PATTERN ANALYSIS OF TXL PROGRAMS

by

ASHIQUR RAHMAN

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
November 2015

Copyright © Ashiqur Rahman, 2015

Abstract

All programming languages need to be updated regularly by adding new features that fill programming needs. Existing approaches to determine new language features are completely manual and are based on language developers' experience, source code analysis, feature requests, and programmer interviews. Although these are acceptable practises, they are time-consuming and require a lot of brainstorming tasks, such as preparing interview questions, understanding ambiguous ideas, finding the common requirements, etc. No research, to our knowledge, has attempted to make the task of language feature identification easier. Through our research, we propose a systematic approach for identifying language features with the help of pattern and clone detection tools that work on source code. It semi-automates the task of feature identification, works quickly, and reduces the effort involved in existing practises. We identify features for the TXL language by implementing our idea and enabling the NiCAD clone detector to perform clone analysis on TXL source code, and developing a pattern detector. After detecting code patterns with the pattern detector and analyzing them, we propose eleven new features that can help improve the feature set of TXL.

Acknowledgments

First of all, I give my praise to the Almighty Allah for his kindness and blessings in my life to reach to this position.

I express my sincerest gratitude to my supervisor Dr. James R. Cordy for all his support and patience during my study. I am grateful for the opportunity he has provided to make my academic career successful.

I would like to give thanks to Doug Martin, PhD candidate in my lab, for his worthy suggestions and proofreading of my thesis. I would also like to give thanks to Dr. Vadim Zaytsev from the University of Amsterdam for his valuable suggestions on my research work.

And finally, I would like to thank my parents and sister, who have always encouraged me during my study. I also thank my wife Tonni, who allowed me to take this long journey just after three months of our marriage.

Statement of Originality

I hereby certify that the research presented in this dissertation is my own, conducted under the supervision of Dr. James R. Cordy. Ideas and techniques that are not a product of my own work are cited, or, in cases where citations are not available, are presented using language that indicates they existed prior to this work.

Contents

Abstract	i
Acknowledgments	ii
Statement of Originality	iii
Contents	iv
List of Tables	vi
List of Figures	vii
Chapter 1: Introduction	1
1.1 Motivation	2
1.2 Contributions	3
1.3 Outline of the Thesis	4
Chapter 2: Background	5
2.1 TXL	5
2.1.1 What is TXL Best Suited For?	5
2.1.2 How Does TXL Work?	6
2.2 NiCAD: Accurate Detection of Near-miss Intentional Clones	12
2.2.1 Software Clones	14
2.2.2 How Does NiCAD Work?	17
2.2.3 How to Use NiCAD?	22
2.3 Summary	23
Chapter 3: Overview	25
3.1 Problem Description: Improving the TXL Feature Set	25
3.2 Feature Recommendation Methodology	27
3.3 Developed Plugins and Tools	30
3.3.1 Clone Detector for TXL	31

3.3.2	Pattern Analysis of TXL Clones	32
3.4	Type of Analysis and Result	32
3.5	Summary	33
Chapter 4:	Clone Detector for TXL	35
4.1	Extractor Program	35
4.2	Blind Renamer Program	37
4.3	Normalizer Program	38
4.4	Summary	46
Chapter 5:	Pattern Analysis of TXL Clones	48
5.1	Programming Language Used for the Pattern Detector	48
5.2	Properties of the Clone Report on which the Pattern Detector Works	49
5.3	How Does the Pattern Detector Work?	49
5.4	Summary	69
Chapter 6:	TXL Project Analysis	71
6.1	Project Description	71
6.1.1	Preprocessing	72
6.2	Grammar Analysis	72
6.2.1	Proposals	73
6.3	Rule Analysis	81
6.3.1	Proposals	82
6.4	Complete TXL Program Analysis	99
6.4.1	Proposals	100
6.5	TXL Project Analysis Summary and Proposal Classification	107
6.6	Summary	111
Chapter 7:	Conclusion and Future Work	112
7.1	Summary	112
7.2	Limitations	113
7.3	Future Work	113
Bibliography		115

List of Tables

6.1 Summary of Analysis 108

List of Figures

2.1	TXL Process (Adapted from TXL Cookbook [8], Figure 1)	7
2.2	General form of define and redefine	8
2.3	Grammar of a sample calculator program	10
2.4	General form of function and rule (Adapted from [9])	11
2.5	Transformation rules of a sample calculator program	14
2.6	Type I clone example	15
2.7	Type II clone example	15
2.8	Type III clone example	16
2.9	Type IV clone example	16
2.10	Sample source code for clone detection	19
2.11	Extracted code fragments by NiCAD from source code of Figure 2.10	20
2.12	Extracted code fragments of Figure 2.11 are normalized (blind re-named) by NiCAD	21
2.13	Clone class report of the source code of Figure 2.10 in XML format with source by NiCAD	22
3.1	Sample Pattern	27
3.2	Programming language feature recommendation process	28
4.1	Code before short form normalization	39

4.2	Code after short form normalization	39
4.3	Rule and function before keyword normalization	40
4.4	Rule and function after keyword normalization	40
4.5	Function before parameter normalization	42
4.6	Function after parameter normalization	42
4.7	Code before construct-deconstruct argument normalization	44
4.8	Code after construct-deconstruct argument normalization	44
4.9	Code before alternative truncation	46
4.10	Code after alternative truncation	46
5.1	A clone class report in HTML generated by NiCAD	54
5.2	A report after retrieving rule files by NiPAT	56
5.3	Clone detection	57
5.4	XML format of the clone class report of Figure 5.1	63
5.5	Parse tree generation	64
5.6	File set generation	65
5.7	Pattern generation by NiPAT	67
5.8	Pattern report by NiPAT	68
5.9	Pattern report of Figure 5.4 in HTML by NiPAT	70
6.1	A representative pattern defining two commonly used XML non-terminals	73
6.2	Code defining and using commonly used non-terminals <i>xml_source_coordinate</i> and <i>end_xml_source_coordinate</i>	74
6.3	Code using <i>xml_source_coordinate</i> and <i>end_xml_source_coordinate</i> de- fined in the library file	75

6.4	A representative pattern showing the use of a non-terminal in its own definition	76
6.5	Code before using <i>self</i>	77
6.6	Code after using <i>self</i>	77
6.7	A representative pattern using the same non-terminal at the beginning of a sequence of non-terminals	79
6.8	Code before using <i>start</i> and <i>end</i>	81
6.9	Code after using keywords <i>start</i> and <i>end</i>	81
6.10	A representative pattern removing identical elements from a <i>repeat</i> -type variable	83
6.11	Example code to find unique elements from a repeat type variable . .	84
6.12	Sample execution of the proposed feature to find unique elements from a variable with repeat type	84
6.13	A representative pattern searching for a subsequence	85
6.14	Code using the proposed subsequence feature	86
6.15	Sample execution of the proposed subsequence feature	86
6.16	A representative pattern skipping the embedded patterns	88
6.17	Example code using the proposed feature to skip the embedded patterns	89
6.18	Sample execution using the proposed extract feature to skip the embedded patterns	90
6.19	A representative pattern performing multiple constructs of the same type	92
6.20	Code to construct multiple variables of the same type using existing <i>construct</i> syntax	93

6.21	Code to construct multiple variables of the same type using the proposed <i>construct</i> syntax	93
6.22	A representative pattern applying a sequence of rules/functions . . .	95
6.23	Code without grouping of rules/functions	96
6.24	Code showing the proposed grouping of rules/functions	96
6.25	A representative pattern showing the use of <i>deconstruct not</i> on the same variable	98
6.26	Code to test a variable against multiple patterns using existing <i>deconstruct not</i> syntax	99
6.27	Code to test a variable against multiple patterns using the proposed <i>deconstruct not</i> syntax	99
6.28	A representative pattern marking extracted patterns with XML tags .	102
6.29	Code using <i>extract</i> and <i>mark</i> with XML tags proposal	103
6.30	Sample execution using <i>extract</i> and <i>mark</i> with XML tag	103
6.31	A representative pattern marking the extracted patterns with XML tags and skipping the embedded ones	106
6.32	<i>Extract</i> and <i>mark</i> patterns with XML tags avoiding the embedded patterns	107
6.33	Sample execution using <i>extract</i> and <i>mark</i> patterns with XML tags avoiding the embedded ones	107

Chapter 1

Introduction

Programming languages can have different types of features, such as multi-threading, support for bit-level operations, platform independence, and so on. In this thesis, we use the term *feature* to mean the support a language provides to its users to perform various programming operations through high level abstraction. For example, the *if-else* statement of the C language that handles program control flow is one of these features. Features can come directly with a language as part of its syntax/semantic specifications, or in the form of library support. While the *if-else* feature is directly integrated in the C language, string handling operations are provided as library features [5]. As use of the language evolves, it is essential to maintain a programming language by adding new features [17]. Unfortunately, there is no easy way to determine which features should be added to an existing language. However, a systematic approach with a pattern detector and clone detector can be followed to reduce the overhead associated with feature identification.

Through this thesis, we explore one such approach to find patterns from TXL [7] source code. The TXL language was developed about 30 years ago [10], and since then feature identification has been performed manually [15]. We find that the use

of some techniques and tools can reduce the efforts needed for finding these needed features. Here, we are concerned about both new additions to the language, and new libraries.

1.1 Motivation

Until now, feature recommendation for existing languages is usually done from the experience of the language developers, interviews with the programmers, and analysis of the developed source code to understand programmers' requirements. Programmers will sometimes also make direct requests for certain features to be added to a language. These methods have several problems associated with them. During interviews, programmers sometimes find it difficult to remember which features could make their task easier at the time of coding. In addition, programmers can express their ideas in ambiguous ways. Manual analysis of complete source code, on the other hand, is very time consuming since the number of lines in source files can be in the thousands, millions, or more. Moreover, analyzers don't actually know what they are looking for and where it is in the code. They can end up with nothing after analyzing a whole program with thousands of lines. Direct requests can tell what a programmer wants, but developers also need to make sure that the requested feature is required in general; otherwise, it may degrade the structure of a language with lots of features that have no use to others.

Source code querying tools, such as BBQ [1], Wiggle [16], Jackpot [2], etc., are available to analyze program source code. They allow users to write their own queries to find target code patterns or idioms. But to use source code querying tools, users need to know what they are looking for. Such tools are good for finding whether

a particular pattern or idioms is used, and for making decisions regarding language evolution from the findings. But they do not help if users don't know the patterns beforehand. Also, some of them do not scale to large programs, and if they do they truncate information in the source code [16].

In this thesis, we propose a systematic approach to identify code patterns from source code of an existing language (TXL) using a pattern and clone detector tool. Program code is the best source to discover programmers needs since it reflects their actual use of the language. The use of clone detection tools on source code tells us what code is repeated by programmers in general, and pattern detectors find the related code segments. It saves language developers effort by narrowing down the length of code to be checked for feature recommendation. It is also faster than finding code patterns by analyzing source by hand. Furthermore, the process can be repeated over several projects very conveniently to ensure that programmers use some pattern in general due to lack of a language feature. There is also no chance of adopting subjective or ambiguous ideas.

1.2 Contributions

Our research makes the following four contributions:

1. We propose a methodology for recommending new features for an existing language.
2. Plugins for the NiCAD [11] clone detector have been developed to perform clone analysis on TXL source files. These plugins enable NiCAD to detect clones from grammar files, rule files, and files containing both grammar and rules.

3. A pattern detector (NiPAT: Pattern Detection of Near-miss Intentional Clones) has been developed to recognize patterns from clone class reports. The pattern detector generates pattern reports in HTML and XML format for analysis by language experts.
4. We propose eleven new features for the TXL language by analyzing existing TXL source code. The analysis has been made implementing the proposed methodology and using the developed tools.

1.3 Outline of the Thesis

The rest of this report is organized into seven chapters. Chapter 2 describes the related background of TXL and NiCAD, which is needed to understand the work presented in this report. It is followed by a brief overview of the research, including our target, approach, tools, analysis, and result types in Chapter 3. We describe our developed programs in Chapter 4 and Chapter 5. Chapter 4 discusses, in detail, the plugin programs that allow us to detect clones from TXL programs, and Chapter 5 gives, in detail, a description of our pattern detector tool which was developed for the purpose of identifying patterns from the detected clones. The analysis we have performed to detect patterns from TXL projects, and the proposals for new features are presented in Chapter 6. Finally, Chapter 7 concludes the thesis with limitations and directions for future research.

Chapter 2

Background

This chapter discusses the related background needed to understand our work. Section 2.1 gives a brief overview of the TXL programming language and Section 2.2 describes the clone detection tool NiCAD with a brief introduction to clones.

2.1 TXL

TXL [7] is a source to source transformation language designed specially to work with programming language notations and features. It was originally developed to extend the capability of Turing programming language to accommodate new language features in the early 1980s [10]. Though TXL was developed mainly to work with programming languages, its enhanced capabilities have made it successful in many other areas, such as software engineering, database applications, web technology, etc. TXL is free to use but not open source.

2.1.1 What is TXL Best Suited For?

TXL is best suited to solve any kind of problem where we need to change the textual form of a given source code to another. It allows all kinds of editing of source code

without breaking the syntax of the language. For example, using TXL we can easily change the structure of a while loop of a C program to a do-while loop or a for loop, we can change the type of a variable to another or rearrange the instructions of a source file, and so on. It makes the task of updating old features to new features very convenient.

If we have the grammar of two different languages, then TXL can be used to change a program written in one language to another language very easily, which is helpful in maintaining legacy systems or changing the platform of a program. We can also perform tasks involving lightweight computations, like performing simple arithmetic operations, sorting, etc. However, unlike most of the other general purpose languages like C, Java, Python etc., it is not well suited (and sometimes incapable of) computing equations that require lots of calculations, or extensive use of logic or graphics, such as text editor or pay roll system.

2.1.2 How Does TXL Work?

Every TXL program consists of a grammatical structure specification and a structural transformation rules [8]. In order to perform a transformation, the TXL processor [4], which is responsible for implementing TXL programs, starts by generating a parse tree of the input (i.e program source code) using the grammar of the given language. TXL uses a recursive descent parser and repetition primitives for this purpose. It then performs transformation operations on the generated parse tree based on the transformation rules. Finally, TXL unparses the tree and generates the output, as shown in Figure 2.1 [8].

Figure 2.1: TXL Process (Adapted from TXL Cookbook [8], Figure 1)

Grammar

The grammar form supported by TXL is similar to Extended Backus-Naur Form (BNF). TXL has its own style and convention to describe nonterminals, terminals, keywords, tokens, etc. of a grammar. It provides several predefined nonterminals which represent the most needed grammar classes, meta characters to develop regular expressions, and other necessary features required to describe the grammar of a language.

