Toward Effective Management of Large-Scale Software

Andrew J. McAllister, PhD
Faculty of Computer Science
University of New Brunswick
Fredericton, NB Canada E3B 5A3
andrewm@unb.ca

Steven A. O'Hara, PhD
Eagle Legacy Modernization, LLC
702 Southwick Avenue
Grovetown, GA 30813
steve@eaglegacy.com

ABSTRACT

This paper outlines challenges the authors have faced over decades of industrial experience with large-scale software analysis and maintenance projects (especially legacy modernization) for multiple organizations where millions of lines of source code are involved. Such projects require large teams cooperating on parsing, analyzing, and manipulating source code. In this context the use of traditional parsing techniques based on context-free grammars has proven problematic. We present the Programmar API, a recently developed parsing approach designed to overcome these problems. This paper describes the industrial experiences that led to our R&D activities. The Programmar approach is designed to enable large teams to effectively extract complete, accurate, up-to-date information from application source code, and to provide this information as the basis for a wide variety of software management tools and activities. We present a framework that relates various types of such activities, and describe a vision for how the Programmar approach can provide significant benefits for the software industry in the future via an open-source distribution approach. This paper is intended to serve as an example of how challenges faced by industry can stimulate research, and as a catalyst for discussion of industry needs and potential future research directions.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *corrections, documentation, enhancement, restructuring, reverse engineering, and reengineering.*

D.3.4 [**Programming Languages**]: Processors – *parsing.*

General Terms

Languages, Documentation.

Keywords

Post-production software management, source code parsing, legacy modernization, reverse engineering.

1. SOFTWARE MANAGEMENT

The passage of time is often an unkind influence when it comes to retaining the value of software. Business needs change.

SER&IP'16, May 17 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4170-7/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897022.2897028

Programming languages and development environments evolve and new ones are introduced. Organizations revise their standards for application architectures and programming styles. Long series of maintenance updates occur, often by a sequence of developers with differing approaches, inconsistent skill levels, and varying depths of understanding of the software architecture and functional requirements. Such updates increase overall application complexity and can introduce bugs, dependencies, and inconsistent programming styles (e.g. Chapter 14 in [1]). This can result in brittle code that is difficult to change without introducing new errors. Documentation created when the software was originally developed becomes out-of-date, less informative, and even worse, potentially misleading. Institutional knowledge about software degrades as personnel change jobs.

Another way of thinking of this is that (a) the quality and completeness of organizational knowledge about software can erode over time, while (b) the pressure to update various characteristics of the software tends to increase over time. Both observations (a) and (b) lead to the need to extract information from source code, improving the ability to answer questions about the code, while observation (b) leads to the need to perform various actions on the software.

In this light, we define Post-Production Software Management (PPSM) to mean: Extracting information from software that has been successfully compiled and executed, and using this information to guide actions that retain and enhance the value of the software to the organization.

We consider PPSM to be more effective when it can be achieved more quickly (for example, by increasing the ease with which tasks can be performed), with reduced cost (which in the software engineering world typically translates into reduced person-days of effort required), and/or with higher quality of results.

1.1 Motivating Example from Industry

As an example of a situation where we have seen PPSM to be immensely challenging, consider the experience of the authors working for several years for a legacy modernization service provider. One author was the Chief Technology Officer, the other a Vice President. Both authors had responsibilities for delivery of services to clients, as well as significant roles in leading the R&D activities of the company. In this context we worked directly with dozens of public- and private-sector organizations worldwide with multi-million-line software holdings, including in government, banking, military, educational, and other sectors.

In working with these clients, we undertook tasks such as the following:

- Document the end-to-end data flows for this (set of) application(s), including flows that cross the boundaries of various languages and technologies. (e.g. [2])

- Identify any dead source code. (e.g. [3, 4])
- Identify and remove all source code clones or near clones. [5]
- Update system and user documentation to be consistent with updated application functionality; [6]
- Transform an application from one set of programming languages to another, usually from legacy languages to a modern technology stack. [7] As part of this task:
  - Redesign the database to a more modern platform, and update all database access actions accordingly;
  - Re-architect the entire application to a modern n-tier architecture; (e.g. [8])
  - Retain all code documentation in place; and
  - Produce high quality source code that is maintainable, which is critical for retaining the future value of the application.

