RegViz: Visual Debugging of Regular Expressions

Fabian Beck[1], Stefan Gulan[2], Benjamin Biegel[2],
Sebastian Baltes[2], and Daniel Weiskopf[1]

[1]VISUS, University of Stuttgart, Germany
{fabian.beck,weiskopf}@visus.uni-stuttgart.de

[2]University of Trier, Germany
{biegel,s.baltes}@uni-trier.de
gulan@informatik.uni-trier.de

ABSTRACT

Regular expressions are a widely used programming technique, but seem to be neglected by software engineering research. Encoding complex string parsing in a very compact notation, their complexity and compactness, however, introduce particular challenges with respect to program comprehension. In this paper, we present RegViz, an approach to visually augment regular expressions without changing their original textual notation. The visual encoding clarifies the structure of the regular expressions and clearly discerns included tokens by function. The approach also provides advanced visual highlighting of matches in a sample text and defining test cases therein. We implemented RegViz as a Web-based tool for JavaScript regular expressions. Expert feedback suggests that the approach is intuitive to apply and increases the readability of regular expressions.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*pretty printing, program editors*

General Terms

Design, Human Factors, Languages

Keywords

Regular expressions, program comprehension, syntax highlighting, software visualization, visual debugging

1. INTRODUCTION

Regular expressions are a useful tool for processing textual data. Already with a short expression, programmers are able to match strings to a grammar, extract substrings from texts, or implement a search feature—at least in theory. In practice, however, regular expressions, though being short, have a quite complex syntax themselves, do not include elements that visually structure the expression, and contain cryptic subexpressions. For instance, what does this short expression do?

```
\[([^\]]+)\]|\(([^\)]+)\)
```

Using traditional means of syntax highlighting might improve the readability of a regular expression, but is limited to coloring characters—structural information is hard to convey. Regular expressions can also be visualized, for instance, as a graph structure [4, 5, 8, 9]. Alternative representations, however, do not only introduce a new notation that has to be learned, but also the problem of a dual representation: insights gained from the visualization need to be mapped to the textual representation of the regular expression.

To overcome these issues, we present RegViz: a hybrid approach to augment the original textual representation of regular expressions with visual elements without using an additional representation. We see our main contributions in designing a visual augmentation of regular expressions that clearly shows the structure of the expression, mapping the visual representation of regular expressions to detected matches in a sample text, and implementing the approach as an Web-based tool for JavaScript regular expressions (`http://regviz.org`).

2. RELATED WORK

Text editors and IDEs format source code according to its structure (pretty printing) and colorize words and expressions (syntax highlighting). Although even visualizations on software metrics and execution behavior have been integrated into text editors [1, 2, 7], regular expressions stay usually unformatted within the code. What exists, however, are visualizations of regular expressions that do not augment the text but completely replace it or use additional diagrams: For teaching scenarios in context of theoretical computer science and compiler construction, a regular expression is transformed in an equivalent automaton, which in turn is visualized as an interactive node-link diagram [5, 8, 9]. Blackwell [4] investigates visual representations of regular expressions in more detail and discusses three different ones: an automaton-like graph representation, a hierarchically structured natural language text, and a visual language working with text and icons. In a software engineering scenario where the regular expression always needs to be represented as text, these visual representations consume much space and lead to a repetitive representation of the regular expression, which we try to avoid.

While research on developing and visualizing regular expressions is rare, many software tools are available. They seem to enjoy quite some popularity and to satisfy a demand,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '14, May 31 – June 7, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2768-8/14/05 ...$15.00.

