Natural Language Checking with Program Checking Tools

Fabrizio Perin Lukas Renggli Jorge Ressia

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

Abstract

Written text is an important component in the process of knowledge acquisition and communication. Poorly written text fails to deliver clear ideas to the reader no matter how revolutionary and ground-breaking these ideas are. Providing text with good writing style is essential to transfer ideas smoothly. While we have sophisticated tools to check for stylistic problems in program code, we do not apply the same techniques for written text. In this paper we present TextLint, a rule-based tool to check for common style errors in natural language. TextLint provides a structural model of written text and an extensible rule-based checking mechanism.

1 Introduction

In a typical programming language the parser and compiler validate the syntax of the program. IDEs often provide program checkers [1] that help us to detect problematic code. The goal of program checkers is to provide hints to developers on how to improve coding style and quality. Today's program checkers [2] reliably detect issues like possible bugs, portability issues, violations of coding conventions, duplicated, dead, or suboptimal code. While a program checker can assist the review process of source code, its suggestions are not necessarily applicable to all given contexts and might need further review of a senior developer.

Most today's text editors are equipped with spelling and grammar checkers. These checkers are capable of detecting a variety of errors in various languages as well as point out invalid grammatical constructs. Despite their sophistication, these tools do not consider common writing conventions and do not provide stylistic suggestions to improve the readability of text. As of today this task is still delegated to editors and reviewers which fulfill it by proof reading.

"To produce a text of good quality the main ideas have to be explained clearly, needless words omitted and statements should be concise, brief and bold instead of timid, vague or undecided." [William Strunk]

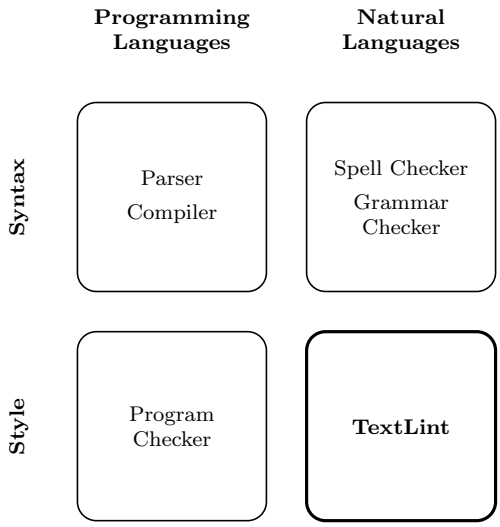

Fig. 1. TextLint as the analogy in natural languages to program checking of source code.

In this paper we take ideas from the domain of program checking and apply them to natural languages (Figure 1). We present TextLint, an automatic style checker following the architecture of SmallLint [3], a popular program checker in Smalltalk. TextLint implements various stylistic rules that we collected over the years, and that are described in *The Elements of Style* [4] and *On Writing Well* [5]. Similar to a program checker TextLint can reliably point out possible problems and suggest to rewrite certain parts of a document to increase its quality. However, as with a program checker, TextLint does not replace a manual reviewing processes, it only helps the writer to improve it.

"It is an old observation that the best writers sometimes disregard the rules of rhetoric. When they do so, however, the reader will usually find in the sentence some corresponding merit, attained at the cost of the violation. Unless he is certain of doing as well, he will probably do best to follow the rules." [William Strunk]

The implementation of TextLint uses Pharo Smalltalk [6] and various Smalltalk libraries: For parsing natural languages we use PetitParser [7], a flexible parsing framework that makes it easy to define parsers and to dynamically reuse, compose, transform and extend grammars. Furthermore, we use Glamour [8], an engine for scripting browsers. Glamour reifies the notion of a browser and defines the flow of data between different user interface widgets.

The contributions of this paper are:

(1) we apply ideas from program checking to the domain of natural language;
(2) we implement an object-oriented model used to represent natural text in Smalltalk;
(3) we demonstrate a pattern matcher for the detection of style issues in natural language; and
(4) we demonstrate a graphical user interface that presents and explains the problems detected by the tool.

