Linguistic Style 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 of 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 pointing 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 who 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]

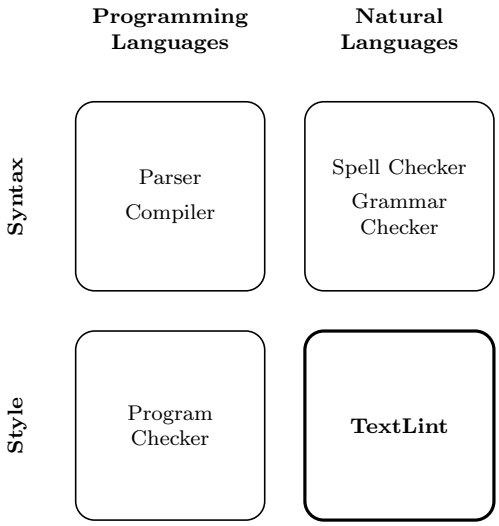

Fig. 1. TextLint as the analogy in natural languages to program checking of source code.

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 [1] 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 one to rewrite certain parts of a document to improve 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 open-source 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. To classify words into types and to detect synonyms we use an open-source dictionary and thesaurus. For the user interface we use Glamour [8], an engine for scripting browsers. For

[1] An online version of TextLint can be tested at http://textlint.lukas-renggli.ch/.

the web interface we use Seaside [9], a framework for developing dynamic web applications. TextLint has been integrated with Emacs and TextMate [2].

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 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 two graphical user interfaces that present and explain the problems detected by the tool.

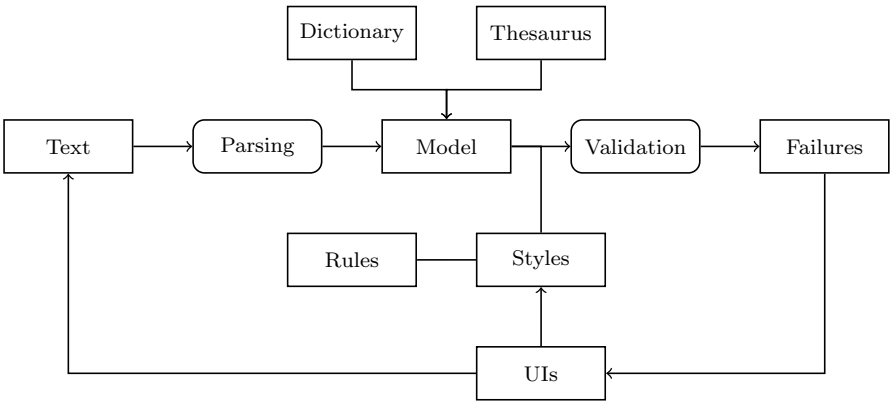

Fig. 2. Data Flow through TextLint.

The remainder of this paper follows the architecture of TextLint as depicted in Figure 2: 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 stylistic checks and explains how natural language features are integrated in TextLint. Section 5 describes how stylistic rules are defined in TextLint. The implementation of the user interfaces is demonstrated in Section 6. Section 7 validates the accuracy of TextLint rules. 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:

- A *Document* models a text document composed of paragraphs.

[2] Follow the installation instructions at https://github.com/DamienCassou/textlint.

- A *Paragraph* models a sequence of sentences up to a break point. Paragraphs are responsible for answering the sentences and words that compose them.
- A *Sentence* is a set of syntactic elements or phrases ending with a sentence terminator.
- A *Phrase* models a sequence of syntactic elements that potentially crosses the boundaries of a sentence or paragraph.
- *Syntactic Elements* model the different tokens of a sentence, they are:
  - A *Word* models vocables or numbers in the text. A word is a sequence of alphanumeric characters.
  - *Punctuation* models periods, commas, parentheses and other punctuation marks that are used in written text to separate paragraphs, sentences and their elements.
  - *Whitespace* models blank areas between words and punctuations. Our model considers spaces, tabs and carriage returns as whitespace.
  - A *Markup* models LaTeX or HTML commands depending on the file type 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 relationships among the elements in the model are depicted in Figure 3.

