Pattern Analysis of TXL Programs

Ashiqur Rahman
School of Computing
Queen's University, Kingston, Canada
E-mail: rahman@cs.queensu.ca

James R. Cordy
School of Computing
Queen's University, Kingston, Canada
E-mail: cordy@cs.queensu.ca

***Abstract*—Modern programming languages evolve, and need to be updated regularly by adding new features that fill new or unexpected programming needs. Existing approaches to determine new language features are completely manual and are based on language developers' experience, source code analysis, feature requests, on-line discussions and programmer interviews. Although these are acceptable practises, they are subjective, time-consuming, and don't always identify real needs. No research, to our knowledge, has attempted to make the task of language feature identification easier. In this paper, we propose a systematic approach for identifying the need for new language features with the help of pattern and clone detection tools that work on source code. Our approach semi-automates the task of language feature identification, works quickly, reduces the effort involved, and avoids subjectivity by supporting new feature proposals with evidence. We demonstrate our idea by analyzing a large set of projects written in the TXL language. After detecting and analyzing code patterns, we propose eleven new features that can help improve the TXL language to support ways that it is really used.**

I. Introduction

Programming languages can have many different kinds of features, such as programming constructs, concurrency support, support for bit-level operations, platform independence, and so on. In this paper, we use the term *feature* to mean the support a language provides to its users to perform various frequent programming operations using a built-in high level abstraction. For example, the *if-then-else* statement of the C language is a feature to support decision making. Features can be built directly into a language as part of its syntax and semantic specifications, or can be provided in the form of standard library support, for example the Java Class Library . As use of the language evolves, it is essential to maintain a programming language by adding new features [1].

Until now, feature recommendation for existing languages is usually done using the experience of the language developers, interviews with programmers, on-line reports and discussions, and manual analysis of source code. Programmers will sometimes also make direct requests for certain features to be added to a language. These methods have several problems associated with them, such as preparing interview questions, understanding ambiguous ideas, finding the common requirements, and so on, and often lead to subjective and controversial decisions. Source code querying tools, such as BBQ [2], Wiggle [3], and Jackpot [4] are available to analyze program source code, allowing users to write own queries to find target code patterns or idioms. Such tools are good for finding whether a particular pattern or idiom is used, and for making decisions regarding language evolution from the findings. But they do not help if users don't know what patterns to look for. Also, some of these do not scale to large programs or corpora, and may truncate information in the source code [3].

We propose instead using a systematic approach based on emergent pattern detection to identify and propose new language features based on a static analysis of existing programs. Using a variant of structured clone detection [5], we propose that patterns of co-incident clone classes in programs can be used to automatically identify candidates for new language features. In this paper we explore one such approach to find frequent patterns in TXL [6] source code by utilizing the NICAD [7] clone detector and our new NIPAT (Pattern Detection for Near-miss Intentional Clones) pattern detector to identify and recommend new features for the TXL language.

The remainder of this paper is organized as follows. Section II gives a brief overview of TXL and NICAD. Our approach and new pattern detection tool are described in Sections III and IV respectively. The results of analysing a large set of TXL projects are given in Section V. Section VI discusses related work and limitations of our research, and finally, Section VII concludes the paper with directions for future research.

II. Background

A. TXL

TXL [6] is a source to source transformation language designed specifically for working with programming language notations and features. Although TXL was originally developed in the 1980's for experimenting with programming languages [8], it has evolved to be used in a wide range of applications in static and dynamic analysis, database technology, web applications and other areas. Hundreds of industrial and academic projects have been implemented in TXL, and in particular the NICAD [9] and SIMONE [10] clone detectors.