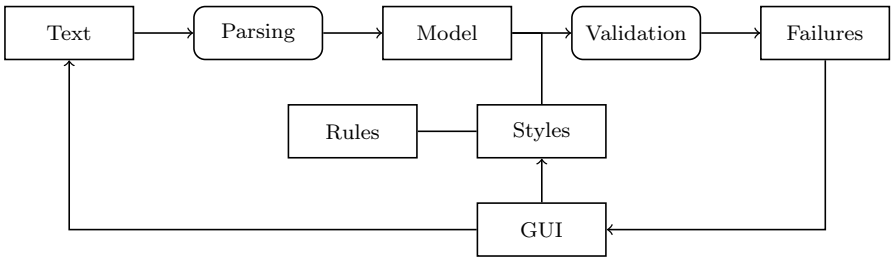

Fig. 2. Data Flow through TextLint.

Figure 2 gives an overview of the architecture of TextLint. Section 2 introduces the natural text model of TextLint and Section 3 details how text documents are parsed and the model is composed. Section 4 presents the rules which model the stylistic checks. Section 5 describes how stylistic rules are defined in TextLint. The implementation of the user interface is demonstrated in Section 6. We summarize related work in Section 8 and conclude and present future work in Section 9.

2 Modeling Text Documents

To perform analyses of written text it is necessary to have a model representing it. TextLint provides the abstractions for modeling written text from a structural point of view. The abstractions provided by our model are:

- The *Document* models a text document composed of paragraphs.
- The *Paragraph* models a sequence of sentences up to a break point. Paragraphs are responsible for answering the sentences and words that compose them.
- The *Sentence* is a set of syntactic elements or phrases ending with a sentence terminator.
- The *Phrase* models a set of syntactic elements of a particular length. A sentence provides access to all potential phrases of a specific size.
- The *Syntactic Elements* model the different tokens of a sentence, they are:
  - The *Word* models vocables or numbers in the text. A word is a sequence of alphanumeric characters.
  - The *Punctuation* models periods, commas, parentheses and other punctuation marks that are used in written text to separate paragraphs, sentences

and their elements.

- The *Whitespace* models blank areas between words and punctuations. Our model considers spaces, tabs and carriage returns as whitespace.
- The *Markup* models LaTeX or HTML commands depending on the filetype of the input.

All document elements answer the message text which returns a plain string representation of the modeled text entity ignoring markup tokens. Furthermore all elements know their source interval in the document. The relationship among the elements in the model are depicted in Figure 3.

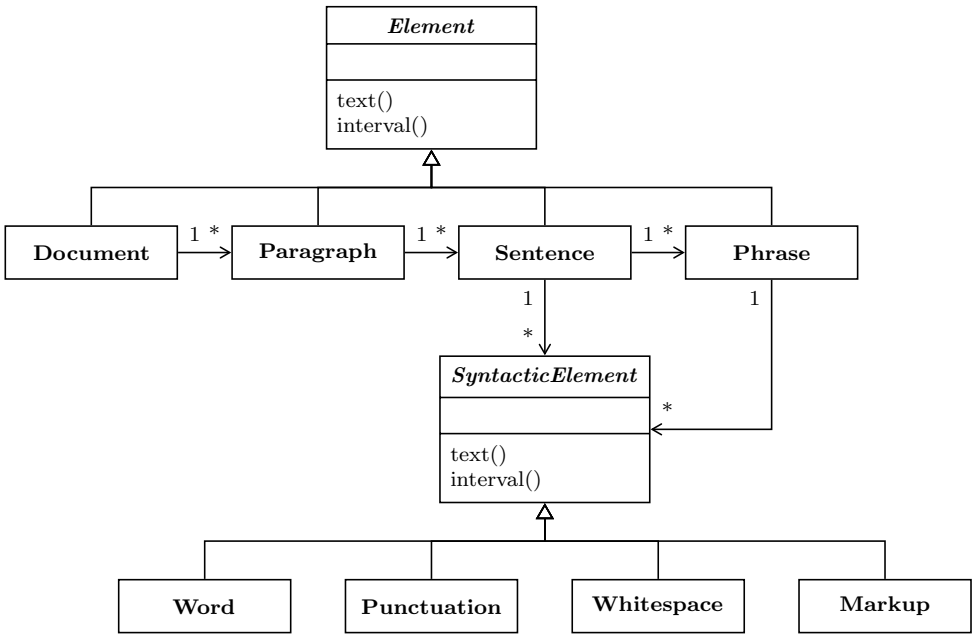

Fig. 3. The TextLint model and the relationships between its classes.

