Algorithm. The important steps in our algorithm to replace iteration with left recursion.

we eliminate portions of productions if they can be generated by the new left recursion productions.

Figure 3 illustrates the left recursive productions given as output by our algorithm when applied to the iterative productions in Figure 4. As described in a previous paragraph, the language generated by the productions in Figure 3 is a superset of the language generated by the productions in Figure 4. This underscores the importance of the identification of candidate nonterminals, and, in particular, of the role of the human in the identification process. Refactoring a grammar to generate a superset of the intended language can be desirable, but performing this refactoring arbitrarily can significantly change the language recognized by the grammar (in addition to degrading its understandability). For example, consider Figure 6, which illustrates how our algorithm can fundamentally change the language recognized by a grammar if misapplied.

IV. GRAMMAR RECOVERY AND REFACTORING SYSTEM

In this section we present a system that implements the methodology described in Section III. We begin in Section IV-A by presenting an overview of the system and thereby illustrating the processes of recovering parse trees and a grammar and refactoring a grammar. In Section IV-B we briefly describe our approach to validation of the recovered parse trees, a process that is presented in more detail in previous work [4]. We then discuss the interoperability of our system in Section IV-C.

A. System Overview

Figure 7 illustrates an overview of our system, which consists of two major subsystems: the parse tree recovery

| Production | Sentence |
|---|-----------------|
| <i>postfix_expression</i> | |
| → <i>primary_expression</i> | o |
| → <i>primary_expression</i> . <i>id_expression</i> | o.x |
| → <i>primary_expression</i> . <i>id_expression</i> (<i>expression_list_opt</i>) | o.foo(p) |
| → <i>primary_expression</i> . <i>id_expression</i> (<i>expression_list_opt</i>) . <i>id_expression</i> | o.foo(p).y |
| → <i>primary_expression</i> . <i>id_expression</i> (<i>expression_list_opt</i>) . <i>id_expression</i> (<i>expression_list_opt</i>) | o.foo(p).bar(q) |

Fig. 4. Iterative Productions. A partial iterative expression of the productions illustrated in Figure 3 and corresponding sentences.

| | |
|--|--|
| <pre> <i>selection_statement</i> → if (<i>condition</i>) <i>statement</i> → if (<i>condition</i>) <i>statement</i> else <i>statement</i> → switch (<i>condition</i>) <i>statement</i> </pre> | <pre> <i>selection_statement</i> → if (<i>condition</i>) <i>statement</i> → <i>selection_statement</i> else <i>statement</i> → switch (<i>condition</i>) <i>statement</i> </pre> |
| (a) Before Refactoring | (b) After Refactoring |

Fig. 6. Result of Misapplying the Left Recursion Recovery Algorithm. The consequences of misapplying the algorithm illustrated in Figure 5.

subsystem and the grammar recovery and refactoring subsystem. The parse tree recovery subsystem, *hylian-c++*, is shown in Figure 7a, and the grammar recovery and refactoring subsystem, *grecovery*, is shown in Figure 7b. *hylian-c++*, shown as a UML package, takes as input one or more C++ source files and produces as output the corresponding parse tree(s) encoded in XML. *grecovery*, also shown as a UML package, takes as input one or more XML-encoded parse trees and produces as output a recursive grammar. Using *hylian-c++* to recover parse trees from C++ source code is a fully automated process; however, using *grecovery* to recover a recursive grammar from parse trees requires human input. Furthermore, construction of *hylian-c++*, which is based upon version 4.0.0 of the GNU C++ parser, was manual process.

parse2xml, shown in Figure 7a as a UML package, is an augmented version of the GNU C++ parser. To create *parse2xml* we instrumented the file `cp/parser.c` from the `g++` source tree. The source code in `cp/parser.c` implements a hand-written, backtracking recursive descent parser. In general, each function in that file recognizes a single nonterminal; that is, the set of productions associated with a single nonterminal. We inserted probes into these functions to generate an XML-encoded trace of the productions exercised by an input program. In total, our patch to `cp/parser.c` adds 1,148 non-blank, non-commented lines of C source code and preprocessor statements. That `cp/parser.c` is structured nicely facilitated the instrumentation task; however, the coding style used, particularly the use of multiple exit paths from a single function, made instrumentation tedious. In addition, because the parser uses backtracking, the output of *parse2xml* is an annotated parse tree that includes subtrees that result from delayed parsing or *tentative parsing*, which is parsing that is abandoned in favor of backtracking.

The *postprocessor*, shown as a UML package in the middle of Figure 7a, is a 233 line C++ program that converts an annotated parse tree to the actual parse tree. We implemented *postprocessor* to perform this conversion in two steps. In the first step, we backpatch delayed parse subtrees. *parse2xml* emits member function bodies and their default parameter lists after the class that contains these constructs, so we must

backpatch the member function bodies and default parameter lists into the appropriate class to obtain a structurally correct parse tree. The second step performed by *postprocessor* is to commit or rollback the subtrees that result from tentative parsing. Each annotated parse tree indicates which tentatively parsed subtrees were accepted (and thus should be committed) as well as which tentatively parsed subtrees were rejected (and thus should be rolled back, or eliminated). In this second step, we remove these annotations and possibly the annotated subtrees. The output of *postprocessor* is a structurally correct, XML-encoded parse tree.

We provide the XML-encoded parse trees produced by *hylian-c++* as input to the *grecovery* system. In the *Recover Grammar* class, shown in the upper left of Figure 7b, we implement the grammar recovery algorithm illustrated in Figure 2; the output of this class is an *Iterative Grammar*, which we provide both to *SynQ* and to the *Refactor Grammar* class. *SynQ* is a metric computation system for grammars [18]; we use *SynQ* to compute the size metrics exploited by our methodology. The human shown in the lower right of Figure 7b must review the size metrics computed by *SynQ* to identify *Candidate Nonterminals*. Once the candidates are identified, they are provided, along with the *Iterative Grammar*, to the *Refactor Grammar* class. This class implements the grammar refactoring algorithm illustrated in Figure 5 and produces as output a *Recursive Grammar*.