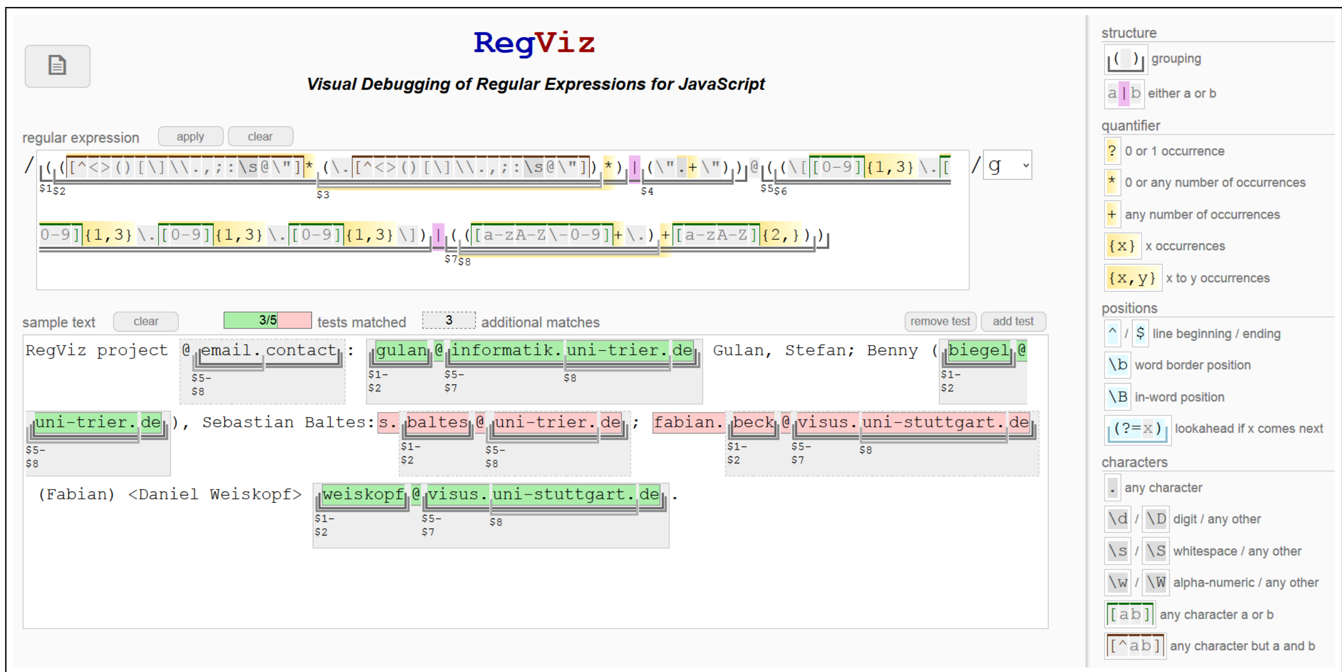


Figure 1: Interface of RegViz showing a (defective) regular expression for matching email addresses.

but systematic evaluations or research regarding their design space is completely lacking. We want to highlight Debuggex[1] and RegexBuddy[2] as particularly visual tools (Goyvaerts and Levithan [6] provide a more exhaustive list of tools). Debuggex shows the regular expression as a graph diagram additionally to representing it as color-coded text. Applying it to a sample text, a slider can be used to synchronously sweep through text and visualization; a list of tests can be specified. RegexBuddy organizes the interface in multiple views: additional to a color-coded regular expression, a treeview shows the hierarchical structure of the expression with nodes labeled in natural language; a debugging view illustrates the matching process. In contrast to our approach, Debuggex and RegexBuddy both require an additional representation for visualizing the structure of a regular expression.

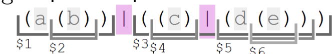
3. REGVIZ APPROACH

Our vision was to improve the readability of regular expressions by enriching their textual notation with visual augmentations as a part of an interactive testing environment. In contrast to previous work, we wanted to avoid an additional representation and the drawbacks discussed that go along with it. We developed the RegViz approach and implemented a prototype as a Web-based tool for JavaScript regular expressions. The software itself is written in JavaScript and its visual appearance is specified in CSS. The full user interface is depicted in Figure 1 visualizing a complex regular expression for matching email addresses. The interface is split into three parts: At the top, a text area shows a visually augmented regular expression, which is editable like plain text though being surrounded by visual elements. Below, a sample text with specified tests can be inserted; the regular expression is applied to this text and matches are visualized. Finally, a legend on the right side explains the encoding.