3 From Strings to Objects

To build the high-level document model from the flat input string we use *PetitParser* [7]. PetitParser is a framework targeted at parsing formal languages (*e.g.,* programming languages), but we employ it in this project to parse natural language input. This is technically difficult because there is no formal grammar for natural languages and the parser has to gracefully accept any input, even when the input does not follow basic rules of writing.

While PetitParser grammars are typically implemented without a separate scanner, in this case we perform the parsing in two separate steps. First, we split the input into markup, word, whitespace and punctuation tokens. Each of these syntactic elements knows its source position in the input file, so that we can map it back to the original position at a later time. Parsing of the

markup is implemented as a strategy for LaTeX, HTML and plain text. This is important to avoid unnecessary noise in the generated language model.

From this input stream of token objects we can build various high-level models of text. First we define predicates for tokens that terminate a document, paragraph or sentence. Then we define the grammar to build document, paragraph and sentence objects as shown below:

```
TextPhraser>>document
  ^ (paragraph starLazy: documentTerminator) , (documentTerminator optional)

TextPhraser>>paragraph
  ^ (sentence starLazy: paragraphTerminator / documentTerminator) ,
    (paragraphTerminator optional)

TextPhraser>>sentence
  ^ (#any asParser starLazy: sentenceTerminator / paragraphTerminator /
    documentTerminator) , (sentenceTerminator optional)
```

The grammar for our model looks more complicated than expected. This is because we need to parse and build a model for any input and scan over paragraphs and sentences even if they are not properly terminated. In future work we plan to use the same parsing infrastructure to build a link grammar model which provides a more sophisticated model for natural language [9].

4 Modeling Rules

A rule reifies an explicit regulation or principle that is accepted as a feature of a writing style. Rules are applied over documents and they analyze properties at different levels of the model. Rules report failures to comply with a certain style feature at document level, sentence level, at phrase level or at word level.

There are two types of rules:

- *Imperative rules* are implemented by calling the document model API. If an imperative rule detects a rule violation it manually instantiates a failure object that knows the failing rule and the context in the model.
- *Declarative rules* check for specific patterns in the document model. Patterns are specified using an internal domain-specific language on top of *PetitParser* [7]. The searching and reporting of violations happens automatically on the complete document model.

4.1 Imperative Rules

The entry point of an *imperative rule* is the method runOn: aDocument. For example, the 'avoid long sentence' is implemented like this. It returns a failure object for each sentence that has more than 40 words.

```
LongSentenceRule>>runOn: aDocument
  ^ aDocument sentences
    inject: OrderedCollection new
    into: [ :results :sentence |
      sentence words size > 40
        ifTrue: [ results add: (RuleFailure on: self in: sentence) ].
      results ]
```

Each imperative rule is implemented as subclass of the TextLintRule class. Rules are required to return additional meta-information such as the name and a rationale giving a description of the rule and how to fix the text.

4.2 Declarative Rules

Most TextLint rules are implemented declaratively. Declarative rules are subclasses of PatternRule and they override the method matchingPattern to return the pattern to be looked for. The class PatternRule provides a series of basic patterns that can be used to compose more complicated patterns:

- word matches any single word.
- word: matches a specific word given as argument.
- wordIn: matches any of the words given in the collection argument.
- wordSatisfying: matches any word that also satisfies the condition given in the block argument.

Similar rules exist for separators such as whitespace and markup tokens, and for punctuation. The returned matcher objects can be composed to more complicated matcher objects using the standard composition operators of PetitParser, such as ',' for sequence and '/' for choice.

For example, the rule 'avoid somehow' is implemented using the following pattern:

```
SomehowRule>>matchingPattern
  ^ (self word: 'somehow')
```

The more complicated rule 'avoid passive voice' is implemented like this:

```
PassiveVoiceRule>>matchingPattern
  ^ (self wordIn: self verbWords) , (self separator star) , ((self wordSatisfying: [ :value
    | value endsWith: 'ed' ]) / (self wordIn: self irregularWords))
```

This rule detects word sequences that start with a verb like 'am', 'are', 'were', ...; followed by zero or more separators; followed by a word ending in '-ed' or one of the irregular passive words like 'awoken', 'born', 'spoken', ...

Similar to the imperative rules declarative pattern have a name and a rationale.