B. System Validation

To recover a correct grammar for a language or language dialect our technique requires a structurally correct parse tree(s). A parse tree is structurally correct if both the terminals and nonterminals from the parsed sentence are structured properly. Thus, our approach to structural validation is two-pronged and comprises validation of correct terminal structure and correct nonterminal structure.

We validate correct terminal structure by comparing source code regenerated from a parse tree to the source code from which the parse tree was generated. Before performing this comparison we must normalize the tokens in both the original and the regenerated source code, because white space,

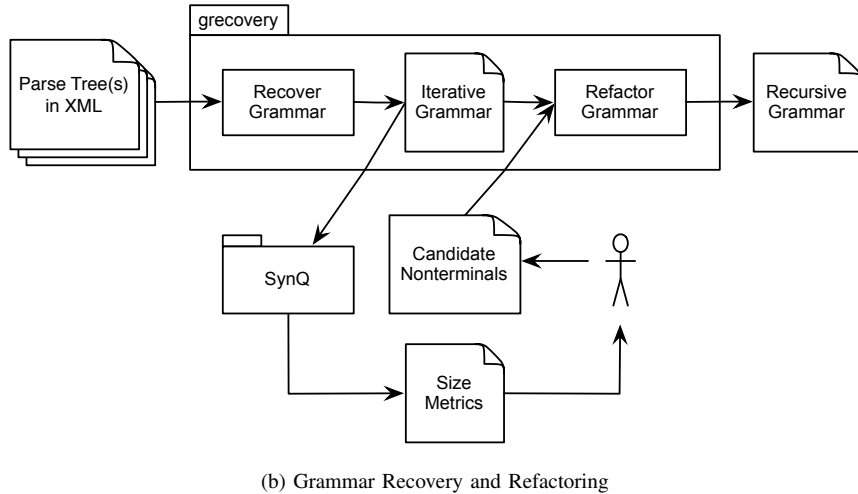
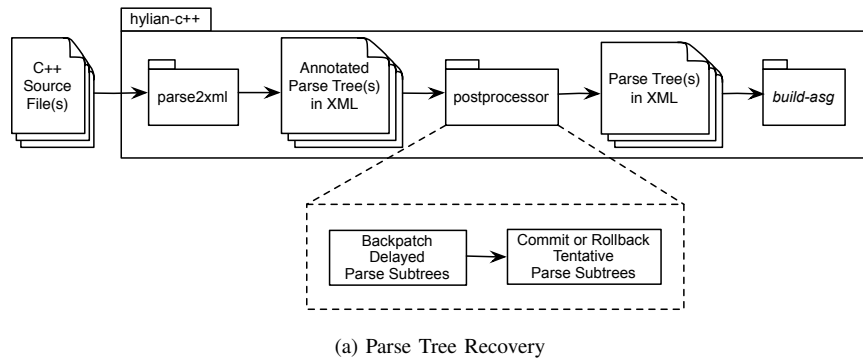


Fig. 7. System Overview. An overview of the architecture of our grammar recovery and refactoring system.

comments, and compiler directives are not represented in a parse tree. Thus, the first step in normalization is to preprocess the original source code and strip any directives that remain, such as `#line` directives. Next, we must normalize numeric tokens, because `g++` (upon which our parse tree generation tool is based) modifies the form of some numeric tokens. For example, `g++` transforms `0.01` to `1e-2`. Finally, we normalize white space by ensuring that exactly one token appears on each line of a source file. Once we have normalized both the original and regenerated source code we perform the comparison using the *diff* utility.

Comparison of the original and regenerated source code is necessary to ensure that the structure of the parse tree is correct. In particular, such a comparison can help to identify missing terminals, improperly ordered terminals, or incorrectly regenerated terminals. Indeed, we found comparison of the original and regenerated source code to be an invaluable debugging aid. However, this comparison does not guarantee that the parse tree is correct; in particular, it does not guarantee the structural integrity of the nonterminals in the parse tree. To validate correct nonterminal structure of an XML parse tree we use its schema and an XML validator, such as *xmllint*. Again, we found this validation to be invaluable, because it uncovered nesting errors in our parse trees during development of *parse2xml*. These errors went undetected by our source regeneration-based validation because nonterminals are not

expressed in regenerated source code.

C. System Interoperability

Interoperability is the cooperation of two or more systems to enable the exchange and utilization of data [28]. Lämmel and Verhoef assert that interoperability of grammars is key and that a recovered grammar must be transformed so that it is useful for another application [10]. Below we describe transformations of our recovered grammar into conforming instances of three schemas: (1) Relax NG, (2) L^AT_EX, and (3) YACC. All transformations are implemented by our system, which can be extended to produce instances of additional schemas as required; one such schemas might be EBNF [29].

Relax NG provides three choices for grammar specification; we use the flattened XML format because of the numerous references to nonterminals and the recursive nature of the C++ grammar. To generate the Relax NG schema, we first designate the `<translation_unit>` element as the root element of the schema. We then group the productions by the left-hand side nonterminal and, for each unique left-hand side, we create `<define>` and `<element>` tags; if there is more than one production for this nonterminal, we also create a `<choice>` tag. Finally, for each production of this nonterminal we generate a `<ref>` tag and attach the terminal or nonterminal on the right-hand side to a `<group>` tag.

To address the need for a human-readable version of the recovered grammar we chose to provide an option to generate L^AT_EX. A human-readable format allows for inspection and comparison of the recovered grammar. Furthermore, it is a first step towards creating a language manual, which rarely exists for a language dialect.

Grammar deployment is the process of turning a given grammar specification into a working parser [30]. The YACC input format is similar to BNF and is ubiquitous. Besides YACC and Bison, many other grammarware tools accept the YACC input format. In particular, SynQ, the grammar metrics computation system written by Power and Malloy [18] and used in our system implementation, accepts YACC formatted input; other grammar measurement systems also accept YACC formatted input [31].

V. CASE STUDY

In this section we present a case study in which we recover and refactor a grammar from a hard-coded parser to demonstrate the feasibility and utility of the methodology we describe in Section III. In particular, we recover a grammar for GNU C++ using the technique we describe in Section V-B and then refactor the recovered iterative grammar using the metrics-guided approach we describe in Section V-C. Finally, we evaluate the recovered and refactored GNU C++ grammar by comparing it to the ISO C++98 grammar.