The grammar of any TXL program must have the definition of a key non-terminal called *program*, which is the goal symbol. TXL provides two structures *define* and *redefine* to define the nonterminals of a grammar. Each define statement can define a new nonterminal consisting of a sequence of terminal symbols, nonterminal symbols, or a combination of both. It also allows us to give alternative definitions of the same nonterminal using the or bar (*|*). Using the *redefine* statement, we can extend or override the definition of an already defined non-terminal. Their general form as

defined by Cordy et al. [9] has been shown in Figure 2.2. Presence of an ellipsis(...) in a redefine statement indicates that the new alternative definitions of a nonterminal is to be merged with its previous definition; otherwise, redefine simply overrides the previous definition of the named nonterminal. To use the name of a nonterminal in its own definition or in some other definition we need to enclose the name of the nonterminal inside square brackets ([]).

```
define name
    alternativeA
    | alternativeB
    | alternativeC
    .
end define
```

(a) General form of the define statement

```
redefine name
    ...
    | alternativeP
    | alternativeQ
    .
end redefine
```

(b) General form of the redefine statement

Figure 2.2: General form of define and redefine

TXL provides certain predefined nonterminals which represent basic grammatical classes of the input items, such as [id], [number], [stringlit], etc. Some of them are used to characterize the given input like [upperid],[lowerid], [upperlowerid], etc. and some are special purpose, like [empty], [key], [srcfilename], etc. To describe certain characteristics of the nonterminals, both user-defined and predefined, TXL offers five nonterminal modifiers: *opt*, *repeat*, *list*, *attr*, *see*, *not*, *push* and *pop*. For example, a nonterminal preceded by the modifier *repeat* means that zero or more instances of that particular nonterminal may appear in the input. Shorthand notations of these nonterminal modifiers are also available for user convenience.

TXL permits the user to define their own compound character sequence or user defined tokens of the grammar using the special keywords *compounds* and *tokens*.

When defining compounds, the user can write a sequence of special (punctuation) characters, like `:=`, `==`, and so on inside the compounds section. To define program tokens the user can use TXL meta characters and operators in order to build regular expressions. TXL defines several meta characters, such as `\d` for digit, `\a` for alphabet, and so on. Meta characters, in combination with regular expression operators, such as `*`, `?`, `()`, etc., can be used to develop regular expressions of any desired pattern. Users can also define their own commenting style by using the keyword *comments*.

Figure 2.3 shows an example of a TXL grammar taken from the `Calculator.txtl` program. This program is available in the TXL Standard Examples Package as TXL resource in the TXL website [4]. Here, items enclosed in square brackets (`[ ]`) indicate nonterminals, symbols preceded by an apostrophe (`'`) are literals and percent (`%`) denotes comments.

```
% Calculator.Txl - simple numerical expression evaluator
% Part I. Syntax specification

define program
    [expression]
end define

define expression
    [term]
    | [expression] [addop] [term]
end define

define term
    [primary]
    | [term] [mulop] [primary]
end define

define primary
    [number]
    | ( [expression] )
end define

define addop
    '+'
    | '-'
end define

define mulop
    '*'
    | '/'
end define
```

Figure 2.3: Grammar of a sample calculator program

Transformation

Using the provided grammar, the TXL processor generates a parse tree of the given input, which is then converted to a new form following the user's instructions in the

function <i>name</i>	rule <i>name</i>
replace [<i>type</i>]	replace [<i>type</i>]
<i>pattern</i>	<i>pattern</i>
by	by
<i>replacement</i>	<i>replacement</i>
end function	end rule
(a) General form of the transformation function	(b) General form of the transformation rule

Figure 2.4: General form of function and rule (Adapted from [9])

transformation. TXL has two distinct features for defining the required transformation: transformation functions and transformation rules. While a rule searches the entire scope tree automatically for a pattern match after each replacement, function does not.

Every TXL program must have a function/rule named *main*, which is where the program starts. Figure 2.4 shows the general format of TXL function and rule. Both have the same format except the differences in keyword and replacement strategy. The minimum instructions that must be given in a function or rule block is the type of the parse tree to be replaced, the pattern to be matched, and the replacement to be done.

The pattern and replacement can be defined using a sequence of terminal symbols and/or variables bound to a subtree of particular types. If the given type of subtree does not hold the given pattern then a rule/function fails. A replacement can be empty if the target type allows.

TXL has three optional components to use in functions and rules. They are *deconstructors*, *constructors* and *conditions*. Deconstruct breaks a variable into pieces and matches it with a given pattern. If the match fails then the rule fails. The construct statement is used to construct variables of a particular type and they are

initialized with a replacement before using in the program. *Where*, *where not*, *where all*, *where not all* are the conditions supported by TXL syntax. They allow the user to apply a sequence of conditional rules on a variable and, based on the applied condition, generates either a success or failure. It is to be mentioned that conditional rules and functions, which are used with conditions, take slightly different form. They use the keyword *match* instead of *replace* and tests the pattern with the scope tree avoiding the *by replacement* portion. All functions and rules can be parameterized if needed.

TXL rules also support exporting and importing variables in the program using the keywords *export* and *import*. Several built-in arithmetic operations, text operations, list operations etc. are also available in TXL.

Figure 2.5 shows an example of TXL transformation rules, which, together with the grammar given in 2.3, makes a complete TXL program. The complete Calculator.txtl program is available in the TXL Standard Examples Package as a TXL resource in the TXL website [4].

It is the usual convention to keep the grammar and transformation rules in separate files. However, TXL is not rigid about that. TXL provides the *include* statement which allows to put the code of one TXL source file inside another. In this way, when creating TXL source files, we can combine multiple files that contain TXL instructions.

2.2 NiCAD: Accurate Detection of Near-miss Intentional Clones

NiCAD [12] is a scalable and flexible software (source code) clone detection method and tool that was developed by James R. Cordy and Chanchal K. Roy. It is distributed

```
% Calculator.Txt1 - simple numerical expression evaluator
% Part 2. Calculator.Txt1 - Transformation rules

rule main
  replace [expression]
    E [expression]
  construct NewE [expression]
    E [resolveAddition]
      [resolveSubtraction]
      [resolveMultiplication]
      [resolveDivision]
      [resolveParentheses]
  where not
    NewE [= E]
  by
    NewE
end rule

rule resolveAddition
  replace [expression]
    N1 [number] + N2 [number]
  by
    N1 [+ N2]
end rule

rule resolveSubtraction
  replace [expression]
    N1 [number] - N2 [number]
  by
    N1 [- N2]
end rule

rule resolveMultiplication
  replace [term]
    N1 [number] * N2 [number]
  by
    N1 [* N2]
end rule
```

(a) Transformation rules of a sample calculator program (*Part-1/2*)


```

rule resolveDivision
  replace [term]
    N1 [number] / N2 [number]
  by
    N1 [/ N2]
end rule

rule resolveParentheses
  replace [primary]
    ( N [number] )
  by
    N
end rule

```

(b) Transformation rules of a sample calculator program (*Part-2/2*)

Figure 2.5: Transformation rules of a sample calculator program

as a TXL-based hybrid clone detection system to detect both exact and near-miss clones [11].

2.2.1 Software Clones

A *clone* is the concept of similarity between two separate entities. For software systems it can be textual similarity or functional similarity. When the source code fragments of two or more software systems show some kind of significant match, then we say they are clones. Development strategy, maintenance benefits, tendency to overcome limitations, and accidental cloning are the main reasons for cloning [14]. It has both advantages and disadvantages. Advantages include quick development, need of fewer expert developers, reliability, and so on. On the other hand, disadvantages include error propagation from one system to another, improper functioning and infringement of copyrights. However, based on the kind of matches and their strength clones can be categorized into the following four types [14].

Type I: This is the strongest clone type where two different code fragments are textually identical except for variations in minor factors like white space, newlines or comments. Figure 2.6(a) and 2.6(b) shows an example of a Type I clone due to the changes in comments and newlines.

<pre>int foo(int i, float f) { int temp; temp=floor(f); i=i*temp; //use the floor value of f return i; // finally return integer value }</pre>	<pre>int foo(int i, float f) { int temp; temp=floor(f); i=i*temp; return i; }</pre>
(a) Original source code	(b) Type I (Modified source code)

Figure 2.6: Type I clone example

Type II: If the identifiers, literals, comments, etc. of the source code are modified keeping the original structure same, then it forms Type II clone. The source code of Figure 2.7(a) has been modified in Figure 2.7(b) by renaming identifiers and functions, and removing the comments. They are an example of a Type II clone.

<pre>int foo(int i, float f) { int temp; temp=floor(f); i=i*temp; //use the floor value of f return i; // finally return integer value }</pre>	<pre>int myfunc(int i, float num) { int temp; temp=ceil(num); i=i*temp; return i; }</pre>
(a) Original source code	(b) Type II (Modified source code)

Figure 2.7: Type II clone example

Type III: When the copied code fragments are modified so that the textual similarity with the original code almost disappears then they become Type III clones.

In addition to the changes allowed in Type I and II cases, they can change statements and add, delete, or rearrange several parts of the copied code. Figure 2.8(b) has fewer statements than Figure 2.8(a). Also, a variable has been renamed and comments have been removed. These differences make it a Type III clone.

<pre>int foo(int i, float f) { int temp; temp=floor(f); i=i*temp; //use the floor value of f return i; // finally return integer value }</pre>	<pre>float myfunc(int i, float num) { int res; res=i*num; return res; }</pre>
(a) Original source code	(b) Type III (Modified source code)

Figure 2.8: Type III clone example

Type IV: This type of clone is derived from the functional meaning of the code fragments. Though the code might not be textually similar, they can perform the same operations. This is the hardest type of clone to be detected. The code in Figure 2.9(a) and Figure 2.9(b) have a wide range of textual dissimilarity but perform the same function.

<pre>int foo(int i, float f) { int temp; temp=floor(f); i=i*temp; //use the floor value of f return i; // finally return integer value }</pre>	<pre>int myfunc(int x, float num) { return x*floor(num); }</pre>
(a) Original source code	(b) Type IV (Modified source code)

Figure 2.9: Type IV clone example

2.2.2 How Does NiCAD Work?

Among the four types of clones described in section 2.2.1, NiCAD detects Type I, II and III clones with high precision and recall [13]. To detect clones, NiCAD goes through three main stages: parsing, normalization, and comparison [11].

During the parsing phase, NiCAD utilizes TXL programs to analyze the input program using the TXL grammar of the input language. TXL supports agile parsing, which allows us to specify different interpretations of the same syntax through grammar overrides and multiple definitions of the same non-terminal. It starts by generating a parse tree of the input program and finding out all code fragments of the target granularity. It then stores them in an XML file, annotating them with the original file name, as well as the beginning and ending line numbers of the extracted portion taken from the original source file. In addition, it supports different pretty printing for the same grammar. If users are not interested in the comments or formatting cues of the input program then they can use TXL's pretty printing feature to strip out those from the extracted code.

This phase has made NiCAD language sensitive since we must provide the island grammar, the mechanism to identify the interesting parts of the input program, for each individual language in which we are interested [12]. We also need to provide a separate extractor for each individual language and granularity, which NiCAD uses to retrieve the target code fragments.

The next phase, normalization, makes NiCAD very flexible. In this stage, users can modify the extracted code fragments to increase the possibility of detecting clones through renaming, removing, adding, changing, or rearranging the text of the extracted code fragments. For instance, NiCAD can rename all the function names of a

C program consistently if user intends to perform a match ignoring such a difference. This phase can be ignored if such transformations are not necessary.

The comparison phase is the last phase that is language insensitive. NiCAD performs a textual comparison using an optimized longest common subsequence (LCS) algorithm to determine the similarity among the extracted code fragments.

Despite normalization in the previous phase, clone detection might fail to detect a copied code fragment as a clone using LCS, since code changes can take place at any arbitrary point and in many different forms, all of which might not be possible to normalize. To avoid such situations, NiCAD allows the user to define a difference threshold value to be used to detect near-miss clones. A difference threshold value, in addition to the normalizer, increases the likelihood of detecting near-miss clones. After deciding the clone relationship through textual comparison, NiCAD generates clone pair and clone class reports in both xml and html format. Clone reports are presented with the original source code and coordinates instead of the pretty printed code fragments. Language sensitiveness in the parsing phase and language insensitiveness in the comparison phase has made NiCAD a hybrid clone detector.

Figure 2.10 to Figure 2.13 presents a scenario of the clone detection process followed by NiCAD. Two different source files written using the C language have been given in Figure 2.10. Now, to detect clones at the function level, NiCAD parses the input source files and extracts the functions, as shown in Figure 2.11. During the extraction, NiCAD strips out the comment, and annotates each of the extracted parts with the original source properties.

<pre>int foo(int i, float f) { int temp; temp=floor(f); i=i*temp; return i; //integer result } float convert(int f) { float cel; cel=(f-32)/9.0; //convert to Celsius return cel; }</pre>	<pre>int myfunc(int x, float num) { int f; f=ceil(num); //nearest top x=x*f; return x; } void printch(int total, char ch) { for (i=0;i<total;i++) printf(ch); }</pre>
(a) Sample source file1	(b) Sample source file2

Figure 2.10: Sample source code for clone detection

```
<source file="file1" startline="1" endline="7">
int foo(int i, float f)
{
    int temp;
    temp=floor(f);
    i=i*temp;    return i;
}
</source>
<source file="file1" startline="9" endline="14">
float convert(int f)
{
    float cel;
    cel=(f-32)/9.0;
    return cel;
}
</source>
<source file="file2" startline="1" endline="7">
int myfunc(int i, float num)
{
    int f;
    f=ceil(num);
    i=i*f;
    return i;
}
</source>
<source file="file2" startline="10" endline="14">
void printch(int total, char ch)
{
    for (i=0;i<total;i++)
        printf(ch);
}
</source>
```

Figure 2.11: Extracted code fragments by NiCAD from source code of Figure 2.10

After collecting all the interesting code fragments in a single XML file, NiCAD will apply a normalizer if user wants to do so. For this scenario, we assume that we want to perform blind renaming on the function names and variable names. Provided the appropriate normalizer, NiCAD generates another XML file, like Figure 2.12 after normalization of the extracted code fragments.

```
<source file="file1" startline="1" endline="7">
int x(int x, float x)
{
    int x;
    x=floor(x);
    x=x*x;    return x;
}
</source>
<source file="file1" startline="9" endline="14">
float x(int x)
{
    float x;
    x=(x-32)/9.0;
    return x;
}
</source>
<source file="file2" startline="1" endline="7">
int x(int x, float x)
{
    int x;
    x=ceil(x);
    x=x*x;
    return x;
}
</source>
<source file="file2" startline="10" endline="14">
void x(int x, char x)
{
    for (x=0;x<x;x++)
        printf(x);
}
</source>
```

Figure 2.12: Extracted code fragments of Figure 2.11 are normalized (blind renamed) by NiCAD

After finishing the normalization phase, NiCAD starts comparing the code fragments and generates reports based on the detected clone relationship. Figure 2.13 shows the clone class report in an XML file. It is one of the four reporting formats that NiCAD generates. If we look closely, we find that in the final report NiCAD has

restored all the comments and formatting cues that were ignored during the intermediate stages. Since NiCAD detects near miss clones using a difference threshold, the clone result for the same source might vary depending on the given value.

```
<clones>
<class classid="1" nclones="2" nlines="7" similarity="96">
<source file="file1" startline="1" endline="7">
int foo(int i, float f)
{
    int temp;
    temp=floor(f);
    i=i*temp;    return i; //integer result
}
</source>
<source file="file2" startline="1" endline="7">
int myfunc(int i, float num)
{
    int f;
    f=ceil(num); //nearest top
    i=i*f;
    return i;
}
</source>
</class>
</clones>
```

Figure 2.13: Clone class report of the source code of Figure 2.10 in XML format with source by NiCAD

2.2.3 How to Use NiCAD?

NiCAD provides a command line interface that can be invoked from Linux, Solaris, Mac OS X or Cygwin command line [11]. When invoking NiCAD we need to provide the desired granularity, language, root directory, and configuration file name in the following command line format:

```
./nicad granularity language systems/systemdir [config]
```

Granularity refers to the code fragment that we want to extract, such as function,

block, rule, etc. To do so, we need to put the TXL extractor inside the **txl** directory of NiCAD. The plugin architecture of NiCAD allows us to extend its ability by adding new extractors and normalizers for new languages. To facilitate that, NiCAD follows a specific file naming convention. The naming format of a new extractor is: **language-extract-granularity**. Here **language** is the file extension name of a particular language without the period. Using the language and granularity, provided in the command line, NiCAD finds out the right extractor from its collection and executes it on all the files of the given system directory including its subdirectories.