Regular expressions [10] are a similar technique to match specific patterns in text. However, we found regular expressions unsuitable in our context because they are external to the domain of natural languages and are neither reusable nor composable. Furthermore, most text documents contain noise not related to the contents, such as LaTeX or HTML markup that cannot be easily handled with regular expressions while keeping the source location of matches.

5 Modeling Style

TextLint supports the definition of various writing styles. A writing style is a set of rules that we want to follow to fulfill our literary objective. We model the style as a composite of rules. Each distinctive style is defined as set of specific rules. Users can build their own styles or compose existing ones.

The following example demonstrates a composition of writing styles used to check this paper:

```
WritingStyle class>>computerSciencePaperStyle
  <style>

  ^ self correctSyntacticStyle + self unclutteredStyle + self boldStyle
```

The computer science style is a composition of three other styles. The operators '+' and '−' are used for adding and for removing styles. The method annotation <style> allows users to extend the system with new styles.

The 'correct syntactic style' checks for the usage of common syntactic errors like 'allow to', 'require to' and 'continuous word repetition' and 'regarded as being'. 'Uncluttered style' validates that unnecessary words are prevented from taking part in sentences qualifiers, and inadequate expressions like 'the truth is'. Finally, the bold style validates that no weakening expressions or words are used in the sentences. Some examples are 'the fact that', 'one of the most',

'avoid stuff', 'avoid thing' and 'avoid somehow'.

Primitive writing styles are built as a composition of rules. The following example shows the 'correct syntactic style' definition:

```
WritingStyle class>>correctSyntacticStyle
  <style>

  ^ WritingStyle
    named: 'Correct Syntactic Style'
    from: AllowToRule new + RequireToRule new + HelpToRule new
      + RegardedAsBeingRule new + WordRepetitionRule new
```

6 Scripting the User Interface

We developed a GUI to ease the issues presentation of TextLint. The application is divided into three main steps: First the user is asked to select a directory of files to check. Then he can choose the desired style to validate against. Finally, a browser is displayed allowing the user to browse through and fix the detected issues. Selected rule violations are highlighted in the text.

The user interface has been developed using Glamour [11], an engine for scripting browsers. Figure 4 shows how the TextLint browser is structured. There are two main parts split into a total of 4 interconnected panes. The arrows in Figure 4 represent the data flow among panels. The upper part of the browser is composed of three panes: the 'files pane' shows the list of the files contained in the folder chosen and in all its subfolders; the 'errors pane'

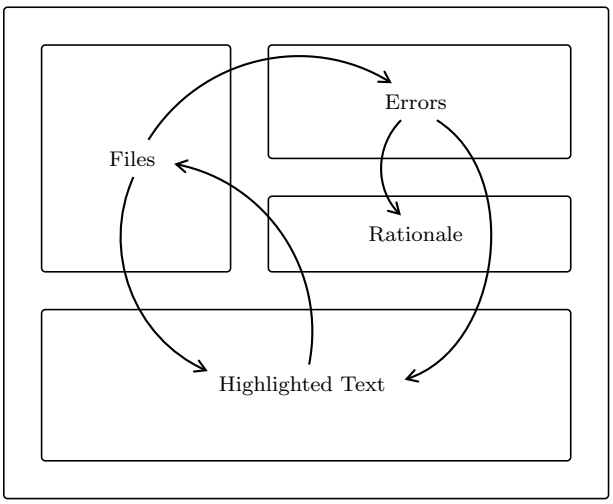

Fig. 4. The implementation of the TextLint window with Glamour. The figure schematically depicts the panes and the data flow between them.

contains a tree of TextLint rules failures; and the 'rationale pane' contains an explanatory text about the selected rule. In the bottom part, the 'text pane', the contents of the selected file is displayed here. The highlighting of issues in the 'text pane' assists the user working through the issues.

A description of how to get started is given on the website of TextLint: http://scg.unibe.ch/research/textlint. The website also provides a one-click distribution running on Microsoft Windows, Apple OS X and Linux.