Our clients consistently reported significant difficulties when they attempted (or contemplated attempting) these types of activities on their own large-scale software holdings. It is a relatively simple matter if there is a need to re-document and/or re-write an application containing only a few thousand lines of code. Up the ante, however, to several million lines of code and PPSM becomes technically challenging, time-consuming, and expensive. Even maintenance efforts to enhance and repair application functionality can be made more difficult due to the sheer size and complexity of the code base. [9]

Our company utilized a partly automated, partly manual process for conducting these types of PPSM projects. We found that fully automated solutions tended to produce poor quality results, and the sheer volume of work to be accomplished tended to overwhelm strictly manual efforts. Significant automation was necessary if projects were to be completed within reasonable timeframes and in a cost-effective manner.

We also found that virtually every project for a new client required the development of new automated functionality. Even if a new project shared the same characteristics as a previous effort (for example, transforming COBOL to Java), each situation presented unique characteristics that required significant custom enhancements to code analysis and generation functionality. For instance, existing applications from different organizations present different coding styles, application architectures, as well as language / environment versions. Moreover, the target results invariably differed, for example by desiring specific user interface or Service-Oriented Architecture (SOA) characteristics.

A fundamental requirement for introducing automation into such efforts is the ability to parse source code. Parsing produces accurate and complete data describing the structure and contents of the code. We organized our project teams into multiple groups. One group of software engineers was in charge of parsing the source code, while other groups used the data produced from parsing to accomplish subsequent processing tasks.

We generated parsers based on Context-Free Grammars (CFGs), similar to the widely used Yet-Another-Compiler-Compiler (YACC) [10]. A CFG is a declarative description of the syntax of a specific programming language. This parsing process relies on a separate token pre-processor (typically LEX, the Lexical Analyzer [11]) and generates Abstract Syntax Trees (ASTs). The generated ASTs are then passed along to teams of developers responsible for creating analysis and transformation tools. Such tools traverse ASTs to search for patterns of sub-trees related to data flow, extract information from those sub-trees, and create a metadata repository with that information.

We were able to achieve success using this approach, however the process also presented significant challenges, as follows:

1.1.1 CFGs Are Dynamic: Our initial expectation related to parsing with CFGs was that the language definitions were static, so once a CFG was either located or developed, it would also tend to remain the same throughout the project. This turned out to be far from the truth. This is because a given language can be represented with many different CFG versions, which produce ASTs with different characteristics. The downstream teams of tool writers tended to run into issues with the ASTs, which might be outright errors, or simply suggestions for how AST structure should be improved (by altering the CFG) to make ASTs easier to work with. As a result CFGs tended to undergo a steady stream of revisions throughout the project.

1.1.2 CFGs Serve Different Purposes: CFGs developed for use with code analysis tools have a different job than those developed for other purposes such as compilers and interpreters. With a compiler, if a program's source code does not successfully parse, then the programmer must change the source code. With analysis and transformation tools the assumption is that all program source code has previously been compiled and executed. If a program's source code does not successfully parse, then the grammar must be revised, or the program is incomplete / in-development / etc. Also, the job of the grammar is only to correctly identify source code structure and contents; there is no need to reject errors. These differences may seem subtle, but they contribute significantly to the tendency for grammars to be updated on an ongoing basis during large-scale PPSM projects.

1.1.3 CFG Updates Were Difficult to Detect and Expensive: As our teams made a series of grammar updates and AST structures changed, it was necessary for the downstream tool writers to revise their tools so they continued to correctly interpret the ASTs and produce accurate results. Unfortunately the constant stream of CFG updates represented a significant communication challenge.

It turned out to be all too common for tool writers to be unaware of all AST changes, which meant their tools no longer worked properly. Sometimes this created execution errors, which was actually the best case scenario, since that forced tool developers to debug the issues and fix the problems. The less desirable situation was when tools continued to execute, with the developers unaware that results were no longer valid. This is one of the main reasons why we found that an analysis approach based on CFGs does not scale well for large-scale software. ***As tool developer team size scales up, undetected tool errors due to CFG updates tend to become more and more common.*** Considerable effort and time can be wasted as other developers unknowingly work with invalid intermediate results. The use of CFG-based parsing thus had a direct, negative impact on project costs and timeframes.

1.1.4 ASTs Required Considerable Searching: AST nodes contain names but no links for navigation to related nodes. For example, an AST node representing a call to a Java method includes the name of that method. That node's subtree would include information about the arguments to be passed to the method's parameters. Assume a team is creating a tool to trace the flow of information through method calls. When the tool encounters a node for a method call, it is necessary then to search somewhere else for information about the definition of that method. That 'somewhere else' might be elsewhere in the AST, or possibly in look-up tables that developers have created to keep track of commonly referenced information such as variable and method definitions. Developing this type of search functionality within source code analysis tools turned out to consume an unexpectedly high amount of effort, time, and budget.