The parameter **[config]** is the name of the configuration file. It is an optional parameter. If omitted, NiCAD uses the default configuration file. Configuration files are located inside the **config** directory of NiCAD using the *.cfg* extension. A configuration file allows the user to specify several settings, including the threshold value and normalization program name. NiCAD processes the extracted code fragments following the configuration and generates the output clone report using clone pair and clone class information in both xml and html format.

2.3 Summary

This chapter has introduced the source transformation language TXL, and the clone detection tool NiCAD which are required to understand the work presented in this paper. The discussion of TXL mainly focuses on those features of TXL that form the textual presentation of the written programs instead of technical details. We have given a short description of software clones, the process NiCAD follows to identify them, and the basic steps one needs to follow to use NiCAD. In the following chapter we present an overview of our research, and describe the developed tools and steps of

our analysis in detail in later chapters.

Chapter 3

Overview

This chapter gives an overview of the problem we intend to solve, our approach, developed tools, and the type of analysis we have performed. First, we describe the concept of improving the TXL feature set in Section 3.1, and then we introduce our approach in Section 3.2. It is followed by a brief overview of our developed programs in Section 3.3. Finally, Section 3.4 gives an idea about the type of analysis we have performed and the type of results we have obtained.

3.1 Problem Description: Improving the TXL Feature Set

All programming languages need to evolve continuously meeting the demand of programmers in order to exist. This is also true for the TXL language [7]. Over the years, many TXL programs have been developed and a closer look reveals that programmers sometimes repeat exact or similar pieces of code in their programs to achieve the same or closely related functionality that are not yet directly supported by any TXL feature. Identification of those required features in a convenient way can help the language developers. Until now, no work exists that automates the task of required

language feature identification for an existing language. Program source code, we believe, is the best practical source to know the needs of programmers, which can help make the evolving decisions of a language. When programmers repeat some exact or similar code to achieve a specific purpose over different projects, they certainly have some special need for them in general. We call such a group of code a *pattern*, which can be used to discover/identify the needs of programmers and to recommend new features for a language. To improve the feature set of the TXL programming language and to make it more user friendly, such patterns can help us recommend necessary syntax or semantic modifications to the language.

Figure 3.1 shows an example of a TXL grammar pattern that we are concerned about. When such a piece of code is repeated over multiple source files or projects, they have some special importance to the programmers. After analyzing their purpose and feasibility, we can consider including their functionality to the TXL feature set to get direct support during coding. It is clear from the code of Figure 3.1 that the programmer intended to implement the grammar of a semicolon separated list. TXL has support for a space separated *repeat* and a comma separated *list* feature, but it does not provide any feature to support semicolon separated list directly. Programmers would need to do that throughout their own code when needed. We can overcome this limitation of TXL by understanding programmers need of a semicolon separated list and adding that feature to TXL.

Any program written using TXL has two parts: a grammar and a transformation. They can exist independently in separate files or can be put together in the same file, but the convention is to put the grammar and transformation rules in separate source files. We can analyze those source files both separately and together to identify the

grammar and rule patterns to recommend new features for TXL in order to keep it evolving.

```
define statement_list
    [repeat statement_opt_semicolon]
end define

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

Figure 3.1: Sample Pattern

3.2 Feature Recommendation Methodology

To improve the feature set of TXL, we started with a large collection of TXL projects and, through the utilization of our developed pattern detector tool NiPAT and the NiCAD clone detector, we achieved our goal. NiCAD, with its existing plugins, is not capable of supporting the process of clone detection for TXL source code. In addition to our own tool NiPAT, we have developed the necessary plugins for NiCAD to enable it to work with TXL source code. Our approach to feature recommendation, shown in Figure 3.2, is described below.

Figure 3.2: Programming language feature recommendation process

Source code analysis using clone detection: The first step to identify features is to start with clone detection. We need to find code clone fragments of the target granularity by analyzing program source code with the help of a clone detector.

We used the clone detection tool NiCAD [11] to find the common code fragments from TXL source code. NiCAD generates two reports: one gives us the result in the form of clone classes; where all exact and near-miss code fragments of the same class are grouped together; and the other gives us the clone pair report showing which code resembles which. The first report, clone class report, is best suited to our task.

Suppose we have three source files: file1.txt, file2.txt, and file3.txt, as shown in the first stage of Figure 3.2. We also assume that file1.txt holds code fragments representing a group of instructions A, B, X, and C, file2.txt holds P, A, R, and X and file3.txt holds 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 have their clones that appear in file1.txt and file2.txt. The power of NiCAD to detect near-miss clone will give us the same result, in this case, fragment X of file2.txt is a textual variant of the fragment X of file1.txt. However, depending on the matching result, A and X might go into different (or the same) clone classes, as shown in the second stage.

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 gives us the groups of exact or similar code fragments arranged in classes with their original source file name. We then used our own tool NiPAT to identify all files that appear together in clone classes.

As can be seen from the second stage of Figure 3.2, code fragment A from file1 and file2 might go to clone class C1 and fragment X might go to clone class C2 similar to NiCAD's generated clone class report. NiPAT allows us to detect out that file1 and file2 appear together in both clone classes C1 and C2, as shown in the second stage of Figure 3.2.

Pattern generation: Patterns can be generated in the third stage by collecting the

code of each of the file sets, detected in the previous step, that appear in a unique list of clone classes. Code collected for such sets make a pattern.

With the help of NiPAT, we collected the code fragments of each of the file sets that appeared in a list of clone classes and made patterns for further analysis by human experts.

For the given example, after determining that file1.txt and file2.txt appear together in clone class C1 and C2, NiPAT will collect their code from clone class C1 and C2 and mark them as a pattern containing code fragments A and X as is shown in the third stage of Figure 3.2.

Pattern analysis and feature recommendation: In the last stage, we need the help of an expert of the target language to analyze the detected patterns and make feasible proposals by understanding the purpose of each identified pattern.

While some patterns may prove to be a promising addition to the feature set of a language, some may seem unrealistic. It should also be mentioned that occasionally the individual code fragment of a pattern might help to recommend a new feature that should not be ignored and sometimes a pattern might show unrelated code fragments that should be discarded. For our example, if the analysis of the code fragments A and X give some feasible idea, then a new feature can be proposed combining their purpose.

3.3 Developed Plugins and Tools

The whole process of making proposals to improve language features is semi-automatic. After identifying the patterns automatically, we need language experts in the last

stage to analyze the identified patterns and to understand what the programmers were doing to make feasible proposals. In the following subsections, we give an overview of our developed plugins and the tools we needed to automate the part of feature recommendation process.

3.3.1 Clone Detector for TXL

As we mentioned in section 2.2, NiCAD provides a plugin architecture [11] that allows us to extend its ability to work with any language. Even though NiCAD is based on TXL, to the best of our knowledge there is no plugin to detect clones in TXL programs.

Our plugin for NiCAD includes extractors and normalizers. Plugin programs developed for one language do not work for another. Depending on the target granularity several extractors might be required for a single language. We have developed a total of three extractors: two extractors extract TXL grammar and rules separately, and the third extracts both grammar and rules together from TXL source code. These extractors provide the necessary grammar blocks *define* and *redefine*, and transformation blocks *rule* and *function* of the TXL language.

The normalizer allows us to make near-miss clones more similar through different forms of transformation. Two normalizers have been used for our purposes to increase the possibility of detecting clones. We have developed one of them, while the other is the generic blind renamer provided with NiCAD.

3.3.2 Pattern Analysis of TXL Clones

We have developed NiPAT to find patterns from the clone class report generated by NiCAD. It takes the XML version of the clone class report as input and finds the set of files that appear together in a list of clone classes. After recognizing the file set and their respective clone class list, it retrieves the source code of each file from the classes and generates patterns. It produces the final pattern report in both XML and HTML formats. The HTML format provides two views: one is a column view and the other is a row view, for easy analysis of the patterns.

3.4 Type of Analysis and Result

By convention, TXL programmers maintain at least two source files to keep the grammar and transformation parts of their program separated. They may use any number of files as traditional include files in their program. They are also free to create source files mixing grammar and transformation rules together. For these reasons, we have performed the following three types of analysis:

Grammar Analysis: To retrieve grammar patterns we perform grammar analysis on source files containing grammar.

Rule Analysis: Rule analysis is performed to detect rule patterns from files containing transformation rules.

Complete TXL Program Analysis: If a source file contains the TXL *main* rule then we have resolved all *include* statements needed to make a complete program in a single source file combining grammar and transformation rules together,

and then we perform grammar and rule analysis together to identify patterns involving both grammar and transformation rules.

All of the above analysis has been performed on each of the individual projects in our collection and on the whole collection as a single project. Finally, we have identified TXL features of following three categories from the retrieved patterns:

- The first category includes proposals that would require TXL syntax/semantic modification to be implemented.
- The second category contains those proposals that are not feasible to add by modifying the syntax/semantics of the language but can be added as library features.
- The last category includes trivial proposals that are not of much significance compared to the previous two categories, but that can be added to TXL as a syntax/semantic modification or as a library feature if desired.

3.5 Summary

In this chapter, we have given an overview of the feature identification problem of the TXL programming language and our approach to solve that through source code analysis, repeated pattern detection, pattern analysis, and pattern recommendation. We have also given a brief description of NiCAD plugins and our tool NiPAT, the type of analysis we have performed, and the category of our obtained results. The next chapter discusses in detail the plugin programs we have developed for NiCAD in order to perform clone analysis on TXL source code. It is followed by the description of

the pattern detector tool NiPAT, source code analysis, and results in the subsequent sections.

Chapter 4

Clone Detector for TXL

To find exact and near-miss clones in TXL programs, we used the NiCAD clone detector [11]. Previously, NiCAD did not have the plugins needed to perform clone detection on TXL programs. However, we have enabled it to do so by developing the required plugin programs. Section 4.1, Section 4.2, and Section 4.3 discuss in detail the extractors, blind renamer, and normalizer we have developed for NiCAD.

4.1 Extractor Program

As we discussed in Section 2.2, NiCAD uses extractor programs written in TXL to extract code segments of the required granularity from the given source file. An extractor itself is a TXL program that has a grammar structure specification and a transformation section. Since our goal was to extract code segments of programs written using TXL, we needed to provide the grammar of the TXL language to the extractor. For this, we used the TXL grammar available from the TXL website [4].

To enable NiCAD to detect clones consisting of only grammar, only rule, and both grammar and rules, we have developed three separate extractors.

- **Grammar Extractor:** The grammar extractor extracts only the *define* and *redefine* statements from a given source file. It excludes all other grammar components, such as compounds, tokens, etc., because they are not an interesting part of the grammar for finding patterns for feature recommendation. Even in the case of a source file mixed with grammar and rules, the extractor can find the grammar segments of the target granularity successfully. The name of our extractor is *grm-extract-grammar.txtl*. It works on *.grm* extension only.
- **Rule Extractor:** All *rules* and *functions* from a TXL source file can be extracted using the rule extractor. The rule extractor works on rule files which ends with *.txtl* extension. However, if a file contains both grammar structure and transformation rules, it extracts only the rules and functions ignoring all other components of the source file. The name of the TXL rule extractor we have developed is *txtl-extract-rules.txtl*
- **Grammar and Rule Extractor:** As opposed to the previous two extractors, the grammar and rule extractor extracts both grammar and rule statements as our target granularities. It extracts *define*, *redefine*, *rule*, and *function* blocks from a given source file. This extractor works on a new type of file extension, called *.jm*, created by us. A file with the *.jm* extension is a complete TXL program source file containing both the grammar structure specification and transformation rules of a program together. The name of the developed grammar and rule extractor is *jgm-extract-txtl.txtl*

4.2 Blind Renamer Program

The blind renamer normalizes code by eliminating the effect of identifier renaming from cloned code. NiCAD provides a TXL source file *generic-rename-blind.txl* to replace the values of all [id] types of the extracted code fragments by x . This process is known as blind renaming. However, the code provided by NiCAD is not a complete program because it needs the definition of a nonterminal named *potential_clone*. The user must still write TXL code to define that nonterminal with the grammar specification of the extracted code fragments. This user-written code together with the code provided in *generic-rename-blind.txl* makes a complete blind renamer program. When the blind renamer is run by NiCAD, it finds all extracted *potential_clones* and replaces the values of all [id] type nodes with x (see Section 2.2.2, Figure 2.12). Blind renaming should be applied in situations when the user wants to perform clone detection ignoring the changes in the names of variables, functions, etc., whose type is [id].

In order to rename three types of our target granularities (grammar, rule, and both grammar and rule together), we have created the following three TXL files with the necessary grammar specification:

- **grm-rename-blind-grammar.txl** defines the *potential_clone* nonterminal with the definition of *define* and *redefine* statements.
- **txl-rename-blind-rules.txl** defines the *potential_clone* with the grammar specification of *rule* and *function*.
- **jm-rename-blind-txl.txl** includes all four TXL structures in which we are interested (*define*, *redefine*, *rule*, and *function*).

4.3 Normalizer Program

The purpose of a NiCAD normalizer is to remove textual dissimilarities of potential clones that are generated by source code editing activities. Identifier renaming also falls into this category but NiCAD treats them specially using the blind renamer program discussed in the previous section. However, the choice of dissimilarities to be removed to normalize source code editing effects depends solely on users. For example, if a user wants to, then can rearrange the code of the extracted fragments to increase the possibility of detecting clones. NiCAD normalizers are also programs written in TXL and they work on the code extracted by the extractors. If both blind renamers and normalizers are applied together then the normalizer works after blind renamer. The normalizer we developed is capable of performing normalization on all kinds of our target granularities. Its name is *txl-normalize-blocks.txl*. A block can be a define, redefine, rule, or function statement. It performs the normalizations below.

Short form normalization: TXL provides short form notation for each of the non-terminal modifiers for user convenience. They are also known as repeaters. For example, instead of writing *[repeat nonterminalname]*, we can write *[nonterminalname*]*. The modifier *opt* can be represented by a question mark at the end of a nonterminal, a list can be represented using a comma, and so on. A complete list of TXL modifiers and their short forms can be found in *The TXL Programming Language* [9] guide. Since NiCAD performs textual comparison to detect clones, it will fail to recognize that *[repeat X]* and *[X*]* are the same. As a result, this dissimilarity will affect the calculation of the similarity value. Therefore, we have normalized all short form notations to their complete forms. Figure 4.1 and 4.2 show the effect of using this normalization on sample code.