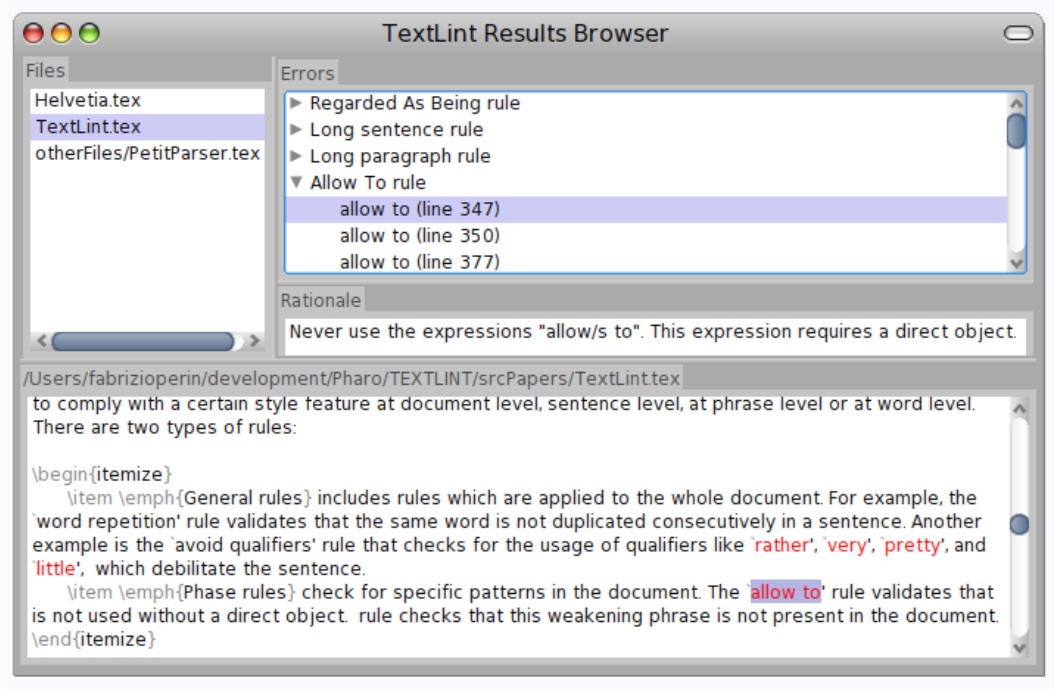

Fig. 5. TextLint browser opened on this paper.

Figure 5 shows an example of a possible interaction with the TextLint browser: The user has selected a file from the files pane triggering TextLint analysis and displaying the problems. In the 'errors pane' the issues are grouped by rule type making it easier to browse through the errors. The user then selects an issue from the tree which displays the rationale for this error and highlights the problem in the 'text pane'. The displayed text is editable, so modifying it and accepting changes saves the file and reruns the rules. In the 'text pane' the elements are highlighted in three different colors: in black is the text that has been analyzed, in red are the issues found, and in grey are the ignored parts such as LaTeX and HTML markup.

7 Validation

In this section we present two empirical studies on the use TextLint.

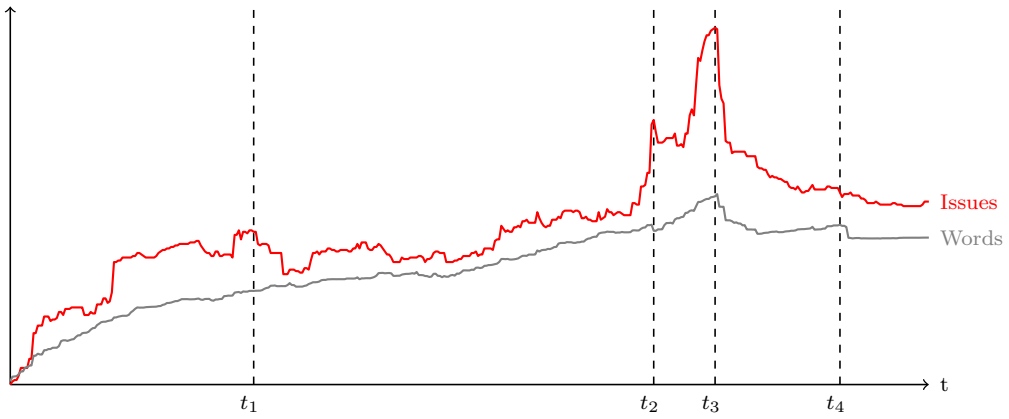

Fig. 6. Evolution of a paper from beginning to publication.

7.1 History of a Paper

Figure 6 depicts the number of stylistic issues detected by TextLint and the number of words in the text. The dashed vertical lines mark interesting moments in the life-time of the document from the beginning to publication.

Up to point t_1 we can see the early life of the paper. A significant amount of text was added and the number of TextLint issues steadily increased over time.