In Section V-A we describe the four test suites that we use in our studies. We constructed multiple test suites because, using the methodology that we described in Section III, we only can recover a terminal, nonterminal, or production if it is exercised by a test case; that is, if it is contained in the parse tree for a test case. Therefore, the coverage of the subject language that we obtain using a test suite is more strongly dependent on the variety than the number of terminals, nonterminals, and productions exercised. To address this need for variety we selected 12 C++ programs, or test cases, from which we constructed four test suites: (1) Benchmarks, (2) Libraries, (3) Applications, and (4) Mozilla.

In Section V-B we report results we obtained when applying our grammar recovery technique to the four test suites. We report results that address the following research questions:

Q1. *How many terminals, nonterminals, and productions are recovered from the parse trees for each test case, each test suite, and all test suites?* Specifically, we wanted to determine whether the sets of terminals, nonterminals, and productions recovered from the parse trees for the GCC test case, part of the Benchmarks test suite, contain the same terminals, nonterminals, and productions recovered from the parse trees for all test cases. That is, we wanted to determine whether any test case exercises parts of the GNU C++ grammar not exercised by the GCC test case. In addition, we wanted to determine whether the Mozilla test cases exercise precisely the same parts of the GNU C++ grammar.

Q2. *How many unique productions are recovered from the parse trees for each test case and each test suite?* We wanted to determine which test cases, if any, make significant unique contributions to the overall recovered grammar. That

is, we wanted to determine if particular test cases contribute the majority of the unique productions, or if all test cases contribute a proportional share of the unique productions. Moreover, we wanted to determine whether any of the four test cases contribute a disproportionately high or low number of unique productions.

Q3. *How significantly can each test case and each test suite be reduced without sacrificing coverage?* We wanted to determine how many translation units (source files) in each test case and each test suite can be removed without removing terminals, nonterminals, or productions from the recovered GNU C++ grammar.

In Section V-C we report results we obtained when applying our metrics-guided grammar refactoring approach to the recovered GNU C++ grammar. In addition, we report information gathered from our comparison of the recovered grammar to the ISO C++98 grammar. We address the following additional research questions:

Q4. *How many candidate nonterminals are identified using the three size metrics?* We wanted to determine whether the size metrics can clearly identify a nonempty set of candidates.

Q5. *How many of the identified candidate nonterminals can be (reasonably) rewritten with left recursion?* We wanted to determine whether candidate nonterminals identified in the investigation of Q4: (a) are actually expressed using iteration and (b) can be rewritten with left recursion without significantly changing the language recognized by the grammar.

Q6. *How similar are the refactored left recursive nonterminals to their counterparts in the ISO C++98 grammar?* We wanted to compare the nonterminals produced by our refactoring algorithm (shown in Figure 5) to their counterparts in the ISO C++98 grammar to determine if our algorithm behaves as expected.

We obtained all results using a Dell™ OptiPlex™ 755 workstation with an Intel® Core™2 Q6600 processor, 4096 MB of RAM, and a 160GB 7200 RPM SATA hard drive on which we installed the Slackware 12.0 operation system. We used xmllint version 20630 to validate the XML-encoded parse trees, we wrote a handler for the SAX parser included with Python version 2.5.1 to recover a grammar from XML-encoded parse trees, and we implemented our left recursion recovery algorithm in Python. Finally, we wrote both Bash and Python scripts to automate execution of the grammar recovery system.

A. Test Suites

Table I lists information about the 12 test cases that form our four test suites. For each test case we list the name, version, number of C++ translation units (TUs), and the approximate number of thousands of non-commented, non-preprocessed lines of code (NCLOC). In C++, a *translation unit* consists of a source file and all files that it includes either directly or transitively, and we used the source code line counter [32] to compute NCLOC. For each test suite we list the three test cases that form the test suite, as well as the total numbers of C++ TUs and NCLOC for the test suite. In the last row of the table we list the total numbers of C++ TUs and NCLOC for

| Test Suite | Test Case | Version | C++ TUs | NCLOC (\approx K) |
|--------------|-----------|----------|---------|----------------------|
| Benchmarks | Dr. Dobbs | 1.0 | 407 | 4 |
| | GCC | 4.0.0 | 1, 318 | 20 |
| | Keystone | 0.6 | 111 | 2 |
| | Subtotal | | 1, 836 | 26 |
| Libraries | Blitz++ | 0.9 | 153 | 95 |
| | Boost | 1.35.0 | 1, 646 | 320 |
| | POOMA | 2.4.1 | 229 | 115 |
| | Subtotal | | 2, 028 | 530 |
| Applications | Doxygen | 1.5.5 | 79 | 188 |
| | FiSim | 1.1b | 23 | 14 |
| | Fluxbox | 1.0.0 | 119 | 38 |
| | Subtotal | | 221 | 240 |
| Mozilla | Gecko | 1.9b5pre | 254 | 215 |
| | Necko | 1.9b5pre | 114 | 75 |
| | XPCOM | 1.9b5pre | 188 | 120 |
| | Subtotal | | 556 | 410 |
| Total | | 4, 641 | 1, 206 | |

TABLE I

TEST SUITES. *The four test suites that we use in our studies. For each test suite we list the test cases that constitute the test suite. For each test case we list the version, the number of C++ translation units (TUs), and the approximate number of thousands of non-commented, non-preprocessed lines of code (NCLOC). The test suites contain 12 test cases, and the test cases contain over 4, 600 C++ TUs and approximately 1.2 million NCLOC.*

the four test suites. The 12 test cases in the test suites contain over 4, 600 C++ TUs and approximately 1.2 million NCLOC.

The first test suite, **Benchmarks**, consists of three benchmarks designed to exercise C++ compilers, particularly their parsers and front ends. The first test case in **Benchmarks** is Dr. Dobbs, a test suite designed by Malloy, et al. [21] to measure the conformance of a C++ parser to the ISO C++98 standard. We included only the translation units that could be successfully compiled by version 4.0.0 of g++. The GCC test case consists of the translation units from the g++.dg directory of the test suite for the C++ compiler included with GCC [33]. While GCC contains many TUs, many of those TUs contain code written to test code optimization or code generation routines, not parser front end routines. The third test case, Keystone, is the combination of a test suite written by the authors to evaluate Keystone [17], an ISO C++98 conformant parser, and a test suite written by the authors to evaluate g⁴re [20], a tool chain for reverse engineering C++ programs. The Keystone test case is intended to complement the Dr. Dobbs test case.