Every TXL program consists of a grammatical structure specification and a set of structural transformation rules [11]. In order to perform a transformation, the TXL processor begins by generating a parse tree of the input using the grammar of the given language. It then performs transformation operations on the generated parse tree based on the transformation rules. Finally, it unparses the transformed tree and generates the output.

```
define statement_list
        [repeat statement_opt_semicolon]
end define

define statement_opt_semicolon
        [statement] [opt ';]
end define
```

Fig. 1. Sample Pattern

1) Grammar: TXL has its own notation to describe the nonterminals, terminals, keywords, and tokens of a grammar. It provides the *tokens* statement to specify terminal symbols, and the *define* and *redefine* statements to specify nonterminals. Several predefined tokens represent commonly used basic symbols, and EBNF-like meta-characters capture commonly used grammatical structures such as lists.

2) Transformation: TXL supports transformation functions and rules for defining the required transformation of the parsed input. While rules search their scope for pattern matches to transform, functions to not. Every TXL program begins with a main function or rule applied to the entire input, and other rules are invoked explicitly on parts of the input in functional programming style. In addition to patterns and replacements, TXL rules optionally use *construct*, *deconstruct* and *where* statements to manipulate and constrain patterns and replacements. The *include* statement provides the ability to combine TXL source files.

B. NICAD

NICAD (Accurate Detection of Near-miss Intentional Clones) [7] is a scalable and flexible code clone detection method for exact and near-miss clones distributed as a TXL-based hybrid clone detection tool [9]. NICAD uses three main stages: parsing, normalization, and text comparison [9]. After analyzing clone relationships through textual comparison, NICAD generates clone pair and clone class reports in both *XML* and *HTML* formats. NICAD detects Type I, II, and III clones with high precision and recall [12], and in a recent experiment was shown to be the most accurate detector of function clones in real systems [13].

III. Approach

When programmers repeat groups of related sections of code, either exactly or slightly modified, to achieve a specific purpose in several different files or projects, they certainly have some special need for them in general. We call such repeated groups of code a *pattern*, which can be used to identify new needs of programmers and hence to recommend new features for a language. Figure 1 shows an example of such a TXL grammar pattern. Clones of each of this pair of related defines are commonly found together in TXL programs.

A. Feature Recommendation Methodology

Our approach to pattern detection and feature recommendation is shown in Figure 2.