This growth decreased between point t_1 and t_2. We can observe that even though some new text is being added the TextLint issues do not increase as much as in the previous part. In this period the authors proof-read and rewrote portions of the paper to accommodate the ideas and to make the paper cohesive to a single story.

Points t_2 and t_3 mark the moments when a native english speaker with experience in paper writing for over 30 years proof-read the document. We can observe in both cases that the number of errors was systematically reduced after each of the interventions. The issues detected did not disappear immediately because the expert author often introduced annotations that were later fixed by the co-authors.

The peak at t_3 marks the time before the paper submission. With the approaching deadline the authors added a lot of new issues. The time period between t_3 and t_4 depicts the correction of most issues and the final preparations of the paper for submission. Later the paper was accepted for publication.

Point t_4 marks a slight increase in text size due to the introduction of passages addressing the reviewers comments. Afterwards, there is an abrupt size reduction due to the elimination of comments and unnecessary text for the camera-ready version.

By comparing the constant growth in size with the heterogeneous change in the number of errors detected by TextLint we can conclude that the quality of the introduced text is more relevant than the amount of text. We can observe that when text is added to the document sometimes the number of errors decreases and sometimes increases. The number of errors depends much more on the stylistic quality of the text introduced than the amount of text introduced.

7.2 Effectiveness of TextLint

To validate the effectiveness of our rules we compared the average number of issues over the complete history of several papers with the number of issues when the final version was submitted for publication. We ran this analysis on 20 papers of our research group that got accepted at international conferences in the past years. As the size of the individual papers significantly varies we normalized all data by dividing through the respective file size.

Figure 7 lists the TextLint rules from the least to the most effective one. This list is not complete since new rules are being added to the tool on regular basis. We see that a few rules do not perform well. For example, the rule *Avoid long paragraph* has 20% more occurrences for the finally published version of the paper than during the writing of the paper. This can have various reasons: either the rule is not well defined, the copy editors do not consider the rule as relevant, or the paper was in a perfect shape from the beginning.

On the other hand many rules in our case-study perform well: For example, over 73% of the rule *Avoid contraction* disappear from the final version of the papers. If TextLint had been used from the beginning of the paper writing, the quality of the text would have been better from the beginning.

Some authors do not always follow all rules, this is the case in the *Avoid currently* rule. Most authors of the papers that we analyzed did not consider this rule as an important style violation.

8 Related Work

There is a wide variety of (commercial) libraries for natural language processing. Most of these libraries do not provide the necessary reusable abstractions to analyze stylistic concerns in text.

Natural Language processing (NLP) is a field of computer science and linguistics concerned with the interactions between computers and human (natural)

