Explanations for Regular Expressions

Martin Erwig and Rahul Gopinath

School of EECS
Oregon State University

Abstract. Regular expressions are widely used, but they are inherently hard to understand and (re)use, which is primarily due to the lack of abstraction mechanisms that causes regular expressions to grow large very quickly. The problems with understandability and usability are further compounded by the viscosity, redundancy, and terseness of the notation. As a consequence, many different regular expressions for the same problem are floating around, many of them erroneous, making it quite difficult to find and use the right regular expression for a particular problem. Due to the ubiquitous use of regular expressions, the lack of understandability and usability becomes a serious software engineering problem.
In this paper we present a range of independent, complementary representations that can serve as explanations of regular expressions. We provide methods to compute those representations, and we describe how these methods and the constructed explanations can be employed in a variety of usage scenarios. In addition to aiding understanding, some of the representations can also help identify faults in regular expressions. Our evaluation shows that our methods are widely applicable and can thus have a significant impact in improving the practice of software engineering.

1 Introduction

Regular expressions offer a limited but powerful metalanguage to describe all kinds of formats, protocols, and other small textual languages. Regular expressions arose in the context of formal language theory, and a primary use has been as part of scanners in compilers. However, nowadays their applications extend far beyond those areas. For example, regular expressions have been used in editors for structured text modification [19]. They are used in network protocol analysis [24] and for specifying events in a distributed system [18]. Regular expressions are used for virus detection using signature scanning [15], in mining the web [14], and as alternatives types for XML data [13]. Moreover, there are many uses of regular expressions outside of computer science, for example, in sociology (for characterizing events that led to placing of a child in foster care) [21] or biology (for finding DNA sequences) [16]. In addition to specific applications, many generic types of information, such as phone numbers or dates, are often presented and exchanged in specific formats that are described using regular expressions.

Despite their popularity and simplicity of syntax, regular expressions are not without problems. There are three major problems with regular expressions that can make their use quite difficult.

– *Complexity*. Regular expressions are often hard to understand because of their terse syntax and their sheer size.
– *Errors*. Many regular expressions in repositories and on the web contain faults. Moreover, these faults are often quite subtle and hard to detect.
– *Version Proliferation*. Because many different versions of regular expressions are stored in repositories for one particular purpose, it is in practice quite difficult to find or select the right one for a specific task.

A major source of these problems is the lack of abstraction in regular expressions, leaving the users no mechanism for reuse of repeated subexpressions. By obscuring the meaning, this also contributes to the explosion of regular expressions that are just variations of others, often with errors.

It is easy to see that lack of abstraction causes regular expressions that do anything non-trivial to grow quite large. We have analyzed all the 2799 regular expressions in the online regular expression repository at `regexplib.com` and found that 800 were at least 100 characters long. Some complex expressions actually exceeded 4000 characters, and the most complex expressions had more than ten nested levels. It is very hard for users to look through what essentially are long sequences of punctuation and understand what such an expression does. That this is a real problem for programmers can be seen, for example, from the fact that the popular website `stackoverflow.com` has over 2000 questions related to regular expressions. (Of those, 50 were asking for variations of dates and 100 questions are about how to validate emails with regular expressions.)

As an example consider the following regular expression. It is far from obvious which strings it is supposed to match. It is even more difficult to tell whether it does so correctly.

```
<\s*[aA]\s+[hH][rR][eE][fF]=f\s*>\s* <\s*[iI][mM][gG]\s+[sS][rR][cC]
=f\s*>[^<>]*<\s*/[iI][mM][gG]\s*>\s*<\s*/[aA]\s*>
```

This complexity makes it very hard to understand non-trivial regular expressions and almost impossible to verify them. Moreover, even simple modifications become extremely prone to errors.

To address these problems and offer help to users of regular expressions, we have developed an ensemble of methods to explain regular expressions. These methods reveal the structure of regular expressions, identify the differences between represented data and format, and can provide a semantics-based annotation of intent.[1] In addition to aiding users in their understanding, our explanations can also be employed to search repositories more effectively and to identify bugs in regular expressions.

The rest of the paper is structured as follows. In Section 2 we discuss shortcomings of regular expressions to reveal the opportunities for an explanation notation. Specifically, we illustrate the problems with complexity and understandability mentioned above. Based on this analysis we then develop in Section 3 a set of simple, but effective explanation structures for regular expressions and demonstrate how these can be computed. We present an evaluation of our work in Section 4 with examples taken from a public

[1] A first explanation of the above example is given in Figure 1, and a more comprehensive explanation follows later in Figure 2.

repository. Finally, we discuss related work in Section 5 and present conclusions in Section 6.