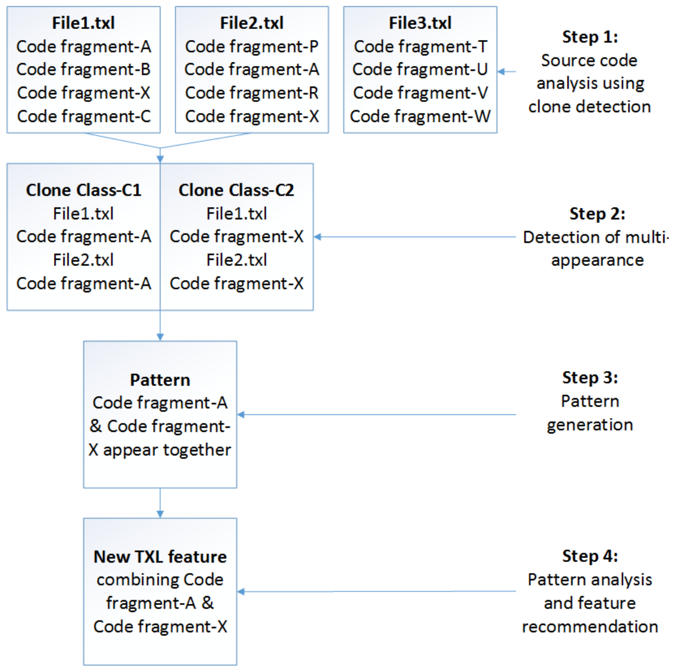


Fig. 2. Feature recommendation process

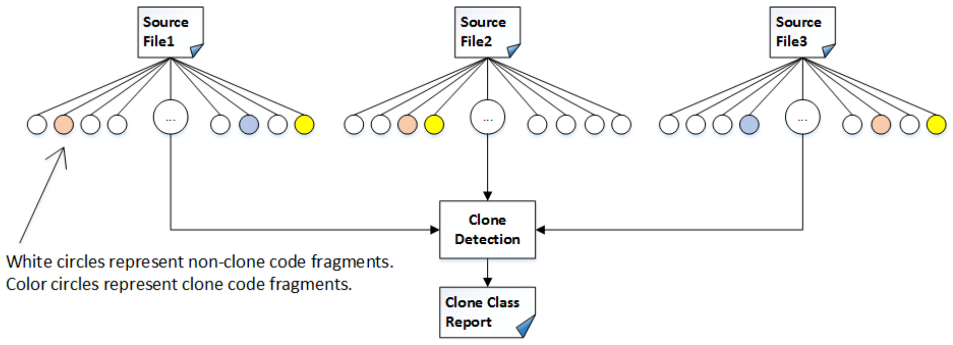
Source code analysis using clone detection: The first step to identify features is to begin with clone detection. We need to find code clones of the target granularity by analyzing program source code with the help of a clone detector. We used the clone detection tool NICAD [9] to find common code fragments in TXL source programs and used the generated clone class report.

For example, suppose we have three source files: *file1.txl*, *file2.txl*, and *file3.txl*, as shown in the first stage of Figure 2. We also assume that *file1.txl* contains instances of code fragments A, B, X, and C, *file2.txl* has fragments P, A, R, and X and *file3.txl* has instances of T, U, V, and W. Analysis of these three files, with a clone detector like NICAD, will tell us that code fragments A and X each have (possibly near-miss) clones that appear in both *file1.txl* and *file2.txl*.

Detection of multi-appearance: After detecting the code clone fragments, we must analyze the clone classes that hold groups of exact or similar code to determine which files appear together as a set in more than one clone class. The clone class report generated by NICAD reports groups of exact or similar code fragments arranged in classes with their original source file name. We then used our tool NIPAT to identify all files that appear together in clone classes.

As shown in Figure 2, code fragment A from *file1.txl* and *file2.txl* might go to clone class C1, and fragment X from each might be in clone class C2 in NICAD's clone class report. NIPAT then allows us to detect that *file1.txl* and *file2.txl* appear together in both clone classes C1 and C2, as shown in the second stage of Figure 2.

Pattern generation: Patterns can then be generated in the third stage by grouping all of the files that appear in two or more same clone classes, and gathering the code fragments from each clone class to form the set of code fragments for the pattern. In this way NIPAT identifies the set of all such co-incident clone class file sets, and then generates a pattern



Source File1
Source File2
Source File3
Clone Detection
Clone Class Report
White circles represent non-clone code fragments.
Color circles represent clone code fragments.

Fig. 3. Clone detection

for each which can be further analyzed by human experts.

For the given example, NIPAT will identify *file1.txl* and *file2.txl* as the file set containing both clone class C1 and C2, and will generate a pattern containing code fragments A and X from each file, as is shown in the third stage of Figure 2.

Pattern analysis and feature recommendation: In the last stage, we need the help of an expert in the target language to analyze the detected patterns and make feature proposals by understanding the purpose of each identified pattern. While some patterns may prove to be a promising addition to the feature set of the language, some may be redundant or unrealistic. In some cases an individual code fragment of a pattern may indicate the need for a new feature rather than the entire pattern, and sometimes a pattern may include unrelated code fragments that are coincident but irrelevant. For our example, if the analysis of the code fragments A and X clearly indicates the need for a new feature or library module, then a new feature can be proposed combining their purpose.

IV. NIPAT: Pattern Detection of Near-miss Intentional Clones

The idea behind our pattern detector is to analyze the clone detector's clone class report, where clones are grouped into clone classes. A clone class holds all clones of the same type from all the given source files. The goal of the pattern detector is to find the set of files of the source project(s) in the clone report that have a sets of clones in common, and to retrieve the respective clone code snippets to make a pattern. We have developed the pattern detector targeting the clone class report generated by NICAD.