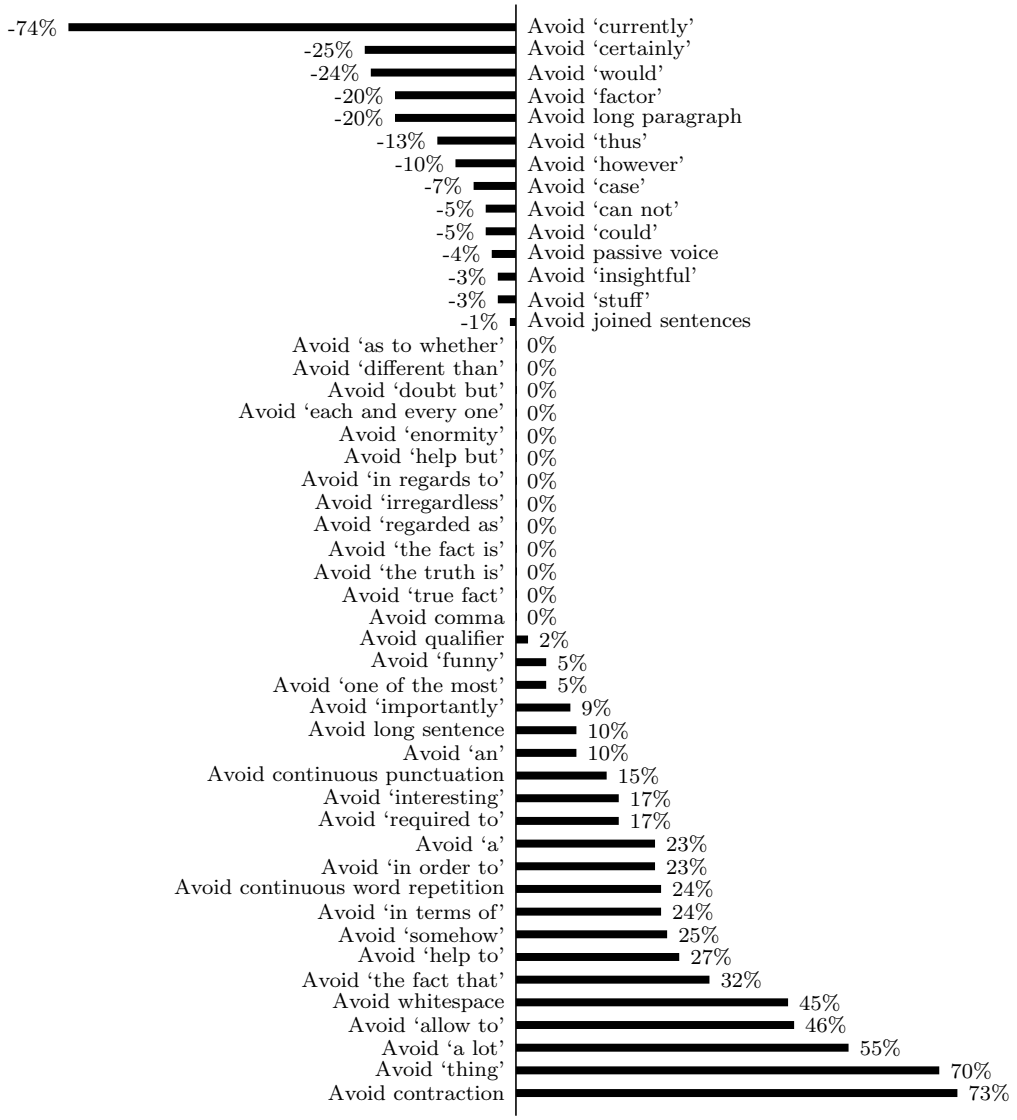

Fig. 7. Effectiveness of various TextLint rules.

languages. NLP is concerned with the natural language generation and understanding. Natural language generation is the process that converts information from a computational representation to readable human language. Natural language understanding works by converting samples of natural language into more formal forms understandable by computer systems. Bates [12] summarizes the NLP problems and state-of-art solutions in detail.

Bird *et al.* present the *Natural Language Toolkit* (NLTK) [13,14] implemented in Python. This tool follows the nomenclature of NLP. It reifies a corpus of text as a large number of sentences. NLTK model is oriented towards parsing, it provides abstraction for tokens, parse trees, tokenization, words and sentences. A sentence is an ordered sequence of token. As other approaches, NLTK does not provide abstractions for other components of written text, nor does it

provide a notion of style or an automatic validation mechanism.

Oda developed *NaturalSmalltalk* [15], a toolkit for analyzing source code and natural languages. NaturalSmalltalk understands Smalltalk code as a series of English words. From a modeling point of view, NaturalSmalltalk only reifies words as a structural unit. There is no other abstraction of written text besides words. It does not deal with style and it was mainly thought to be applied to Smalltalk. When loaded, NaturalSmalltalk creates the corpus by analyzing the Smalltalk image, processing all source code and interpreting it as English language text.

Slator and Temperley propose *link grammars* [9]. A link grammar consists of a set of words each of which has linking requirements. The link requirement specifies conditions that if satisfied allow a word to be connected to other words. The underlying model reifies verbs, nouns, adjectives, *etc.* and defines different linking restriction for each word using huge dictionaries. The link grammar model was not conceived for validating style but as a way of checking the grammatical structure of sentences.

Klein and Manning [16] presented a novel *generative model* for natural language which uses a different model for representing the data. Syntactic structures (PCFG) and lexical dependencies structure are kept separated to accomplish conceptual simplicity and a good level of performance. This approach follows the intuition that several models of written text are required to detect various stylistic problems.

9 Conclusion and Future Work

We have presented TextLint, an automatic style validation tool for written text. TextLint reifies the different elements of written text as an extensible object-oriented model. A specific style is modeled as a set of rules that validate written text. Our contributions are:

(1) We provide a model which reifies structurally written text.
(2) We have presented a novel rule-based system for checking written text following the principles of program checkers such as Lint. By modeling style and stylistic rules as first class objects we have accomplished an extensible text validation system.
(3) We have successfully applied PetitParser in natural language parsing.
(4) We have demonstrated a matching mechanism to specify and detect specific phrases in written text.
(5) We have proposed a user interface for conveniently browsing and fixing style issues in text.

