

# Lightweight Structured Visualization of Assembler Control Flow based on Regular Expressions

Sibel Toprak, Arne Wichmann, and Sibylle Schupp  
 Hamburg University of Technology, Institute for Software Systems  
 Schwarzenbergstr. 95e, 21073 Hamburg, Germany  
 {sibel.toprak, arne.wichmann, schupp}@tuhh.de  
 http://www.sts.tu-harburg.de

**Abstract**—regVIS is a tool for viewing directed graphs with start and end nodes. It applies a new visualization technique, which uses regular expressions as a meta-representation of all the paths in an input graph; the result is a containment-based and structured visualization of that graph. The tool can be configured to derive these regular expressions from the input graph using either the Brzozowski algebraic method or the transitive closure method.

regVIS can be used in combination with the binary code analysis tool IDA (Interactive Disassembler), either integrated or standalone, to view the control flow graph (CFG) of assembler code. The resulting visualization, which restructures the control flow and can thus help reduce program comprehension efforts, is called *control flow blocks* (CFB).

In this paper, we present the workings of regVIS and evaluate the new CFB visualization it produces against the traditional CFG visualization in an explorative user study. The study suggests that the CFB is better for analyzing and navigating along specific execution paths, while the CFG is better for getting an overview of the overall control flow.

## I. INTRODUCTION

Whenever one needs to understand a program, for which the original source code is no longer available, working with the binaries is the only option. With the aid of binary code analysis tools, such as the industry standard disassembler IDA (*Interactive Disassembler*, [www.hex-rays.com](http://www.hex-rays.com)), assembler code can be recovered for large portions of the binaries, which is then more human-readable, yet still unmanageable for program understanding purposes. Most binary code analysis tools therefore provide additional support for examining various aspects of the assembly code semantics. Usually, control flow analysis constitutes a good starting point. Insights into how the control possibly flows between the basic blocks of a program during its execution can be gained by taking a look at the *control flow graph* (CFG), a commonly used program visualization technique that conveys the control flow information using the visual property of linkage [1].

For large program functions, however, these CFGs are impractical to use. Due to their size, the task of exploring and navigating through them becomes an arduous one: Viewing the portion of interest in such a CFG requires a considerable amount of horizontal and vertical panning. Also, automatically generated graph layouts are not always optimal for code exploration. As such, keeping track of the overall control flow, especially while trying to examine the code details, is difficult.

Deriving a *regular expression* (RE) over the basic blocks of a CFG yields a concise *one-dimensional* representation of all valid execution paths for the corresponding program function. In such a regular expression, the concatenation, alternation, and quantification operators embody the control flow information, abstracting the control flow into explicit structures like sequence, choice, and loop, respectively. Mapping it to a visual language composed of constructs that make use of the visual property of containment [1], yields a different visualization of the control flow; we call it *control flow blocks* (CFB).

In short, the CFB is an overlay of all valid execution paths, each one represented by the sequence of basic blocks that it is composed of. It allows the viewer to analyze a specific execution path in a focused manner, while all other information that are not of interest, remain hidden. Fitting the height of the visualization to the screen space reduces the panning necessary to see other portions of a selected execution path to a single direction only.

We present **regVIS** (*Regular Expression-based Graph Visualizer*), a tool that receives a CFG in Graph Description Language (GDL) [2], [3] format and is capable of visualizing CFBs.

The remainder of this paper is structured as follows: Section II discusses related work. In Section III, the theoretical foundations are laid for computing a regular expression that represents all possible execution paths in a CFG, while details concerning the generation of the CFB are described in Section IV. Section V introduces the regVIS tool. Following this, the design of the experiment that was conducted to evaluate the control flow visualizations against each other, the results obtained from the experiment as well as any threats to the validity of these results are presented in Section VI. Section VII concludes this paper.

## II. RELATED WORK

Related research can be divided into program restructuring, visualizing graphs and programs, and regular expression theory.

Our goal is a structured, linearized CFG representation, which presents a node's contents as well as the overall structure and can be used for program comprehension.

This idea originates in Structured Programming [4], where the use of explicit control structures, for instance for condi-

tional or iterative execution, to structure programs is recommended. The well-known Nassi-Shneiderman Diagrams [5] introduced a notation that enforces the use of these structures by omitting a notation for the discouraged goto. Our visualization mimics the structured programming idioms by using regular expression structures and shows them as visual elements.

The concept of dimensional reduction of flowcharts is not new, as Knuth [6] writes about systematic program restructuring. He discusses the elimination as well as the introduction of goto-statements and concludes that:

"In two dimensions it is possible to perceive *go to* structure in small examples, but we rapidly lose out ability to understand larger and larger flowcharts; some intermediate levels of abstraction are necessary."

Other works on mechanized program restructuring for refactoring [7], compiler construction [8], or decompilation [9], [10] employ various graph transformations or rewriting of the abstract syntax tree, but either concentrate on concrete target languages, or require complex transformation systems, or cannot fully eliminate goto-statements. In difference to these works, we pass the input graph's nodes unchanged and calculate the new representation from its linkage.

Linearization of programs is also used for signature generation of malware executables [11], which applies CFG structuring to create a signature that allows a simple comparison, but hides concrete workings of the nodes.

A large body of work exists on graph visualization [12] and on software visualization [1]. Yet, graph visualization concepts like (selective) zooming and panning, and conventional 2-dimensional graph layouts cannot help to achieve the desired dimensional reduction. The common reduction strategies for large graphs hide the nodes' contents and are therefore also unsuitable for our goal. For hierarchical graphs, treemaps [13] provide a good method of structuring the visualization, but lack support for loop structures in the graph. Software visualization provides the above mentioned structured diagrams, which we employ to display the structure of our computed regular expression.

For the construction of the regular expression from a deterministic finite automaton (DFA) [14], [15] we use the transitive closure method [16] and the Brzowski algebraic method [17], [18]. While optimal conversions from a DFA [19] as well as normal forms [20] for regular expressions exist, these simpler methods are fully suitable for our visualization needs.

### III. PRELIMINARIES

The equivalence of *deterministic finite automata* (DFA) and regular expressions within the Chomsky hierarchy turns out to be quite useful for our purposes: Because CFGs are conceptually similar to DFAs, the problem of transforming a CFG into a regular expression is reduced to the problem of converting a DFA to an equivalent regular expression. For this, textbook solutions, such as the Brzowski algebraic method, exist. All relevant concepts will be reviewed in this section.