```
define records
    [repeat record] [opt propoerties] % Using full form of modifiers
    | [list record] [opt category]
end define

define emp_records
    [record*] [propoerties?] % Using short form of modifiers
    | [record,] [category?]
end define
```

Figure 4.1: Code before short form normalization

```
define records
    [repeat record] [opt propoerties]
    | [list record] [opt category] % No effect
end define

define emp_records
    [repeat record] [opt propoerties] % Changed to full form
    | [list record] [opt category]
end define
```

Figure 4.2: Code after short form normalization

Keyword normalization: The normalizer also renames the keywords *function*, *re-define* and *list* to *rule*, *define* and *repeat* respectively. Rules and functions have semantic differences significant to the TXL processor but their textual forms do not except the difference in keywords. The code a user can write using a function, can also be written as a rule, so they are a promising source of clones. But the difference in keywords will certainly lower the value of the measured similarity between two such pieces of code. As a result, near-miss clone detection might fail. The same is true for define and redefine. While lists allow commas between elements, repeats do not. Otherwise, lists and repeats have the same meaning. So the normalizer renames *list* to *repeat*. Figure 4.4 shows the changes due to keyword normalization applied on the sample code of Figure 4.3.

```

rule sort                                % Keyword rule
  replace [repeat number]
    N1 [number] N2 [number] Rest [repeat number]
  where
    N1[< N2]
  by
    N2 N1 Rest
end rule                                % Keyword rule

function changePos                      % Keyword function
  replace [repeat number]
    N1 [number] N2 [number] Rest [repeat number]
  where
    N1[> N2]
  by
    N2 N1 Rest
end function                            % Keyword function

```

Figure 4.3: Rule and function before keyword normalization

```

rule sort                                % No Change
  replace [repeat number]
    N1 [number] N2 [number] Rest [repeat number]
  where
    N1[< N2]
  by
    N2 N1 Rest
end rule                                % No Change

rule changePos                          % Changed to rule
  replace [repeat number]
    N1 [number] N2 [number] Rest [repeat number]
  where
    N1[> N2]
  by
    N2 N1 Rest
end rule                                % Changed to rule

```

Figure 4.4: Rule and function after keyword normalization

Parameter normalization: This normalizer removes parameters from rules and functions. The appearance of statements inside a rule or function is significant

to understanding patterns, but the parameters of a rule or function does not play any important role. For example, while a rule might perform an operation after receiving a list of parameters, another rule can do the same or a similar task by deconstructing a parameter. However, the presence of parameters in one rule, and the absence or difference in the number of parameters with the other can make a huge difference to a clone detector based on textual comparison. For that reason, we have normalized the parameters of rules and functions. The effect of such normalization has been illustrated by Figure 4.5 and 4.6.

```

function changeNumber N1 [number] N2 [number]    %Function with parameter
  replace [number]
    _ [number]
  by
    N1 [+ N2]
end function

function changeValue Num [repeat number]        % Function with parameter
  deconstruct Num
    N1 [number] N2 [number]
  replace [number]
    _ [number]
  by
    N1 [+ N2]
end function

```

Figure 4.5: Function before parameter normalization

```

function changeNumber                % Parameter removed
  replace [number]
    _ [number]
  by
    N1 [+ N2]
end function

function changeValue                % Parameter removed
  deconstruct Num
    N1 [number] N2 [number]
  replace [number]
    _ [number]
  by
    N1 [+ N2]
end function

```

Figure 4.6: Function after parameter normalization

Construct and deconstruct normalization: TXL allows `[type]`, *not* and `*` to be used with *deconstruct* as optional arguments and `[type]` as a mandatory argument with the construct statement [9]. `[type]` is a non-terminal, which can take different types of modifiers and user defined names; *not* is a keyword; and `*` is

used to denote a searching deconstruct. These arguments can make near-miss clones go undetected by raising textual dissimilarity. To increase the possibility of clone detection, the normalizer removes these arguments from constructor and deconstructor. For example, the argument *not* with a deconstruct instructs the TXL processor to continue execution when a pattern match fails. On the other hand, programmers can write the same code without a *not* to allow the TXL processor to continue when a pattern match succeeds. If we remove the *not*, textual dissimilarity will be reduced. For that reason, removal of those arguments from deconstructs and constructs can increase the likelihood of clone detection. Figure 4.8 shows the effect of such argument removal from the deconstruct of Figure 4.7.

```

function headSame Num [repeat number]
  replace [number]
    N [number]
  deconstruct Num          % No optional argument
    N _ [repeat number]
  by
    1
end function

function headDifferent Num [repeat number]
  replace [number]
    N [number]
  deconstruct not Num      % Using optional argument
    N _ [repeat number]
  by
    0
end function

```

Figure 4.7: Code before construct-deconstruct argument normalization

```

function headSame Num [repeat number]
  replace [number]
    N [number]
  deconstruct Num          % No change
    N _ [repeat number]
  by
    1
end function

function headDifferent Num [repeat number]
  replace [number]
    N [number]
  deconstruct Num          % Optional argument removed
    N _ [repeat number]
  by
    0
end function

```

Figure 4.8: Code after construct-deconstruct argument normalization

Alternative truncation: If we rename all identifiers blindly then sometimes we get *define* and *redefine* containing the same alternative repeatedly. In some cases,

the length of such a repetition is too long to get a match with a clone of shorter length. To avoid such situations, the normalizer truncates long repetition of blind alternatives and keeps up to three. This increases the chance of detecting clones when performing blind normalization. The code of Figure 4.10 illustrates the effect of alternative truncation done on the code of Figure 4.9 .


```
define x
    [x]
    | [x] % Long list of repeated alternatives
    | [x]
    | [x]
    | [x]
    | [x]
    | [x]
    | [x]
    | [x]
    | [x]
end define

define x
    [x]
    | [x] % Short list of repeated alternatives
    | [x]
    | [x]
    | [x]
end define
```

Figure 4.9: Code before alternative truncation

```
define x
    [x]
    | [x] % Same alternatives truncated
    | [x]
    | [x]
end define

define x
    [x]
    | [x] % Same alternatives truncated
    | [x]
    | [x]
end define
```

Figure 4.10: Code after alternative truncation

4.4 Summary

In this chapter, we have described the extractors, blind renamers, and normalizers that we have developed to enable NiCAD to perform clone analysis on TXL source

code. In the next chapter, we discuss in detail our pattern detection tool NiPAT, and present our analysis and results in the following chapters.

Chapter 5

Pattern Analysis of TXL Clones

The plugins we have developed for NiCAD to perform clone analysis on TXL source code have been described in the previous chapter. With the help of the developed plugins, NiCAD extracts and normalizes the code of the required granularity from the given source files. Then it generates clone reports by analyzing the extracted code. When the clone detection finishes, the pattern detector works to find patterns from the detected clones. In this chapter, we discuss the pattern detector NiPAT (Pattern Detection of Near-miss Intentional Clones) that we have developed to find patterns inside the NiCAD generated clone report.

5.1 Programming Language Used for the Pattern Detector

NiPAT has been developed using Python [3] and TXL. It uses Python for collecting files, fixing broken XML tags, and implementing the main logic of pattern identification. It also makes an external call to a TXL program to resolve the include instructions, which instruct that code from some other files must be put at the place where the instructions are found.

5.2 Properties of the Clone Report on which the Pattern Detector Works

The goal of the pattern detector is to find the set of files that have a set of clones in common and to retrieve the respective clone code snippets to make a pattern. It does this by using the clone report generated by clone detection.

The idea behind our pattern detector is to utilize clone report where clones are grouped into clone classes. A clone class holds clones of the same type from all the given source files. A good clone detector is expected to be efficient enough to put the right clones in the same group.

We have developed the pattern detector targeting the clone class report generated by NiCAD. As shown in Figure 5.1, each clone class of NiCAD gives a list of files and their clone code, which are grouped together based on the nominal size of the code and their similarity values. Individual clone instances from the same file might go under different classes, depending on their size and similarity value to the clones of its group. The pattern detector finds all of them from the clone report and puts them together in a pattern.

5.3 How Does the Pattern Detector Work?

NiPAT creates an environment to perform all tasks, including clone detection, without leaving the program. It prepares the files for detecting clones, invokes a command prompt for running NiCAD, and then implements the core pattern detection process on the detected clones. NiPAT starts by giving three choices to the user to find patterns from TXL source files. The user can select any one of them at a time. The first choice retrieves only grammar patterns, the second finds patterns consisting of only rules, and the third selection retrieves patterns consisting of both TXL grammar

Clone class 1, 2 fragments, nominal size 37 lines, similarity 100%

Lines 175 - 200 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl

```

rule convertFunctionDefinitions
  % Find each function definition and match its input source coordinates
  skipping [system_element]
  replace $ [system_element]
    FileName [srcfilename] LineNumber [srcclinenumber]
    'System '{
      Body [repeat default_element]
      EndFileName [srcfilename] EndLineNumber [srcclinenumber]
    '}'

  % Convert file name and line numbers to strings for XML
  construct FileNameString [stringlit]
    _ [quote FileName]
  construct LineNumberString [stringlit]
    _ [quote LineNumber]
  construct EndLineNumberString [stringlit]
    _ [quote EndLineNumber]

  % Output is XML form with attributes indicating input source coordinates
  by
    <source file=FileNameString startline=LineNumberString endline=EndLineNumberString>
      'System '{
        Body [unmarkEmbeddedFunctionDefinitions]
      '}'
    </source>
  end rule

```

Lines 163 - 188 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl

```

rule convertFunctionDefinitions
  % Find each function definition and match its input source coordinates
  skipping [model_list]
  replace $ [model_list]
    FileName [srcfilename] LineNumber [srcclinenumber]
    'Model '{
      Body [repeat default_element]
      EndFileName [srcfilename] EndLineNumber [srcclinenumber]
    '}'

  % Convert file name and line numbers to strings for XML
  construct FileNameString [stringlit]
    _ [quote FileName]
  construct LineNumberString [stringlit]
    _ [quote LineNumber]
  construct EndLineNumberString [stringlit]
    _ [quote EndLineNumber]

  % Output is XML form with attributes indicating input source coordinates
  by
    <source file=FileNameString startline=LineNumberString endline=EndLineNumberString>
      'Model '{
        Body [unmarkEmbeddedFunctionDefinitions]
      '}'
    </source>
  end rule

```

(a) NiCAD clone class report (*Part-1/5*)

Clone class 2, 2 fragments, nominal size 16 lines, similarity 100%

Lines 202 - 213 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl

```

rule unmarkEmbeddedFunctionDefinitions
  replace $ [system_element]
    FileName [srcfilename] LineNumber [srclinenumber]
    'System '{
      Body [repeat default_element]
      EndFileName [srcfilename] EndLineNumber [srclinenumber]
    }
  by
    'System '{
      Body
    }
end rule

```

Lines 190 - 201 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl

```

rule unmarkEmbeddedFunctionDefinitions
  replace $ [model_list]
    FileName [srcfilename] LineNumber [srclinenumber]
    'Model '{
      Body [repeat default_element]
      EndFileName [srcfilename] EndLineNumber [srclinenumber]
    }
  by
    'Model '{
      Body
    }
end rule

```

Clone class 3, 2 fragments, nominal size 16 lines, similarity 100%

Lines 163 - 178 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl

```

function orBothListAndGreaterByKind Element1 [default_element] Element2 [default_element]
  replace [number]
    0
    deconstruct Element1
      _ [default_list_element]
    deconstruct Element2
      _ [default_list_element]
    deconstruct * [id] Element1
      E1Id [id]
    deconstruct * [id] Element2
      E2Id [id]
    where
      E1Id [> E2Id]
    by
      1
end function

```

Lines 146 - 161 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl

```

function orBothSingleAndGreaterByKind Element1 [default_element] Element2 [default_element]
  replace [number]
    0
    deconstruct Element1
      _ [default_single_element]
    deconstruct Element2
      _ [default_single_element]
    deconstruct * [id] Element1
      E1Id [id]
    deconstruct * [id] Element2
      E2Id [id]
    where
      E1Id [> E2Id]
    by
      1
end function

```

(b) NiCAD clone class report (*Part-2/5*)

Clone class 4, 2 fragments, nominal size 10 lines, similarity 90%

Lines 152 - 161 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl

```

function main
  replace [program]
    P [program]
  construct Functions [repeat model_list] % [repeat model_element]
    - [ ^ P ] % Extract all functions from program
      [convertFunctionDefinitions] % Mark up with XML
      [filterIrrelevantElements] % Special for Simulink - take out irrelevant stuff
  by
    Functions
end function

```

Lines 164 - 173 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl

```

function main
  replace [program]
    P [program]
  construct Functions [repeat system_element]
    - [ ^ P ] % Extract all functions from program
      [convertFunctionDefinitions] % Mark up with XML
      %% [filterIrrelevantElements] % Special for Simulink - take out irrelevant stuff
  by
    Functions
end function

```

Clone class 5, 2 fragments, nominal size 8 lines, similarity 100%

Lines 103 - 110 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl

```

function toRight IfGreater [number] ThisElement [default_element]
  deconstruct not IfGreater
    1
  replace * [repeat default_element]
    Elements [repeat default_element]
  by
    Elements [ . ThisElement ]
end function

```

Lines 94 - 101 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl

```

function toLeft IfGreater [number] ThisElement [default_element]
  deconstruct IfGreater
    1
  replace * [repeat default_element]
    Elements [repeat default_element]
  by
    Elements [ . ThisElement ]
end function

```

Clone class 6, 2 fragments, nominal size 7 lines, similarity 100%

Lines 222 - 228 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl

```
rule removeIrrelevantElements
  replace [repeat default_element]
    IrrelevantElement [irrelevant_element]
    RestOfElements [repeat default_element]
  by
    RestOfElements
end rule
```

Lines 210 - 216 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl

```
rule removeIrrelevantElements
  replace [repeat default_element]
    IrrelevantElement [irrelevant_element]
    RestOfElements [repeat default_element]
  by
    RestOfElements
end rule
```

Clone class 7, 2 fragments, nominal size 6 lines, similarity 100%

Lines 215 - 220 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl

```
rule filterIrrelevantElements
  replace $ [system_element]
    System [system_element]
  by
    System [removeIrrelevantElements]
end rule
```

Lines 203 - 208 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl

```
rule filterIrrelevantElements
  replace $ [model_list]
    Model [model_list]
  by
    Model [removeIrrelevantElements]
end rule
```

(d) NiCAD clone class report (*Part-4/5*)

Clone class 8, 2 fragments, nominal size 6 lines, similarity 100%

Lines 251 - 256 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl

```
function orEqualAndNameGreater Type2 [id] N1 [stringlit] N2 [stringlit]
  match [id]
    Type2
  where
    N1 [> N2]
end function
```

Lines 287 - 292 of examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl

```
function orEqualAndDstGreater SB2 [stringlit] DB1 [stringlit] DB2 [stringlit]
  match [stringlit]
    SB2
  where
    DB1 [> DB2]
end function
```

(e) NiCAD clone class report (*Part-5/5*)

Figure 5.1: A clone class report in HTML generated by NiCAD

and rules.

1. Grammar only
2. Rule only
3. Grammar and Rule

After getting the user's choice, the pattern detector goes through the four steps below.

- Source file collection
- Clone analysis
- Clone report filtering
- Pattern detection

Source file collection: Based on the user's selection NiPAT searches the specified directory to gather all source files of the given extensions into a single directory.

Usually, a TXL project keeps its grammar specification and transformation rules in separate files with `.grm` and `.txl` extensions, respectively. However, users are free to use any other extension name they like to store them. All of those extension names must be given to the pattern detector to retrieve the files.

For example, if we use two different extensions, `.grm` and `.Grammar`, in our project to name TXL source files holding a grammar specification, then we have to provide both extension names to the pattern detector in order to get all grammar specifications. The pattern detector retrieves all files ending with the given grammar file extensions and, if it finds files ending with an extension other than `.grm`, renames them consistently with the `.grm` extension at the end. All file names must have the `.grm` extension in the end to use NiCAD properly during grammar pattern detection.

The same is true for retrieving the rule files but the pattern detector searches for files ending with rule file extensions and they are consistently named with the `.txl` extension at the end.

The scenario is different if the user wants to find patterns consisting of both grammar and rules together. In that case, the pattern detector selects a rule file first and resolves all include instructions by collecting code from the instructed files, and puts all the code together in a single file. To distinguish such files from other TXL files, it uses a new `.jm` file extension. A file with `.jm` extension is a complete TXL program having a main rule/function together with all necessary grammar specifications and transformation rules. Though the pattern detector

is mainly developed using Python, it gets help from a TXL program to resolve the include instructions. That TXL program opens the given file and, if it finds any include instructions in the file, then it reads the code from the file and replaces the include instruction with the obtained code.

At the end of the file retrieval process, NiPAT generates a report showing the path where the retrieved files have been stored, the source path, and the retrieval success message for each of the files having the given extensions in the given directory and its subdirectories. This report helps during analysis to trace back the origin of a source file. Figure 5.2 shows a sample file retrieval report generated by NiPAT after collecting the rule files of a project. It should be noted that our pattern detector does not move the original file, it just makes a copy.