As future work we imagine the following improvements:

- We would like to improve the collection of rules and styles for different domains, *i.e.,* business, criticism, humor, *etc.*
- We also plan to add TextLint specific annotations that cause certain rules to be ignored in the marked context.
- We plan on exploring the introduction of different points of view over the same written text. We imaging employing link grammars to provide an even higher-level view onto the same text. This would allow us to implement rules like 'no more than two chained adjectives', or 'be careful with the creation of adverbs out of adjectives'.
- We intend to further simplify the definition of rules. Helvetia [17] is a language workbench for defining embedded languages and could provide the necessary infrastructure to define patterns even more naturally.
- We aim at improving the GUI to work as a full text editor. The initial purpose of the TextLint browser was to have an instrument to ease the analysis of different files and to inspect and fix the result of the analysis.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010) and of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" (Project no. 2234, Oct. 2007 – Sept. 2010).

References

[1] S. Johnson, Lint, a C program checker, in: UNIX programmer's manual, AT&T Bell Laboratories, 1978, pp. 78–1273.

[2] D. Hovemeyer, W. Pugh, Finding bugs is easy, ACM SIGPLAN Notices 39 (12) (2004) 92–106.

[3] D. Roberts, J. Brant, R. E. Johnson, A refactoring tool for Smalltalk, Theory and Practice of Object Systems (TAPOS) 3 (4) (1997) 253–263.

[4] W. S. Jr., E. White, The Elements of Style, 4th Edition, Allyn and Bacon, 2000.

[5] W. Zinsser, On Writing Well: The Classic Guide to Writing Nonfiction, anniversary. Edition, B&T, 2006.

[6] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.
URL http://pharobyexample.org

[7] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical dynamic grammars for dynamic languages, in: 4th Workshop on Dynamic Languages and Applications (DYLA 2010), Malaga, Spain, 2010.
URL http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf

[8] P. Bunge, Scripting browsers with Glamour, Master's thesis, University of Bern (Apr. 2009).
URL http://scg.unibe.ch/archive/masters/Bung09a.pdf

[9] D. T. Daniel D. K. Sleator, Parsing english with a link grammar, in: 3rd International Workshop on Parsing Technologies, 1991.

[10] R. I. Bull, A. Trevors, A. J. Maltopn, M. W. Godfrey, Semantic grep: Regular expressions + relational abstraction, in: Proceedings Ninth Working Conference on Reverse Engineering (WCRE'02), IEEE Computer Society, 2002, pp. 267–276.

[11] P. Bunge, T. Gîrba, L. Renggli, J. Ressia, D. Röthlisberger, Scripting browsers with Glamour, European Smalltalk User Group 2009 Technology Innovation Awards, glamour was awarded the 3rd prize (Aug. 2009).
URL http://scg.unibe.ch/archive/reports/Bung09bGlamour.pdf

[12] M. Bates, Models of natural language understanding, Voice communication between humans and machines - National Academy of Sciences (1994) 238–253.

[13] S. Bird, Nltk-lite: Efficient scripting for natural language processing, in: In Proc. of the 4th International Conference on Natural Language Processing (ICON, Publishers, 2005, pp. 11–18.

[14] S. Bird, E. Klein, E. Loper, Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit, O'Reilly, Beijing, 2009.
URL http://www.nltk.org/book

[15] T. Oda, NaturalSmalltalk (Dec. 2006).
URL http://map.squeak.org/package/624ed871-4e89-4343-8652-af38a873d0b4/

[16] D. Klein, C. D. Manning, Fast exact inference with a factored model for natural language parsing, in: In Advances in Neural Information Processing Systems 15 (NIPS, MIT Press, 2003, pp. 3–10.

[17] L. Renggli, T. Gîrba, O. Nierstrasz, Embedding languages without breaking tools, in: T. D'Hondt (Ed.), Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10), Vol. 6183 of LNCS, Springer-Verlag, 2010, pp. 380–404.
URL http://scg.unibe.ch/archive/papers/Reng10aEmbeddingLanguages.pdf