1.1.5 ASTs Are Inconsistent Across Languages: Large-scale applications tend to involve multiple languages and notations, all of which need to be parsed. CFGs tend to be developed for individual programming notations, which means we frequently generated separate ASTs with entirely different structures and element names for different parts of a client's application. Downstream tool developers were forced to process multiple unconnected ASTs, and to make sense of how information in one type of AST was related to information in other types of ASTs. This consumed considerable time and effort. The ability of tool writers to connect references across different languages was partially dependent on the skill of CFG developers to make the interconnections obvious. Again, the decision to parse using CFGs resulted in hidden costs we had not anticipated.

1.1.6 Parsing Legacy Programming Languages: Context Free Grammars are insufficient to interpret the syntactic rules of some programming languages. This necessitated messy and expensive workarounds, such as source code pre- or post-processing. In many cases this was because the languages were created before current parsing and compiling techniques became widely used. Examples we encountered where CFGs were insufficient include:

- Column-sensitive aspects of COBOL and the Report Program Generator (RPG): Comments in COBOL are identified by having a character in column 7. The identity and meaning of many RPG elements are determined by the column in which they appear.
- Interpretation of COBOL level numbers: The relative value of two consecutive level numbers determines whether the second clause should appear in the AST either (a) at the same level as the preceding clause, (b) as a sub-tree of the preceding clause, or (c) at the same level as some earlier clause.
- Context-sensitive tokenization in Algol: The sequence A-B should be recognized as a subtraction between two variables in some sections of an Algol program, whereas in other parts of the same program this should be recognized as a single hyphenated identifier.
- Context-sensitive parsing in Unisys Mapper: An integer number following an identifier must be parsed differently depending on what information the identifier represents. The program element represented by the number cannot be identified by the syntax alone.

1.2 An Improved Approach

After the authors left the aforementioned legacy modernization firm, Eagle Legacy Modernization was formed with the goal of developing an improved alternative to CFGs for parsing within the context of PPSM projects. The result is the Programmar approach. An earlier paper provides a technical overview of how parsing with Programmars is accomplished [12]. The focus of this paper is on the value that Programmars can provide to the software industry.

The paper is organized as follows. Section 2 provides a brief overview of a new approach for parsing software called the Programmar technique, which is specifically designed to provide the basis for effective PPSM. Readers desiring more technical detail can find it in [12]. In Section 3 we describe a number of ways that using Programmars for large-scale PPSM overcomes the challenges described in Section 1.1. Section 4 presents a framework that describes the various types of activities organizations typically undertake to accomplish PPSM. We use this framework to show how the Programmar approach relates to the challenges faced by the software industry, and how our research results will be provided to industry members. The final section concludes the paper.

2. PARSING WITH PROGRAMMARS

The term 'Programmar' is derived by combining 'program' and 'grammar,' and refers to a new approach for parsing source code [12]. The Programmar approach is specifically designed to overcome the challenges described in Section 1.1.

A Programmar is a set of Java classes that contain all the rules and logic required to parse source code for one or more programming languages. All of the elements needed to describe the computer programming language(s) to be parsed are embedded in the Java classes as fields, inner classes, and methods. The Programmar API provides the parser functionality required to parse source code using Programmars.

Instead of ASTs, parsing creates instances of the classes defined in the Programmar. Collectively these instances form a Programmar Semantic Tree, or PST. The PSTs are currently persisted as XML files or as Java programs. PSTs are an extension of ASTs that include semantic information, such as cross references.

Programmars are similar in some respects to CFGs. This is intended to promote familiarity for those experienced in working with existing parsing approaches. With traditional approaches, program elements are separated into terminals and non-terminals [13]. The same is true with Programmars. Terminals represent textual elements encountered in the source code being parsed, such as string literals, comments, numbers, keywords, identifiers, and punctuation. Non-terminals are more complex program elements such as statements, statement blocks, methods, classes, or lists of other program elements. Non-terminals are formed by combining one or more other elements.

With traditional parsing approaches, the rules for terminals are often defined using regular expressions [11]. With Programmars, Java methods are used for parsing terminals. This has the advantages that:

- The full definition of the programming language to be parsed is included within the Programmar, including terminals and non-terminals. There is no need for a token pre-processing step or for separate specifications;
- The full expressiveness of a general purpose programming language (Java) can be used for defining terminals. This includes the possible use of regular expressions;
- The Programmar API pre-defines several of the most common types of literals. Examples include TerminalLiteralToken, TerminalNumberToken, and TerminalPunctuationToken. We have found this can save considerable time and effort, since specifying and debugging either regular expression or CFG-like rules for floating point numbers or string literals can be challenging and time-consuming; and
- The Programmar API also includes special pre-defined terminals to ease the task to creating custom definitions for special cases. For example, Python has very strict rules for indentation. The Programmar API defines a Start-of-line terminal type that can be used to handle the indentation logic correctly.