```
---- Summary Report on TXL Rule File Retrieval----
TXL Rule Extension searched for:
('.txl', '.irul', '.rul', '.rule', '.i', '.module')

Retrieved File Stored in:
C:\...\Simone-1.3\TXLfiles <-- Original Source File Path
mdl-extract-models.txl <-- C:\...\Simone-1.3\txl
                           \mdl-extract-models.txl <CopySuccess>
mdl-extract-systems.txl <-- C:\...\Simone-1.3\txl
                           \mdl-extract-systems.txl <CopySuccess>
mdl-rename-blind-systems.txl <-- C:\...\Simone-1.3\txl
                           \mdl-rename-blind-systems.txl <CopySuccess>
mdl-transform-sort-systems.txl <-- C:\...\Simone-1.3\txl
                           \mdl-transform-sort-systems.txl <CopySuccess>

Total file found: 4
Copied Successfully: 4
Same File found: 0
Unknown Copy Error: 0
```

Figure 5.2: A report after retrieving rule files by NiPAT

Figure 5.3: Clone detection

Clone analysis: NiPAT does not perform clone analysis by itself. After collecting all files of the required type, it invokes the command prompt, for user convenience, to run NiCAD. The user gives the appropriate command to NiCAD providing the path of the directory where the pattern detector has stored all collected files of the target type. NiCAD then performs clone analysis on the files of the given directory and generates clone reports. Figure 5.3 illustrates the process to get the clone class report, and Figure 5.4 shows a sample clone class report generated by NiCAD on which NiPAT works. This report is the XML format of the HTML report presented in Figure 5.1. When the user finishes running NiCAD, the pattern detector gets control again and proceeds.

Clone report filtering: NiCAD generates clone pair and clone class reports in XML and HTML formats, but the pattern detector only uses the XML version of the clone class report. It might happen that the collected code in the clone report contains special characters reserved for XML syntax, and that has happened for *pcid="7"* and *pcid="2"* of Figure 5.4. In this case, the pattern detector

```

<clones>
<systeminfo processor="nicad3" system="txlfiles" granularity="rules-blind-normalized"
threshold="10%" minlines="6" maxlines="2500"/>
<cloneinfo npcs="28" npairs="8"/>
<runinfo ncompares="37" cputime="62"/>
<classinfo nclasses="8"/>

<class classid="1" nclones="2" nlines="37" similarity="100">
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl"
startline="175" endline="200" pcid="7">
rule convertFunctionDefinitions
  % Find each function definition and match its input source coordinates
  skipping [system_element]
  replace $ [system_element]
    FileName [srcfilename] LineNumber [srclinenumber]
    'System '{
      Body [repeat default_element]
      EndFileName [srcfilename] EndLineNumber [srclinenumber]
    '}'

  % Convert file name and line numbers to strings for XML
  construct FileNameString [stringlit]
    _ [quote FileName]
  construct LineNumberString [stringlit]
    _ [quote LineNumber]
  construct EndLineNumberString [stringlit]
    _ [quote EndLineNumber]

  % Output is XML form with attributes indicating input source coordinates
  by
    <source file=FileNameString startline=LineNumberString endline=EndLineNumberString>
      'System '{
        Body [unmarkEmbeddedFunctionDefinitions]
      '}'
    </source>
end rule
</source>
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl"
startline="163" endline="188" pcid="2">
rule convertFunctionDefinitions
  % Find each function definition and match its input source coordinates
  skipping [model_list]
  replace $ [model_list]
    FileName [srcfilename] LineNumber [srclinenumber]

```

(a) NiCAD clone class report in XML (*Part-1/6*)

```

    'Model '{
        Body [repeat default_element]
        EndFileName [srcfilename] EndLineNumber [srclinenumber]
    '}

% Convert file name and line numbers to strings for XML
construct FileNameString [stringlit]
    _ [quote FileName]
construct LineNumberString [stringlit]
    _ [quote LineNumber]
construct EndLineNumberString [stringlit]
    _ [quote EndLineNumber]

% Output is XML form with attributes indicating input source coordinates
by
    <source file=FileNameString startline=LineNumberString endline=EndLineNumberString>
        'Model '{
            Body [unmarkEmbeddedFunctionDefinitions]
        '}
    </source>
end rule
</source>
</class>

<class classid="2" nclones="2" nlines="16" similarity="100">
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl"
startline="202" endline="213" pcid="8">
rule unmarkEmbeddedFunctionDefinitions
    replace $ [system_element]
        FileName [srcfilename] LineNumber [srclinenumber]
        'System '{
            Body [repeat default_element]
            EndFileName [srcfilename] EndLineNumber [srclinenumber]
        '}
    by
        'System '{
            Body
        '}
end rule
</source>
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl"
startline="190" endline="201" pcid="3">
rule unmarkEmbeddedFunctionDefinitions
    replace $ [model_list]
        FileName [srcfilename] LineNumber [srclinenumber]

```

(b) NiCAD clone class report in XML (*Part-2/6*)

```

        'Model '{
            Body [repeat default_element]
            EndFileName [srcfilename] EndLineNumber [srclinenumber]
        '}
    by
        'Model '{
            Body
        '}
end rule
</source>
</class>

<class classid="3" nclones="2" nlines="16" similarity="100">
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl"
startline="163" endline="178" pcid="22">
function orBothListAndGreaterByKind Element1 [default_element] Element2 [default_element]
    replace [number]
        0
        deconstruct Element1
            _ [default_list_element]
        deconstruct Element2
            _ [default_list_element]
        deconstruct * [id] Element1
            E1Id [id]
        deconstruct * [id] Element2
            E2Id [id]
        where
            E1Id [> E2Id]
        by
            1
end function
</source>
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl"
startline="146" endline="161" pcid="21">
function orBothSingleAndGreaterByKind Element1 [default_element] Element2 [default_element]
    replace [number]
        0
        deconstruct Element1
            _ [default_single_element]
        deconstruct Element2
            _ [default_single_element]
        deconstruct * [id] Element1
            E1Id [id]
        deconstruct * [id] Element2
            E2Id [id]

```

(c) NiCAD clone class report in XML (*Part-3/6*)

```

        where
            E1Id [> E2Id]
        by
            1
    end function
</source>
</class>

<class classid="4" nclones="2" nlines="10" similarity="90">
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl"
startline="152" endline="161" pcid="1">
function main
    replace [program]
        P [program]
    construct Functions [repeat model_list] % [repeat model_element]
        _ [ ^ P] % Extract all functions from program
            [convertFunctionDefinitions] % Mark up with XML
            [filterIrrelevantElements] % Special for Simulink - take out irrelevant stuff
    by
        Functions
    end function
</source>
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl"
startline="164" endline="173" pcid="6">
function main
    replace [program]
        P [program]
    construct Functions [repeat system_element]
        _ [ ^ P] % Extract all functions from program
            [convertFunctionDefinitions] % Mark up with XML
        %% [filterIrrelevantElements] % Special for Simulink - take out irrelevant stuff
    by
        Functions
    end function
</source>
</class>

<class classid="5" nclones="2" nlines="8" similarity="100">
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl"
startline="103" endline="110" pcid="17">
function toRight IfGreater [number] ThisElement [default_element]
    deconstruct not IfGreater
        1
    replace * [repeat default_element]

```

(d) NiCAD clone class report in XML (*Part-4/6*)


```

        Elements [repeat default_element]
    by
        Elements [. ThisElement]
end function
</source>
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl"
startline="94" endline="101" pcid="16">
function toLeft IfGreater [number] ThisElement [default_element]
    deconstruct IfGreater
    1
    replace * [repeat default_element]
        Elements [repeat default_element]
    by
        Elements [. ThisElement]
end function
</source>
</class>

<class classid="6" nclones="2" nlines="7" similarity="100">
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl"
startline="222" endline="228" pcid="10">
rule removeIrrelevantElements
    replace [repeat default_element]
        IrrelevantElement [irrelevant_element]
        RestOfElements [repeat default_element]
    by
        RestOfElements
end rule
</source>
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl"
startline="210" endline="216" pcid="5">
rule removeIrrelevantElements
    replace [repeat default_element]
        IrrelevantElement [irrelevant_element]
        RestOfElements [repeat default_element]
    by
        RestOfElements
end rule
</source>
</class>

<class classid="7" nclones="2" nlines="6" similarity="100">
<source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-systems.txl"
startline="215" endline="220" pcid="9">
rule filterIrrelevantElements
    replace $ [system_element]
        System [system_element]

```

(e) NiCAD clone class report in XML (*Part-5/6*)

```

        by
            System [removeIrrelevantElements]
        end rule
    </source>
    <source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-extract-models.txl"
    startline="203" endline="208" pcid="4">
        rule filterIrrelevantElements
            replace $ [model_list]
                Model [model_list]
            by
                Model [removeIrrelevantElements]
            end rule
        </source>
    </class>

    <class classid="8" nclones="2" nlines="6" similarity="100">
    <source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl"
    startline="251" endline="256" pcid="26">
        function orEqualAndNameGreater Type2 [id] N1 [stringlit] N2 [stringlit]
            match [id]
                Type2
            where
                N1 [> N2]
            end function
        </source>
    <source file="examples/moretxl/moretxl/simone-1.3/txlfiles/mdl-transform-sort-systems.txl"
    startline="287" endline="292" pcid="28">
        function orEqualAndDstGreater SB2 [stringlit] DB1 [stringlit] DB2 [stringlit]
            match [stringlit]
                SB2
            where
                DB1 [> DB2]
            end function
        </source>
    </class>