3.1 Visually Augmented Regular Expressions

The heart of the approach is the visual enrichment of regular expressions. The design as illustrated in Figure 1 highlights the structure of a regular expression mainly through horizontal lines top and bottom of the expression and discerns special-purpose tokens by color. In the following, we enumerate the syntactic constituents of regular expressions, recapitulate their use, and comment on our design decisions for each.

One of the most important augmentations are those that clarify the **structure** of a regular expressions—something that is nearly concealed in its plain-text representation. First, regular expressions include parentheses to group tokens within the expression; these *capturing groups* can be used for referencing (matched) parts of the regular expression when postprocessing the matches. In JavaScript, group ID `$1` refers to the first group (with respect to opening parentheses), `$2` to the second etc. Our tool exposes groups through horizontal lines attached to the bottom of the expression. The stacking of these lines reflects the nesting hierarchy of groups. The start and end positions of a group are marked by vertical lines and the group ID is provided below.



Another structural element in a regular expression is a vertical bar character, which denotes the *alternation* operator: either the subexpression to the left or right of the operator has to be matched. We highlight this inconspicuous, yet important token by using a light purple background.

Regular expressions further allow us to specify a number of repeated consecutive matchings of the same subexpression. This is denoted by postfixing this subexpression—which may be a single character, a set of characters, or a group—with

[1] `https://www.debuggex.com/`
[2] `http://www.regexbuddy.com/`

a **quantifier** operator. A quantifier is any of `?`, `+`, `*`, or a range in curly brackets (e.g., `{1,3}`). A yellow horizontal gradient identifies quantifier occurrences quite prominently. The gradient is darker to the left to emphasize the association with the left subexpression, i.e., the postfix character of quantifiers. The range of a quantifier's influence is visualized by an additional yellow line above characters (e.g., `a*`) and character sets (e.g., `[A-Z]*`) and below groups (e.g., `(a)*`).

An *anchor* specifies a **position** at which a match is allowed to occur. Possible anchors are `^`, `$`, `\b`, or `\B`, indicating the beginning or ending of a line or a word. A light blue background clearly discerns anchors from other entities so that they cannot be mistaken for plain characters. A related concept is the *lookahead*, which defines the context that must follow a match. A lookahead may be positive: `(?=r)`, or negative: `(?!r)`, where `r` is again a regular expression. This makes the lookahead a postfix operator of arbitrary complexity. As illustrated in the following example, we visualize lookahead subexpressions like a special group, marked by a blue bottom line, which integrates into the nesting of groups. The start and end of the lookahead are highlighted in light blue to make them instantly distinguishable from start and end of capturing groups.

`(University)(?= (of )?(Trier|Stuttgart))`
$1 $2 $3

Finally, the atomic building blocks of every regular expressions are the specific **characters** that should be matched. While most characters can be included in an expression 'as is', some need to be escaped because they have a special meaning otherwise. Both escaped and non-escaped plain characters have a light gray background and are separated by small white gaps (e.g., `a\[\.b<`). It is further possible to specify sets, or *classes*, of characters in square brackets: we visualize these by adding a top border to the subexpression in square brackets (e.g., `[A-Z]`). By starting the selection with `^`, character sets enumerate the characters that should not be included; we use dark red instead of dark green for the top border to indicate this kind of negative selection (e.g., `[^A-Z]`). Certain frequently occurring classes are given short placeholders. For example, `.` denotes any character, `\d` denotes a digit, `\D` represents every character but a digit, etc. Those expressions are discerned from plain characters by a slightly darker background color.

3.2 Visually Indicated Matches and Testing

The regular expression is applied to a sample text. If the expression has valid syntax, all detected matches are marked as gray boxes in the sample text. There matching mode can be selected by the combo box next to the expression: the flag `g`, for *global*, specifies that all matches should be identified instead of just the first while flag `i` switches from *case-sensitive* to *case-insensitive* mode.

Groups in regular expressions correspond to certain substrings of the match. Programmers might use groups for structuring the regular expression as well as for referencing a part of a match. In the process of debugging a regular expression, the group structure in the match could provide details on an error in the expression. We thus visualize the groups in the matches as well, using the same bottom lines and labels that highlight the groups in the expression.