2.1 Sample Programmar

Non-terminals within Programmars are defined using one class for each non-terminal. Each of these classes must extend one of several abstract non-terminal classes built-in to the Programmar API. As an example, consider a traditional grammar for a simple language L as shown in Figure 1.

```
    <L> ::= <P>+;
    <P> ::= <X> [<Y>];
    <X> ::= 'a' | 'b';
    <Y> ::= 'c' | 'd' | 'e';
```

Figure 1: Traditional Grammar

Figure 1 describes a language made up of single-letter keywords delimited by white space (e.g. blanks). A source file consistent with Figure 1 consists of one or more instances of the non-terminal P, each of which is comprised of a non-terminal X, optionally followed immediately by a non-terminal Y. An equivalent Programmar class for P is presented in Figure 2.

```
public class P extends TokenSequence {
    public KeywordChoice x
            = new KeywordChoice("a", "b");
    public @OPT KeywordChoice y
            = new KeywordChoice("c", "d", "e");
}
```

Figure 2: Programmar Class Example

The abstract TokenSequence class is built-in to the Programmar API, and provides Programmar writers with a convenient way to specify sequences of elements. Similarly, the built-in KeywordChoice class handles the common need to specify a set of keywords that can appear as a given element in a language. The notation @OPT is used to specify optional elements.

Each Programmar requires a top-level class that extends Language, such as that shown in Figure 3. The TokenList<> class represents a list of one or more elements.

```
public class Letters_Program extends Language {
    public TokenList<P> pList;
}
```

Figure 3: Top-level Programmar Class Example

The Programmar parsing process creates instances of the types of classes shown in Figures 2 and 3. Collectively these instances form a PST, which can be stored as an XML file or as Java code that regenerates the PST.

As an example, assume the Programmar provided above is used to parse the following text: `a b c`

The Programmar parser generates the XML in Figure 4. PSTs in XML form can be used by tools that perform subsequent analysis and processing tasks. ("T" means Token, "TT" is Token Type, "N" is Name, and "V" is Value.)

The central idea behind parsing with Programmars is to use reflection [14] to fill in the PST with the results of the parsing process. Reflection is essentially the ability of the Programmar parser to examine Programmar source code and use this knowledge to make decisions about how the parse should proceed. The parser uses this strategy to infer grammar rules from the Programmar classes. This property of Eagle Legacy's proprietary parser is what enables Programmars to be written in a declarative (rather than procedural) form.

Instances are named in the Programmars, which results in named PST entries in Figure 4. This is unlike an AST where elements are

```
<Program Language="L" Tokens="7">
  <T TT="Letters_Program">
    <T N="pList" TT="List">
      <T TT="P">
        <T N="x" TT="KeywordChoice" V="a"/>
      </T>
      <T TT="P">
        <T N="x" TT="KeywordChoice" V="b"/>
        <T N="y" TT="KeywordChoice" V="c"/>
      </T>
    </T>
  </T>
</Program>
```

Figure 4: XML Representation of a PST

anonymous. This means specific instances can be referred to from elsewhere in the Programmar or in associated source code analysis programs. This turns out to be extremely valuable when analyzing source code.

The Programmar technique has been used to successfully parse millions of lines of code written in dozens of computer programming languages such as Assembler, Fortran, PL/I, RPG, Java, Visual Basic, Delphi, DOS, SQL, Python, C++, and many more. For example, Figure 5 shows a summary of nearly seven million lines of successfully parsed industrially-sourced source code across multiple projects. This parsing took place during the development of the Programmar API, while developing Programmars for several widely used languages. The unparsed C files are due to missing macro definitions, while the unparsed HTML & Javascript files are due to embedded Django.