2 Deficiencies of Regular Expressions

There are many different forms of regular expressions and many extensions to the basic regular expressions as defined in [12]. Perl regular expressions provide the largest extension set to the above. However, the extensions offered by Perl go far beyond regular languages and are very hard to reason about. Therefore, we have chosen the POSIX regular expression extensions [20] as the basis of our work along with short forms for character classes, which provide a good coverage of the commonly used regular expression syntax while still avoiding the complexity of Perl regular expression syntax.[2]

Some principal weaknesses of the regular expression notation have been described by Blackwell in [1]. Using the cognitive dimensions framework [2] he identifies, for example, ill-chosen symbols and terse notation as reasons for the low understandability of regular expressions. Criticism from a practical perspective comes, for example, from Jamie Zawinsky, co-founder of Netscape and a well-known, experienced programmer. He is attributed with the quote "Some people, when confronted with a problem, think: 'I know, I'll use regular expressions.' Now they have two problems." [11].

In the following we point out some of the major shortcomings of regular expressions. This will provide input for what explanations of regular expression should accomplish.

(1) Lack of abstraction. A major deficiency of regular expressions is their lack of an abstraction mechanism, which causes, in particular, the following problems.

- *Scalability*. Regular expressions grow quite large very quickly. The sheer size of some expressions make them almost incomprehensible. The lack of a naming mechanism forces users to employ copy-paste to represent the same subexpression at different places, which impacts the scalability of regular expressions severely.
- *Lack of structure.* Even verbatim repetitions of subexpressions cannot be factored out through naming, but have to be copied. Such repetitions are very hard to detect, but knowledge about such shared structure is an important part of the meaning of a regular expression. Therefore, regular expressions fail to inform users on a high level about what their commonalities and variabilities are.
- *Redundancy*. The repetition of subexpression does not only obscure the structure and increase the size, it also creates redundancies in the representations, which can lead to update anomalies when changing regular expressions. This is the source of many faults in regular expressions and has a huge impact on their maintainability.
- *Unclear intent*. Both of the previously mentioned problems make it very hard to infer the intended purpose of a regular expression from its raw syntax. Without such knowledge it is impossible to judge a regular expression for correctness, which also makes it hard to decide whether or not to use a particular regular expression. Moreover, the difficulty of grasping a regular expression's intent makes it extremely hard to select among a variety of regular expressions in a repository and find the right one for a particular purpose.

[2] For simplicity we do not consider Unicode.

All these problems directly contribute to a lack of understandability of regular expressions and thus underline the need for explanations. The problem of unclear intent also points to another major shortcoming of regular expressions.

(2) Inability to exploit domain knowledge. Abstract conceptual domains are structured through metaphorical mappings from domains grounded directly in experience [3]. For example, the abstract domain of time gets its relational structure from the more concrete domain of space. Children learning arithmetic for the first time commonly rely on mapping the abstract domain of numbers to their digits [7]. The abstract concepts are easier to pick up if users are provided with a mapping to a less abstract or more familiar domain. One of the difficulties with regular expressions is that it is a formal notation without a close mapping [2] to the domain that the user wants to work with, which makes it difficult for new users to pick up the notation [1]. Moreover, there is no clear mental model for the behavior of the expression evaluator.

(3) Role Expressiveness. A role expressive notational system must provide distinctive visual cues to its users as to the function of its components [2]. Plain regular expressions have very few beacons to help the user identify the portions of regular expressions that match the relevant sections of input string. The users typically work around this by using subexpressions that correspond to major sections in the input, and by using indentation to indicate roles of subexpressions visually.

(4) Error Proneness. Regular expressions are extremely prone to errors due to the fact that there is no clear demarcation between the portions of regular expression that are shared between the alternatives, and those portions that are not, and thus have a higher contribution towards the variations that can be matched.

In the next section we will address the problems of scalability, unclear structure, and redundancy through explanation representations that highlight their compositional structure and can identify common formats.

3 Explanation Representations and Their Computation

The design of our explanation representations is based on the premise that in order to understand any particular regular expression it is important to identify its structure and the purpose of its components. Moreover, we exploit the structures of the domain that a particular regular expression covers by providing semantics-based explanations that can carry many syntactic constraints in a succinct, high-level manner. Two other explanation structures that are obtained through a generalization process are briefly described in the Appendix.