#### A. Control Flow Graphs

A CFG is a node-link-based visualization that contains all execution paths in a program function: The nodes corresponds to the *basic blocks* (BB), while the directed links connecting the nodes represent the control flow between the respective basic blocks. A basic block may have multiple incoming and outgoing links, meaning that it has as many possible predecessors and successors respectively. There are two types of basic blocks in the CFG that pose exceptions: *Entry blocks* do not have any predecessors, whereas *exit blocks* do not have any successors.

All control flow enters through an entry block and leaves through one of the exit blocks. Once the control flow reaches a basic block, the sequence of instructions that it is composed of, is executed as a unit. Upon completion, the control flow is transferred to one successive basic block. A sequence of transitions in the CFG constitutes a valid execution path, if it starts in an entry block and ends in an exit block.

#### B. Deterministic Finite Automata

A DFA  $\mathcal{A}$  is formally [14], [15] denoted by the quintuple

$$\mathcal{A} = (S, \Sigma, \delta : S \times \Sigma \rightarrow S, s_0, F),$$

where  $S$  is a finite set of *states*,  $\Sigma$  is a finite *input alphabet*,  $\delta$  is the *transition function* that determines the next state given a state and some input symbol,  $s_0 \in S$  is the *start state* and  $F \subseteq S$  is the set of *final states*. It is *deterministic* because in any of its states and for any input symbol that is accepted at that particular state, there is only one possible next state it can transition to.

A sequence of input symbols is called an *input string*. A DFA is said to accept any such string if that string makes it transition from its start state  $s_0$  to any final state in  $F$ . The set of all strings that the DFA  $\mathcal{A}$  accepts, makes up its *language*  $L(\mathcal{A})$ . Languages accepted by DFAs are called *regular languages*. The same regular language may be accepted by the multiple DFAs. The canonical form is the one with the least number of states, called the *minimal DFA*.

#### C. Regular Expressions

In formal language theory, regular expressions [14], [15] are concise descriptions of regular languages. The language generated by a regular expression  $E$  is denoted by  $L(E)$ . The following constants are defined as basic regular expressions:

- A regular language is defined over a finite alphabet  $\Sigma$ . Each symbol  $a \in \Sigma$  constitutes a regular expression.
- The *empty set*  $\emptyset$  is a regular expression that denotes the empty language, that is  $L(\emptyset) = \{\}$ . This constant is the *zero element* in the algebra of regular expressions. For the DFA corresponding to  $\emptyset$  there is no input string that could take it from its start state to its final state.
- The *epsilon*  $\epsilon$  is a regular expression that denotes the language only containing the empty string. In the algebra of regular expressions, this constant represents the *one element*. The corresponding DFA consists of a single state, which is both a start and a final state.

More complex regular expressions can be built inductively over these basic ones in finitely many steps by means of the following three algebraic regular expression operators. Let  $A$  and  $B$  be two regular expressions, such that:

- The *alternation*  $A + B$  denotes the set union of  $L(A)$  and  $L(B)$ , that is  $L(A + B) = L(A) \cup L(B)$ , where  $L(A + B)$  contains  $|L(A)| + |L(B)|$  strings as a result of this operation. In the literature, the symbols  $+$  and  $\mid$  are used interchangeably.
- The *concatenation*  $A \cdot B$  denotes the product of the languages  $L(A)$  and  $L(B)$ , that is  $L(A \cdot B) = \{a \cdot b \mid a \in L(A), b \in L(B)\}$ . This operation yields a set of strings that is formed by taking any string  $a$  in  $A$  and concatenating it with any string  $b$  in  $B$ , resulting in a total of  $|L(A)| \cdot |L(B)|$  strings in  $L(A \cdot B)$ . This operator is usually omitted.
- The *star* operator applied to  $A$ , as in  $A^*$ , is an abbreviating for  $\epsilon + A + A \cdot A + \dots$ . It denotes the closure of  $L(A)$ , such that  $L(A^*) = (L(A))^*$ . This operation yields a set of strings that is constructed by concatenating  $L(A)$  with itself any number of times. This can be denoted more formally like so:  $(L(A))^* = \bigcup_{i \geq 0} L(A)^i$ , where  $(L(A))^i = L(A) \cdot (L(A))^{i-1}$ , and  $(L(A))^0 = \{\epsilon\}$ .

Although the above operators are sufficient to construct any regular expression and hence, to express any regular language, the following two additional quantification operators are introduced as syntactic sugar. Let  $A$  be a regular expression:

- Using the *plus* operator, regular expression of the form  $A \cdot A^*$  or  $A^* \cdot A$  can be rewritten as  $A^+$ . The result is the positive closure of  $L(A)$ , that is  $L(A^+) = (L(A))^+$ , which is basically the same as the closure defined above, except that it does not contain  $\epsilon$ :  $(L(A))^+ = \bigcup_{i \geq 1} L(A)^i$ .
- With the *optional* operator regular expressions of the form  $\epsilon + A$  or  $A + \epsilon$  can be written as  $A^?$  instead.

As for the order, in which the operators are applied in a regular expression, the quantifiers, namely *star* ( $*$ ), *plus* ( $+$ ) and *optional* ( $?$ ) are of highest precedence, followed by the *concatenation* operator ( $\cdot$ ) and, finally, the *alternation* operator ( $+$ ). The precedence can be overridden using parentheses.

#### D. Simplification of Regular Expressions

In order to keep the size of the regular expressions manageable, the transformation laws that are listed in Table I can be applied. These are equivalences between regular expressions, such that rewriting one regular expression as the other does not have any effect on the language that is represented.

Simplifying a regular expression computationally can be quite time-consuming, if the application of the transformation laws is implemented to be reversible. If we want to avoid this by applying these transformation laws unidirectionally, how the resulting regular expression ends up looking, depends on the order, in which they are applied. This is due to the associative and commutative nature of the binary alternation operation.