`fabian.beck@visus.uni-stuttgart.de`
$1-$2 $3 $5-$7 $8

Due to alternation operators and quantifiers, it is possible that some groups are not matched and represented within the match (e.g., missing group `$4` in the above example). Please note that, in contrast to highlighting in regular expressions, several groups may start at the same text positions; labels, hence, need to be aggregated at these positions (e.g., `$5-$8`). The representation of a group in the regular expressions is connected to the representation of the same group in the sample text through interaction: hovering the mouse over the opening or closing parenthesis of the group in the regular expression, the group markers are highlighted in blue color in both the regular expression and the sample text.

We also integrated a simple testing feature into RegViz, which—in contrast to Debuggex—does not require an additional view. A test is a user-defined substring of the sample text, which should be matched by the regular expression. A statistic above the sample text window summarizes how many of the specified tests are matched and whether additional matches (that do not cover specified tests) exist. Matched tests in the sample text are marked with a green box, non-matched tests with a red one. Tests can be added and removed by marking the respective text with the mouse and clicking the add or remove button next to the text field. Theses tests are basically regression tests that are applied whenever the regular expression is changed. They, hence, could indicate that a change has introduced a new bug when tests that matched before do not match anymore. As well, users might specify tests in the sample text before writing the regular expression—a form of *test-driven development* [3].

3.3 Application Example

Discussing RegViz in a realistic application, we take a closer look at the email example in Figure 1. The regular expression shows a slightly modified version of a solution originally posted at stackoverflow.com by Peter Mortensen[3]. The sample text that we use for testing consists of names and addresses in various notations, also including a substring that only looks like an email address (`@email.contact`). For illustration purposes, we introduced a bug into the regular expression such that not all email addresses in the sample text are matched, but the fake email address is.

Although matching of email addresses sounds like a simple problem, a generic solution is quite complex. The presented variant consists of two main segments, which are quickly identified by looking at the nesting structures of groups: the local part of the address in front of `@` (group `$1`) and the domain part after (group `$5`). Either part is split by an alternation operator: the local part either consist of sequences of certain characters

`[^<>()[\]\\.,;:\s@\"]*`

separated by fullstops (group `$2`) or arbitrary characters enclosed in quotation marks `\".+\"` (group `$4`); the domain part is either an IP address (group `$6`) or a resolved Internet domain (group `$7`). Without visual augmentation, the example is very difficult to read, not only because of its length but also because non-escaped parentheses are used in sets as plain characters (see character set above) and can be easily mistaken for closing group parentheses.

In the sample text, all five email addresses are marked as tests, but only three of them are matched correctly. The

[3] `http://stackoverflow.com/questions/46155/validate-email-address-in-javascript`

highlighted groups within the matches reveal that all domain parts of the addresses are identified correctly. A bug, hence, should be in the local part, which is only partly matched in two of the examples. These two examples have in common that they consist of two words split by a fullstop—group `$3` should have been matched, but—as the visualization reveals—is not. The quantifier `+` is missing after the character set in group `$3`. Another issue is that the character sequence is optional (`*`), but should not be (`+`)—this finally prevents `@email.contact` to be matched. The fixed version of group `$2` and `$3` is:

```
([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)
$2                     $3
```

4. EXPERT FEEDBACK

We sent RegViz to colleagues who are either experts in software engineering, visualization, or Web development and asked for constructive feedback. The goal of this preliminary evaluation was to collect educated opinions on strengths and shortcomings of the approach as well as to get ideas for future development. We provided unlimited access to RegViz including several examples (combinations of regular expressions, sample texts, and tests) and allowed the experts to use the tool as long as they like. Our inquiry did not dictate a specific task but just included a short questionnaire to obtain semi-structured free-text feedback. Ten experts replied, who we name E1–E10 in the following. We summarize their general, high-level feedback but skip technical details and minor comments.