3.1 Structural Analysis and Decomposition

Large regular expressions are composed of smaller subexpressions, which are often very similar to each other. By automatically identifying and abstracting common subexpressions, we can create a representation that directly reveals commonalities. Our method is a variation of a rather standard algorithm for factoring common subexpressions. We

illustrate it with the example given in the introduction. We begin by grouping maximally contiguous sequences of characters that do not contain bracketed expressions such as `(···)`, `[···]` or `|` into bracketed expressions. This step is not necessary if the expression does not contain `|`. It preserves the meaning of the regular expression because brackets by themselves do not change the meaning of a regular expression.

For example, here are the first several subsequences identified in that way for the introductory example (each separated by white space).

```
<\s* [aA] \s+ [hH] [rR] [eE] [fF] =f\s*>\s*<\s* [iI] [mM] [gG]
\s+ [sS] [rR] [cC] =f\s*> [^<>]* ...
```

This step essentially amounts to a tokenization of the regular expression character stream.

The next step is to introduce names for sequences (not containing space specifications) that are recognized as common entities that appear more than once in the expression.

In the extraction of subexpressions we make use of a number of heuristics for generating names for the subexpressions since the choice of names has a big influence on the understandability of the decomposed expression [4, 8]. For example, a plain name represents the regular expression that matches the name, such as *img* = `img`. Since many applications of regular expressions involve the description of keywords whose capitalization does not matter, we also use the convention that underlined names represent regular expressions that match any capitalization of that name, such as $\underline{a}$ = `[aA]` or $\underline{img}$ = `[iI][mM][gG]`.

In the example, common expressions are expressions that represent upper- or lowercase characters, such as `[aA]`, which will therefore be replaced by their names. After this replacement, the token sequence for our example looks as follows.

```
<\s* a \s+ h r e f =f\s*>\s*<\s* i m g \s+ s r c =f\s*> [^<>]* ...
```

This grouping process can now be iterated until no more groups can be identified for naming. In the example we identify several new names as follows.

```
<\s* a \s+ href =f\s*>\s*<\s* img \s+ src =f\s*> [^<>]* ...
```

The advantage of the described naming conventions is that they provide an implicit naming mechanism that simplifies regular expressions without introducing any overhead.

Another naming strategy is to generate names from equivalent POSIX character class names, such as *alpha* = `[a-zA-Z]`, *alphas* = *alpha*`*`, *digit* = $[0-9]$, and *digits* = *digit*`*`. In the case of sequence of one or more matches, we capitalize the word, that is, *Alphas* = *alpha*`+` and *Digits* = *digit*`+`. We also replace sequences that are identified as matching a continuous sequence of numbers with a range notation. For example, `[0 .. 12]` = `[0-9]|1[0-2]` and `[00 .. 99]` = `[0-9][0-9]`.

Finally, we also use the convention that a blank matches `\s*`, that is, a sequence of zero or more spaces, and that a boxed space ␣ matches `\s+`, that is, a sequence of one or more spaces. Redundant brackets that do not enclose more than one group are also removed in this step. Applying some of these simplifications, we finally obtain for our example expression the decomposition shown in Figure 1.

```
< a␣href=f > < img␣src=f >[^<>]*< /img > < /a >
```

Fig. 1. Decomposition for the embedded image regular expression.

This representation makes it much clearer what the original regular expression does, namely matching an image tag embedded in a link that matches the same file name `f`. This is a common pattern frequently used in programs and scripts that scan web pages for pictures and download the image files.

Our decomposition method can also identify parameterized names for subexpressions that are variations of one another. Although, this technique can further reduce redundancy, it also makes explanations more complex since it requires understanding of parameterized names (which are basically functions) and their instantiation (or application). Since it is not clear at this point whether the gained abstraction warrants the costs in terms of complexity, we ignore this feature in this paper.

Although the overall structure of regular expressions is revealed through decomposition, the approach described so far is only partially satisfying. In particular, the different functions of format and data expressions are not yet distinguished, and opportunities for semantic grouping have not been exploited yet. We will address these two issues next.

3.2 Format Analysis

Most non-trivial regular expressions use punctuation and other symbols as separators for different parts of the expression. For example, dates and phone numbers are often presented using dashes, or parts of URLs are given by names separated by periods. We call such a fixed (sub)sequence of characters that is part of any string described by a regular expression a *format*. Since all strings contain the format, it does not add anything to the represented information. Correspondingly, we call the variable part of a regular expression its *data part*.

By definition the individual strings of a format must be maximal, that is, in the original regular expression there will always be an occurrence of a (potentially empty) data string between two format strings. The situation at the beginning and end of a format is flexible, that is, a format might or might not start or end the regular expression.