</clones>

```

(f) NiCAD clone class report in XML (*Part-6/6*)

Figure 5.4: XML format of the clone class report of Figure 5.1

Figure 5.5: Parse tree generation

fails to analyze the report with broken XML syntax. In order to perform clone analysis successfully, it scans the clone report first for broken XML tags and, if any, fixes them, and steps into the final phase of pattern identification.

Pattern detection: This is the last phase of pattern recognition. In this phase, NiPAT generates a parse tree of the filtered clone results. Figure 5.5 shows the parse tree that will be generated by our pattern detector from the clone class report of Figure 5.3. It then finds all unique file names from the parse tree and generates all possible combinations of files as shown in Figure 5.6. Next, NiPAT selects each file set one by one and maps them onto the parse tree to find the set of clone classes where each particular file set appears. After collecting the set of clone classes for a particular file set, NiPAT retrieves the clone code snippets from the clone class set related to that file set and marks the retrieved set of code clones as a pattern as shown in Figure 5.7.

The pattern detector reports patterns in XML and HTML formats. The HTML format generates two different views, column view and row view, for

Figure 5.6: File set generation

easy analysis of the detected patterns by human experts. The column view shows the code of a pattern in columns and the row view shows them in rows. Figure 5.8 shows a sample column view pattern report of our given example.

After analyzing the clone class report of Figure 5.4, NiPAT finds a pattern where code from two files, *mdl-extract-systems.txt* and *mdl-extract-models.txt*, appear together in five clone classes (1, 2, 4, 5, and 7). Figure 5.9 shows the generated pattern report in column view HTML format.

(a) Pattern generation of the file set having ($f1$, $f2$) (b) Pattern generation of the file set having ($f2$, $f3$)

(c) Pattern generation of the file set having (f3, f4)

(d) Pattern generation of the file set having (f1, f2, f3)

Figure 5.7: Pattern generation by NiPAT

Figure 5.8: Pattern report by NiPAT

PatternID: 1 Total files in set: 2 [examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-systems.tbl, 'examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-models.tbl] Appears in total 5 clone classes: [1, 2, 4, 6, 7]	
Class: 1 of 5 ...Code in NiCAD Clone Class: 1 ...Similarity in NiCAD origin Class: 100	
examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-systems.tbl CloneClass= 1 Similarity= 100	examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-models.tbl CloneClass= 1 Similarity= 100
<pre> rule convertFunctionDefinitions % Find each function definition and match its input source coordinates skipping [system_element] replace \$ [system_element] FileName [srcfilename] LineNumber [srclineno] 'System '{ Body [repeat default_element] EndFileName [srcfilename] EndLineNumber [srclineno] '}' % Convert file name and line numbers to strings for XML construct FileNameString [stringlit] _ [quote FileName] construct LineNumberString [stringlit] _ [quote LineNumber] construct EndLineNumberString [stringlit] _ [quote EndLineNumber] % Output is XML form with attributes indicating input source coordinates by <source file=FileNameString startline=LineNumberString endline=EndLineNumberString> 'System '{ Body [unmarkEmbeddedFunctionDefinitions] '}' </source> end rule </pre>	<pre> rule convertFunctionDefinitions % Find each function definition and match its input source coordinates skipping [model_list] replace \$ [model_list] FileName [srcfilename] LineNumber [srclineno] 'Model '{ Body [repeat default_element] EndFileName [srcfilename] EndLineNumber [srclineno] '}' % Convert file name and line numbers to strings for XML construct FileNameString [stringlit] _ [quote FileName] construct LineNumberString [stringlit] _ [quote LineNumber] construct EndLineNumberString [stringlit] _ [quote EndLineNumber] % Output is XML form with attributes indicating input source coordinates by <source file=FileNameString startline=LineNumberString endline=EndLineNumberString> 'Model '{ Body [unmarkEmbeddedFunctionDefinitions] '}' </source> end rule </pre>
Class: 2 of 5 ...Code in NiCAD Clone Class: 2 ...Similarity in NiCAD origin Class: 100	
examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-systems.tbl CloneClass= 2 Similarity= 100	examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-models.tbl CloneClass= 2 Similarity= 100
<pre> rule unmarkEmbeddedFunctionDefinitions replace \$ [system_element] FileName [srcfilename] LineNumber [srclineno] 'System '{ Body [repeat default_element] EndFileName [srcfilename] EndLineNumber [srclineno] '}' by 'System '{ Body '}' end rule </pre>	<pre> rule unmarkEmbeddedFunctionDefinitions replace \$ [model_list] FileName [srcfilename] LineNumber [srclineno] 'Model '{ Body [repeat default_element] EndFileName [srcfilename] EndLineNumber [srclineno] '}' by 'Model '{ Body '}' end rule </pre>

(a) Pattern report of Figure 5.4 in HTML by NiPAT (*Part-1/2*)

5.4 Summary

The purpose of the pattern detector is to find the set of cloned code that appears commonly in a set of source files. The pattern detector we have developed to find patterns by analyzing NiCAD generated clone results has been discussed in this chapter. We have given, in detail, the steps it follows starting from taking the user's choice of pattern detection to the final pattern report. When pattern detection finishes, human experts work on them to understand their meaning in order to recommend new language features. In the following chapters, we present our analysis and results

Class: 3 of 5 ...Code in NiCAD Clone Class: 4 ...Similarity in NiCAD origin Class: 90	
<pre> examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-systems.td CloneClass= 4 Similarity= 90 function main replace [program] P [program] construct Functions [repeat system_element] - (^ P] % Extract all functions from program [convertFunctionDefinitions] % Mark up with XML %% [filterIrrelevantElements] % Special for Simulink - take out irrelevant stuff by Functions end function </pre>	<pre> examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-models.td CloneClass= 4 Similarity= 90 function main replace [program] P [program] construct Functions [repeat model_list][repeat model_element] - (^ P] % Extract all functions from program [convertFunctionDefinitions] % Mark up with XML [filterIrrelevantElements] % Special for Simulink - take out irrelevant stuff by Functions end function </pre>
Class: 4 of 5 ...Code in NiCAD Clone Class: 6 ...Similarity in NiCAD origin Class: 100	
<pre> examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-systems.td CloneClass= 6 Similarity= 100 rule removeIrrelevantElements replace [repeat default_element] IrrelevantElement [irrelevant_element] RestOfElements [repeat default_element] by RestOfElements end rule </pre>	<pre> examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-models.td CloneClass= 6 Similarity= 100 rule removeIrrelevantElements replace [repeat default_element] IrrelevantElement [irrelevant_element] RestOfElements [repeat default_element] by RestOfElements end rule </pre>
Class: 5 of 5 ...Code in NiCAD Clone Class: 7 ...Similarity in NiCAD origin Class: 100	
<pre> examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-systems.td CloneClass= 7 Similarity= 100 rule filterIrrelevantElements replace % [system_element] System [system_element] by System [removeIrrelevantElements] end rule </pre>	<pre> examples/moretd/moretd/simone-1.3/bdfiles/mdl-extract-models.td CloneClass= 7 Similarity= 100 rule filterIrrelevantElements replace % [model_list] Model [model_list] by Model [removeIrrelevantElements] end rule </pre>

(b) Pattern report of Figure 5.4 in HTML by NiPAT (*Part-2/2*)

Figure 5.9: Pattern report of Figure 5.4 in HTML by NiPAT

in detail.

Chapter 6

TXL Project Analysis

In the previous chapter, we discussed the pattern detector tool NiPAT, which can be used to find programming patterns in TXL source code. In this chapter, we give an overview of our collection of TXL projects, type of analysis, and the proposals we made to enhance the TXL feature set.

6.1 Project Description

We have collected a total 83 TXL projects for analyzing with our pattern detector tool. These projects were taken from the TXL project collection of user applications, and from the example applications on the TXL website [4]. Projects from the website are freely available for download and come from various fields, such as general purpose language translation, mark-up language processing, software re-architecture, graph processing, etc. As mentioned on the website, these projects were developed by TXL staff and users.

6.1.1 Preprocessing

To avoid unnecessary comparison among files, we performed some manual preprocessing on the collected projects, such as, deletion of files from projects that contain a main rule/function but do nothing except print the input program by including grammar and rule files. We deleted such files to avoid comparison of a file with itself due to inclusion. For the same reason, we removed inclusion of the same file by more than one source file within the same project, and across projects. Fortran, SybAPT2MSSPB7, PHP345, SoftwareGraph, etc. are some examples of such projects.

After preprocessing, we were left with a total 370 files containing TXL code in 83 TXL projects. It is difficult to determine the exact number of grammar and rule files separately because rule files may also contain grammar, and sometimes users use non-standard file extensions to store both grammar and rules. As a result, their type is not clear. However, we obtained a total of 210 files containing a main rule/function.

6.2 Grammar Analysis

The purpose of grammar analysis is to find patterns involving only grammar specifications. We performed the following two types of analysis on the collected projects:

- **Grammar Analysis of Individual Projects:** During this analysis we considered each project separately to find grammar patterns of the particular project.
- **Grammar Analysis of Merged Collection:** We put all grammar files from the collected projects into a single directory, and then performed pattern analysis to find grammar patterns that appear across projects.

6.2.1 Proposals

After retrieving the grammar patterns using NiPAT, we analyzed them to understand their meaning and propose new features. Instead of presenting the whole pattern in this report, we present representative code from each pattern since they are often too big to fit within a short number of pages. A change in clone class is shown by a sequence of dots (.) in the pattern. By analyzing the grammar patterns, we make the following proposals that can help improve the grammar syntax and semantics of TXL.

Grammar Proposal 1: Predefined non-terminals to mark extracted patterns with XML source coordinates.

```
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
```

Figure 6.1: A representative pattern defining two commonly used XML non-terminals

When users extract patterns from program source files, they usually need to know the source file name, and the beginning and ending line number of the extracted patterns. By default, TXL provides those in plain text format, but grammar pattern analysis reveals that programmers often define two common non-terminals to mark the extracted code patterns with XML tags. The pattern that helps us come to this

decision is shown in Figure 6.1. It is recommended to provide these non-terminals as library features for users. In addition to reducing user effort, it will make their code shorter. Figure 6.3 shows the change that the proposed feature would make in coding, compared to the existing practise of coding shown in Figure 6.2, to achieve the same purpose.

```

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

define system_element
    % Input form
    [srcfilename] [srclinenumber]
    'System '{ [NL] [IN]
        [repeat default_element] [EX]
    [srcfilename] [srclinenumber]
    '}' [NL]

    % Output form
    | [opt xml_source_coordinate]
    'System '{ [NL] [IN]
        [repeat default_element] [EX]
    '}' [NL]
    [opt end_xml_source_coordinate]
end define

```

Figure 6.2: Code defining and using commonly used non-terminals *xml_source_coordinate* and *end_xml_source_coordinate*

```
include xmlLibrary % Defines xml_source_coordinate &
                  % end_xml_source_coordinate

define system_element
    % Input form
    [srcfilename] [srclinenumber]
    'System '{ [NL][IN]
        [repeat default_element] [EX]
    [srcfilename] [srclinenumber]
    '}' [NL]

    % Output form
    | [opt xml_source_coordinate] % Defined in library file
    'System '{ [NL][IN]
        [repeat default_element] [EX]
    '}' [NL]
    [opt end_xml_source_coordinate] % Defined in library file
end define
```

Figure 6.3: Code using *xml_source_coordinate* and *end_xml_source_coordinate* defined in the library file

Grammar Proposal 2: A keyword to mention a non-terminal in its own definition.

```
define PrivateSequenceStmt
    [LblDef] 'private [EOS]
    |   [LblDef] 'sequence [EOS]
end define

.....

define ElseWhere
    [Where] [ElsewhereStmt]
    |   [ElseWhere] [AssignmentStmt]
end define

define StructureComponent
    [VariableName] [FieldSelector]
    |   [StructureComponent] [FieldSelector]
end define

define WhereConstruct
    [Where] [EndWhereStmt]
    |   [ElseWhere] [EndWhereStmt]
end define
```

Figure 6.4: A representative pattern showing the use of a non-terminal in its own definition

From the grammar pattern of Figure 6.4, we find that programmers sometimes need to refer a non-terminal in its own definition. Until now, the only way to do this was to write the name of the non-terminal inside its definition like other non-terminals. Such practise does not distinguish the defining non-terminal from other non-terminals, and as a result, it is not clear at a glance that it refers to the same non-terminal being defined. Besides, if non-terminal names are very close, they may cause confusion,

and the user will need to go back to check the name of the defining non-terminal. To avoid this, TXL could provide a key non-terminal *[self]*, to allow the user to specify that the defining non-terminal will come at the place of *[self]*. It would make the code more easily understandable. Figure 6.5 shows a piece of code where the defining non-terminal is treated like other non-terminals in its definition. On the other hand, Figure 6.6 shows a piece of code where it is privileged with a special key non-terminal *[self]* that eliminates the need of going back to check whether it is the defining non-terminal or not.

```
define else_where
    [where] [else_where_stmt]
    | [else_where] [assignment_stmt]
end define
```

Figure 6.5: Code before using *self*

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

Figure 6.6: Code after using *self*

Grammar Proposal 3: Selected non-terminals to repeat at the start or end of a sequence of alternative definitions.

Sometimes a particular non-terminal needs to be repeated at the beginning or end of each alternative definition of a defining non-terminal. The definition of the *file_description_clause* non-terminal of Figure 6.7 presents such an example. Though, the rest of this pattern is uninteresting, we have shown them to present the whole pattern retrieved by NiPAT. However, there are two ways to repeat a non-terminal at the


```

define file_description_clause
    [TAB_25] [block_clause]
    | [TAB_25] [record_clause]
    | [TAB_25] [data_clause]
    | [TAB_25] [label_clause]
    | [TAB_25] [FD_value_clause]
    | [TAB_25] [linage_clause]
    | [TAB_25] [code_set_clause]
    | [TAB_25] [top_clause]
    | [TAB_25] [bottom_clause]
    | [TAB_25] [recording_clause]
    | [TAB_25] [report_clause]
    | [TAB_25] [sort_option_clause]
end define

.....

define cond_condition_primary
    ( [condition_expression] )
    | [TRUE_or_FALSE]
    | [class_condition]
    | [sign_condition]
    | [relation_condition]
    | [io_condition]
    | [error_condition]
    | [identifier] [opt IS_TRUE_or_FALSE]
end define

.....

define description_clause
    [picture_clause]
    | [redefines_clause]
    | [opt 'IS] 'EXTERNAL
    | [opt 'IS] 'GLOBAL
    | [usage_clause]
    | [sign_clause]
    | [synchronization_clause]
    | [justification_clause]
    | [blank_when_clause]
    | [value_clause]
    | [occurs_clause]
end define

.....

```

(a) A representative pattern using the same non-terminal at the beginning of a sequence of non-terminals (*Part-1/2*)

```

define alternate_key_clause
    [NL] 'ALTERNATE [opt 'RECORD] [opt 'KEY]
    [opt 'IS] [key_identifier] [opt WITH_DUPLICATES]
end define

.....

define key_comparison
    'KEY [opt 'IS] [opt 'NOT] [relational_operator] [key_identifier]
end define

.....

define record_key_clause
    [NL] 'RECORD [opt 'KEY] [opt 'IS] [key_identifier] [opt password_clause]
end define

.....

define merge_order
    [opt 'ON] [ASCENDING_or_DESCENDING] [opt 'KEY] [repeat key_identifier+]
end define

define sort_order
    [opt 'ON] [ASCENDING_or_DESCENDING] [opt 'KEY] [repeat key_identifier+]
end define

.....

define ordering_clause
    [ASCENDING_or_DESCENDING] [opt 'KEY] [opt 'IS] [repeat key_identifier+]
end define

.....

define nominal_key_clause
    [NL] 'NOMINAL [opt 'KEY] [opt 'IS] [key_identifier]
end define

define relative_key
    [NL] 'RELATIVE [opt 'KEY] [opt 'IS] [key_identifier]
end define

```

(b) A representative pattern using the same non-terminal at the beginning of a sequence of non-terminals (*Part-2/2*)

Figure 6.7: A representative pattern using the same non-terminal at the beginning of a sequence of non-terminals

beginning or end of a sequence of alternatives. One way is to write the name of the non-terminal with each alternative definition. Another way is to define a second non-terminal holding all the alternative definitions, except the non-terminal to be repeated, and to place the non-terminal before or after the defined non-terminal.

Instead of writing the same name repeatedly, or going through a second non-terminal, we can allow the user to repeat a non-terminal at the beginning or end of each alternative definition by defining two keywords *start* and *end*. A non-terminal name given with the *start* and *end* keywords will be repeated for all alternatives between them. It will keep the code clean and increase flexibility. Figure 6.9 presents cleaner code using the keywords *start* and *end* in the code of Figure 6.8.

```

define file_description_clause
    [TAB_25] [block_clause]
    | [TAB_25] [record_clause]
    | [TAB_25] [data_clause]
    | [TAB_25] [label_clause]
    | [TAB_25] [FD_value_clause]
    | [TAB_25] [linage_clause]
    | [TAB_25] [code_set_clause]
    | [TAB_25] [top_clause]
    | [TAB_25] [bottom_clause]
    | [TAB_25] [recording_clause]
    | [TAB_25] [report_clause]
    | [TAB_25] [sort_option_clause]
end define

```

Figure 6.8: Code before using *start* and *end*

```

define file_description_clause
    [TAB_25 start] [block_clause]
    | [record_clause]
    | [data_clause]
    | [label_clause]
    | [FD_value_clause]
    | [linage_clause]
    | [code_set_clause]
    | [top_clause]
    | [bottom_clause]
    | [recording_clause]
    | [report_clause]
    | [TAB_25 end] [sort_option_clause]
end define

```

Figure 6.9: Code after using keywords *start* and *end*

6.3 Rule Analysis

The purpose of rule analysis is to find patterns involving TXL rules and/or functions. The following two types of rule analysis was performed on the collected projects:

- **Rule Analysis of Individual Projects:** Analysis of individual projects was performed to retrieve patterns consisting of sets of TXL rules and/or functions

within a project.

- **Rule analysis of Merged Collection:** This analysis was performed after putting all the rule files in the same directory to get the rule patterns that show up across different projects.

6.3.1 Proposals

Using NiPAT, we analyzed TXL files and retrieved patterns involving only TXL rules and/or functions. In this section, we present concise representative code from selected patterns that lead to some proposals. We use a sequence of dots (.) inside a representative pattern to mean change in clone class. After analyzing the retrieved patterns we propose the following features.

Rule Proposal 1: Allow users to find unique elements from a TXL *repeat* type variable.

A variable with the *repeat* keyword holds elements of the same type. When needed, programmers remove nodes holding the same terminal value from such a variable to make each element unique by applying their own technique. The pattern of Figure 6.10 reveals this. TXL does not provide any feature to find the unique elements of a *repeat* type variable. A new TXL feature to find unique elements from a *repeat* variable would save programmers the effort of writing their own.

Figure 6.11 shows a sample code snippet using the proposed feature to find unique elements from a *repeat* variable *Exprsn*. We have used *!!* as a shorthand notation to indicate that it will invoke the appropriate TXL built-in function to find unique elements from the given variable, and will help to generate the *UniqueExprsn* variable

```

function deleteDuplicates
  replace [repeat date_type]
    ADateType [date_type] RestDates [repeat date_type]
  by
    ADateType
    RestDates [deletedDT ADateType]
               [deleteDuplicates]
end function

.....

rule deletedDT ADT [date_type]
  replace [repeat date_type]
    ADT RestDates [repeat date_type]
  by
    RestDates
end rule

```

Figure 6.10: A representative pattern removing identical elements from a *repeat*-type variable

in our example, with the unique elements from *Exprsn*. Figure 6.12 shows what we expect to get in *UniqueExprsn* after executing `[!]` on *Exprsn*.