2.2 Comparison to Other Tools

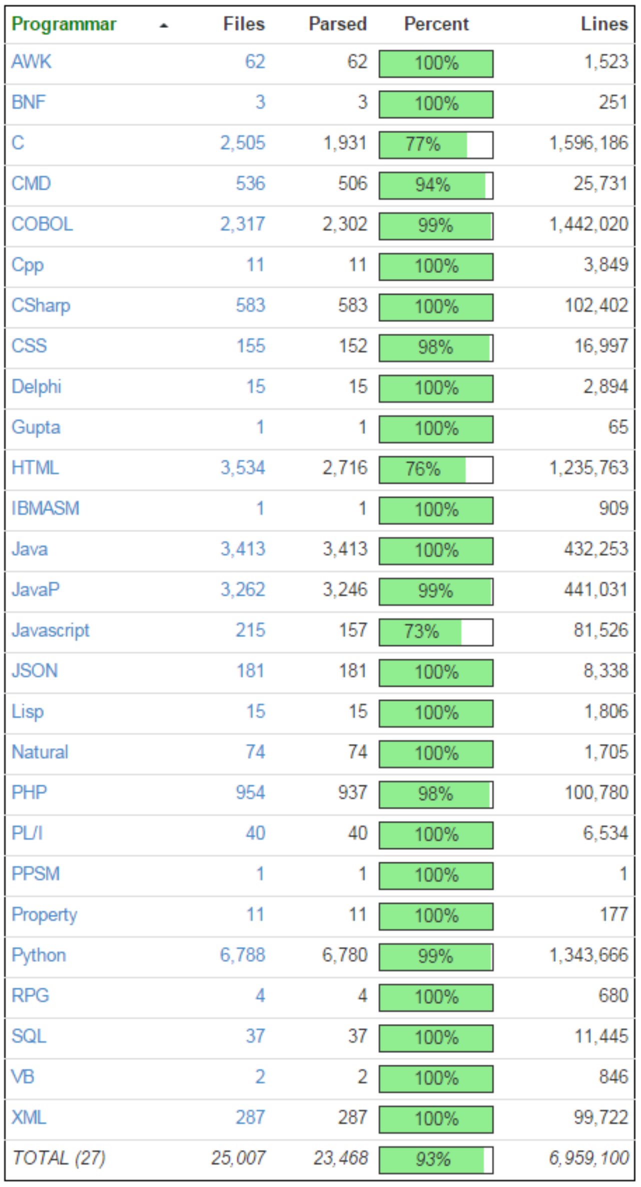
In terms of equivalence with other parsing approaches, the Programmar parser uses a top-down approach with no look-ahead (i.e., it is an LL(0) grammar [13]). No token pre-processor is required.

The use of a Programmar differs from a Recursive Descent Parser (RDP) [13] because Programmars use a declarative way of representing computer languages, as described above. Other than terminal nodes, there is no logic required in a Programmar. An RDP, in contrast, uses programming logic for matching each node in the grammar.

Tools such as ctags, opengrok, lxr, and doxygen (see for example [15]) are also used for source code analysis tasks, but serve a fundamentally different purpose as compared with a full parsing approach such as with Programmars. These tools rely on regular expressions to help locate various syntactic elements within source code files, which means their expressive power and ability to fully parse source code files is quite limited when compared with CFGs, let alone in comparison to context-sensitive Programmars. These tools cannot provide the complete parsing results required for large-scale PPSM.

3. ADDRESSING THE CHALLENGES

Although Programmars tend to be slightly more verbose than CFGs, the Programmar approach is designed to enable faster and more cost-effective development of PPSM functionality with a reduced likelihood of errors in the results. This is accomplished by addressing the challenges discussed in Section 1.1 as follows. A similar project approach as described above is assumed. One team of developers is responsible for Programmar development, which typically involves updates to the Programmars throughout a project. Other teams write tools to traverse PSTs and perform various analysis and processing tasks.



Programmar	Files	Parsed	Percent	Lines
AWK	62	62	100%	1,523
BNF	3	3	100%	251
C	2,505	1,931	77%	1,596,186
CMD	536	506	94%	25,731
COBOL	2,317	2,302	99%	1,442,020
Cpp	11	11	100%	3,849
CSharp	583	583	100%	102,402
CSS	155	152	98%	16,997
Delphi	15	15	100%	2,894
Gupta	1	1	100%	65
HTML	3,534	2,716	76%	1,235,763
IBMASM	1	1	100%	909
Java	3,413	3,413	100%	432,253
JavaP	3,262	3,246	99%	441,031
Javascript	215	157	73%	81,526
JSON	181	181	100%	8,338
Lisp	15	15	100%	1,806
Natural	74	74	100%	1,705
PHP	954	937	98%	100,780
PL/I	40	40	100%	6,534
PPSM	1	1	100%	1
Property	11	11	100%	177
Python	6,788	6,780	99%	1,343,666
RPG	4	4	100%	680
SQL	37	37	100%	11,445
VB	2	2	100%	846
XML	287	287	100%	99,722
TOTAL (27)	*25,007*	*23,468*	*93%*	*6,959,100*