A format can thus be represented as an alternating sequence of format strings interspersed with a wildcard symbol "•" that stands for data strings. For example, possible formats for dates are •-•-• and •/•/•, and the format for our embedded image tag expression is the pattern <•=f><•=f>•</•></•>. The wildcard symbol matches any number of characters so long as they are not part of the format. This notion of a format can be slightly generalized to accommodate alternative punctuation styles, as in the case for dates. Thus a regular expression can have, in general, a set of alternative formats. Note that because there is no overlap between the format and data strings, identifying the format also helps with the structuring of the data.

The computation of formats can be described using a simple recursive definition of a function *format*, which computes for a given regular expression e a set of formats. The function is defined by case analysis of the structure of e. A single constant c is

itself a format, and the format for a concatenation of two expressions $e \cdot e'$ is obtained by concatenating the formats from both expressions (successive $\bullet$ symbols are merged into one). We cannot derive a format in the above defined sense from a repetition expression since it cannot yield, even in principle, a constant string.[3] The formats of an alternation $e|e'$ are obtained by merging the formats of both alternatives, which, if they are "similar enough", preserves both formats, or results in no identification of a format (that is, $\bullet$) otherwise.

$$\begin{aligned} \mathit{format}(c) &= \{c\} \\ \mathit{format}(e \cdot e') &= \{f \cdot f' \mid f \in \mathit{format}(e) \wedge f' \in \mathit{format}(e')\} \\ \mathit{format}(e|e') &= \cup_{f \in \mathit{format}(e), f' \in \mathit{format}(e')} \mathit{align}(f, f') \\ \mathit{format}(e^*) &= \bullet \end{aligned}$$

Two formats are *similar* when they can be aligned in a way so that the positions of the wild-cards and the fixed strings align in both expressions. If this is the case, the function *align* will return both formats as result, otherwise it will return a wildcard. For example, the formats f = `•-•-•` and f' = `•/•/•` are similar in this sense, and we have $\mathit{align}(f, f') = \{$`•/•/•`, `•-•-•`$\}$.

3.3 User-Directed Intent Analysis

Our analysis of repositories revealed that many regular expressions have subexpressions in common. Sometimes they share exact copies of the same subexpression, while sometimes the expressions are very similar.

This indicates that there is a huge need, and also an opportunity, for the reuse of regular expressions. Specifically, the commonality can be exploited in two ways. First, if someone wants to create a new regular expression for a specific application, it is likely that some parts of that expression exist already as a subexpression in a repository. The problem is how to find them. If the names used in the decompositions produced as part of our explanations were descriptive enough, then these domain-specific names could be used to search or browse repositories. This will work increasingly well over the middle and long run, when explanations with descriptive names have found their way into existing repositories.

The purpose of user-directed intent analysis is to provide explanations with more descriptive names, which aids the understanding of the regular expression itself, but can also be exploited for searching repositories as described. As an example, consider the following regular expression for dates.

```
(((0[13578]|[13578]|1[02])/([1-9]|[1-2][0-9]|3[01]))|((0[469]|[469]|11)
/([1-9]|[1-2][0-9]|30))|((2|02)-([1-9]|[1-2][0-9])))/[0-9]{4}
```

[3] Although we have considered the notion of *repetition formats*, we did not include them since they would have complicated the format identification process considerably. On the other hand, our analysis presented in Section 4 seems to indicate that the lack of repetition formats can limit the applicability of format analysis.

The fact that it describes dates is not inherent in the regular expression itself. However, once we are given this piece of information, we try to match its subexpressions to annotated date (sub)expressions in the repository. In this case, we find that `0[13578]|[13578]|1[02]` is used to describe months that have 31 days. At this point, this is just a guess, which can be communicated to the user. Once the user confirms the interpretation, this information can be exploited for an explanation.

Specifically, we can apply decomposition with a name that reflects this information. Moreover, we can suggest potential improvements to the expression. For example, in this case we can suggest to replace the first two alternatives `0[13578]|[13578]` by an optional prefix, that is, by `0?[13578]`. If we continue the interpretation, we can identify two other components, months with 30 days and February, the month with 29 days. Accepting all suggested interpretations, decomposition will thus produce the following explanation.

((*month-with-31-days*/*31-days*)| (*month-with-30-days*/*30-days*)| (*february*/*29-days*))/*year*
where
month-with-31-days = `0?[13578]|1[02]`
month-with-30-days = `0?[469]|11`
february = `2|02`
31-days = `[1 ... 31]`
30-days = `[1 ... 30]`
29-days = `[1 ... 29]`
year = `[0-9]{4}`

From our analysis of online repositories and discussion sites of regular expressions we found that the majority of regular expressions that are actively used and that users ask about are taken from 30 or so categories, which shows that with limited annotation effort one can have a big impact by providing improved regular expression decomposition using intention analysis.