The strategy that will be explained briefly in what follows, is pretty straightforward and has proven itself to be quite

TABLE I  
TRANSFORMATION LAWS FOR REGULAR EXPRESSIONS

<b>Associativity:</b>	$A + B = B + A$ $A + (B + C) = (A + B) + C$
<b>Commutativity:</b>	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
<b>Identities:</b>	$A \cdot \epsilon = \epsilon \cdot A = A$ $A \cdot \emptyset = \emptyset \cdot A = \emptyset$ $A + \emptyset = \emptyset + A = A$ $(\emptyset)^* = \epsilon$ $(\epsilon)^* = \epsilon$ $(\emptyset)^+ = \emptyset$ $(\epsilon)^+ = \epsilon$ $(\emptyset)^? = \epsilon$ $(\epsilon)^? = \epsilon$
<b>Distributivity:</b>	$A \cdot B + A \cdot C = A \cdot (B + C)$ $A \cdot C + B \cdot C = (A + B) \cdot C$
<b>Idempotency:</b>	$A + A = A$
<b>Shifting:</b>	$(A \cdot B)^* \cdot A = A \cdot (B \cdot A)^*$
<b>Additional:</b>	$A + \epsilon = \epsilon + A = A^?$ $A \cdot A^* = A^* \cdot A = A^+$

$A$ ,  $B$ , and  $C$  are regular expressions.

effective: The simplification process is organized in two steps and uses recursion. The transformation laws in Table I are applied only from left to right. In the first step, the intermediate results obtained at each step of the process of building a regular expression from simpler ones are simplified as much as possible. Here, only those transformation rules come to use that involve the algebraic operators, namely the concatenation, alternation and star operators. While iterating over the associativity, commutativity, distributivity, idempotency and shifting rules, in that order, the regular expression tree is traversed top down. Once the overall regular expression is obtained and the first step is completed, the transformation rules involving the additional plus and optional operators, are applied to it in a second step.

#### E. Brzozowski's Algebraic Method

Brzozowski presents an algebraic method [17], [18] for converting a DFA into a regular expression: A system of linear equations, called *derivatives* or *characteristic equations* in the literature, is created, which describes the behavior of the automaton. By solving this system of equations, the regular expression that is equivalent to the DFA with respect to the language it accepts, can be obtained. Let the DFA be denoted by  $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ . Furthermore, let  $S$  be a finite set of  $n$  states, where each state is uniquely identifiable by a number  $i$  between 1 and  $n$ .

The construction of the characteristic equations starts by associating each state  $i \in S$  with an unknown  $X_i$ . This unknown represents the set of input strings that make  $\mathcal{A}$  move from the state  $i$  to one of its final states in  $F$ . Each  $X_i$  can be described by an alternation over terms of the form  $a_{i,j} \cdot X_j$ . These terms represent the sequences of transitions over different successor

states of  $i$  that it takes for the automaton to reach a final state; each of the sequences is composed, as expressed with the concatenation operation, of the transition between the state  $i$  to a state  $j \in S$  on input  $a_{i,j} \in \Sigma$  and the sequence of transitions from  $j$  to any final state in  $F$  denoted by the unknown  $X_j$ . The alternation over these indicates that there is a choice between these sets of paths. The absence of a transition between the state  $i$  and a state  $j$  is denoted by  $\emptyset$ , this is usually omitted in the equations. If a state  $i$  belongs to the set of final states  $F$ , then  $\epsilon$  is also one of the terms in the equation  $X_i$ . Later on, it will act as a terminal in the process of solving the equations.

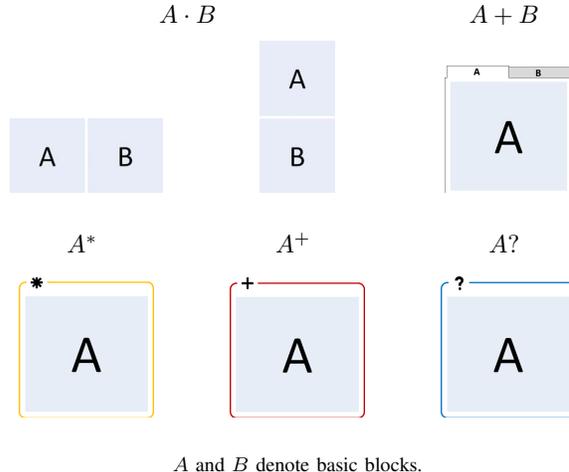
The resulting system of characteristic equations can be solved by substitution. Starting off with the characteristic equations corresponding to the final states of  $\mathcal{A}$ , the occurrences of the unknowns in all the other equations are eliminated, until only the characteristic equation corresponding to the start state  $s_0$  is left, which yields the regular expression that is searched for.

When an unknown appears on both sides of the language equation  $X_i$ , which happens when there is an edge from state  $i$  to itself, further substitution is not possible. In such a case, *Arden's rule* is applied, which states that a characteristic equation of the form  $X = A \cdot X + B$  can be transformed into  $X = A^* \cdot B$ . Applying this theorem solves the situation at hand, since it prevents the same unknown from occurring on both side of the equation as a result. After having isolated the unknown, the substitution process can be proceeded with.

Basically, Brzozowski's method starts from the automaton's final states and eliminates in each step the immediate predecessors of the states that were eliminated in the previous step. There are other methods for transforming a DFA into a regular expression, for example the transitive closure method. In comparison, however, this one creates relatively compact regular expressions. That is the why we confined ourselves to only discuss Brzozowski's method in detail.

#### IV. VISUALIZATION APPROACH

The CFB is a control flow visualization just like the CFG. It is based on regular expressions that are derived over the basic blocks of some CFG and capture the set of all paths that can be traversed in it. This creates a one-dimensional and structured representation of the control flow: While all execution paths in the CFG are projected onto a plane, the regular expression operators abstract the control flow into explicit structures like loop, choice, and optional execution. These structures are then simply mapped to a visual language composed of constructs that make use of the visual property containment to convey the control flow information. The resulting visualization contains as much control flow information as the CFG, but is easier to navigate and explore specific execution paths with. This section presents the visual language constructs, the workflow behind the CFB generation, and the expected output CFB for an example CFG.



$A$  and  $B$  denote basic blocks.

Fig. 1. The CFB Visual Language

##### A. Visual Language

The CFB is an overlay of all possible execution paths in a CFG, each of which is characterized by the sequence of basic blocks that lie along it. The constructs that make up its visual language, are designed to allow the viewer to switch between the different layers of the visualization. Figure 1 shows the mapping between the basic components of a regular expression computed for some CFG and these visual language constructs. The basic blocks in the regular expression are represented by simple boxes, which display the instructions contained in them. At the same time, the regular expression operators that achieve the structuring of the control flow, are represented by containers that enclose the basic blocks that they apply to.