NIPAT provides an interactive environment to perform all tasks, including clone detection, without leaving the tool. It prepares the files for detecting clones, invokes the command prompt to run NICAD, and then implements the pattern detection process on the detected clone classes. The pattern detector goes through four steps- Source file collection, Clone analysis, Clone report filtering, and Pattern detection.

Source file collection: Based on the user's selection NIPAT searches a specified source directory to gather all source files of a given file type into a single directory, and generates a report to allow users to trace back to the original source files. The source directory can contain any number of different projects' source trees.

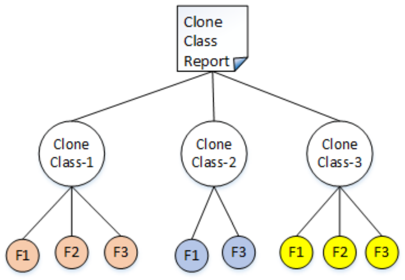
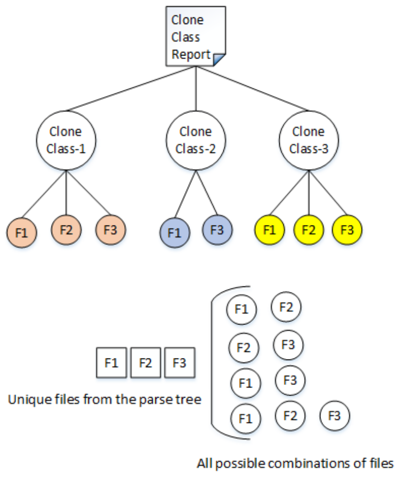


Fig. 4. Parse tree generation



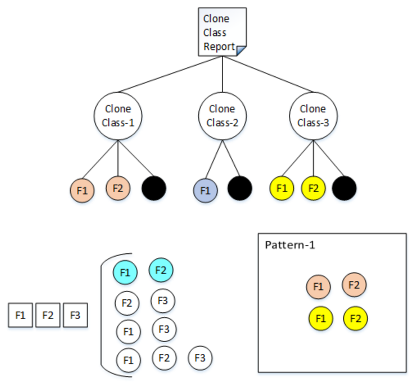
Clone Class Report
Clone Class-1
Clone Class-2
Clone Class-3
F1
F2
F3
Unique files from the parse tree
All possible combinations of files

Fig. 5. File set generation

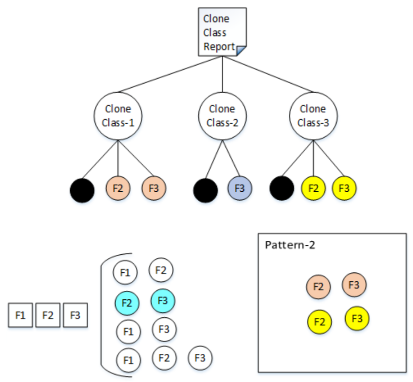
Clone analysis: After collecting all files of the required type, it invokes the command prompt to run NICAD. The user gives the NICAD command to run clone detection on the collected files of the target type. NICAD performs the clone analysis and generates a clone class report. Figure 3 illustrates the process to get the clone class report. When the user finishes running NICAD, control returns to the pattern detector.

Clone report filtering: The pattern detector then analyzes the XML version of the clone class report generated by NICAD. Because NICAD reports raw source fragments and does not generate pure XML, NIPAT scans the clone report and encodes any problems.