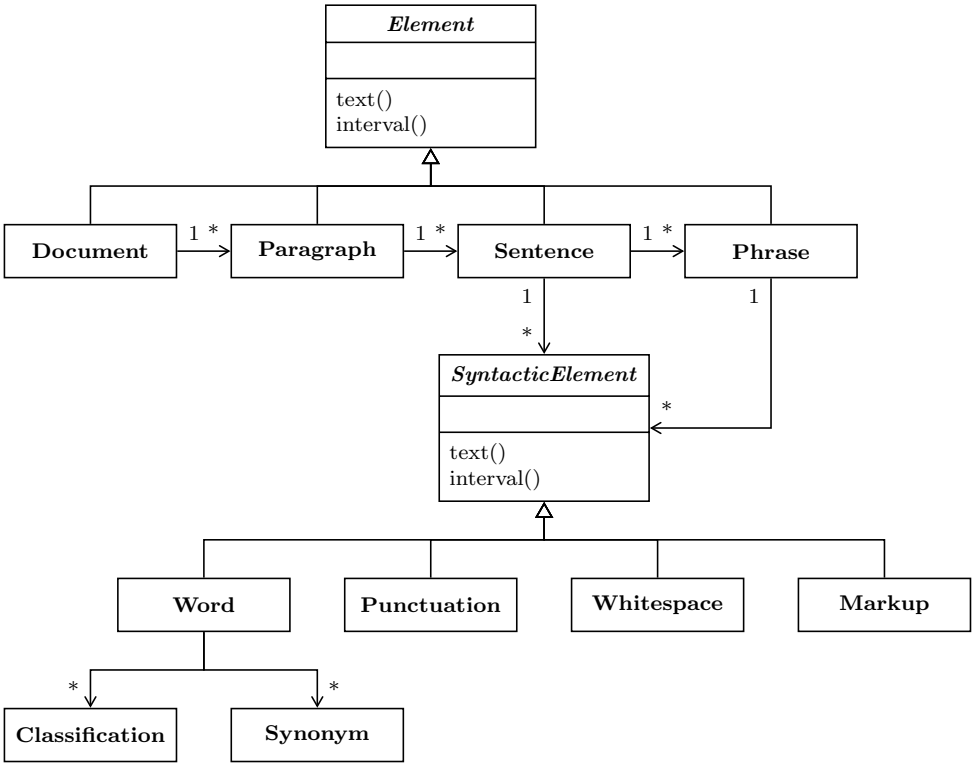

Fig. 3. The TextLint model and the relationships between its classes.

Each word in our model is classified using 264 028 words from Wiktionary [3], an open-source dictionary of the English language. We use Wiktionary to categorize all words into parts of speech (noun, pronoun, adjective, verb, adverb, preposition, conjunction and interjection) and other useful categories (abbreviation, acronym, article, number, prefix, suffix, *etc.*).

Depending on its position in a sentence the same word can belong to multiple categories. For example, the word 'welcome' is categorized by our system as an adjective, an interjection, a noun and a verb. Reducing the number of possible classifications is difficult and requires the grammatical structure of the sentence to be analyzed. The TextLint style rules do not solely depend on these classifications and thus the false-positives do not reduce the quality of our rules. Additionally, 91.9% of the words are classified to exactly one category, 7.5% to exactly two categories, and less than 1% to three or more categories.

Furthermore, for each word in our model we detect possible synonyms using the public domain thesaurus available in the Project Gutenberg [10]. Our English thesaurus contains 25 953 words with 1 816 509 synonyms in total.

3 From Strings to Objects

To build the high-level document model from a flat input string we use PetitParser [7]. PetitParser is a framework targeted at parsing formal languages (*e.g.,* programming languages), but we employ it here 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 text at a later time. Scanning of the markup is implemented using the strategy design pattern [11] to detect the language specific tokens in LaTeXand HTML.