Figure 5: Sample Parsing Summary Across Languages

3.1 Impact of Programmar Changes

In the Programmar approach, downstream impact of a change to any part a Programmar will be detected immediately, because the Java Programmar code will be an integral part of any Java tools written for downstream processing. The combined code will not successfully compile unless and until the tool code is fully consistent with the Programmar. If somebody were to change the name of an element in a Programmar, all references to that name within the tool code would become invalid until they were updated to be consistent with the changed element. Each PST has a version number embedded within it, so it is possible to detect out-of-date PST's. This allows projects to scale to much more significant levels. It is now possible to have dozens of developers working on the same project, processing many computer languages, with much reduced impact of effort and time delays associated with downstream detection of PST structure changes. This addresses the concerns described in Sections 1.1.1, 1.1.2, and 1.1.3.

3.2 Facilitating Cross References

As mentioned in Section 1.1.4, ASTs document references from one program element to another by naming the element. For example, an AST node for a procedure call merely names the procedure. There is no further information attached to it. If you write a tool to analyze or transform such procedure calls, you will have to search the rest of the AST to find out what is in that procedure. With the PST version, the procedure call instance contains within it a reference to the actual definition of that called procedure, including all of its parameters, return type, statements, etc. This greatly simplifies the task of writing analysis and conversion tools. Some of the work in connecting references to definitions is accomplished as part of the parsing process, which lessens the effort required to create tools for subsequent processing tasks. This addresses the concerns described in Section 1.1.4.

3.3 Multiple Languages Concurrently

Some computer languages, such as HTML for web pages, include other languages inside of them, such as Javascript, CSS or PHP in the case of HTML. Using CFGs, options include (a) attempting to separate and parse the languages separately (which is far from ideal), or (b) creating a complex, monolithic CFG covering all sub-languages. A Programmar can draw upon the expressive powers of Java. With Java encapsulation, the main Programmar (e.g. HTML) can simply reference other Programmars (e.g. Javascript). This addresses the concerns discussed in Section 1.1.5.

Additional advantages arise when considering multiple programming languages that include similarities, or those for which multiple language variants exist. A traditional CFG is typically built to describe just one programming language. With the Programmar approach, the components common to all variations of a particular programming language can be placed into an abstract Programmar class. For example, there are major variations of languages like RPG. A File specification has the same meaning across each variation, so an abstract RPGFile class can be used to define the common elements. The minor syntactic differences between RPG variations can then be represented by concrete classes that extend the abstract class.

Variations on a computer programming language can also be handled using inheritance. For example, there are both fixed width (80 column) and free-format COBOL programs. Their meanings are virtually identical, but the syntax is different. With a traditional grammar, the whole grammar might be copied and edited for each variation. With Programmar inheritance, only the local changes need to be considered and the rest can be inherited from the main Programmar.

3.4 Context-Sensitive Processing

Any Java method written to parse a particular type of terminal node has access to the current context in which an instance of that terminal is encountered within the source code being parsed. This Java method can examine any information relevant to understanding the correct meaning of that terminal. For example, a COBOL level number can be correctly interpreted by examining preceding level numbers in the partially completed PST. When parsing a Java program that involves method overriding or overloading, method calls can be resolved during parsing by examining and comparing arguments and parameters. This enables method call nodes within the resultant PST to include references to the appropriate method definitions. This addresses the concerns discussed in Section 1.1.6.

3.5 Enabling Shared Processing

Most terminal nodes are somewhat similar across programming languages. Generic processors for numbers, literal strings, punctuation, etc. are all made available by the Programmar API to use when writing Programmars. For example, parsing functionality for hexadecimal (hex, base 16) numbers can generally be implemented in just a few lines of Java by extending the generic hex number processor, and simply declaring their hex prefix or suffix. Comments, floating point numbers and string literals are simpler to implement in a Programmar than a traditional grammar, because they can utilize the built-in generic methods. An abstract Syntax class aids in managing the nuances of programming languages, such as case sensitivity and line continuations. An abstract Project class helps to decide which files to process, how to identify languages, how to override base Programmar classes, and other related functionality.

4. A FRAMEWORK FOR THE FUTURE

One goal of our research is to provide building blocks so organizations can perform effective large-scale PPSM. Figure 6 shows the hierarchy of functionality involved in achieving this goal.