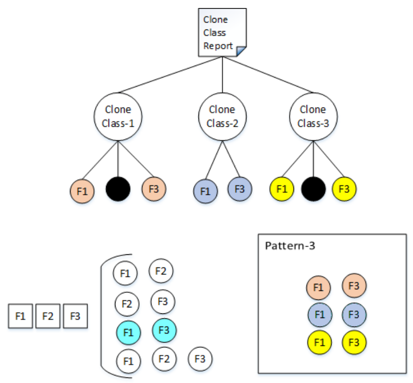
Pattern detection: In the last phase, NIPAT generates a parse tree for the filtered clone results from the clone class



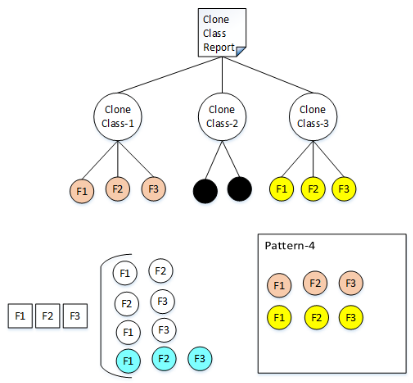
(a) Pattern generation for the file set (f1, f2)



(b) Pattern generation for the file set (f2, f3)



(c) Pattern generation for the file set (f3, f4)



(d) Pattern generation for file set (f1, f2, f3)

Fig. 6. Pattern generation by NIPAT

report (Figure 4). It finds all unique file names in the parse tree and generates all possible combinations of files as shown in Figure 5. Next, NIPAT selects each file set one by one and maps them onto the parse tree to find the set of common clone classes in which each particular file set appears. NIPAT retrieves the clone code fragments from the clone classes related to that file set and marks the retrieved set of code clones as a pattern (Figure 6). Figure 7 shows a sample final report from the pattern analysis.

V. TXL Project Analysis

As a first application of our method, we collected a total 83 TXL programming language projects to analyze using NIPAT. These projects are a mixture of standard examples and production user applications in TXL. To avoid unnecessary comparison and noise in the pattern results, we performed some manual preprocessing on the collected projects to remove trivial examples and redundant copies of projects source files. After preprocessing, we were left with a total 370 unique files containing TXL code. The first two columns of Table I show the number of TXL grammar and rule files in our project set.

Because TXL has two distinct sublanguages (grammars and transformation rules), we performed three different kinds of pattern analysis, first on each project individually, and then on the full set of projects as a collection. The following outlines the feature proposals deriving from our pattern analysis. Unfortunately there is no room in this short paper to include the patterns themselves, which can be quite large.

- **Grammar Analysis:** To retrieve grammar patterns from source files containing grammar definitions.
- **Rule Analysis:** To detect transformation rule patterns from files containing rules.
- **Complete TXL Program Analysis:** To identify patterns involving both grammar and transformation rules.

Table I summarizes the number of patterns detected of each kind, and the number of new language feature proposals deriving from expert examination of the patterns. As we can see, the

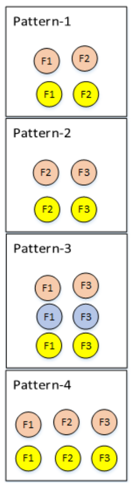


Fig. 7. Example NIPAT pattern report

TABLE I
SUMMARY OF ANALYSIS

	#Files	#Lines	#NICAD Clone Classes	#NiPAT Patterns	NiPAT Time(s)	#Proposals
Grammar	92	41,769	402	86	1.75	3
Rules	278	67,723	573	95	3.37	6
Grammar and Rules	210	113,331	1,798	113	23.04	2

```
define xml_source_coordinate
    '< [SPOFF] 'source [SP] 'file=[stringlit]
        [SP] 'startline=[stringlit]
        [SP] 'endline=[stringlit] '> [SPON][NL]
end define

define end_xml_source_coordinate
    '< [SPOFF] '/ 'source '> [SPON] [NL]
end define
```

Fig. 8. Frequently co-defined non-terminals

```
define else_where
        [where] [else_where_stmt]
    |   [self] [assignment_stmt] % Use of [self]
end define
```

Fig. 9. Code using *self*

majority of patterns were considered either uninteresting or not easy to abstract as new language features or libraries. However, the analysis generated eleven proposals for new features.

A. Grammar Proposals

Grammar Proposal 1: Predefined non-terminals to mark extracted patterns with XML source coordinates. Grammar pattern analysis reveals that programmers often define two common non-terminals, shown in Figure 8, to mark the extracted code patterns with XML tags. It is recommended to provide these non-terminals as part of a new XML library to avoid this repetition and standardize its use.