```

rule main
  replace [repeat expression]
    Exprsn [repeat expression]
  construct UniqueExprsn [repeat expression]
    Exprsn [!!] % [!!] finds unique elements from repeat variable Exprsn
  by
    UniqueExprsn [evaluateExprsn] % Evaluates expressions
end rule

```

Figure 6.11: Example code to find unique elements from a repeat type variable

```

Sample elements of Exprsn:
  1+2
  1+2
  4+5
  6+2

Elements of UniqueExprsn after executing Exprsn[!!]:
  1+2
  4+5
  6+2

```

Figure 6.12: Sample execution of the proposed feature to find unique elements from a variable with repeat type

Rule Proposal 2: Allow users to find whether a *repeat* type variable is a subsequence of another variable of the same type.

By analyzing the pattern shown in Figure 6.13, we conclude that the user attempted to find a subsequence in a TXL *repeat* variable. A *repeat* type variable holds elements of the same type. TXL provides an easy way to find whether a repeat type variable is same as another variable of its type using *deconstruct*, but it does not provide any way to find whether a *repeat* variable is a subsequence of another variable of the same type. The pattern of Figure 6.13 shows the need for this. A TXL feature to find subsequence in a *repeat* type variable would certainly help in coding when needed.

The code in Figure 6.14 shows an example where we check whether a *repeat*

```

function matches PartialUid [repeat id]
  deconstruct PartialUid
    FirstId [id] RestPartialIds [repeat id]
  match [repeat id]
    FirstId RestUids [repeat id]
  where
    RestUids [matches RestPartialIds]
              [matchEmpty RestPartialIds]
end function

.....

function matchEmpty PartialIds [repeat id]
  deconstruct PartialIds
    %% Empty

  match [repeat id]
    Ids [repeat id]
  where
    Ids [isOne]
        [isEmpty]
end function

```

Figure 6.13: A representative pattern searching for a subsequence

variable *Num* is a subsequence of another variable *NumMain* using the proposed feature. It is assumed in our coding that the subsequence notation (@) will invoke the needed TXL function to find if the left variable is a subsequence of the right, and generate either True or False based on the result. Expected results from two sample executions using the proposed feature have been shown in Figure 6.15. In the case of the first input, the program detects a subsequence, but does not detect a subsequence for the second input.


```

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] % '@' assumed subsequence check symbol
  by
    % Empty if subsequence
end function

```

Figure 6.14: Code using the proposed subsequence feature

```

Sample input-1:
  12 39 48 56 77 pqrs 12 39 48

Sample output-1:
  12 39 48 56 77 pqrs

Sample input-2:
  17 88 56 bvtty 39 48

Sample output-2:
  17 88 56 bvtty 39 48

```

Figure 6.15: Sample execution of the proposed subsequence feature

Rule Proposal 3: Skip the embedded patterns during extraction.

Programmers sometimes want to get only the uppermost instances of a non-terminal, ignoring embedded ones of the same type which is clear from the pattern of Figure 6.16. A new feature could be provided to avoid the embedded patterns during

```

rule convertFunctionDefinitions
    % Find each function definition and match its input source coordinates
    skipping [system_element]
    replace $ [system_element]
        FileName [srcfilename] LineNumber [srclinenumber]
        'System '{
            Body [repeat default_element]
            EndFileName [srcfilename] EndLineNumber [srclinenumber]
        '}'

    % Convert file name and line numbers to strings for XML
    construct FileNameString [stringlit]
        _ [quote FileName]
    construct LineNumberString [stringlit]
        _ [quote LineNumber]
    construct EndLineNumberString [stringlit]
        _ [quote EndLineNumber]

    % Output is XML form with attributes indicating input source coordinates
    by
        <source file=FileNameString startline=LineNumberString
        endline=EndLineNumberString>
            'System '{
                Body [unmarkEmbeddedFunctionDefinitions]
            '}'
        </source>
end rule

.....

rule unmarkEmbeddedFunctionDefinitions
    replace $ [system_element]
        FileName [srcfilename] LineNumber [srclinenumber]
        'System '{
            Body [repeat default_element]
            EndFileName [srcfilename] EndLineNumber [srclinenumber]
        '}'
    by
        'System '{
            Body
        '}'
end rule

.....

```

(a) A representative pattern skipping the embedded patterns (*Part-1/2*)

```

function main
  replace [program]
    P [program]
  construct Functions [repeat system_element]
    _ [^ P]          % Extract all functions from program
  [convertFunctionDefinitions] % Mark up with XML
  %% [filterIrrelevantElements] % Special for Simulink
                                - take out irrelevant stuff

  by
    Functions
end function

.....

rule removeIrrelevantElements
  replace [repeat default_element]
    IrrelevantElement [irrelevant_element]
    RestOfElements [repeat default_element]
  by
    RestOfElements
end rule

.....

rule filterIrrelevantElements
  replace $ [system_element]
    System [system_element]
  by
    System [removeIrrelevantElements]
end rule

```

(b) A representative pattern skipping the embedded patterns (*Part-2/2*)

Figure 6.16: A representative pattern skipping the embedded patterns

extraction.

Figure 6.18 shows the expected output by executing the code of Figure 6.17 on the sample input. In the code of Figure 6.17, to invoke the proposed feature to skip the embedded patterns during extraction, we have used the double hat notation ($\hat{\hat{}}$). From the expected output, we see that the proposed feature skips the *if y* block when it is nested within the *if x* block.

```
rule main
  replace [program]
    P [program]
  construct MyIf [repeat if_statement]
    _ [ $\hat{\hat{P}}$ ] %  $\hat{\hat{\phantom{x}}}$  skips the embedded patterns
  by
    MyIf
end rule
```

Figure 6.17: Example code using the proposed feature to skip the embedded patterns

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

Sample output:
Myfile 3 % File name and start line number.
if x:
    y=4-x
    if y: % Embedded pattern is skipped
        x=x*3
Myfile 6

Myfile 5
    if y:
        x=x*3
Myfile 6
```

Figure 6.18: Sample execution using the proposed extract feature to skip the embedded patterns

Rule Proposal 4: *construct* can be modified to perform multiple constructs of the same type.

The existing *construct* feature of TXL allows programmers to perform one construct at a time. But programmers sometimes need to write multiple constructs of the same type one after another, as shown by the pattern of Figure 6.19. It would be more suitable if the *construct* instruction were allowed to perform multiple constructs of the same type in a single instruction.

Figure 6.20 shows the existing approach to performing multiple constructs of the same type, and Figure 6.21 shows the change the proposed feature would have.

```

rule markIfUsedByIdent
    Statement [informationModelStatement_48]
    mentioned_pattern [informationModelStatement_48]

    deconstruct mentioned_pattern
        pW [patternWords_666]
        ':
        eVP [entityVectorPattern_76]
        sT [statementTerminator_22]

    construct search_pattern [informationModelStatement_48]
        'Unment_ pW ': eVP sT

    construct replace_pattern [informationModelStatement_48]
        'Ment_ pW ': eVP sT

    construct added_pattern [informationModelStatement_48]
        'Ident_ pW ': eVP sT

    replace [repeat informationModelStatement_48]
        search_pattern
        rest [repeat informationModelStatement_48]

    % exit if not found. Else, mark the pattern as having been
    % mentioned, and as the subject of an Identifiers statement.

    by
        replace_pattern
        rest [. added_pattern]
end rule

.....

function buildIdentAtomicPattern
    pEV [primeEntityVariable_79]
    sAP [simpleAttributePhrase_129]

    % From the parameters, construct an atomicPattern statement

    construct sAPD_nul [singleAttributePhraseDeclaration_82]
        'Nul

    % nul

```

- (a) A representative pattern performing multiple constructs of the same type (*Part-1/2*)

```

construct sAPD [singleAttributePhraseDeclaration_82]
  sAPD_nul [convertSAP1_to_SAPD sAP]
  %type 1 sAP
  [convertSAP2_to_SAPD sAP]

%type_2 sAP

construct new_mentioned_pattern [informationModelStatement_48]
  'Pattern ': pEV '( sAPD ' ) '.

replace [repeat informationModelStatement_48]
  mentioned_patterns [repeat informationModelStatement_48]
by
  mentioned_patterns [. new_mentioned_pattern]
end function

.....

function getEVP
  Statement [informationModelStatement_48]
  oDC [opt domainsClause_216]
  eII [entityIdentifierIndicator_669]
  eAPlist [entityAttributePhrase_125]

  deconstruct eAPlist
    pEV [primeEntityVariable_79]
    '(
      lsAP [list simpleAttributePhrase_129+]
    ')

  construct nullSAPs [repeat simpleAttributePhrase_129]
    % ( empty )

  construct sAPs [repeat simpleAttributePhrase_129]
    nullSAPs [^ eAPlist]

  %      [message '" atomic ident Att Spec "[print]

  replace [repeat informationModelStatement_48]
    mentioned_patterns [repeat informationModelStatement_48]
  by
    mentioned_patterns [buildIdentAtomicPattern pEV each sAPs]
end function

```

(b) A representative pattern performing multiple constructs of the same type (*Part-2/2*)

Figure 6.19: A representative pattern performing multiple constructs of the same type

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

Figure 6.20: Code to construct multiple variables of the same type using existing *construct* syntax

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

Figure 6.21: Code to construct multiple variables of the same type using the proposed *construct* syntax

Rule Proposal 5: Group a sequence of transformation rules/functions applied to some variable.

TXL programmers often apply a sequence of rules/functions to variables. The pattern in Figure 6.22 shows such an example. A new feature could be provided to group them using the keyword *transformation*. It will help to locate and rearrange transformation sequences quickly. Also, the name of a transformation would increase the readability of program code. Figure 6.24 shows the change due to grouping of transformation sequences of the code of Figure 6.23.


```

function MG_160_SET_id_TO_TRUE
  replace [LSCobol_program]
    P [LSCobol_program]
  by P
    [markUnstableLevel88DescriptionInLevel1]
    [markUnstableLevel88DescriptionInRecord]
    [convertSetConditionToTrue]
    [removeOptNameOfConditionalVariable]
end function

function MG_38_EXAMINE_Statement
  replace [LSCobol_program]
    P [LSCobol_program]
  by P
    [replaceExamineByInspect]
    [translateTallyingUntilFirstClause]
    [translateReplacingUntilFirstClause]
    [propagateTallyingClauseInInspect]
    [insertMoveZeroToTally]
    [deleteRedundantMoveZero]
    [insertTallyFor]
end function

function MG_39_EXHIBIT_Statement
  replace [LSCobol_program]
    P [LSCobol_program]
  by P
    [expand_EXHIBIT_NAMED]
    [replace_EXHIBIT_NAMED_Identifier_by_DISPLAY]
    [replace_EXHIBIT_NAMED_Literal_by_DISPLAY]
    [report_EXHIBIT]
end function

function ObsoleteFileHandling
  replace [LSCobol_program]
    Program [LSCobol_program]
  by
    Program
      [MG_34_ApplyCoreIndex]
      [MG_34_ApplyReorgCriteria]
      [MG_34_NominalKey]
      [MG_35_ApplyRecordOverflow]
end function

```

- (a) A representative pattern applying a sequence of rules/functions (*Part-1/2*)

```

function main
  replace [root]
    P [LSCobol_program]
  by P
    [preProcess]
    [message "LS370 1.7d (13 Sep 96) (c) 1996 Legasys Corporation"]
%    [LANGLVL_1_to_LANGLVL_2]
    [LANGLVL_2_to_CMPR2]
    [CMPR2_to_NOCMPR2]
    [NOCMPR2_to_COBOL370]
    [postProcess]
end function

.....

function MG_37_CURRENT_DATE_Special_Register
  replace [LSCobol_program]
    P [LSCobol_program]
  by P
    [expandMoveCurrentDate]
    [declareDateBlock]
    [replaceMoveCurrentDate]
end function

function MG_46_Relation_Conditions
  replace [LSCobol_program]
    P [LSCobol_program]
  by P
    [correct_EQUAL_TO]
    [correct_GreaterThanSign_THAN]
    [correct_LessThanSign_THAN]
end function

```

(b) A representative pattern applying a sequence of rules/functions (*Part-2/2*)

Figure 6.22: A representative pattern applying a sequence of rules/functions

```

rule main
  replace [expression]
    E [expression]
  construct NewE [expression]
    E [resolveAddition]    % Sequence of transformations on E
      [resolveSubtraction]
      [resolveMultiplication]
      [resolveDivision]
      [resolveBracketedExpressions]

  where not
    NewE [= E]
  by
    NewE
end rule

```

Figure 6.23: Code without grouping of rules/functions

```

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

rule main
  replace [expression]
    E [expression]
  construct NewE [expression]
    E [evaluateExpression] % Executes transformations in sequence
  where not
    NewE [= E]
  by
    NewE
end rule

```

Figure 6.24: Code showing the proposed grouping of rules/functions

```

function createOtherBuffer
  match [name]
    fileId [name]
  deconstruct not fileId
    'input
  deconstruct not fileId
    'output
  import TXLargs [repeat stringlit]
  construct NewArgs [repeat stringlit]
    TXLargs [fileNameMissingError fileId]
  deconstruct TXLargs
    fileName [stringlit] RestArgs [repeat stringlit]
  export TXLargs
    RestArgs
  construct WriteBuff [writeBuffer]
    _
  construct ReadBuff [readBuffer]
    ""
  construct Buffer [fileBuffer]
    fileId fileName WriteBuff ReadBuff 0
  import FileBuffers [repeat fileBuffer]
  export FileBuffers
    FileBuffers [. Buffer]
end function

.....

rule makeSpaces Width [integernumber]
  replace [integernumber]
    Length [integernumber]
  where
    Length [Width]
  construct String [stringlit]
    " "
  construct SingleSpace [id]
    _ [unquote String]
  import Spaces [charlit]
  export Spaces
    Spaces [quote SingleSpace]
  by
    Length [+ 1]
end rule

```

(a) A representative pattern showing the use of deconstruct not on the same variable (*Part-1/2*)

```

.....

function removeQuotes
  replace [operand]
    CLit [charlit]
  construct NewCLit [id]
    _ [unquote CLit]
  by
    NewCLit
end function

```

(b) A representative pattern showing the use of *deconstruct not* on the same variable (*Part-2/2*)

Figure 6.25: A representative pattern showing the use of *deconstruct not* on the same variable

Rule Proposal 6: *deconstruct not* can be modified to perform multiple *deconstructs* together.

The TXL feature *deconstruct* allows program execution to continue when a variable holds a given pattern. On the other hand, *deconstruct not* allows execution to proceed when a variable does not hold a given pattern. The existing syntax of *deconstruct not* allows the user to test a variable against only one pattern. However, programmers sometimes want to use *deconstruct* instruction with more than one given patterns, as shown by the pattern in Figure 6.25. The traditional way of doing this is to repeat the *deconstruct* instruction according to the number of tests needed. A change in *deconstruct* to check the value of a variable against multiple patterns would save programmers time and make the code shorter.

Figure 6.26 and Figure 6.27 show example code where we want to increase the value of a *[number]* type variable when it does not hold any of the two values 0 and 100. While Figure 6.26 shows the existing approach, Figure 6.27 shows the proposed approach. Change in the capability of *deconstruct* makes it possible to test a variable

against multiple patterns by a single instruction.