3.4 Combined Explanations

The different explanation representations explored so far serve different purposes and each help explain a different aspect of a regular expression. Ultimately, all the different representations should work together to provide maximal explanatory benefit. To some degree this cooperation between the explanations is a question of GUI design, which is beyond the scope of the current paper.

We therefore employ here the simple strategy of listing the different explanations together with the regular expression to be explained. We add one additional representational feature, namely the color coding of identified subexpressions. This enhances the explanation since it allows us to link subexpressions and their definitions to their occurrence in the original regular expression. As an example, consider the explanation of the embedded image regular expression shown in Figure 2.

For coloring nested definitions, we employ a simple visualization rule with preference to the containing color. This leaves us with non-nested colors. This rule can be seen in action in Figure 3 where the occurrence of the subexpression *29-days* as part of the expressions representing *30-days* and *31-days* is not color coded in the original regular expression.

▶REGULAR EXPRESSION

```
<\s*[aA]\s+[hH][rR][eE][fF]=f\s*>\s*<\s*[iI][mM][gG]\s+[sS][rR][cC]=f\s*>
[^<>]*<\s*/[iI][mM][gG]\s*>\s*<\s*/[aA]\s*>
```

▶STRUCTURE

```
< a␣href=f > < img␣src=f >[^<>]*< /img > < /a >
```

▶FORMAT(S)

```
<•=f><•=f>•</•></•>
```

Fig. 2. Combined explanation for the embedded image regular expression.

4 Evaluation

We have evaluated our proposed regular expression explanations in two different ways. First, we analyzed the explanation notation using the cognitive dimensions framework [2]. The results are presented in Section 4.1. Then in Section 4.2, we show the result of analyzing the applicability of our approach using the `regexplib.com` repository.

4.1 Evaluating the Explanation Notation Using Cognitive Dimensions

Cognitive Dimensions [2] provide a common vocabulary and framework for evaluating notations and environments. The explanation notation was designed to avoid the problems that the original regular expression notation causes in terms of usability and understandability. The cognitive dimensions provide a systematic way of judging that effort. Here we discuss a few of the cognitive dimensions that immediately affect the explanation notation.

Viscosity The concept of *viscosity* measures the resistance to change that often results from redundancy in the notation. High viscosity means that the notation is not supportive of changes and maintenance.

While regular expressions score very high in terms of viscosity, our explanation structures for regular expressions have low viscosity since the abstraction provided by naming and decomposition allows redundancy to be safely eliminated. Moreover, the automatic decomposition identifies and removes duplicates in regular expressions. For example, consider an extension of the regular expression for dates, shown in Figure 3, that uses either `/` or `-` as a separation character. Using raw regular expressions, we need to duplicate the entire expression, and add it as an alternative at the top level. In contrast, the explanation representation can safely factor out the commonality and avoid the redundancy.

▶REGULAR EXPRESSION

```
((((0[13578]|[13578]|1[02])/([1-9]|[0-2][0-9]|3[01]))|((0[469]|[469]|11)/
([1-9]|[0-2][0-9]|30))|((2|02)/([1-9]|[0-2][0-9])))/[0-9]{4}
```

▶STRUCTURE

((*month-with-31-days*/*31-days*)|(*month-with-30-days*/*30-days*)|(*february*/*29-days*))/*year*
where
month-with-31-days = 0?[13578]|1[02]
month-with-30-days = 0?[469]|11
february = 2|02
31-days = [1 .. 31]
30-days = [1 .. 30]
29-days = [1 .. 29]
year = [0-9]{4}

▶FORMAT(S)

•/•/•

Fig. 3. Combined explanation for the date expression.

Closeness of Mapping Any notation talks about objects in some domain. How closely the notation is aligned to the domain is measured in the dimension called *closeness of mapping*. Usually, the closer the notation to the domain is, the better since the notation then better reflects the objects and structures it talks about.

In principle, regular expressions are a notation for a rather abstract domain of strings, so considering closeness of mapping might not seem very fruitful. However, since in many cases regular expressions are employed to represent strings from a specific domain, the question actually *does* matter. Our notation achieves domain closeness in two ways. First, intent analysis provides, in consultation with the user, names for subexpressions that are taken from the domain and thus establish a close mapping that supports the explanatory value of the regular expression's decomposition. While this obviously can be of help to other users of the expression, this might also benefit the user who adds the names in the decomposition in gaining a better understanding by moving the expression's explanation closer to the domain. Second, explanations employ a specific notation for numeric ranges that is widely used and can thus be assumed to be easily understood. Again, Figure 3 provides an example where the ranges and their names for 30 days, 31 days, and 29 days are closer to the intended list of numbers than the corresponding original parts of the regular expressions.