For the sake of simplicity, let  $A$  and  $B$  be two basic blocks in a program function. But the following would also apply, if  $A$  and  $B$  were more complex regular expressions:

The *concatenation* operator applied to  $A$  and  $B$ , as in  $A \cdot B$ , denotes that  $B$  is executed right after  $A$ . In the visualization, the two basic blocks are shown in sequence, either one below the other or both next to each other depending on whether the orientation of the visualization is to be vertical or horizontal. The *alternation* operator applied to  $A$  and  $B$ , as in  $A + B$ , denotes the choice between the two basic blocks as to which one is to be executed next. The corresponding visual construct consists of tabs, one for every possible case. It only displays one case at a time and allows the viewer to switch from one case to any other by selecting the respective tab. As for the quantifiers, they denote variants of iterative execution: Applied to the basic block  $A$ , the *star* operator ( $A^*$ ) indicates that it is executed zero or more times, the *plus* operator ( $A^+$ ) implies that it is executed at least once, and the *optional* operator ( $A^?$ ) states that it is either executed once during a program run or not at all. In the visualization, a border with a small indicator for the respective quantifier type is put around the terminal representing  $A$ .

## B. Workflow

The workflow for generating the CFB for some given CFG can be divided into three steps, as illustrated in Figure 2.

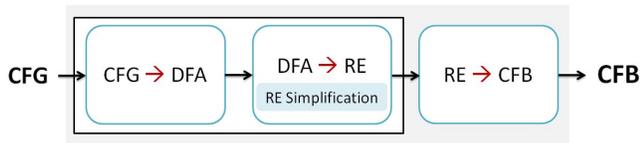


Fig. 2. The process of generating the CFB.

First, the CFG is turned into a DFA. Both are conceptually very similar: The CFG is also deterministic in its behavior, in that it is not possible to transfer the control flow to two different basic blocks at the same time. Such being the case, only a few minor modifications are required to transform the CFG into a DFA: All nodes of the CFG are reinterpreted as states of a DFA; each node is assigned a unique identifier  $i$  and is then added to the set  $S$ . A new state, linked to all states representing an entry block, is introduced. This state is designated as the start state  $s_0$ . All states corresponding to an exit block in the CFG are declared as final states of the DFA by adding them to the set  $F$ . The alphabet  $\Sigma$  contains the identifiers of all states. The directed links in the CFG are reinterpreted as transitions of the DFA. Each transition is labeled with the identifier of the target state, which can be thought of as the address of the next basic block that will be jumped to during program execution.

Next, a regular expression that accepts the same language as the DFA is derived by means of the Brzowski algebraic method or the transitive closure method. The problem is that both methods produce different regular expressions for the same input. The difference lies in how bloated these are due to basic blocks occurring multiple times in the regular expression. Ultimately, this problem of *basic block duplication* leads to different and unmanageable CFBs to be produced for the same CFG, which is undesirable. For a good compact visualization outcome, the underlying regular expression should be *minimal*: It should be as short as possible in length, while the set of execution paths that it represents, remains the same. Simplifying the regular expression according to the strategy previously described in Section III-D eliminates unnecessary basic block duplication and results in a minimized regular expression.

Finally, the obtained regular expression is mapped straightforwardly to the respective constructs in the visual language, which completes the workflow.

## C. Example

Figures 3 to 5 give an example of what the output of the process described in Section IV-B looks like. The CFG at hand (see Figure 3) contains assembly code in its nodes that corresponds to the code snippet (see Figure 4). In this CFG, the Blocks that contain a conditional jump at their end use red and green arrows to show the alternative next blocks. That snippet shows a program function written in C, which

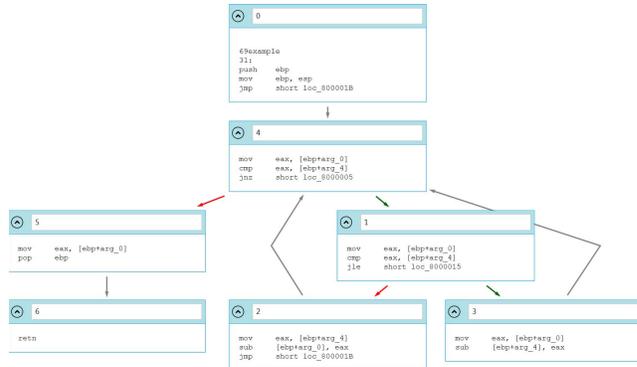


Fig. 3. Example CFG for the Euclidean algorithm.

```

int example(int a,int b) {
    while(a!=b)
        if(a > b)
            a=a-b;
        else
            b=b-a;
    return a;
}
  
```

Fig. 4. Implementation of the Euclidean algorithm in C.

implements the Euclidean algorithm for computing the greatest common divisor of two integer numbers. The derivation of a regular expression from the CFG using Brzowski’s method in combination with our systematic simplification approach yields

$$0 \cdot 4 \cdot (1 \cdot (2 + 3) \cdot 4)^* \cdot 5 \cdot 6$$

as a structured representation of its control flow, in which the numbers correspond to the basic block identifiers. The outcome of mapping the regular expression components to the respective constructs in the visual language is the CFB that is also shown in Figure 5.

This example shows the duplication of block number 4, the loop condition, which is traversed at least once, regardless of an execution of its loop. We accept such shared node duplications, as trade off for displaying complete paths. If at any point in time the node duplication in the visualization becomes a problem, it is possible to widen the regular expressions to accept more paths and thereby reduce node duplication.

## V. TOOL

This section introduces regVIS, a tool for generating the CFB for a given CFG. It accepts CFGs specified in GDL and can be used in combination with IDA, in which CFGs are internally represented in the same format. It is possible to integrate regVIS into IDA, but it can also be used as a standalone tool, as IDA is capable of exporting the GDL representation of CFGs to file. These files have the extension `.gdl` and can then be loaded manually into regVIS.