Grammar Proposal 2: A keyword to mention a non-terminal in its own definition. Several patterns indicated that TXL programmers often recursively refer a non-terminal in its own definition, introducing a potential renaming issue that has caused problems. Instead, TXL could provide a key non-terminal *[self]*, to explicitly refer to the containing definition without introducing a name dependency. Figure 9 shows a piece of code where the defining non-terminal is privileged with a special key non-terminal *[self]*.

Grammar Proposal 3: Selected non-terminals to repeat at the start or end of a sequence of alternative definitions. A frequently observed pattern involves a particular non-terminal repeated at the beginning or end of every alternative in the definition of a non-terminal. Instead of writing the same name repeatedly, or going through a second non-terminal, we could add a feature to allow the user to use two keywords *start* and *end* to achieve the same effect, as illustrated in Figure 10.

B. Rule Proposals

Rule Proposal 1: Uniqueify the elements of a TXL [repeat] sequence. A *repeat* type variable in TXL represents a sequence of elements of the same type. A very frequent rule pattern we uncovered was a seemingly standard set of rules to uniqueify the elements of such a sequence. Since the problem is so common, we propose a new TXL built-in rule [!!] to remove all identical elements from a *repeat* sequence (Figure 11).

```
define file_description_clause
        [TAB_25 start] [block_clause]
    |   [record_clause]
    |   [data_clause]
    |   [label_clause]
    |   [recording_clause]
    |   [report_clause]
    |   [TAB_25 end] [sort_option_clause]
end define
```

Fig. 10. Code using keywords *start* and *end*

```
rule main
    replace [repeat expression]
        Exprsn [repeat expression]
    construct UniqueExprsn [repeat expression]
        Exprsn [!!]  % [!!] finds unique elements
    by
        UniqueExprsn [evaluateExprsn]
end rule
```

Fig. 11. Finding the unique elements in a *repeat* sequence

Rule Proposal 2: Testing whether a repeat *sequence is a subsequence of another sequence of the same type.* Our analysis uncovered a relatively common pattern of rules used to match a subsequence in a TXL *repeat* sequence. A TXL feature to test whether one [repeat] sequence is a subsequence of another would remove the need for this pattern and open up opportunities for TXL to optimize the test. The code in Figure 12 shows an example that checks whether a the [repeat number] sequence *Num* is a subsequence of the [repeat number] sequence *NumMain* using a proposed new [@] built-in rule for this test.

```
function main
    replace [program]
        Num1 [repeat number] Str [stringlit]
        Num2 [repeat number]
    by
        Num1 Str
        Num2 [clearIfSubSequence Num1]
end function

function clearIfSubSequence NumMain [repeat number]
   replace [repeat number]
      Num [repeat number]
   where
      Num [@ NumMain]   % '@' testing subsequence
   by
      % Empty
end function
```

Fig. 12. Code using the proposed [@] subsequence feature

```
rule main
    replace [program]
        P [program]
    construct MyIf [repeat if_statement]
         _ [^^ P]  % skips embedded [if-statement]s
    by
        MyIf
end rule
```

Fig. 13. Proposed feature to skip the embedded patterns

```
rule main
    replace [number]
        Num [number]
    construct Num1, Num2 [number]  % constructs both
        5, 10
    by
        Num1 [+ Num2]
end rule
```

Fig. 14. Code to construct multiple variables of the same type

Rule Proposal 3: Skip embedded instances during extraction. We found that a frequently used pattern was designed to extract only the uppermost instances of a non-terminal from a scope, ignoring embedded instances of the same type. The pattern of rules to achieve this is awkward, inefficient and difficult to understand, since it first extracts all instances and then removes the embedded ones. We propose that a new variant of the TXL built-in extract rule [^] could be added to simply avoid the embedded patterns during extraction. For the moment we propose a new built-in rule [^^] with this semantics (Figure 13).