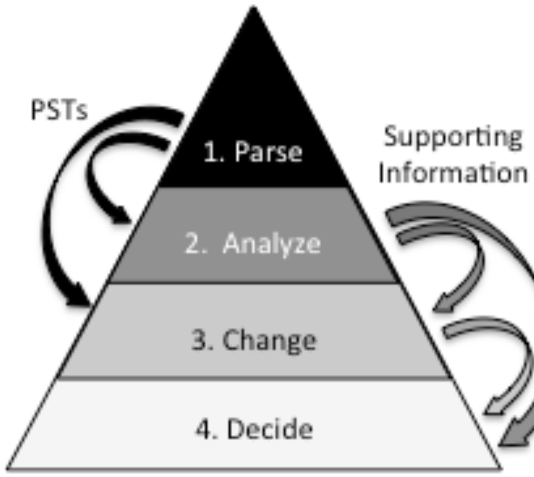


Figure 6: A PPSM Framework

4.1 Level 1: Parse

At the tip of the pyramid in level 1 is the ability to parse large-scale software repositories, thus creating an accurate, complete, detailed, up-to-date, and searchable understanding of an organization's source code. The Programmar approach provides this information in the form of PSTs.

We have developed a Programmar parser that produces PSTs as well as tracing output parsing process, both in plain text and html. This is available as an API for parsing programs over the web. Other functionality directly related to the parsing level in Figure 6 that has already been developed includes:

- Sample Programmars for a wide variety of modern and legacy programming languages, which have been used to successfully parse millions of lines of source code in those languages;
- A parsing progress monitor that shows what percentage of source code files were successfully parsed, details about any failed parsing attempts, as well as a variety of statistics for successfully parsed files such as number of lines, size of the resulting parse output, frequency counts showing how many instances of each Programmar rule were identified, etc.;
- Given a Programmar, a generator to automatically create an equivalent context free grammar. Note that full equivalence is not possible in cases where the Programmar performs context-sensitive processing. Also, terminal nodes defined in a Programmar are not converted by this tool;
- A pretty printer to regenerate a version of the original program from the PST, but in a canonical form. Indentation can be fixed, extra spaces eliminated, capitalization standardized, etc. Programmar element annotations are available to assist with the output formatting, such as @NEWLINE, @NOSPACE, @INDENT, and @OUTDENT;
- A program inspector that generates an html report for a parsed program, with color codes for all the different kinds of terminal nodes;
- A visual debugger for Programmars with commands like Step-in, Step-over, and Continue; and
- A macro processor for languages like C, PL/I and COBOL. Generally, not all macros need to be expanded, so controls are available to choose which macros to expand and which to leave intact.

4.2 Level 2: Analyze

Level 2 in Figure 6 includes a wide variety of functionality to extract various types of useful information from an organization's software holdings. Examples include but are certainly not limited to the following, all of which make direct use of PSTs:

- Fundamental to most analysis tools is a strongly connected network of references and definitions. For example, we have begun work on tools to connect variable references to their definitions. Such references can be within a single source code file, might span multiple such files, or can even be between multiple programming languages. Some languages include features like reflection and runtime compilation that complicate the process of identifying connections between program elements, so in some cases analysis tools must be capable of marking references as indeterminate;
- Using program-specific information to feed into various types of application documentation, such as user manuals and technical documentation;
- Extraction of various types of source code metrics, such as complexity or quality measurements; and
- Development of software search / query dashboards to enable IT workers to ask questions such as the following:
  - Where are the most exceptions being thrown?
  - Where is the most CPU time spent?
  - Which applications are never (or rarely) used?
  - Where do we have obsolete or deprecated code?

Beyond the parsing level, we recognize that much of the functionality in level 2 and onward in Figure 6 is likely to be organization-specific. Thus our strategy is to provide as much functionality as possible in an open source format to the community at large. This is already in place for sample Programmars at github.com/oharasteve/eagle. Eagle Legacy will provide an initial set of Programmars for commonly used languages and hope the community at large will create new Programmars, modify existing Programmars, and build associated PPSM tools as organization-specific projects proceed. It is anticipated that there will be language-specific champions in the open source community who will monitor and control Programmar changes for consistency and accuracy.