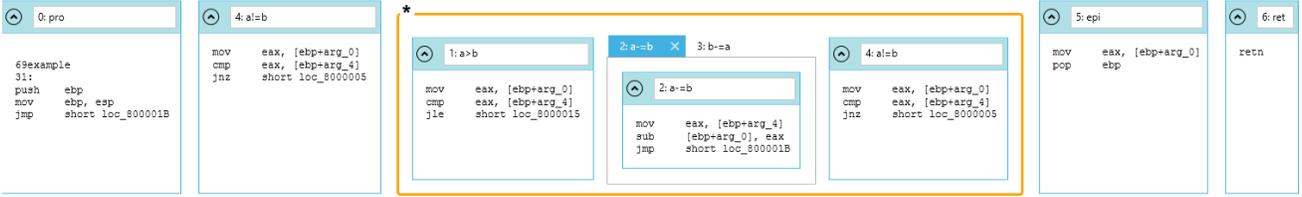


Fig. 5. Expected output CFB for the example CFG.

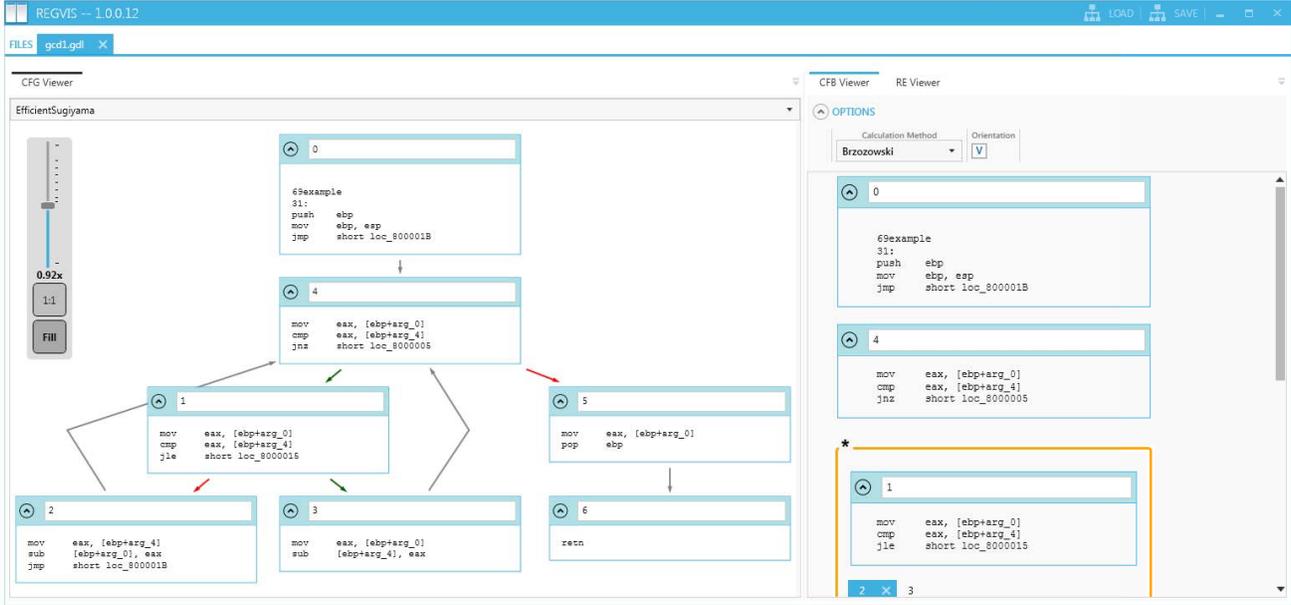


Fig. 6. The graphical user interface of regVIS.

The graphical user interface, of which a snapshot is provided in Figure 6, is designed to be clear in its structure: Much of the interface area is designated to rendering the different program outputs, which also includes the CFG in addition to the CFB. Each one is displayed in a separate resizable and dockable panel, which makes it possible to view them in parallel. Apart from the *CFB Viewer* and the *CFG Viewer*, there is also an *RE Tree Viewer* showing the tree representation of the computed regular expression. The color scheme is kept consistent throughout the different views. Multiple files can be loaded to the program at the same time. In all views, it is possible to manually put a description of a basic block’s content into the box headers that propagate throughout the views. The renaming of the basic blocks in the CFB can be stored and restored.

There are certain settings and certain functionalities that are specific to a particular view. In the CFB Viewer, one can switch between different regular expression derivation methods. Currently, one can either select the Brzozowski algebraic method (see Section III-E) or the transitive closure method to compute the regular expression for the input CFG. Moreover, one can view the CFB in horizontal or vertical

layout. The alternative paths that are hidden in the tab controls can be viewed as tool tips that appear when the mouse is placed over the tab headers in the CFB. By renaming the basic blocks in the headers, one can, by hand, create regular expressions, which are abstract representations of the path semantics. The one for our example CFG would be like so

$$pro \cdot a \neq b \cdot (a > b \cdot (a = b + b = a) \cdot a \neq b)^* \cdot epi \cdot ret$$

In the CFG Viewer it is possible to select between different graph layouts, however sticking with the default graph layout is recommended. Due to the limitations imposed by the used graph rendering library, the layout of the CFG is not ideal: Loops are not rendered, moving the basic blocks around results in weirdly bended arrows, and the positioning of the basic blocks in the CFG sometimes changes when opening the same GDL file again. However, it is sufficient for our purposes.

## VI. EVALUATION

This section presents the design of the experiment, which was conducted to validate the claims that were made, up to this point, about the CFB with respect to the convenience in navigating along and exploring specific execution paths, the

obtained results, and also any threats that could affect the validity of these results.

### A. Experiment Design

The conducted experiment falls into the category of *Evaluating User Experience* (UE) according to Lam, Bertini et al.’s taxonomy of empirical studies in information visualization [21]. Among all proposed methodological alternatives listed for this category, a *usability test* seemed the most viable option; an *informal evaluation* or a *field observation* were out of question due to limited availability of domain experts for the former and lack of extensive time for the latter method. In the usability test, we had participants analyze the assembler code of two simplified message protocols, using regVIS to view the CFG for one and the CFB for the other protocol. The participants were given program comprehension tasks to make sure that they familiarized themselves enough with the peculiarities of each control flow visualization to then ask them to give feedback about their experience as users. We were interested in finding out what the participants perceived as strengths and weaknesses about these visualizations and which one they found more suitable to perform certain tasks with. Ultimately, we wanted to see which visualization would be preferred overall.