The second test suite, **Libraries**, consists of the test suites for three C++ class template libraries: Blitz++, Boost, and POOMA. Blitz++ is a scientific computation library that uses templates to achieve performance on par with Fortran [34]. Boost is a highly-regarded free library that is peer-reviewed and portable [35]. POOMA is a collection of class templates that can be used to write parallel PDE solvers [36]. We chose these test cases because they are listed on the *GCC Testing Efforts* page [37], they are listed in the *GCC Release Criteria*, and they are each well-known for making heavy use of templates. However, as these test cases are class template libraries, we had to instantiate them to compile them (and thus to obtain parse trees for them). Therefore, to construct

the **Libraries** test suite we used translations units from the test suites provided with each of the test cases.

The third test suite, **Applications**, consists of three C++ applications: Doxygen, FiSim, and Fluxbox. Doxygen is a documentation system for C, C++, and Java [38]; FiSim is a scientific modeling tool for advanced engineering of fiber and film [39]; and Fluxbox is a light-weight X11 window manager built for speed and flexibility [40]. We chose these applications for their variety of size and application, including a language processing tool, a scientific application, and a window managing system. We specifically chose FiSim for its use of the Loki library [41], which make heavy use of templates.

The fourth test suite, **Mozilla**, consists of three modules from the open source, cross-platform web and email application suite [42]: Gecko, Necko, and XPCOM. Gecko (a.k.a. layout) is a cross-platform rendering engine that forms the core of the Mozilla browser. Necko (a.k.a. network) is the Mozilla network library and provides a platform-independent API for several layers of networking ranging from the transport to the presentation layer. XPCOM is a cross-platform variant of the well-known Component Object Model (COM). In addition to Mozilla being a popular application, it is also frequently used as a test suite for C++ fact extractors and other C++ reverse engineering tools. For example, iPlasma is an integrated environment for analysis of object-oriented software systems written in the C++ or Java languages. The scalability of the iPlasma environment is demonstrated by its ability to handle large-scale projects in the size of millions of lines of code, including Mozilla [43], [44]. Similarly, Mozilla is used to evaluate the TUAnalyzer system [45] and the Columbus reverse engineering tool [46].

In the next two sections we report the results of applying our grammar recovery system to the four test suites.

B. Grammar Recovery Study

In this section we report results we obtained when applying our grammar recovery technique to the four test suites. These results address research questions Q1–Q3 from the beginning of this section. In particular, Table II addresses Q1, Table III addresses Q2, and Table IV addresses Q3.

1) *Measurements of the Recovered Grammars*: Table II lists size metrics for the grammars that we recovered for each test case, for each test suite, and for all test suites. For each recovered grammar we list three size metrics: number of terminals (TERM), number of nonterminals (VAR), and number of productions (PROD). The first size metric, TERM, is listed in the third column of the table. For each test case, TERM is the number of unique terminals found in the recovered grammars for the translation units in the test case. For each test suite, TERM is listed in a row labeled **Combined** and is the number of unique terminals found in the recovered grammars for the test cases in the test suite. For all test suites, TERM is listed in a row labeled **Recovered** and is the number of unique terminals found in the recovered grammars for all test suites. The second metric, VAR, is located in the fourth column of the table, while the third

| Test Suite | Test Case | TERM | VAR | PROD |
|--------------|-----------|------|-----|------|
| Benchmarks | Dr. Dobbs | 90 | 125 | 368 |
| | GCC | 136 | 138 | 612 |
| | Keystone | 117 | 132 | 462 |
| | Combined | 137 | 138 | 620 |
| Libraries | Blitz++ | 123 | 131 | 486 |
| | Boost | 127 | 133 | 723 |
| | POOMA | 122 | 125 | 517 |
| | Combined | 127 | 133 | 765 |
| Applications | Doxygen | 107 | 113 | 401 |
| | FiSim | 120 | 133 | 514 |
| | Fluxbox | 119 | 131 | 515 |
| | Combined | 122 | 133 | 581 |
| Mozilla | Gecko | 115 | 121 | 525 |
| | Necko | 113 | 123 | 445 |
| | XPCOM | 117 | 123 | 462 |
| | Combined | 119 | 124 | 553 |
| Recovered | | 137 | 138 | 930 |

TABLE II

GRAMMAR SIZE. *This table lists the sizes of the grammars recovered for the test cases in our four test suites.*

| Test Suite | Test Case | # This Suite | % This Suite | All Suites | % All Suites |
|--------------|-----------|--------------|--------------|------------|--------------|
| Benchmarks | Dr. Dobbs | 3 | 0.5 | 0 | 0.0 |
| | GCC | 130 | 21.0 | 63 | 6.8 |
| | Keystone | 5 | 0.8 | 3 | 0.3 |
| | Combined | N/A | N/A | 74 | 8.0 |
| Libraries | Blitz++ | 9 | 1.2 | 9 | 1.0 |
| | Boost | 221 | 28.9 | 113 | 12.2 |
| | POOMA | 30 | 3.9 | 24 | 2.6 |
| | Combined | N/A | N/A | 154 | 16.6 |
| Applications | Doxygen | 22 | 3.8 | 8 | 0.9 |
| | FiSim | 40 | 6.9 | 8 | 0.9 |
| | Fluxbox | 34 | 5.9 | 9 | 1.0 |
| | Combined | N/A | N/A | 27 | 2.9 |
| Mozilla | Gecko | 71 | 12.8 | 35 | 3.8 |
| | Necko | 11 | 2.0 | 3 | 0.3 |
| | XPCOM | 14 | 2.5 | 3 | 0.3 |
| | Combined | N/A | N/A | 47 | 5.0 |

TABLE III

UNIQUE PRODUCTIONS. *This table lists the numbers of productions uniquely recovered from each test case and test suite.*

metric, PROD, is located in the fifth column of the table. Both of these metrics are calculated over the same sets of recovered grammars as the first metric.