Rule Proposal 4: The construct statement can be modified to perform multiple constructs of the same type. The existing *construct* feature of TXL allows programmers to construct one variable at a time. But pattern analysis found that programmers often need to write sequences of multiple constructs of the same type one after another. We propose to extend the construct feature to support lists of constructs together, as shown in Figure 14.

Rule Proposal 5: Sequences of transformation rules applied together. We discovered a pattern that indicates that TXL programmers frequently apply a set of transformation rules as a group. We propose that a new feature could be provided to directly support abstraction of such groups as a *trasformation*, as shown in Figure 15.

Rule Proposal 6: Multiple deconstructs together. TXL *deconstruct* clauses test whether a TXL variable's bound parse tree matches a pattern. Pattern analysis found that a frequent pattern of use involved sequences of *deconstructs* of the same variable. We propose that the *deconstruct* feature could be modified to check the value of a variable against multiple patterns, using a new & syntax, as shown in Figure 16.

C. Grammar and Rule Proposals

Some of the patterns we discovered in TXL programs involved related sets of grammar definitions and rules that often appeared together to achieve a certain result. The most extensive such patterns occurred in the TXL source of the NICAD clone detector language plugins, each of which performs a similar set of transformations on a different source language.

```
transformation evaluateExpression
    [resolveAddition]
    [resolveSubtraction]
    [resolveMultiplication]
    [resolveDivision]
end transformation

rule main
    replace [expression]
        E [expression]
    construct NewE [expression]
        E [evaluateExpression] % Apply whole group
    where not
        NewE [= E]
    by
        NewE
end rule
```

Fig. 15. Proposed grouping of rules into a "transformation"

```
rule main
    replace [number]
        Num [number]
    deconstruct not Num
        0 & 100
    by
        Num [+ 1]
end rule
```

Fig. 16. Testing a variable against multiple patterns

The following two proposals would be more appropriately an application-specific library than a general TXL feature.

Grammar and Rule Proposal 1: Mark extracted elements with XML source tags. Because there are several NICAD language plugins in the TXL projects we analyzed, one of the largest patterns we found consisted of a number of grammar definitions and rules related to identifying. extracting, and marking NICAD potential clone fragments with their source coordinates in XML. This demonstrates another application of our technique: project-specific patterns. While we could propose that TXL provide a feature to directly mark extracted code fragments with with source coordinates as XML tags, a better proposal would be to make a new TXL library for general use. Figure 17 and Figure 18 show example code and expected output using our proposed library.

Grammar and Rule Proposal 2: Mark extracted elements with XML tags skipping embedded ones. An observed variant of the pattern of *Grammar and Rule Proposal 1* showed that sometimes programmers need to skip embedded patterns. Thus another library operation could be added mark the extracted patterns with XML tags skipping the embedded ones and

```
include "xmlsourcetag.txl"  % new library

rule main
    replace [program]
        P [program]
    construct MyIfs [repeat if_statement]
        _ [xml^ P]  % library operation to
                    % extract and mark with
                    % source coordinates
    by
        MyIfs
end rule
```

Fig. 17. Code using extract and mark with XML tags

```
Sample input:
  p=3
  x=p-2
  if x:
    y=4-x
    if y:
      x= x*3
  z=x/2

Sample output:
<source file="myfile" startlinenumber="3"
  endlinenumber="6">
  if x:
    y=4-x
  <source filename="myfile" startlinenumber="5"
    endlinenumber="6">
    if y:
      x=x*3
  </source>
</source>
<source filename="myfile" startlinenumber="5"
  endlinenumber="6">
    if y:
      x=x*3
</source>
```

Fig. 18. Extracted instances using extract and mark with XML

```
include "xmlsourcetag.txl"  % same library

rule main
    replace [program]
        P [program]
    construct MyIf [repeat if_statement]
        _ [xml^^ P] % library operation to
                    % extract and mark skipping
                    % embedded instances
    by
        MyIf
end rule
```