4.3 Level 3: Change

Level 3 represents tasks that make updates to application software, including:

- Software maintenance updates, whether for correcting discovered problems, adding desired functionality, adapting to different environments, improving software quality, or proactive changes to prevent potential future issues. Any of these forms of software maintenance can be enhanced by having a thorough and up-to-date understanding of the source code as provided by levels 1 and 2; and
- Transforming software to meet new needs, which might mean changing to a different set of programming languages and/or a new technology stack, or changes to achieve consistency with programming or architectural standards. Examples include migrating from COBOL to Java or from a siloed application to a service-oriented architecture. Our experience with Automation-Enabled Modernization (AEM) shows the value of information from levels 1 and 2 in performing such transformations [8].

4.4 Level 4: Decide

Level 4 represents decision making processes such as effort / cost estimation and cost / benefit analysis for software projects, project planning, proposal development, and choosing between options (e.g. how to integrate software holdings following a corporate acquisition). Such processes are highly complex and do not usually make direct use of parsing results. There is, however, often a residual impact. For example, project planning relies on effort estimates, which are typically more accurate when based on up-to-date software metrics and other information obtained via level 2.

It is our vision that efforts at all four levels across a variety of organizations will begin to show cumulative effects in the near to medium future. We anticipate widespread recognition that PPSM poses challenges different from those that organizations face when developing software to begin with. With that in mind, it makes sense that programming languages as well as their development and runtime environments could be designed with the effectiveness of PPSM in mind. Imagine new releases of Java or .NET with vendor-supplied Programmars provided as an integral part of the release.

5. CONCLUSION

This paper outlines how the Programmar approach addresses current challenges with post-production large-scale software management and paves the way for an industry-wide solution to evolve via the open source community. Our ultimate goal is to grow to support organizations with ultra-large-scale software holdings likely to be in the billion lines-of-code range, such as the US government, the US military, Apple, Google, Microsoft, Yahoo, and others. These organizations face particular challenges due to the size and complexity of their software. In such environments we have observed the following:

- The people who created the software to be managed are often no longer available. Large organizations tend to see significant staff mobility and turnover. This underscores the importance of effective software management capabilities; and
- Quite a variety of different operating systems, databases, programming languages, and other technology stack components tend to be in the mix. This requires large organizations to build bridges between applications created with different technologies, which means overall complexity increases at an incredible rate. The PPSM challenges are increased significantly.

These factors tie in directly to the challenges posed with CFG-based software processing as described in Section 1.1, as well as the advantages offered by the Programmar approach as described in Section 3. We are highly optimistic that the Programmar approach will enable organizations with millions and even billions of lines of source code to more effectively manage their software.

6. REFERENCES

[1] H. van Vliet, Software Engineering: Principles and Practice, 3rd Edition, Wiley, 2008.

[2] F. Li, A. Pop & A. Cohen, Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs, IEEE Micro, Vol. 32, No. 4, July 2012, 19-31.

[3] É. Payet & F. Spoto, Static analysis of Android programs, Information and Software Technology, Vol. 54, No. 11, Nov. 2012, 1192-1201.

[4] E. Arisholm, L.C. Briand & A. Føyen, Dynamic Coupling Measurement for Object-Oriented Software, IEEE Transactions on Software Engineering, Vol. 30, No. 8, Aug. 2004, 491-506.

[5] D. Rattan, R. Bhatia & M. Singh, Software clone detection: A systematic review, Information and Software Technology, Vol. 55, No. 7, July 2013, 1165-1199.

[6] G. Canfora, M. Di Penta & L. Cerulo, Achievements and Challenges in Software Reverse Engineering, Communications of the ACM, Vol. 54, No. 4, Apr. 2011, 142-151.

[7] A.J. McAllister, Automation-Enabled Code Conversion, Proceedings SERP '10: International Conference on Software Engineering Research and Practice, Las Vegas, NV, July 2010, 11-17.

[8] L. Shklar & R. Rosen, Web Application Architecture: Principles, Protocols and Practices, 2nd Edition, Wiley, 2009.

[9] G. Booch, The large-scale structure of software-intensive systems, Interface Focus, Vol. 2, No. 1, Feb. 2012, 91-100.

[10] S.C. Johnson, Yacc: Yet Another Compiler-Compiler, AT&T Bell Laboratories, 1975.

[11] J.R. Levine, T. Mason, & D. Brown Lex & Yacc, O'Reilly & Associates, 1992.

[12] S.A. O'Hara, Programmars: A Revolution in Computer Language Parsing, SERP '15: International Conference on Software Engineering Research and Practice, Las Vegas, NV, July 2015, 125-131.

[13] A.V. Aho, M.S. Lam, R. Sethi & J.D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, 2006.

[14] I.R. Forman & N. Forman, Java Reflection in Action, Manning Publications, 2004.

[15] ctags Linux man page: http://linux.die.net/man/1/ctags