From this input stream of token objects we can build a high-level model of the 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:

[3] http://en.wiktionary.org/

```
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 defined by 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 an even better model for natural language [12].

4 Modeling Rules

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

Regular expressions [13] are an alternative mechanism 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.

We support 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 can be specified in two ways using an *internal* or *external domain-specific language* on top of PetitParser. In both cases the searching and reporting of violations happens automatically on the complete document model.

Rules are required to return additional meta-information such as its name and a rationale giving a description of the rule and how to fix the text.

4.1 Imperative Rules

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

```
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. Depending on the level of granularity that a rule requires there are other methods which can be overridden to specify the rule behavior.

```
TextLintRule>>runOnParagraph: aParagraph
TextLintRule>>runOnSentence: aSentence
TextLintRule>>runOnWord: aWord
```

If a rule should only be applied to a paragraph scope then it is simpler to use the method runOnParagraph: aParagraph as an entry point.

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 word ‘that’ should never be preceded by a comma. In German this is mandatory but not in English.

```
NoCommaBeforeThatRule>>matchingPattern
  ^ (self word) , (self separator star) , (self punctuation: ',') , (self separator star) , (
    self word: 'that')
```

This rule detects sequences of words followed by zero or more separators, then a comma, zero or more separators, and the word ‘that’. The requirement to define separators makes a rule definition more complicated, but it gives us the full flexibility to reason about all parts of the input document. An example where separators matter is the following rule, which triggers when one or more accidental separators occurs in front of a punctuation mark:

```
NoSeparatorsBetweenWordAndPunctuationMarkRule>>matchingPattern
  ^ (self word) , (self separator plus) , (self punctuationIn: #(',' '.' ':' ';' '?' '!'))
```

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

```
PassiveVoiceRule>>matchingPattern
  ^ (self wordIn: self verbWords) , (self separator star) , ((self wordSatisfying: [ :word
    | word text 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’, . . .

Some stylistic rules validate a certain semantic structure rather than a syntactic one. For example, the concatenation of two or more adjectives makes text more difficult to read. This form of clutter is frequent in written text and often misused by authors to add artificial importance to certain noun. A better solution is to use a single adjective encompassing the meaning of both adjectives.

Using our internal matching language we can define the described rule as follows:

```
{adjective} {adjective}
```

The resulting matcher is very similar to the ones we have defined above. Albeit less powerful than the previous rule definitions, the simple query language allows users without a background in programming languages to define new rules to detect stylistic issues concisely:

- A word matches any case-insensitive occurrence of that word.
- A punctuation character matches any occurrence of that punctuation character.
- A whitespace matches a possibly empty sequence of whitespace or markup characters.
- A word enclosed in curly braces matches any word of the specified type: abbreviation, adjective, adverb, article, conjunction, contraction, interjection, name, noun, number, participle, particle, preposition, pronoun, or verb.

For example, to find lists of nouns or adjectives that are not separated by a comma we can use the following query string. Each line describes an alternative pattern:

```
{adjective} {adjective} and {adjective}
{adjective} {adjective} or {adjective}
{noun} {noun} and {noun}
{noun} {noun} or {noun}
```

TextLint's natural language facilities are not limited to classifying words. TextLint also provides the synonyms of a given word. The thesaurus is useful for detecting adverbs that are implied by the following word, as described by William Zinsser [5]. For example, 'effortlessly easy' is an overstatement, since if something is easy it is by definition effortless. The same logic applies to phrases like 'extremely loud', 'slightly spartan' and 'slightly small'. We can detect such word combinations with the rule:

```
AdverbSynonymRule>>matchingPattern
  | firstWord |
  ^ (self wordSatisfying: [ :word | firstWord := word. word classifiesAs: #adverb ]) ,
    (self separator star) ,
    (self wordSatisfying: [ :word | firstWord synonyms includes: word ])
```

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 to add and remove styles. The method annotation <style> allows users to extend the system with new styles.

The 'correct syntactic style' checks for the usage of common grammatical errors like 'allow to', 'require to', 'continuous word repetition' and 'regarded as being'. 'Uncluttered style' validates that unnecessary words do not take part in sentence qualifiers. 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 two GUIs to present issues detected by TextLint. The first interface was developed using Glamour [14] to provide a fat-client implementation with an integrated editor, and the second was built with Seaside [9] to provide an online version of TextLint.

6.1 Glamour

The main user interface of TextLint has been developed using Glamour [14], an engine for scripting browsers. Figure 4 shows how the TextLint browser is

modeled in Glamour. The arrows in Figure 4 represent the data flow among the panels. The upper part of the browser is composed of four panes: the 'files pane' shows the list of the files contained in the folder chosen and in all its subfolders, the 'issues pane' contains a tree of failures of TextLint rules, the 'rationale pane' contains an explanatory text about the selected rule, and the 'export pane' contains a button that allows the user to export the computed list of issues in plain text. In the bottom part of the browser there is one pane, called 'text pane', that displays the contents of the selected file. The highlighting of issues in the 'text pane' assists the user in working through the issues.

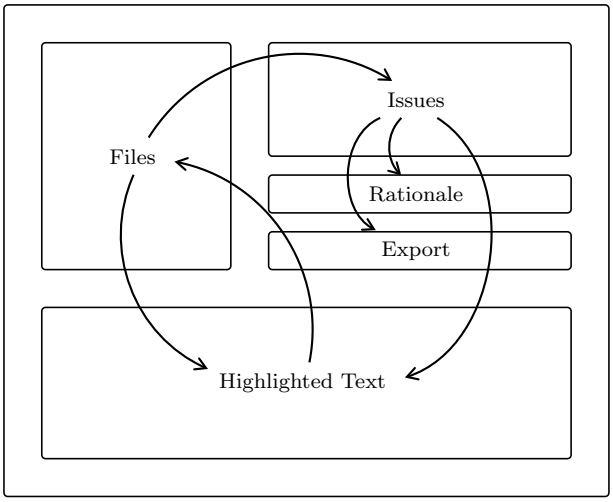

Fig. 4. The implementation of the TextLint window with Glamour. The figure schematically depicts the panes and the data flow between them.