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

Figure 6.26: Code to test a variable against multiple patterns using existing *deconstruct not* syntax

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

Figure 6.27: Code to test a variable against multiple patterns using the proposed *deconstruct not* syntax

6.4 Complete TXL Program Analysis

As we have mentioned in Section 3.4, a complete TXL program is a source file containing a main rule or function with all necessary grammar specification and transformation rules of the program. We have analyzed such complete source files to find patterns involving both grammar specifications and transformation rules.

- **Complete TXL Program Analysis on Individual Projects:** Like grammar analysis and rule analysis on individual projects, we have performed complete program analysis on each individual project.
- **Complete TXL Program Analysis on Merged Collection:** Similar to the previous two analyses, we have performed complete program analysis on the merged collection to get patterns holding both grammar specification and transformation rules that appear across multiple projects.

6.4.1 Proposals

Using NiPAT, we find patterns from complete TXL programs. Like the previous two sections, we present representative code only from those patterns from which we can make proposals to improve the feature set of TXL. After analyzing all retrieved patterns, we make the proposals given below.

Grammar and Rule Proposal 1: Mark extracted patterns automatically with XML tags.

The pattern shown in Figure 6.28 combines both a grammar and a transformation rule. By analyzing the pattern, we find that users defined two common non-terminals, and used those in the rule *ConvertFunctionDefinitions* to convert the file name and source coordinates of the extracted patterns to XML format and annotate the extracted patterns. Programmers do this because, by default, TXL provides source file name and coordinates of the extracted patterns in plain text format. To reduce programming effort, TXL could provide a feature that would directly mark the extracted patterns with XML tags.

```

rule convertFunctionDefinitions
  replace [method_definition]
    FileName [srcfilename]
    LineNumber [srclinenumber]
    FunctionHeader [method_header]
    ' {
      FunctionBody [opt statement_list]
      EndFileName [srcfilename]
      EndLineNumber [srclinenumber]
    }
    Semi [opt ';']
  construct FileNameString [stringlit]
    _ [quote FileName]
  construct LineNumberString [stringlit]
    _ [quote LineNumber]
  construct EndLineNumberString [stringlit]
    _ [quote EndLineNumber]
  construct XmlHeader [xml_source_coordinate]
    <source file=FileNameString
      startline=LineNumberString
      endlime=EndLineNumberString>
  by
    XmlHeader
    FunctionHeader
    ' {
      FunctionBody
    }
    </source>
end rule

.....

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

.....

```

(a) A representative pattern marking extracted patterns with XML tags (*Part-1/2*)


```

function main
  replace [program]
    P [program]
  construct Functions [repeat method_definition]
    _ [^ P] [convertFunctionDefinitions]
  by
    Functions [removeOptSemis]
              [removeEmptyStatements]
end function

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

.....
define checked_expression
  ' checked
  ' (
    [expression]
  ')
end define

define default_value_expression
  ' default
  ' (
    [type]
  ')
end define

define sizeof_expression
  ' sizeof
  ' (
    [unmanaged_type]
  ')
end define

define unchecked_expression
  ' unchecked
  ' (
    [expression]
  ')
end define

```

(b) A representative pattern marking extracted patterns with XML tags (*Part-2/2*)

Figure 6.28: A representative pattern marking extracted patterns with XML tags

The code of Figure 6.29 shows example code using our proposed feature. We have used the notation $xml^{\wedge}$ to indicate that it marks the extracted patterns directly with XML tags. We have shown sample input and the expected output in Figure 6.30.

```
rule main
  replace [program]
    P [program]
  construct MyIf [repeat if_statement]
    _ [xml^ P]    % Extract and mark with XML tags
  by
    MyIf
end rule
```

Figure 6.29: Code using extract and mark with XML tags proposal

Sample input:

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

Sample output:

```
% File name and Line numbers are in XML format
< 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>
```

Figure 6.30: Sample execution using extract and mark with XML tag

Grammar and Rule Proposal 2: Mark extracted patterns automatically with XML tags skipping the embedded ones.

The pattern of Figure 6.31 is similar to the pattern presented in *Grammar and Rule Proposal 1*, but it holds another piece of code that creates scope for a different TXL feature. The interesting piece of code is a rule (*unmarkEmbeddedCompoundStatements*), which shows that when marking patterns with XML tags, programmers wanted to skip the embedded patterns. So, another new feature could be added to TXL to mark the extracted patterns with XML tags skipping the embedded ones and preventing them from being tagged. This is similar to the previous proposal but it proposes not to mark the embedded patterns during extraction.

We have given an example of such a feature in the code of Figure 6.32 using the notation $xml^{\wedge}$. Expected output in Figure 6.33 shows that the proposed feature does not mark the embedded patterns with XML tags, as opposed to the feature of the previous proposal.

```

rule convertCompoundStatements
  skipping [compound_statement]
  replace [compound_statement]
    FileName [srcfilename]
    LineNumber [srcclinenumber]
    ' {
      CompoundBody [compound_statement_body]
      EndFileName [srcfilename]
      EndLineNumber [srcclinenumber]
    }
    Semi [opt ';']
  construct FileNameString [stringlit]
    _ [quote FileName]
  construct LineNumberString [stringlit]
    _ [quote LineNumber]
  construct EndLineNumberString [stringlit]
    _ [quote EndLineNumber]
  construct XmlHeader [xml_source_coordinate]
    <source file=FileNameString
      startline=LineNumberString
      endline=EndLineNumberString>
  by
    XmlHeader
    {
      CompoundBody [unmarkEmbeddedCompoundStatements]
    }
    </source>
end rule

```

```

.....

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

```

```

.....

```

(a) A representative pattern marking the extracted patterns with XML tags and skipping the embedded ones (*Part-1/2*)

```

rule unmarkEmbeddedCompoundStatements
  replace [compound_statement]
    FileName [srcfilename]
    LineNumber [srclinenumber]
    ' {
      CompoundBody [compound_statement_body]
      EndFileName [srcfilename]
      EndLineNumber [srclinenumber]
    }
    Semi [opt ';]
  construct Empty [opt xml_source_coordinate]
  by
    Empty
    ' {
      CompoundBody
    }
end rule

.....
function main
  replace [program]
    P [program]
  construct Compounds [repeat compound_statement]
    - [^ P] [convertCompoundStatements]
  by
    Compounds [removeOptSemis]
               [removeEmptyStatements]
end function

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

.....
rule removeEmptyStatements
  replace [repeat declaration_or_statement]
    ';'
    More [repeat declaration_or_statement]
  by
    More
end rule

```

(b) A representative pattern marking the extracted patterns with XML tags and skipping the embedded ones (*Part-2/2*)

Figure 6.31: A representative pattern marking the extracted patterns with XML tags and skipping the embedded ones

```

rule main
  replace [program]
    P [program]
  construct MyIf [repeat if_statement]
    _ [xml^^ P] % xml^^ provides XML tags and avoid the embedded patterns
  by
    MyIf
end rule

```

Figure 6.32: Extract and mark patterns with XML tags avoiding the embedded patterns

```

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

Sample output:
% File name and Line numbers are in XML format
< source file=myfile startlinenumber=3 endlinenumber=6>
if x:
  y=4-x
  if y:      % Not marked with XML tags
    x=x*3
</source>
<source filename=myfile startlinenumber=5 endlinenumber=6 >
  if y:
    x=x*3
</source>

```

Figure 6.33: Sample execution using extract and mark patterns with XML tags avoiding the embedded ones

6.5 TXL Project Analysis Summary and Proposal Classification

As we mentioned previously in Section 6.1.1, it is difficult to clearly specify the exact number of grammar and rule files separately because there is no rigid rule in the language to put the grammar and transformation rules of a TXL program in separate

files with standard extensions. In Table 6.1, we show the number of grammar and rule files by limiting ourselves on the known standard extensions. Files having *.grm* and *.grammar* extensions in the collected projects have been counted as grammar files. Files with *.txl*, *.rul* and *.i* extensions have been considered as transformation rule files, ignoring the fact that they may also contain grammar definitions. The last row of the table, grammar and rules, are complete TXL program source files that were prepared by putting all grammar and transformation rules in a single source file after resolving all include instructions of files with a *main* rule or function, and the *.txl* extension. Lines have been counted using Cygwin's `wc -l <filename>` command [6].

Table 6.1: Summary of Analysis

	#Files	#Lines	#NiCAD Clone Classes	#Patterns	#Proposals
Grammar	92	41,769	402	86	3
Rules	278	67,723	573	95	6
Grammar and Rules	210	113,331	1,798	113	2

As can be seen from Table 6.1, the number of rule files is 278. On the other hand, the number of complete TXL program source files is 210. The reason behind this difference is that, in the case of only rule files, we count all rule files regardless of the presence of a main rule/function. But in the second case, a *main* rule/function must be present in the source file. So, rule files with no *main* rule/function were not counted in the latter case. The total number of grammar and rule instructions counted separately is very close to, but not exactly, the number of all instructions counted by putting all grammar and rule instructions of a program into a single source file. This is due to the non-standard file extensions that were used to include

instructions, as we did not recognize them as grammar or rule files when counting them separately.

We have made a total of eleven recommendations by analyzing 83 TXL projects. From grammar files NiCAD generated 402 clone classes and NiPAT detected 86 patterns. By analyzing the grammar patterns, we have made three proposals. Six proposals have been made from the rule patterns and two by analyzing patterns from complete TXL programs. The proposals we made for improving the TXL feature set can be categorized into the following three groups based on the needed modification to the language.

- **Proposals requiring TXL syntax/semantic modification:** To include the proposals of this group into TXL feature set, language developers would need to make changes to TXL syntax/semantics. The following proposals fall in this category:
 - Rule Proposal 1: Allow users to find unique elements from a TXL *repeat* type variable.
 - Rule Proposal 2: Allow users to find whether a *repeat* type variable is a subsequence of another variable of the same type.
 - Rule Proposal 3: Skip the embedded patterns during extraction.
 - Rule Proposal 4: *construct* can be modified to perform multiple constructs of the same type.
 - Rule Proposal 5: Group a sequence of transformation rules/functions applied to some variable.

- Rule Proposal 6: *deconstruct not* can be modified to perform multiple deconstructs together.
- **Proposals as library feature:** Proposals of this category is difficult or impossible to add to the language by modifying its syntax but can be added as library features.
 - Grammar Proposal 1: Predefined non-terminals can be provided to mark extracted patterns with XML source coordinates.
 - Grammar and Rule Proposal 1: Mark extracted patterns automatically with XML tags.
- **Trivial proposals:** Proposals of this category do not bear much significance compared to the previous two categories, but it would be good to have them in the language feature set. They might be added to the feature set by modifying the syntax/semantics, or as library feature.
 - Grammar Proposal 2: A keyword can be reserved to mention a non-terminal in its own definition.
 - Grammar Proposal 3: Selected non-terminals can be allowed to repeat at the start or end of a sequence of alternative definitions.
 - Grammar and Rule Proposal 2: Mark extracted patterns automatically with XML tags skipping the embedded ones.

6.6 Summary

In this chapter, we have presented the analysis we performed on the collected TXL projects to retrieve patterns. We have also presented proposals to improve the feature set of the TXL language after analyzing the retrieved patterns. In the next chapter, we draw conclusions by summarizing the thesis, limitations, and future research.

Chapter 7

Conclusion and Future Work

7.1 Summary

The thesis presented in this report describes a novel idea to find new features needed to improve the TXL, or another existing programming language, by analyzing source code. Identifying the language features that are truly needed by programmers is one of the most important steps of language feature enhancement. The traditional way of doing this is to analyze source code and to ask programmers about their experience using the language, which is time consuming and also has several other problems like memory recall, vagueness, etc. We have proposed a method that minimizes the amount of code that must be analyzed and eliminates the need to ask programmers about the tasks they need to perform frequently due to lack of language features. The idea presented here will certainly save language developers some of the effort needed to find the required features for a language. Furthermore, it will speed up the overall process of language feature enhancement.

Our proposed method uses a pattern detector tool that takes advantages of clone reports to find patterns from existing source code of a language. Language experts

then analyze those patterns to understand and propose new features for the language. The NiPAT pattern detector tool we have implemented to find patterns from TXL source code uses the clone class report generated by the NiCAD clone detector. We have developed the necessary plugins to enable NiCAD to perform clone analysis on TXL source code. By analyzing the report generated by our pattern detector, we have recommended new features for the TXL language.

7.2 Limitations

The accuracy of the pattern detection process presented in this thesis is related to the accuracy of the clone detector. If the clone detector fails to find or group the clones into classes appropriately, then the pattern detection process fails to retrieve patterns. Another limitation of this research is the manual analysis of the detected patterns. We have attempted to detect the patterns automatically, but we did not make any proposal to help analyze the patterns – it depends on language experts. NiPAT also has a limitation that it currently only works with NiCAD.

7.3 Future Work

The research presented here proposes a semi-automated system to feature recommendation, since language experts need to analyze the patterns in the last stage to understand their meaning. The system could be made more automatic in the future by automating the pattern analysis process further.

As another direction, NiPAT can be improved in the future to work with other clone detection tools, and analysis of the detected patterns can be done to separate the dissimilar portions of the cloned code of a pattern for easy analysis. This will

reduce the effort needed by experts to go through all of the cloned code of a detected pattern. In addition, the output generated by the pattern detector can be color coded for easy recognition of the syntactic structure of the language during analysis.

Bibliography

- [1] Antlersoft. Browse-by-Query. <http://browsebyquery.sourceforge.net>. Accessed: 2015-11-18.
- [2] Netbeans. Jackpot. <http://wiki.netbeans.org/Jackpot>. Accessed: 2015-11-18.
- [3] Python Software Foundation. Python. <https://www.python.org>. Accessed: 2015-01-12.
- [4] TXL. The TXL Website. <http://www.txl.ca>. Accessed: 2015-08-31.
- [5] Wikipedia. C Standard Library. https://en.wikipedia.org/wiki/C_standard_library. Accessed: 2015-11-18.
- [6] Wikipedia. Wc (Unix). https://en.wikipedia.org/wiki/Wc_%28Unix%29. Accessed: 2015-11-18.
- [7] James R Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
- [8] James R Cordy. Excerpts from the TXL cookbook. In *Generative and Transformational Techniques in Software Engineering III*, pages 27–91. Springer, 2011.

-
- [9] James R Cordy, Ian H Carmichael, and Russel Halliday. The TXL programming language-version 10. *Kingston: Queen's University at Kingston and Legasys Corporation*, 2000.
 - [10] James R Cordy, Charles D Halpern-Hamu, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.
 - [11] James R Cordy and Chanchal K Roy. The NICAD clone detector. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 219–220. IEEE, 2011.
 - [12] Chanchal K Roy and James R Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181. IEEE, 2008.
 - [13] Chanchal K Roy and James R Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 157–166. IEEE, 2009.
 - [14] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. Technical report, Technical Report 541, Queens University at Kingston, 2007.

-
- [15] Adrian D Thurston and James R Cordy. Evolving TXL. In *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on*, pages 117–126. IEEE, 2006.
 - [16] Raoul-Gabriel Urma and Alan Mycroft. Source-code queries with graph databases-with application to programming language usage and evolution. *Science of Computer Programming*, 97:127–134, 2015.
 - [17] Raoul-Gabriel Urma, Dominic Orchard, and Alan Mycroft. Programming language evolution workshop report. In *Proceedings of the 1st Workshop on Programming Language Evolution*, pages 1–3. ACM, 2014.