In general, the experts seem to agree that the approach is accessible as they describe it as "intuitive" (E1, E6, E7), "clear" (E3, E5), "self-explaining" (E3), "easy to understand" (E8), "helpful" (E2), or "useful" (E9, E10). Although we only provided minimal explanations, none of the experts had reasonable problems using the tool. Only secondary features had not been totally clear to everybody, like testing (E5) or different matching modes (E6, E9).

The visual augmentation, which discerns our approach from others, is largely described as beneficial: All experts except E1 explicitly mention the visual encoding of the regular expressions as one of the main advantages of RegViz. E3 and E4 also rate the highlighting of groups in the sample text as particularly important; E2, E3, E5, E6, E9, and E10 highlight the testing features and visualization. Nevertheless, also some doubts were raised on the extensive use of visuals: in the opinion of some experts, the tool could be somewhat overwhelming at first sight (E1, E9), some visual encodings tend to hinder readability (E3), the visualization clutters the display (E7, E8), or it consumes much space and attention (E4).

Beyond requested minor improvements, the experts suggested some larger extensions and application scenarios for RegViz. One group of suggestions targets an improved debugging like to visually explain a match in detail (E1), to help finding out why something was not matched (E3), or to support identifying the causes of syntax errors (E4). E2 and E7 put forward the idea of marking substring as negative tests that should not be matched. Extending editing features, E4, E5, and E6 propose making the legend clickable so that users can edit the regular expressions using these buttons. Also integrating the tool into an IDE was suggested (E3, E4, E6), or combining it with a graph-based visualization of regular expressions (E4, E9). Besides being used by programmers for software development, RegViz might also be leveraged for teaching students (E1, E2, E9).

5. CONCLUSIONS AND FUTURE WORK

Filling the gap between visualization and textual notation, we have presented the new idea of a hybrid approach for representing regular expressions. Embedded into a environment to test and debug these expressions, we call the approach RegViz. Feedback collected from experts suggests that RegViz is easy to use and helps understanding regular expressions. In the near future, we plan to improve RegViz according to the experts' suggestions, in particular, to reduce the visual load of the interface further. We also want to pick up and implement the suggested extensions like explanations of missed matches and an improved interactive editing of regular expressions. The enhanced version will be made publicly available on `http://regviz.org`.

6. ACKNOWLEDGMENTS

We thank all experts for their highly valuable feedback.

7. SOLUTION OF THE QUIZ

The regular expression presented in Section 1 matches text in (non-nested) square brackets (group `$1`) or parentheses (group `$2`).

```
\[([^\]]+)\]|\(([^\)]+)\)
  $1          $2
```

8. REFERENCES

[1] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf. Visual monitoring of numeric variables embedded in source code. *VISSOFT '13: Proceedings of the 1st IEEE Working Conference on Software Visualization*, 2013.

[2] F. Beck, O. Moseler, S. Diehl, and G. D. Rey. In situ understanding of performance bottlenecks through visually augmented code. In *ICPC '13: Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 63–72. IEEE, 2013.

[3] K. Beck. *Test-Driven Development: By Example.* Addison-Wesley Professional, 2003.

[4] A. Blackwell. SWYN: A visual representation for regular expressions. In *Your Wish is My Command: Programming by Example*, pages 245–270. Morgan Kauffman, 2001.

[5] B. Braune, S. Diehl, A. Kerren, and R. Wilhelm. Animation of the generation and computation of finite automata for learning software. *Automata Implementation*, pages 39–47, 2001.

[6] J. Goyvaerts and S. Levithan. *Regular Expressions Cookbook.* O'Reilly Media, 2009.

[7] M. Harward, W. Irwin, and N. Churcher. In situ software visualisation. In *ASWEC '10: Proceedings of the 21st Australian Software Engineering Conference*, pages 171–180. IEEE Computer Society, 2010.

[8] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. *ACM SIGCSE Bulletin*, 32(1):6–10, 2000.

[9] N. Moreira and R. Reis. Interactive manipulation of regular objects with FAdo. *ACM SIGCSE Bulletin*, 37(3):335–339, 2005.