The data listed in Table II shows that the grammar recovered for the GCC test case contains all but one of the 137 recovered terminals and that it contains all of the 138 recovered nonterminals. However, note that not only does this grammar contain only 65% of the total productions recovered from all test suites, but also that this grammar does not even contain the most productions of the 12 test cases. In particular, the grammar recovered for the Boost test case contains 723 productions, which is 12% more than the grammar recovered for GCC. Nonetheless, the grammar recovered for Boost does not contain 10 of the terminals and 5 of the nonterminals recovered by other test cases in the test suites. We also note that the Mozilla test cases do exercise different parts of the GNU C++ grammar, although there is significant overlap.

Table III lists statistics about the number of unique productions contributed by each test case and by each test suite to the grammars that we recovered for each test suite and for all test suites. For the recovered grammar for each test case we list four statistics; for the recovered grammar for each test suite we list only two statistics. We first describe the two statistics that apply only to the recovered grammars for the test cases; we then describe the two statistics that apply to the recovered grammars for the test cases as well as the recovered grammars for the test suites.

The first statistic is located in the third column of the table and is the number of unique productions contributed to the grammar for this test suite. For example, for the Gecko test case the value in the third column is 71; this means that the recovered grammar for Gecko contains 71 productions that are not found in either the recovered grammar for the Necko test case or the recovered grammar for the XPCOM test case. The second statistic is located in the fourth column of the table and is the number in the third column of the table divided by

the total number of unique productions for the grammar for this test suite. Continuing the previous example, for the Gecko test case the value in the fourth column is 12.8; this means that by uniquely contributing 71 of the 553 productions in the recovered grammar for the Mozilla test suite, Gecko uniquely contributes 12.8% of the productions in that grammar.

The third statistic in Table III is listed in the fifth column and is the number of unique productions contributed to the grammar for all test suites. For example, for the Gecko test case the value in the fifth column is 35; this means that the recovered grammar for Gecko contains 35 productions that are not found in the recovered grammars for the other 11 test cases. Similarly, for the Mozilla test suite the value in the fifth column of the table is 47; this means that the recovered grammar for Mozilla contains 47 productions that are not found in the recovered grammars for the other three test suites. The fourth statistic is listed in the sixth column of the table and is the number in the fifth column divided by the total number of unique productions for the grammar for all test suites. Continuing the previous example, for the Gecko test case the value in the sixth column is 3.8; this means that by uniquely contributing 35 of the 930 productions in the recovered grammar for all test suites, Gecko uniquely contributes 3.8% of the productions in that grammar. Similarly, for the Mozilla test suite the value in the sixth column is 5.0; this means that by uniquely contributing 47 of the 930 productions in the recovered grammar for all test suites, Mozilla uniquely contributes 5.0% of the productions in that grammar.

For three of the four test suites, one of the three test cases in the test suite contributes the majority of unique productions; only in the Applications test suite is there no test case that does so. Yet, there is no test case among the 12 that does not contribute a unique production within its test suite, though 7 of the 12 test cases uniquely contribute less than 5% of the productions in their respective test suites. When considering

| Test Suite | Test Case | # C++ TUs | % Reduction |
|--------------|-----------|-----------|-------------|
| Benchmarks | Dr. Dobbs | 91 | 77.6 |
| | GCC | 129 | 90.2 |
| | Keystone | 32 | 71.2 |
| | Subtotal | 252 | 86.3 |
| Libraries | Blitz++ | 14 | 90.8 |
| | Boost | 85 | 94.8 |
| | POOMA | 17 | 92.6 |
| | Subtotal | 116 | 94.3 |
| Applications | Doxygen | 26 | 67.1 |
| | FiSim | 8 | 65.2 |
| | Fluxbox | 29 | 75.6 |
| | Subtotal | 63 | 71.5 |
| Mozilla | Gecko | 52 | 79.5 |
| | Necko | 28 | 75.4 |
| | XPCOM | 35 | 81.4 |
| | Subtotal | 115 | 79.1 |
| Total | | 546 | 88.2 |

TABLE IV

PERCENTAGE REDUCTION OF TEST SUITES. *The reduced numbers of C++ translation units (TUs) and the percentage reductions for the individual test cases and test suites, as well as the aggregate reduction for the test suites.*

productions uniquely contributed among all 12 test cases, we note that Dr. Dobbs has a total of zero. We also experimented with the FTensor library [47] and found that it too failed to contribute a production among those recovered from our 12 test cases. Finally, we note that the GCC and Boost test cases each contribute disproportionately high numbers of unique productions.

2) *Reduction of the Test Suites:* Hennessy and Power present a test suite reduction technique for grammar-based software that is based on rule coverage [16]. When performing grammar recovery, we store data that allows us to apply an analogous technique. When recovering the grammar for a test case, we store in a set the name of the translation unit that contributes each new production. Upon completion of grammar recovery the resulting set of translation units represents a (possibly) reduced version of the test case, known as the *essential set*. If we apply the grammar recovery system to the reduced test case, we will obtain the same grammar as we would from the unreduced test case. However, because we did not attempt to order the translation units (we simply ordered them alphabetically), we cannot guarantee that we compute the minimal test case (set of translation units). Further, as noted by Hennessy and Power, given the essential set, choosing the minimal test case that covers the productions is equivalent to the minimum cardinality hitting set, which is an intractable problem [48].

Table IV lists percentage reductions achieved using the above technique. For each test case we list two numbers: the number of translation units in the essential set and the percentage reduction in the number of translation units. We also list the aggregate values for each test suite and for all test suites. The first number is located in the third column of the table and is the number of translation units in the essential set. For example, for the Gecko test case the value in the third column is 52; this means that all of the productions from

the recovered grammar for Gecko can be recovered from a particular set of 52 translation units. The second number is located in the fourth column of the table and is the percentage reduction in the number of translation units. Continuing the previous example, for the Gecko test case the value in the fourth column is 79.5; this means that by eliminating all but 52 of the 254 translation units for the Gecko test case, we have reduced the Gecko test case by 79.5%.