Nine computer science students, who were either about to complete their Bachelor degree or were already enrolled in a Master study program, and one research assistant participated in this study. All ten participants were selected on the basis of previous experience with reading and understanding assembly code. Considering their academic standing, it was assumed that all of them knew about control flow graphs and regular expressions sufficiently well.

Multiple sessions were offered on different dates to fit the participants’ schedules. All sessions were conducted in a lab room on university premises and took about two hours. One moderator was present at each session and made sure that the experiments took place under examination conditions. All participants were provided with their own separate desks and a 23-inch Full-HD screen, which they could connect their own notebooks to if they wanted. All of the notebooks had Windows 7 or 8 and the .NET-Framework installed on them, which is necessary to run regVIS. The distributed test sheets were printed in black and white on A4-sized paper. The moderator passed around a USB stick containing regVIS, for those who had not downloaded it yet, as well as all input files necessary to complete the tasks, right before the start of the experiment.

The organization of the experiment is outlined in a detailed manner in Table II. The time was allocated to the four different stages keeping in mind that participants would probably be reluctant to spare more than two hours. For the sake of obtaining useful feedback, ensuring that the participants have enough time to play around with the visualizations, was important. Therefore, as much time as possible within the existing constraints was made available for the analysis of the message protocols with the aid of the two different control

TABLE II  
ORGANIZATION OF THE EXPERIMENT

Stage	Time
<b>Introduction</b>	05 min
<b>Training</b>	15 min
<ul style="list-style-type: none"> <li>• <i>Self-assessment questions</i> concerning programming experience and assembler knowledge</li> <li>• <i>Reading section</i> on how to produce the CFB for some CFG including a small example to illustrate the transformation process</li> <li>• A small memory game as <i>self-test question</i>, where CFG patterns corresponding to the basic high-level control flow structures have to be mapped to the corresponding CFB patterns</li> <li>• <i>Tutorial</i> on how to use the regVIS tool using a simple CFG as working example in preparation for the upcoming <b>Use</b> stage of the experiment</li> </ul>	
<b>Use</b>	2 x 25 min
5 x CFG first, 5 x CFB first Answering questions about the semantics of two simplified message protocols, where one was to be analyzed using the CFG and the other using the CFB. The time given to analyze one protocol was 25 minutes. Half of the questions were the same for both protocols, the rest was specific to each protocol. Each participant examined the visualizations in a different order: 5 participants worked with the CFG first, while the remaining 5 participants started with the CFB.	
<b>Feedback</b>	30 min
<ul style="list-style-type: none"> <li>• <i>Task 1</i>: Choosing the visualization that is more suitable to perform a program comprehension task with for a total of eight tasks</li> <li>• <i>Task 2</i>: Listing any previous knowledge that has made the questions in the <b>Use</b> section easier to solve</li> <li>• <i>Task 3</i>: Giving opinion about anything else noteworthy about the control flow visualizations or regVIS in general</li> </ul>	

flow visualizations. In addition, effort was put into creating tasks that were simple, yet allowed enough exposure to the visualization techniques for the participants to identify the strengths and weaknesses. Despite all that, it would not have been realistic to expect the participants to actually complete all the tasks correctly within that time frame; this was also emphasized during the introductory segment of the experiment.

In the **Use** part of the experiment, the participants had to examine two protocol parsers using the CFB and CFG visualizations. Both parsers comprised 40 lines of C-Code each. In a general questions part, the participants had to answer overall questions concerning the parser operation. Individual questions to allowed message sequences and dependencies during processing were posed in a special part for each parser. The actual data and questions can be found on the regVIS website [22].

The tasks were predominantly multiple-choice. Free spaces were provided additionally for textual answers and the participants were encouraged to make use of these to explain the reasoning behind their choices. This was especially necessary for the first task of the feedback section, where the participants were driven to choose the one visualization among the two that

TABLE III  
SELF-ASSESSMENT

Competence	min	max	avg	med
Programming	1 year	16 years	5.6 years	5 years
Assembler	1 week	3 years	7.9 months	3.5 months

TABLE IV  
DETAILED FEEDBACK

Tasks	G	B	X
<i>Strategic</i>			
Exploring Neighbors First (Breadth-First Search)	5	3	2
Exploring Paths First (Depth-First Search)	1	8	1
<i>Structural</i>			
Finding the Predecessors and Successors of a Basic Block	6	4	0
Detecting Data Dependencies	2	3	5
Detecting Clustering or Proximity	4	5	1
<i>Contextual</i>			
Navigating through the Visualization	3	6	0
Searching for a Specific Basic Block in the Visualization	6	2	2
Keeping Track of the Overall Control Flow	4	3	3
<i>Overall Preference</i>	4	3	3

G: Graph, B: Block, X: Undecided

they deemed more suitable to perform a certain graph analysis or program comprehension task with. Here, the empty spaces served as an opportunity to backpedal from their choice in case there was something about the chosen visualization that they did not find optimal either. If there were any other aspects that had caught their eye but that were not explicitly asked about in the previous tasks, such as further strengths and weaknesses of the visualization techniques, bugs in the implementation or feature requests, they had the chance to write extensively about them in the third and last task, where sufficient free space was left for that purpose.

## B. Results

The results of the experiment that was described detailedly in Section VI-A, will be presented and discussed in what follows. For starters, Table III is a summary of the rough estimates that the participants provided in the beginning of the experiment when asked to assess their programming experience and assembler knowledge in years, months, weeks or days. The intention was to have some means for characterizing the participants.

The evaluation of the Use part of the experiment can be seen in the performance column of Table V. For each participant, points were given for correct answers. These were made relative to the best performer of each task, to make the tasks and performance in individual visualizations comparable.

There were a total of three tasks for soliciting user feedback.

In the first task, the study participants had to choose for each one of eight basic program comprehension tasks the one visualization among the possible two that was, in their opinion, more suitable to perform it. Table IV shows the eight

tasks, for which the choice had to be made together with the respective distribution in votes. Note that, despite having only two options to choose from, where **G** stands for the CFG and **B** represents the CFB, there were certain instances where participants marked either both options or none at all. These votes were then counted as undecided (**X**). Based on the votes, a winner can be identified for each task. A summary of the arguments that were provided in the textual answers will be considered in the discussion of the results of this task in the following.