The workflow to use the application is divided into two main steps: First the user is asked to select a directory of files to check. Then a browser is presented allowing the user to select files and to fix the detected issues. Selected rule violations are highlighted in the text and can be directly fixed from within the tool. The tool can be downloaded from scg.unibe.ch/research/textlint.

Figure 5 shows a running instance of the TextLint browser: The user has selected a file from the files pane triggering TextLint analysis and displaying the problems. In the 'issues pane' the issues are grouped by rule type. 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.

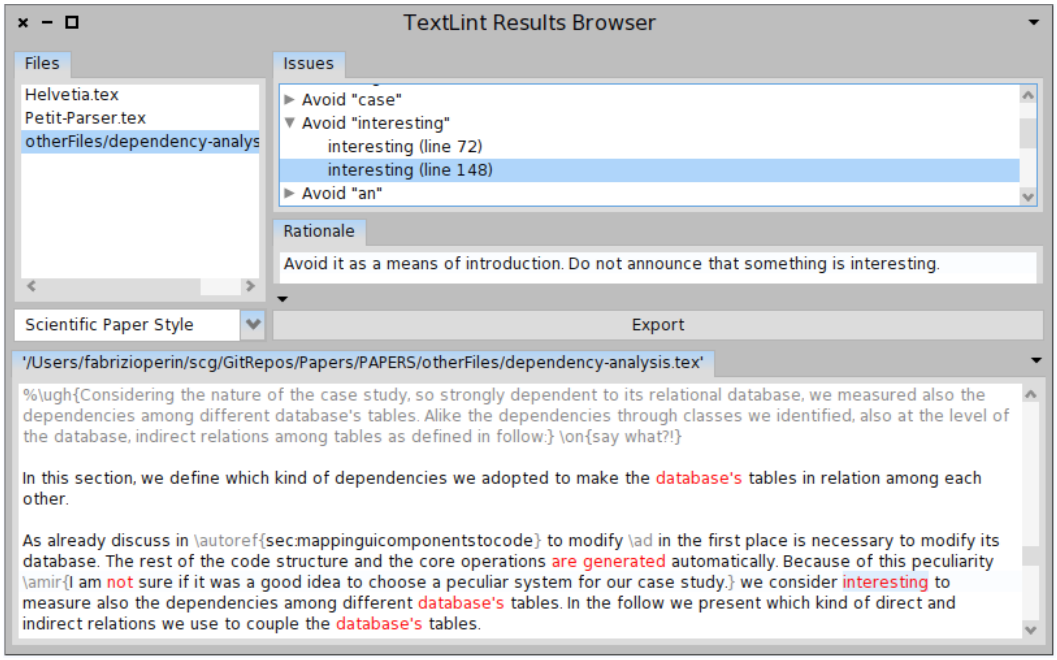

Fig. 5. Screenshot of the TextLint browser open on a scientific paper.

6.2 Seaside

The web application of TextLint at textlint.lukas-renggli.ch provides a quick and easy way for authors to check their text style, and play with the pattern matching infrastructure of TextLint. The web interface is implemented using Seaside [9], a framework to develop dynamic web applications.

The web application presents the user with a large empty text pane. The user needs to paste the text representation of the document to be analyzed and click the button. After the text has been analyzed two vertically placed panes are displayed as depicted in Figure 6. The upper pane holds a list with the detected issues with their respective rationale. The lower pane displays the analyzed text with the detected issues highlighted. The rationales of issues can also be seen by hovering the mouse over the highlighted text.

7 Validation

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

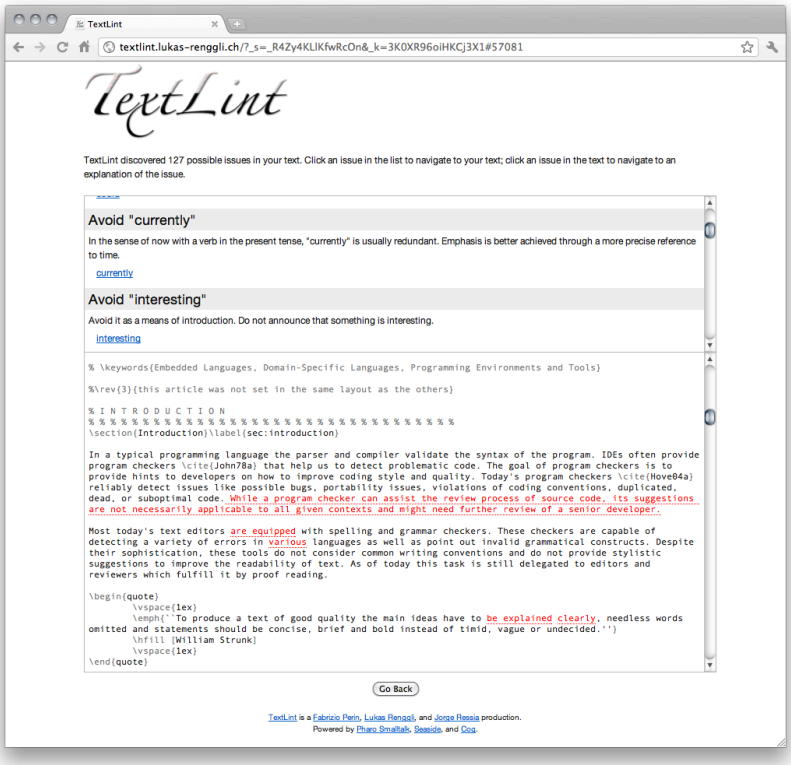

Fig. 6. The Seaside web interface for TextLint.

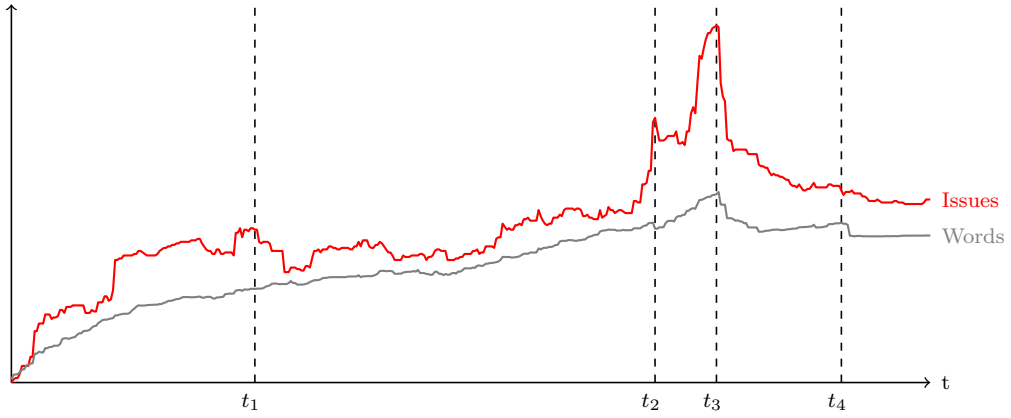

Fig. 7. Evolution of a paper from beginning to publication.

7.1 History of a Paper

Figure 7 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 story of the paper more cohesive.

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 to highlight issues 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 many 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 it 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 recent years. As the size of the individual papers significantly varies we normalized all data by dividing by the respective file size.