The results in Table IV show that we can achieve an 88.2% reduction in the number of C++ translation units in our test suites without sacrificing the recovery of any terminals, non-terminals, or productions. We did not leverage this reduction in our studies or even measure the time savings that result. However, we note that the largest reduction, 94.3%, is for the **Libraries** test suite and that this test suite accounted for the vast majority of the running time consumed by our grammar recovery process [4]. Furthermore, this result has obvious utility to anyone who wishes to recover a grammar for a C++ dialect from test cases.

C. Grammar Refactoring Study

In this section we report results we obtained when applying our metrics-guided grammar refactoring approach to the GNU C++ grammar recovered in Section V-B. These results address research questions Q3 and Q4 from the beginning of this section. In addition, we report information gathered by inspection of the refactored grammar and from our comparison of the recovered grammar to the ISO C++98 grammar. This information addresses question Q5 from the beginning of this section.

As described in Section III, the three grammar size metrics PROD, AVS, and HAL can be computed at the nonterminal granularity. That is, we can compute the value of the metric for a nonterminal and its set of right-hand sides and we did so for each of the 138 nonterminals in the recovered GNU C++ grammar. When examining the results we found that one nonterminal had values for PROD and AVS that were one order of magnitude larger than the values for any other nonterminal. Further, we found that two nonterminals had values for HAL that were one order of magnitude larger than the values for any other nonterminal. More specifically, the values of PROD and AVS were particularly large for `postfix_expression` and the value of HAL was particularly large for `direct_declarator` and `direct_abstract_declarator`. Indeed, upon inspection of these three nonterminals, we found that each had expressions that were expressed using iteration. Thus we identified these three nonterminals as candidates for refactoring using the algorithm that we present in Section III-B.

Table V lists data about the recovered grammar before refactoring, the recovered grammar after refactoring by replacing iteration with left recursion, and (for comparison) the ISO C++98 grammar. In particular, we list the numbers of terminals, nonterminals, and productions, as well as the numbers of nonterminals that use left and right recursion. Only two values differ between the recovered and refactored grammars; specifically, the number of productions and the number of

| | | | |
|--|------|--|------|
| <i>postfix_expression</i> | | <i>postfix_expression</i> | |
| → <i>primary_expression</i> | (1) | → <i>primary_expression</i> | (1) |
| → <i>postfix_expression</i> [<i>expression</i>] | (2) | → <i>postfix_expression</i> [<i>expression</i>] | (2) |
| → <i>postfix_expression</i> (<i>expression_list_opt</i>) | (3) | → <i>postfix_expression</i> <i>expression_list_opt</i> | (3) |
| → <i>simple_type_specifier</i> (<i>expression_list_opt</i>) | (4) | → <i>simple_type_specifier</i> | (4) |
| → <i>typename</i> :: _{opt} <i>nested_name_specifier</i> <i>identifier</i> | (5) | → <i>typename</i> <i>nested_name_specifier</i> <i>identifier</i> | (5) |
| (<i>expression_list_opt</i>) | | → <i>typename</i> :: <i>nested_name_specifier</i> <i>template_id</i> | (6) |
| → <i>typename</i> :: _{opt} <i>nested_name_specifier</i> | | <i>template_id</i> <i>expression_list_opt</i> | (7) |
| <i>template_opt</i> <i>template_id</i> (<i>expression_list_opt</i>) | (6) | → <i>postfix_expression</i> . <i>id_expression</i> | (8) |
| → <i>postfix_expression</i> . <i>template_opt</i> <i>id_expression</i> | (7) | → <i>postfix_expression</i> . <i>template</i> <i>id_expression</i> | (9) |
| → <i>postfix_expression</i> -> <i>template_opt</i> <i>id_expression</i> | (8) | <i>expression_list_opt</i> | (9) |
| → <i>postfix_expression</i> . <i>pseudo_destructor_name</i> | (9) | → <i>postfix_expression</i> -> <i>template</i> <i>id_expression</i> | (10) |
| → <i>postfix_expression</i> -> <i>pseudo_destructor_name</i> | (10) | <i>expression_list_opt</i> | (10) |
| → <i>postfix_expression</i> ++ | (11) | → <i>postfix_expression</i> -> <i>id_expression</i> | (11) |
| → <i>dynamic_cast</i> < <i>type_id</i> > (<i>expression</i>) | (13) | → <i>postfix_expression</i> . <i>pseudo_destructor_name</i> | (12) |
| → <i>static_cast</i> < <i>type_id</i> > (<i>expression</i>) | (14) | <i>expression_list_opt</i> | (12) |
| → <i>reinterpret_cast</i> < <i>type_id</i> > (<i>expression</i>) | (15) | → <i>postfix_expression</i> -> <i>pseudo_destructor_name</i> | (13) |
| → <i>const_cast</i> < <i>type_id</i> > (<i>expression</i>) | (16) | <i>expression_list_opt</i> | (13) |
| → <i>typeid</i> (<i>expression</i>) | (17) | → <i>postfix_expression</i> ++ | (14) |
| → <i>typeid</i> (<i>type_id</i>) | (18) | → <i>postfix_expression</i> -- | (15) |
| | | → <i>dynamic_cast</i> < <i>type_id</i> > (<i>expression</i>) | (16) |
| | | → <i>static_cast</i> < <i>type_id</i> > (<i>expression</i>) | (17) |
| | | → <i>reinterpret_cast</i> < <i>type_id</i> > (<i>expression</i>) | (18) |
| | | → <i>const_cast</i> < <i>type_id</i> > (<i>expression</i>) | (19) |
| | | → <i>typeid</i> (<i>expression</i>) | (20) |
| | | → <i>typeid</i> (<i>type_id</i>) | (21) |
| | | → (<i>type_id</i>) { <i>initializer_list</i> } | (22) |

(a) ISO Version

(b) Recovered and Refactored Version

Fig. 8. Comparison of Nonterminals Expressed Using Left Recursion. *Two versions of postfix_expression, both expressed using left recursion.*

| Grammar | TERM | VAR | PROD | LR | RR |
|------------|------|-----|------|----|----|
| Recovered | 137 | 138 | 930 | 0 | 39 |
| Refactored | 137 | 138 | 601 | 3 | 39 |
| ISO | 117 | 184 | 479 | 27 | 6 |

TABLE V

GRAMMAR SIZE METRICS. *This table lists size metrics for the recovered grammar before and after refactoring, and the ISO C++98 grammar.*

nonterminals using left recursion. Note that the number of productions for our recovered grammar has decreased by 329; this can largely be attributed to *postfix_expression* having 337 productions before the refactoring. Also note that the recovered grammar now contains three nonterminals that use left recursion; in particular, the three nonterminals that we identified as candidates using size metrics.