Role Expressiveness This dimension tries to measure how obvious the role of a component is in the solution as a whole. In our explanation notation the formats produced by format analysis separate the fixed strings from the data part, which directly points to the roles of the format strings. The format also identifies more clearly a sequence of

Type	Selected	Type	Selected
Email	38	URI	74
Numbers	107	Strings	91
Dates and Times	134	Address and Phone	104
Markup or code	63	Miscellaneous	173

Table 1. Regular expression in different domains in the regexplib.com repository

matchings, which supports a procedural view of the underlying expression. Moreover, roles of subexpressions are exposed by structure analysis and decomposition techniques.

4.2 Applicability of Explanations

We have evaluated the potential reach and benefits of our methods through the following research questions.

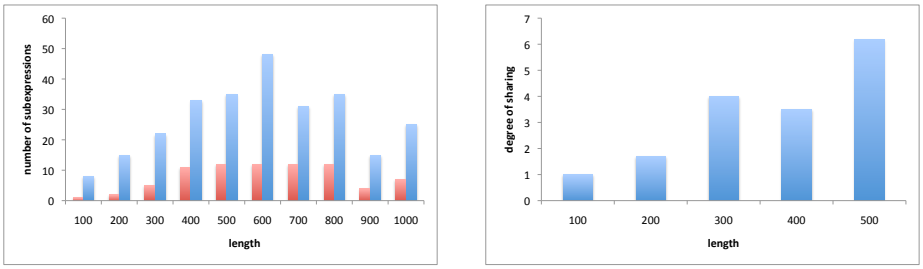
RQ1: Is the method of systematic hierarchical decomposition applicable to a significant number of regular expressions?
RQ2: How effective is our method of computing formats?
RQ3: Can we accurately identify the intended meanings of subexpressions in a regular expression given the context of the regular expression?
RQ4: How well does error detection work? In particular, can we identify inclusion errors for the regular expressions in real-world expressions?

The fourth research question came up during our work on RQ3. We noticed subtle differences in similar regular expressions and suspected that these were responsible for inaccuracies. We have developed a method of regular expression generalization that generates candidates of compatible regular expressions that can then be inspected for inclusion/exclusion errors, that is, strings that are incorrectly accepted/rejected by a regular expression.

Setup The publicly available online regular expression repository at `regexplib.com` was used for all our evaluations. This repository contains a total of 2799 user-supplied regular expressions for a variety of matching tasks. For our evaluations we manually eliminated eight syntactically invalid regular expressions as well as 18 expressions that were longer than 1000 characters to simplify our inspection and manual annotation work, leaving a total of 2773 regular expressions for analysis. Of these, 800 have been assigned to different domains. The total numbers of regular expressions in each domain are given in Table 1.

Test Results For the evaluation of the first research question, we identified the subexpressions using our decomposition algorithm. For each expression we recorded the number of its (non-trivial)[4] subexpressions and averaged these numbers over regular expressions of similar length. That is, average was taken over intervals of 100 characters each. These are given by the long bars in Figure 4 on the left. The short bars show the averaged number of common subexpressions (that is, subexpression that occurred at least twice).

[4] A basic subexpressions was identified as the longest sequence of tokens that does not contain another group or subexpression.



(a) Frequency of subexpressions (b) Degree of sharing

Fig. 4. Subexpression and Sharing Statistics. There was almost no sharing in expressions that are longer than 500 characters.

Considering one category in more detail, of the 134 regular expressions in the date-time category, 103 contained subexpressions. The maximum number of subexpressions was 11, and on average each expression contained about 3.25 subexpressions. The maximum number of nesting levels was 10, and on average expressions had 4 levels of nesting. The numbers in the other categories are very similar. We also determined the degree of sharing, that is, the number of times each identified subexpression occurred in the regular expression. The average degree of sharing, again averaged over expressions within a certain length range, is shown in Figure 4 on the right.

The most repeated single subexpression occurred 21 times. The total number of repeated subexpressions per regular expression was on average 3.9, the regular expression that contained the most repetitions had 42 repeated subexpressions. These numbers show that (1) there is quite a variety in number and repetitions of subexpressions, and (2) that decomposition has much structure to reveal and is widely applicable.