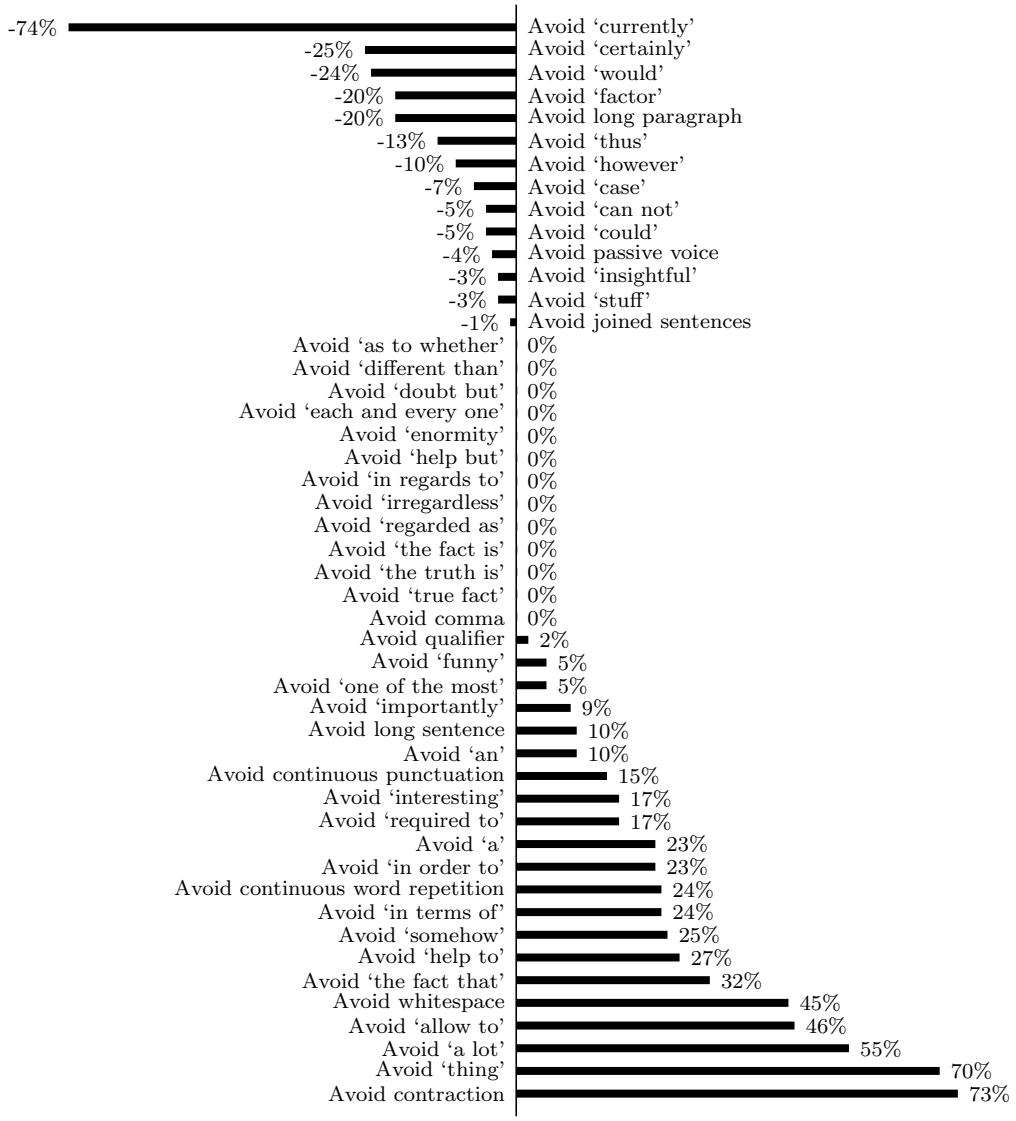

Fig. 8. Effectiveness of various TextLint rules.

Figure 8 lists the TextLint rules from the least to the most effective ones. 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 violations 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, which 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

A wide variety of (commercial) libraries for natural language processing exists. 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) 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 a representation understandable by computer systems. Bates [15] summarizes the NLP problems and state-of-art solutions in detail.

Bird *et al.* present the *Natural Language Toolkit* (NLTK) [16,17] 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 abstractions 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* [18], 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 designed 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* [12]. 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 restrictions 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 [19] 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.

LanguageTool [20] is an Open Source style and grammar checker for English, French, German, Polish, Dutch, Romanian, and other languages. Rules are defined with xml files. Complex rules are defined in Java. One key drawback of this tool is that there is no style abstraction, thus different writing styles cannot be modeled. A big set of rules is present for each language, however some rules are only syntactic checks. For example, a rule checks for the phrase 'can be build' which should be fixed by using the word 'built'. There are also rules checking for 'some king' instead of 'kind', and 'I has' instead of have.

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 presented a light-weight approach for natural language analysis.
(6) We have proposed two user interfaces 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 plan to add TextLint-specific annotations that cause certain rules to be ignored in the marked context.