The recovered grammars (before and after refactoring) contain 20 terminals not contained in the ISO grammar; these terminals are GNU extensions to C++, such as the keyword `__restrict__`. Additionally, the recovered grammars contain 46 less nonterminals than the ISO grammar; however, this is largely due to the 41 distinct productions that only implement optionality in the ISO grammar. The recovered grammars typically do not use distinct productions to implement optionality, but rather simply introduce a lambda production for the optional nonterminal. The two most striking differences between the recovered grammars and the ISO grammar are in the number of productions and the use of recursion. Before

refactoring, the recovered grammar contains almost twice as many productions as the ISO grammar; this is because iteration is used in the original recovered grammar. Also, while the recovered grammars actually use more recursion than the ISO grammar, because that recursion originates from a recursive descent parser, the recovered grammars use right recursion rather left recursion.

Figure 8b illustrates the 22 productions for *postfix_expression* that remain in the recovered grammar after refactoring. A comparison of these productions to those in Figure 8a yields several items of interest. Firstly, the sets of productions are very similar and, in fact, significant subsets of the productions are identical. Secondly, we note that production 22 in Figure 8b is a GNU extension. Thirdly, some optionality in the ISO grammar is expressed as multiple productions in the recovered grammar. Finally, note that in the recovered grammar, parentheses are included in the productions for the nonterminal *expression_list_opt*; that is, *expression_list_opt* in the recovered grammar is equivalent to (*expression_list_opt*) in the ISO grammar.

VI. THREATS TO VALIDITY

Our study has limitations that impact the validity and generalizability of our findings. In this section, we identify some of these limitations and discuss their impact on our study.

A central objective of our work is to recover a grammar for GNU C++ from parse trees. To obtain these parse trees, we manually instrumented the GNU C++ parser, which is located

in `cp/parser.c` and consists of nearly 17,000 lines of C source code. Manual instrumentation of the parser was tedious and difficult due to its scale and complexity, and use of the instrumented parser in our study is threat to internal validity. To address this threat, we regenerated source code from the parse trees and compared this code the original source code. In addition, we compiled both versions of the source code and compared the generated outputs. These comparisons provide some evidence of the correctness of our instrumentation.

Another threat to internal validity arises from our use of test cases to generate the parse trees from which we recover the GNU C++ grammar. In particular, the completeness of our recovered grammar depends on how many terminals, nonterminals, and productions are exercised by the test cases in the four test suites that we used in the study. That is, if our test cases fail to exercise a given terminal, nonterminal, or production, then that terminal, non-terminal or production will not appear in the recovered grammar. To address this threat, we gathered four disparate test suites, marked by their variety of application and domain. Moreover, we have compared the recovered grammar to the ISO C++98 grammar and shown that our grammar has more terminals, non-terminals and productions than the ISO C++98 grammar. This provides some evidence that we are converging on complete recovery of the GNU C++ grammar. Nevertheless, we have no certainty that we have recovered all of the terminals, nonterminals or productions in the GNU C++ grammar.

Because our technique for grammar recovery is based on parse trees, which are frequently utilized for testing and debugging of language-dependent software, we believe that the technique has the potential to be applied widely. However, we recover only one grammar in our study, which is a threat to external validity. Further, we have only applied our metrics-based approach to grammar refactoring to this single grammar. It is possible that this approach succeeded due to specific characteristics of the grammar and that the approach will not succeed when applied to other grammars; this is a another threat to external validity.

VII. CONCLUDING REMARKS

We have described a methodology for grammar recovery from a hard-coded parser. The methodology comprises manual instrumentation of the GNU C++ parser, a technique for automatic grammar recovery from parse trees, a semi-automatic metrics-guided approach to refactoring an iterative grammar to obtain a recursive grammar. We presented algorithms for recovering a grammar from a parse tree and for rewriting nonterminals expressed using iteration with left recursion.

We have shown the feasibility and utility of our methodology by performing a case study in which we recovered and refactored a grammar for GNU C++ and answered important research questions related to the methodology. We used four test suites consisting of 4,600 C++ translation units and 1.2 million lines of non-commented code to conduct this study, but showed that 88.2% of those translation units could be eliminated without sacrificing the recovery of any terminals, nonterminals or productions. We also investigated the use of

grammar size metrics for identifying candidate nonterminals for refactoring and found that by refactoring the identified nonterminals we could reduce the size of our recovered grammar by 329 productions while making it more legible.

We believe that our contribution in this paper fill a void in the grammar recovery research delineated in reference [7]. In particular, no previously published research describes a methodology for grammar recovery from a hard-coded parser. Moreover, there is no published research that describes an approach for refactoring an iterative grammar to obtain a recursive grammar. Finally, our methodology is capable of handling a large, industrial strength language, namely GNU C++.