Since the evaluations of the second, third, and fourth research questions all required manual verification, we had to limit the scope of the considered regular expressions. For the second research question we used a randomized sample of 100 regular expressions selected from the complete repository. Format analysis was able to identify formats for 55 of those. The results are also shown in Table 2. We would have expected a larger number of formats. We suspect the rather low number is due to our ignoring repetition formats. A follow-up study should investigate how many more formats can be identified by using repetition formats.

Property	Found (Total)
Formats	55 (100)
Intentions	440 (1513)
Inclusion errors	39 (280)

Table 2. Analysis Applicability

For the evaluation of the third research question, we chose all of the 134 regular expressions from the date-time category. We applied our algorithm to identify the intent of each of the 1513 subexpressions that were contained in the 134 expressions. We could identify 440 subexpressions as having a specific intent. These results are also shown in Table 2. We believe that this number is quite encouraging and demonstrates that intent analysis can be widely applicable.

With regard to the fourth research question, we chose a randomized sample of 100 regular expressions from the date-time category. These regular expressions contained

280 subexpressions that we could identify from the context. We could detect 39 inclusion errors in these samples, but no exclusion errors. These results are summarized in Table 2. We believe that finding 39 (potential) faults in 134 regular expressions as a by-product of an explanation is an interesting aspect. Some of these are false positives since the intent provided by our analysis might be overly restrictive and not what the creator of the regular expression under consideration had in mind. Still, warnings about even only potential faults make users think carefully about what regular expression they are looking for, and they can thus further the understanding of the domain and its regular expressions.

4.3 Threats to Validity

The limited sample size may skew our results if the samples are not representative. Another threat to the validity is that the website `regexplib.com` may not be representative. A different threat to our research findings could be in our estimation of effectiveness of our techniques using cognitive dimensions framework rather than a user study.

5 Related Work

The problems of regular expressions have prompted a variety of responses, ranging from tools to support the work with regular expressions to alternative language proposals.

Prominent among tools are debuggers, such as the Perl built-in regular expression debugger and *regex buddy* (see `regexbuddy.com`). The Perl debugger creates a listing of all actions that the Perl matching engine takes. However, this listing can become quite large even for fairly small expressions. Regex buddy provides a visual tool that highlights the current match. There are also many tools that show subexpressions in a regular expression. The best known is rework [23], which shows the regular expression as a tree of subexpressions. However, it does not support naming or abstraction of common subexpressions. Several tools, such as Graphrex [6] and RegExpert [5], allow the visualization of regular expressions as a DF. All these approaches share the same limitations with respect to providing high-level explanations. Specifically, while these approaches help users understand why a particular string did or did not match, they do not provide any explanations for what the general structure of the regular expression is and what kind of strings the regular expression will match in general.

In the following we discuss a few alternatives that have been proposed to regular expressions.

Topes provides a system that allows a user to specify a format graphically without learning an arcane notation such as regular expressions [22]. The system internally converts specifications to augmented context-free grammars, and a parser provides a graded response to the validity of any string with respect to an expected format. The system also allows programmers to define "soft" constraints, which are often, but not necessarily always true. These constraints help in generating graded responses to possibly valid data that do not conform to the format.

Blackwell proposes a visual language to enter regular expressions [1]. It also provides a facility to learn regular expressions from given data (programming by example). The

system does provide multiple graphical notations for representing regular expressions. However, it is not clear how well this notation will scale when used for more complex regular expressions.

Lightweight structured text processing is an approach toward specifying the structure of text documents by providing a pattern language for text constraints. Used interactively, the structure of text is defined using multiple relationships [17]. The text constraint language uses a novel representation of selected text as collections of rectangles or region intervals. It uses an algebra over sets of regions where operators take region sets as arguments and generate region sets as result.

While most of these (and other) approaches arguably provide significant improvements over regular expressions, the fact that regular expressions are a de facto standard means that these tools will be used in only specific cases and that they do not obviate the need for a general explanation mechanism.

We have previously investigated the notion of explainability as a design criterion for languages in [9]. This was based on a visual language for expressing strategies in game theory. A major guiding principle for the design of the visual notation was the *traceabilty* of results. A different, but related form of tracing was also used in the explanation language for probabilistic reasoning problems [10].

Since regular expressions already exist, we have to design our explanation structures as extensions to the existing notation. This is what we have done in this paper. In particular, we have focused on structures that help to overcome the most serious problems of regular expressions—the lack of abstraction and structuring mechanisms. In future work we will investigate how the notion of traceability in the context of string matching can be integrated into our set of explanation structures.