- Other languages than English have different rules for written style. We plan to start introducing support for other languages.

- 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 that exploit semantic information. Words in specific parts of a document will have a reduced set of potential classifications thus enhancing the accuracy of TextLint rules.
- We intend to further simplify the definition of rules. Helvetia [21] is a language workbench for defining embedded languages and can provide the necessary infrastructure to define patterns even more naturally.
- We aim to better integrate TextLint into commonly used text editors. Currently plugins for Emacs and TextMate exist; supporting other editors will extend the reach of TextLint and ease the analysis of different file types.

Acknowledgments. We thank Oscar Nierstrasz for his feedback on early drafts of this paper. 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). We also gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 – Sept. 2012).

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] S. Ducasse, L. Renggli, C. D. Shaffer, R. Zaccone, M. Davies, Dynamic Web Development with Seaside, Square Bracket Associates, 2010, http://book.seaside.st/book.
URL http://book.seaside.st/book

[10] G. Ward, Moby Thesaurus List: Words and phrase lists – English, Project Gutenberg, 2002.
URL http://www.gutenberg.org/ebooks/3202

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional, Reading, Mass., 1995.

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

[13] 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.

[14] 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

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

[16] 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.

[17] 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

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

[19] 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.

[20] D. Naber, A rule-based style and grammar checker, Master's thesis, University of Bielefeld (2003).
URL http://danielnaber.de/languagetool/download/style_and_grammar_checker.pdf

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