The first set of program comprehension tasks were strategic in nature. For *Breadth-First Search*, the majority of participants chose **G** with the argument that it was less effort to track edges than clicking through the tab navigation in **B**. Those in favor of **B** stated that scrolling was necessary only in one direction to find the neighbors, while **G** required a lot of panning in two directions. Those that remained undecided, were of the opinion that the suitability of the visualization techniques depended on the size of the assembler code; accordingly, **B** would probably be more preferable to **G** for larger program functions. As for *Depth-First Search*, **B** was chosen by the majority who felt that the navigation through unidirectional scrolling and tab selection was the most in line with the objective of analyzing a specific execution path, while the one participant opting for **G** considered it to be the better means for maintaining an overview of all execution paths.

The second set of program comprehension tasks can be characterized as being structural. For *Finding the Predecessors and Successors of a Basic Block*, most of the participants selected **G**, because they deemed tracking the edges to find the predecessors and successors here to be of less effort and far less confusing than navigating through **B** via tab selection. Those who chose **B**, reasoned that less scrolling was required and the positions of the predecessors and successors were more predictable due to the linearity in the layout. However, there was one participant who mentioned that there might be the risk of overlooking an occurrence of a basic block due to the problem with basic block duplication, introduced in Section IV-B, in the underlying regular expression. With regard to *Detecting Data Dependencies*, the majority of the participants remained undecided, with a few stating that data dependencies are easier analyzable along paths. The next task, *Detecting Clustering or Proximity*, was interpreted by the participants differently for each visualization: They thought of clustering and proximity as the closeness in placement of the basic blocks in **G** as defined by the graph layout algorithm and as the togetherness of blocks indicated by the structures in **B**. Here, the majority chose **B**.

The third and last set of program comprehension tasks were contextual. The majority voted for **B** for *Navigating through the Visualization*, giving the convenience of unidirectional scrolling and tab selection as reasons again. There was one person among those opting for **B**, however, who considered the loss of the jump conditions information as a disadvantage. Those who chose **G** over **B**, stated that they preferred one static view, where the information necessary for navigating is

available at one glance and is not hidden, as it is in **B**. As far as *Searching for a Specific Basic Block in the Visualization* is concerned, the majority opted for **G**. They argued that less clicking was necessary in a static view in order to navigate to the basic block that is of interest, since the information to do so does not have to be retrieved first. There was one participant, who mentioned that he was expecting it to be different for larger program functions. Another participant found this task difficult to perform with **B**, as basic blocks can belong to multiple paths in the visualization. Hence, the task of searching for a basic block actually becomes the task of searching for all occurrences of that block in the visualization. There were also those participants who could not decide between the two visualization approaches stating that both did the job. As for the task of *Keeping Track of the Overall Control Flow*, most users preferred **G**. One person explained his choice with the availability of the jump conditions in **G**, which were missing in **B**. A participant, who remained undecided, expressed in his textual answer that he considered both visualizations to be rather complicated in their own ways.

Finally, the study participants had to declare their overall opinion as to whether **G** or **B** is their preferred visualization approach to perform program comprehension tasks with. Here, **G** was selected by a slight majority.

In the second task, the participants were asked to list any previous knowledge that had helped them in using of the visualizations and completing the program comprehension tasks during the experiment. Having an understanding of assembler code, knowing how message protocols work, and being familiar with regular expressions as well as graphs in general were mentioned here, most of which had been anticipated already. In the third task, the participants had the opportunity to write about anything else that had caught their eye and that they wanted to give their opinion on.

About the CFG, the participants stated that they found the indication of the jump conditions helpful, that much scrolling was necessary to reach a basic block but that it was easier to determine where to scroll to, and that the overlapping links between the basic blocks were confusing. Some participants requested some additional features for regVIS; these were undoing manual changes to the placement of the basic blocks in the graph layout, double-clicking on arrows to quick-follow long paths and zooming in our out by using the mouse wheel instead of clicking on the designated control. With respect to the CFB, the participants wrote that the compactness and the abstraction of the control flow into explicit structures constituted an improvement to the user experience but that this could look different for larger program functions, that no Full-HD monitor is necessary to view it, that the path summary tool tips were somewhat helpful but disappeared too quickly, and that the absence of indicators for jump conditions in the visualization posed a serious weakness. Requested features were panning with the mouse wheel, making the basic blocks resizable to save screen space, making scrolling faster, displaying pop-up windows for subroutines, and hiding unnecessary information better. There were points referring

TABLE V  
OVERALL EVALUATION

Participant	Task Order	Performance	$\Sigma$ Feedback			Preference
			G	B	X	
1	G $\rightarrow$ B	B	3	1	4	G
2	G $\rightarrow$ B	X	3	4	1	X
3	G $\rightarrow$ B	X	4	4	0	G
4	B $\rightarrow$ G	B	4	4	0	G
5	B $\rightarrow$ G	G	6	2	0	G
6	G $\rightarrow$ B	B	1	6	1	B
7	B $\rightarrow$ G	G	1	6	1	B
8	G $\rightarrow$ B	B	2	3	3	X
9	B $\rightarrow$ G	B	3	5	0	B
10	B $\rightarrow$ G	X	2	3	3	X

G: Graph, B: Block, X: Undecided

TABLE VI  
INFLUENCE OF TASK ORDER

Order	Performance			Preference		
	G	B	X	G	B	X
G $\rightarrow$ B	0 (0%)	3 (60%)	2 (40%)	2 (40%)	2 (40%)	1 (20%)
B $\rightarrow$ G	2 (40%)	2 (40%)	1 (20%)	2 (40%)	1 (20%)	2 (40%)
Total	2 (20%)	5 (50%)	3 (30%)	4 (40%)	3 (30%)	3 (30%)

to regVIS in general, too. The graphical user interface, for instance, was perceived as well-structured. One participant suggested that the RE Viewer be improved by making the basic blocks collapsible and adding the node naming functionality here as well, arguing that it would be a good complement for the CFB Viewer. The remaining feedback statements were about the design of the experiment. Some participants felt that more training in using the CFB would have been better and that the difficulty of the program comprehension tasks, together with lacking understanding of assembler code in general, made it hard to complete them within the given time.

Table V shows the feedback results for every study participant ( $\Sigma$  *Feedback*) together with his or her preferred control flow visualization (*Preference*). Also contained in that table is the order that the visualizations were used in (*Task Order*), indicated with  $\rightarrow$ , and the visualization the respective user performed better with (*Performance*).