6 Conclusions

We have identified several representations that can serve as explanations for regular expressions together with algorithms to automatically (or semi-automatically in the case of intention analysis) produce these representations for given regular expressions.

By comparing raw regular expressions with the annotated versions that contain decomposition structures, formats, and intentional interpretations, it is obvious—even without a user study—that our methods improve the understanding of regular expressions. The use of the developed explanation structures is not limited to explain individual regular expression. They can also help with finding regular expressions in repositories and identifying errors in regular expressions. This demonstrates that explanation structures are not simply "comments" that one might look at if needed (although even that alone would be a worthwhile use), but that they can play an active role in several different ways to support the work with regular expressions. Our evaluation shows that the methods are widely applicable in practice.

The limitations of regular expressions have prompted several designs for improved languages. However, it does not seem that they will be replaced with a new representation anytime soon. Therefore, since regular expressions are here to stay, any support that can help with their use and maintenance should be welcome. The explanation structures and algorithms developed in this paper are a contribution to this end.

References

1. Blackwell, A.F.: See What You Need: Helping End-users to Build Abstractions. J. Visual Languages and Computing 12(5), 475–499 (2001)
2. Blackwell, A.F., Green, T.R.: Notational Systems - The Cognitive Dimensions of Notations Framework. HCI Models, Theories, and Frameworks: Toward and Interdisciplinary Science pp. 103–133 (2003)
3. Boroditsky, L.: Metaphoric structuring: understanding time through spatial metaphors. Cognition 75(1), 1 – 28 (2000)
4. Bransford, J.D., Johnson, M.K.: Contextual prerequisites for understanding: Some investigations of comprehension and recall. J. Verbal Learning and Verbal Behavior 11(6), 717–726 (1972)
5. Budiselic, I., Srbljic, S., Popovic, M.: RegExpert: A Tool for Visualization of Regular Expressions. In: EUROCON, 2007. The Computer as a Tool. pp. 2387–2389 (2007)
6. Graphrex. `http://crotonresearch.com/graphrex/`
7. Curcio, F., Robbins, O., Ela, S.S.: The Role of Body Parts and Readiness in Acquisition of Number Conservation. Child Development 42, 1641–1646 (November 1971)
8. Derek, M.J.: The New C Standard: An Economic and Cultural Commentary. Addison-Wesley Professional (2003)
9. Erwig, M., Walkingshaw, E.: A Visual Language for Representing and Explaining Strategies in Game Theory. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing. pp. 101–108 (2008)
10. Erwig, M., Walkingshaw, E.: Visual Explanations of Probabilistic Reasoning. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing. pp. 23–27 (2009)
11. Friedl, J.: Now you have two problems. `http://regex.info/blog/2006-09-15/247`
12. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc. (2006)
13. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular Expression Types for XML. In: In Proc. of the International Conf. on Functional Programming (ICFP). pp. 11–22 (2000)
14. Hur, J., Schuyler, A.D., States, D.J., Feldman, E.L.: SciMiner: web-based literature mining tool for target identification and functional enrichment analysis. Bioinformatics (Oxford, England) 25(6), 838–840 (Mar 2009)
15. Lockwood, J.W., Moscola, J., Kulig, M., Reddick, D., Brooks, T.: Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware. In: In Military and Aerospace Programmable Logic Device (MAPLD). p. 10 (2003)
16. Mahalingam, K., Bagasra, O.: Bioinformatics Tools: Searching for Markers in DNA/RNA Sequences. In: BIOCOMP. pp. 612–615 (2008)
17. Miller, R.C., Myers, B.A.: Lightweight Structured Text Processing. In: USENIX Annual Technical Conf. pp. 131–144 (1999)
18. Nakata, A., Higashino, T., Taniguchi, K.: Protocol synthesis from context-free processes using event structures. In: Int. Conf. on Real-Time Computing Systems and Applications. pp. 173–180 (1998)
19. Pike, R.: Structural Regular Expressions. In: EUUG Spring Conf. pp. 21–28 (1987)
20. Regular Expressions. `http://en.wikipedia.org/wiki/Regular_expression`
21. Sanfilippo, L., Voorhis, J.V.: Categorizing Event Sequences Using Regular Expressions. IASSIST Quarterly 21(3), 36–41 (1997)
22. Scaffidi, C., Myers, B., Shaw, M.: Topes: reusable abstractions for validating data. In: Int. Conf. on Software Engineering. pp. 1–10 (2008)
23. Steele, O.: . `http://osteele.com/tools/rework/`
24. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic Network Protocol Analysis. In: Annual Network and Distributed System Security Symp. (NDSS 08) (2008)