Fig. 19. Code using extract and mark skipping embedded instances

preventing them from being tagged. Figure 20 shows the result of running the code of Figure 19 on the same input.

```
Sample output:
<source file="myfile" startlinenumber="3"
  endlinenumber="6">
  if x:
    y=4-x
    if y:
      x=x*3
</source>
<source filename="myfile" startlinenumber="5"
  endlinenumber="6">
    if y:
      x=x*3
</source>
```

Fig. 20. Extracted instances using extract and mark skipping embedded

VI. Limitations and Related Work

A limitation of this research is the need for manual analysis of detected patterns. While NIPAT can detect clone patterns automatically, thus far we have no proposal to automate analysis and generation of feature suggestions. While any clone detector [14] could be used in our process, the effectiveness of our pattern detector directly depends on the precision of the identified clone classes. We began with NICAD for this reason [12], and at the moment NIPAT works only with NICAD.

While there are many other efforts in software pattern detection, the most clearly related work is that of Basit and Jarzabek on “structured clones” [5], which are closely related to our clone patterns. Our work differs in its use of NICAD rather than metrics to detect clone classes at higher precision, and in its concentration on patterns at the program level. Basit’s “internal” structured clones and those at the directory level and above seem to have no role in language feature identification.

VII. Conclusion and Future Work

The work presented in this paper describes a new process for semi-automatically identifying potential new language features for a programming language by analyzing the source code of applications written in the language. Our proposed method uses a pattern detector tool that utilizes clone class reports to find common patterns in existing source code. Language experts then analyze the patterns to understand and propose new features for the language.

Currently, we are exploring the potential of our approach to find patterns in Java programs. In future, language feature recommendation could be improved by automating pattern analysis to better filter detected patterns and help guide the language expert.

References

[1] R.-G. Urma, D. Orchard, and A. Mycroft, “Programming language evolution workshop report,” in *Proc. 1st Workshop on Programming Language Evolution*, 2014, pp. 1–3.

[2] Antlersoft, “Browse-by-query,” http://browsebyquery.sourceforge.net, accessed: 2015-11-18.

[3] R.-G. Urma and A. Mycroft, “Source-code queries with graph databases-with application to programming language usage and evolution,” *Sci. Comput. Progr.*, vol. 97, pp. 127–134, 2015.

[4] Netbeans, “Jackpot,” http://wiki.netbeans.org/Jackpot, accessed: 2015-11-18.

[5] H. A. Basit and S. Jarzabek, “A data mining approach for detecting higher-level clones in software,” *IEEE Trans. Software Eng.*, vol. 35, no. 4, pp. 497–514, 2009.

[6] J. R. Cordy, “The TXL source transformation language,” *Sci. Comput. Progr.*, vol. 61, no. 3, pp. 190–210, 2006.

[7] C. K. Roy and J. R. Cordy, “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in *ICPC 2008*, 2008, pp. 172–181.

[8] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow, “TXL: A rapid prototyping system for programming language dialects,” *Comput. Lang.*, vol. 16, no. 1, pp. 97–107, 1991.

[9] J. R. Cordy and C. K. Roy, “The NICAD clone detector,” in *ICPC 2011*, 2011, pp. 219–220.

[10] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, “Models are code too: Near-miss clone detection for simulink models,” in *ICSM 2012*, 2012, pp. 295–304.

[11] J. R. Cordy, “Excerpts from the TXL cookbook,” in *Generative and Transform. Techniques in Softw. Eng. III*. Springer, 2011, pp. 27–91.

[12] C. K. Roy and J. R. Cordy, “A mutation injection-based automatic framework for evaluating code clone detection tools,” in *Mutation’09*, 2009, pp. 157–166.

[13] J. Svajlenko and C. K. Roy, “Evaluating clone detection tools with BigCloneBench,” in *ICSME 2015*, 2015, pp. 131–140.

[14] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.