What remains to be examined is whether any peculiarities exist in the data that could have influenced the *Preference* in any way. Looking at Table V, there is no clear correlation between the *Performance* and the *Preference*. However, one peculiarity can be detected: The preferred visualization of a participant was rarely the one that he performed better with. This is the most visible for the CFB, as the majority of participants that had actually performed better with that visualization, still ended up choosing the CFG as the overall more preferable one. Table VI shows the distribution of participants among **G**, **B**, and **X** with respect to *Performance*

and *Preference* with and without considering the *Task Order*. There seems to be no effect on the *Preference* that could be attributed to the *Task Order* judging from the similarity in how the participants are distributed for the two possible orders, in which they were exposed to the control flow visualizations. However, an influence of the *Task Order* on the *Performance* is observable: One would expect the total distribution of the participants to be reflected in the distributions for each case of *Task Order*, but this is not the case. Generally speaking, the first control flow visualization in both sequences had a lower percentage of participants performing better with it, indicating a biasing training effect.

All in all, the results speak for the validity of the claims made about the CFB: As expected, the CFB was perceived by the users as the better means for navigating along and exploring execution paths compared to the CFG. Nevertheless, depending on the program comprehension task that is to be performed, not being able to view multiple paths can turn out to be of disadvantage. This is why, for program comprehension purposes, the best approach seems to be using the CFG and the CFB together.

### C. Threats To Validity

Although large participant groups are desirable in empirical studies for the sake of obtaining generally valid results, we had to settle for a small sample size for this usability test, because more people with some previous experience with assembler code were unavailable. Despite the effort of simplifying the program comprehension tasks, the lack of proficiency in analyzing and understanding assembler code within short time as well as lack of exposure to the CFB might have caused a bias in the overall results in favor of the CFG. Also, all participants were aware of our relation to this work, which could have led to subjectivity in the feedback.

## VII. CONCLUSION AND FUTURE WORK

In an effort to improve understanding of program functions in assembler language, a new regular-expression-based graph visualization, the CFB, was introduced to view the control flow in a structured manner with: A regular expression is computed for a CFG in order to abstract the control flow into explicit structures and is then mapped to a visual language consisting of constructs conveying the control flow information with the visual property of containment. We also introduced regVIS, a tool that both the CFG and the CFB of assembler code can be generated with. This tool came to use in a usability test, conducted to evaluate the two control flow visualizations against each other. It turned out that both complement each other well, because each visualization possesses features that make it more suitable for certain program comprehension tasks: While the CFB allows focusing on specific execution paths, the CFG allows one to view the control flow in a broader context. As a byproduct, we received useful input on how to improve the user experience of regVIS; the implementation of the requested features and a more comprehensive user study in terms of participation and detailedness remain as future work.

## REFERENCES

- [1] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.
- [2] *aiSee User Manual for Windows and Linux*, Version 3.4.3, AbsInt Angewandte Informatik GmbH, December 2009.
- [3] G. Sander, "Graph layout through the VCG tool," in *Graph Drawing*, ser. Lecture Notes in Computer Science, R. Tamassia and I. Tollis, Eds. Springer Berlin Heidelberg, 1995, vol. 894, pp. 194–205.
- [4] E. W. Dijkstra, "Structured programming," O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. London, UK: Academic Press Ltd., 1972, ch. Chapter I: Notes on Structured Programming, pp. 1–82.
- [5] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *ACM SIGPLAN Notices*, vol. 8, no. 8, pp. 12–26, Aug. 1973.
- [6] D. E. Knuth, "Structured programming with go to statements," *ACM Computing Surveys*, vol. 6, no. 4, pp. 261–301, Dec. 1974.
- [7] T. A. Bunter, "A non-deterministic approach to restructuring flow graphs," Department of Computer Science, Columbia University, Tech. Rep. CUCS-019-93, 1993.
- [8] A. M. Erosa and L. J. Hendren, "Taming control flow: A structured approach to eliminating goto statements," *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*, no. 1, pp. 229–240, 1994.
- [9] U. Lichtblau, "Decompilation of control structures by means of graph transformations," in *Mathematical Foundations of Software Development*, ser. Lecture Notes in Computer Science, H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, Eds. Springer Berlin Heidelberg, 1985, vol. 185, pp. 284–297.
- [10] C. Cifuentes, "Structuring decompiled graphs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, T. Gyimóthy, Ed. Springer Berlin Heidelberg, 1996, vol. 1060, pp. 91–105.
- [11] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing*, vol. 107, pp. 61–70, 2010.
- [12] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. van Wijk, J.-D. Fekete, and D. Fellner, "Visual analysis of large graphs: State-of-the-art and future research challenges," *Computer Graphics Forum*, vol. 30, no. 6, pp. 1719–1749, Sep. 2011.
- [13] B. Johnson and B. Shneiderman, "Tree-maps: A space-filling approach to the visualization of hierarchical information structures," in *Proceedings of the 2nd Conference on Visualization '91*, ser. VIS '91. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 284–291.
- [14] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [15] J. E. Savage, *Models of Computation: Exploring the Power of Computing*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.
- [16] S. C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies*, ser. Annals of Mathematics Studies, C. E. Shannon and J. McCarthy, Eds. Princeton University Press Princeton, NJ, USA, 1956, vol. 34.
- [17] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM*, vol. 11, no. 4, pp. 481–494, 1964.
- [18] R. Kain, *Automata Theory: Machines and Language*, ser. McGraw-Hill computer science series. Krieger, 1972.
- [19] S. Gulan and H. Fernau, "An optimal construction of finite automata from regular expressions," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Hariharan, M. Mukund, and V. Vinay, Eds., vol. 2. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008, pp. 211–222.
- [20] H. Gruber and S. Gulan, "Simplifying regular expressions," in *Language and Automata Theory and Applications*, ser. Lecture Notes in Computer Science, A.-H. Dediu, H. Fernau, and C. Martín-Vide, Eds. Springer Berlin Heidelberg, 2010, vol. 6031, pp. 285–296.
- [21] H. Lam, E. Bertini, P. Isenberg, C. Plaisant, and S. Carpendale, "Empirical studies in information visualization: Seven scenarios," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 9, pp. 1520–1536, Sept 2012.
- [22] "regVIS website," <http://www.sts.tu-harburg.de/projects/regvis/regvis.html>, 2014.