REFERENCES

- [1] P. Klint, R. Lämmel, and C. Verhoef, "Toward an engineering discipline for grammarware," *ACM TOSEM*, vol. 14, no. 3, pp. 331–380, 2005.
- [2] M. Črepinšek, M. Mernik, and V. Zumer, "Extracting grammar from programs: Brute force approach," *ACM SIGPLAN Notices*, vol. 40, no. 4, pp. 29–38, Apr. 2005.
- [3] P. T. Devanbu, "GENOA — a customizable, front-end-retargetable source code analysis framework," *ACM TOSEM*, vol. 8, no. 2, pp. 177–212, 1999.
- [4] E. B. Duffy and B. A. Malloy, "An automated approach to grammar recovery for a dialect of the c++ language," in *WCRE*, 2007, pp. 11–20.
- [5] R. Lämmel and C. Verhoef, "Semi-automatic grammar recovery," *SPE*, vol. 31, no. 15, pp. 1395–1438, Oct. 2001.
- [6] A. Sellink and C. Verhoef, "Generation of software renovation factories from compilers," in *ICSM*, 1999, pp. 245–255.
- [7] R. Lämmel and C. Verhoef, "Cracking the 500-language problem," *IEEE Software*, vol. 18, no. 6, pp. 78–88, Nov./Dec. 2001.
- [8] A. Sellink, H. Sneed, and C. Verhoef, "Restructuring of cobol/cics legacy systems," *SCP*, vol. 45, no. 2–3, pp. 193–243, 2002.
- [9] M. van den Brand, M. Sellink, and C. Verhoef, "Obtaining a cobol grammar from legacy code for reengineering purposes," 1997.
- [10] R. Lämmel, "Grammar testing," in *FASE*, 2001, pp. 201–216.
- [11] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.
- [12] F. Javed, B. R. Bryant, M. Črepinšek, M. Mernik, and A. Sprague, "Context-free grammar induction using genetic programming," in *ACMSE*. New York, NY, USA: ACM, 2004, pp. 404–405.
- [13] W. Lohmann, G. Riedewald, and M. Stoy, "Semantics-preserving migration of semantic rules after left recursion removal in attribute grammars," in *LDTA*, 2004.
- [14] E. Bouwers, M. Bravenboer, and E. Visser, "Grammar engineering support for precedence rule recovery and compatibility," in *LDTA*, Mar. 2007, pp. 82–96.
- [15] J. Harm and R. Lammel, "Two-dimensional approximation coverage," *Informatica (Slovenia)*, vol. 24, no. 3, 2000.
- [16] M. Hennessy and J. F. Power, "An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software," in *ASE*, Nov. 2005, pp. 104–113.
- [17] B. A. Malloy, T. H. Gibbs, and J. F. Power, "Decorating tokens to facilitate recognition of ambiguous language constructs," *SPE*, vol. 33, no. 1, pp. 19–39, 2003.
- [18] J. F. Power and B. A. Malloy, "A metrics suite for grammar-based software," *Journal of Software Maintenance*, vol. 16, no. 6, pp. 405–426, 2004.
- [19] C. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism SDF — reference manual," *SIGPLAN Notices*, vol. 24, no. 11, pp. 43–75, 1989.
- [20] N. A. Kraft, B. A. Malloy, and J. F. Power, "A tool chain for reverse engineering C++ applications," *SCP*, vol. 69, no. 1–3, pp. 3–13, December 2007.
- [21] B. A. Malloy, S. A. Linde, E. B. Duffy, and J. F. Power, "Testing C++ compilers for ISO language conformance," *DDJ*, pp. 71–80, Jun. 2002.
- [22] B. A. Malloy, T. H. Gibbs, and J. F. Power, "Progression toward conformance for C++ language compilers," *DDJ*, pp. 54–60, Nov. 2003.
- [23] American National Standards Institute, *International Standard: Programming Languages – C++*, ser. ASC X3. ISO/IEC JTC 1, Sep. 1998, no. 14882:1998(E).

- [24] A. van Deursen and P. Klint, "Domain-specific language design requires feature descriptions," 2001.
- [25] J. Paakki, "Attribute grammar paradigms - a high-level methodology in language implementation," *ACM Comput. Surv.*, vol. 27, no. 2, pp. 196–255, 1995.
- [26] A. Dubey, P. Jalote, and S. Aggarwal, "Learning context free grammar rules from a set of programs," *IET Software*, vol. 2, no. 3, pp. 223–240, 2008.
- [27] Y. Sakakibara, "Grammatical inference in bioinformatics," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 7, pp. 1051–1062, 2005.
- [28] N. A. Kraft, B. A. Malloy, and J. F. Power, "An infrastructure to support interoperability in reverse engineering," *IST*, vol. 49, no. 3, pp. 292–307, March 2007.
- [29] ISO/IEC, *International Standard: Information Technology – Syntactic Metalanguage – Extended BNF*. ISO/IEC JTC 1, 1996, no. 14977:1996(E).
- [30] J. Kort, R. Lämmel, and C. Verhoef, "The grammar deployment kit," in *ENTCS*, M. van den Brand and R. Lämmel, Eds., vol. 65, no. 3. Elsevier Science Publishers, 2002.
- [31] T. Alves and J. Visser, "Metrification of SDF grammars," Departamento de Informática, Universidade do Minho, Tech. Rep., May 2005.
- [32] "SCLC," <http://www.cmcrossroads.com/bradapp/clearperl/sclc.html>.
- [33] "GCC, the GNU Compiler Collection," <http://www.gnu.org/software/gcc/>.
- [34] "Blitz++," <http://www.oonumerics.org/blitz/>.
- [35] "Boost," <http://www.boost.org/>.
- [36] "FreePOOMA," <http://www.nongnu.org/freepooma/>.
- [37] "GCC Testing Efforts," <http://www.gnu.org/software/gcc/testing/>.
- [38] D. van Heesch, "Doxygen," <http://stack.nl/dimitri/doxygen/>.
- [39] C. L. Cox, E. B. Duffy, and J. von Oehsen, "FiSim: An integrated model for simulation of industrial fibre and film processes," *Plastics, Rubber and Composites*, vol. 33, no. 9-10, pp. 426–437, Nov. 2004.
- [40] "Fluxbox," <http://www.fluxbox.org/>.
- [41] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
- [42] "The Mozilla Project," <http://www.mozilla.org/>.
- [43] L. R. Group, "iPlasma: An integrated platform for quality assessment of object-oriented design," 2007, <http://loose.upt.ro/iplasma/>.
- [44] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2006.
- [45] T. Gschwind, M. Pinzger, and H. Gall, "Tuanalyzer – analyzing templates in C++ code," in *WCRE*, 2004, pp. 48–57.
- [46] A. Beszédés, R. Ferenc, and T. Gyimóthy, "Columbus: A reverse engineering approach," in *STEP 2005*, 2005, pp. 93–96.
- [47] "FTensor," <http://www.oonumerics.org/FTensor/>.
- [48] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.