Fire works and fire station

a finite automata and regular expression playground

Frishert, M.

Award date:
2005

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

FIRE Works & FIRE Station

a Finite Automata & Regular Expression Playground

by
M. Frishert

Supervisor:	prof. dr. B.W. Watson
Advisors:	prof. D.G. Kourie dr. A.C. Telea

Berkeley, March 2005

Abstract

We present a new software toolkit for the construction and manipulation of regular "things", such as regular languages, regular expressions, and finite automata. The toolkit, FIRE Works, differs from toolkits in the same field in that it enforces a high level consistency between these different approaches to regular languages.

On top of this toolkit we have developed a software application, FIRE Station, for the visualization and manipulation of the graph datastructures underlying FIRE Works. We present a formal description, architectural and detailed design of both the toolkit and application.

Samenvatting

We presenteren een nieuwe software-toolkit bestemd voor het construeren en manipuleren van reguliere "dingen", zoals reguliere talen, reguliere expressies, en eindige automaten. De toolkit, genaamd FIRE Works, verschilt van andere toolkits in hetzelfde gebied door consistentie af te dwingen tussen de verschillende benaderingen van reguliere talen.

Op basis van de toolkit hebben we een software toepassing ontwikkeld: FIRE Station. Deze stelt de gebruiker in staat om de graaf structuren, waarop FIRE Works is gebaseerd, visueel weer te geven en te manipuleren. We presenteren een formele beschrijving, architectuur en detail ontwerp van zowel de toolkit, als de toepassing.

'Give a man a fire and he's warm for a day, but set fire to him and he's warm for the rest of his life.'
– Terry Pratchett

Contents

List of Figures

Preface

This document presents the master's thesis for my study *Technische Informatica* at the Technische Universiteit Eindhoven. The research and practical work that is part of this thesis took place from January 2003 to May 2004 within the *Software Construction (SoC)* group of the Department of Mathematics and Computing Science's Division of Computing Science. During this period, I was supervised by Prof. Dr. Bruce W. Watson, head of the group.

Acknowledgements

The research in this thesis was performed while I was immigrating to the United States of America. I would like to thank my advisor, Bruce Watson, for being flexible in allowing me to perform the research remotely, work at my own pace and for managing to fit a few visits to Berkeley, California into his busy schedule.

I thank Derrick Kourie and Alex Telea for being on my thesis committee and providing valuable comments and feedback on my thesis and this report.

I want to thank my wife, Diana, whom I met shortly before starting on this project, and my parents, for their love and support during my research.

Finally I would like to thank my friends from the time I was at the TU/e, as well as those from Berkeley. Especially, I would like to thank Loek Cleophas for all our brainstorming sessions, constructive criticism, and help in reviewing this report.

Michiel Frishert
Berkeley, March 2005

Part I

Prologue

Chapter 1

Introduction

In this chapter we describe the problem statement and its context, as well as an overview of the structure of this thesis report.

1.1 Problem Statement

Regular language algorithms have a wide range of applications. For example, they are used in computational linguistics, biotechnology, network intrusion detection, compilers and data compression. Having been around since the 1960's, it is one of the older research topics in computer science, and there exists a large body of literature on the topic. For a structured overview of many of the algorithms, see Watson's "Taxonomy and Toolkits of Regular Language Algorithms" [Wat95].

Two concepts are central to regular language algorithms: *regular expressions*, which can be used to symbolically write down (infinite) regular languages in a finite way, and *finite state automata*, which can be used by a computer to recognize elements of a regular language. Since regular expressions are easy for human operators to write, but computers work more readily with finite automata, most regular language algorithms are concerned with (some aspect of) converting a regular expression into a finite state automaton, or with making those automata more efficient.

There exist quite a few toolkits that implement regular language algorithms, for example FIRE Engine [Wat95], Vaucanson [LPRGS03], FSM [MPR98], FSA [vN97]. There is also a number of tools and applications to visualize finite automata, such as JFLAP [CFR04], Vaucanson-G [LS02] and Graphviz [GN00].

The toolkits have one property in common: once a finite state automaton has been generated from a regular expression, it is very hard to automatically retrieve a human-readable regular expression. Finding the original regular expression from which the automaton was constructed is impossible, since the number of regular expressions yielding the same automaton is infinite.

It is technically possible and straightforward to obtain a regular expression for any finite automaton, see for example [Brz64], [HU79]. However, it is a well known problem that the regular expressions obtained from these algorithms are generally (much) larger than strictly

necessary (we note the exception of the position automaton [CZ00]). We will call tools that do not retain their regular expressions *lossy*, and the opposite of lossy *loss-less*.

The terms lossy and loss-less are more commonly known in the field of computer graphics, giving us a good analogy. Computer graphics software can operate on *bitmaps* (a finite matrix of pixels that are either black or white), or with *vectors graphics* (a mathematical description of shapes). The vectors graphics can be scaled and manipulated over and over without losing any quality, while similar operations on a bitmap quickly degrade the image quality.

We have constructed a toolkit that can implement regular language algorithms in a loss-less fashion. This means that we keep a regular expression for each state of a finite automaton, once it is has been provided by the user, or computed from some regular language algorithm. On top of this toolkit we have implemented an application that uses graphs to visualize the results of the toolkit. The toolkit is entitled FIRE Works, the application FIRE Station. *FIRE* is short for FInite automata and Regular Expressions.

In this thesis we describe both the theoretical underpinnings and the software design and implementation details of both the toolkit and application

We have limited the scope of this thesis to manipulation of regular languages. Future extensions to both FIRE Works and FIRE Station may include, amongst others, string, multi-string and approximate string pattern matching, as well as regular relations and transducers.

1.2 Thesis Structure

This thesis is divided into three main parts. This chapter belongs to Part I, as does the next, which introduces the basic formalisms and notations used throughout the rest of the thesis. Additional notation will be introduced as needed in the other chapters.

Part II describes in detail the theoretical concepts of FIRE Works and FIRE Station, as well as providing formal specifications of various elements. Chapter 3 introduces the general concepts underpinning FIRE Works, specifically layers and nodes. Chapter 4 through 10 discuss various kinds of those layers. Chapter 11 provides a details on graph layout algorithms in FIRE Station.

Details on the implementation of FIRE Works and FIRE Station are provided in Part III. Chapter 13 discusses the high level architectural components, while Chapter 14 discusses the detailed software design. Chapter 15 goes into details on various implementation issues.

Finally, Chapter 16 of Part IV contains some concluding remarks, as well as providing some suggestions for future extensions.

Chapter 2

Preliminaries

In this chapter we introduce the basic definitions used throughout this thesis. Subsequent chapters will introduce thesis specific definitions as they become relevant.

2.1 Notations and Conventions

Convention 2.1. The following general naming conventions are used:

- Σ for the alphabet
- $a, b, c, \ldots$ for alphabet symbols
- i, n for natural numbers (elements of $\mathbb{N}$)
- x, y, z for strings over the alphabet
- u, v, w for nodes
- U for list of nodes
- V, W for set of nodes
- $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}$ for regular expressions

2.2 Basic definitions

Notation 2.2 (Quantifications). A basic understanding of the meaning of *quantifications* is assumed. We use the following notation:

$$(\oplus a : R(a) : f(a))$$

where $\oplus$ is the associative and commutative *quantification operator* (with unit $e_\oplus$), a is the *dummy variable* introduced, R is the *range predicate* on the dummy, and f is the *quantified expression*. By definition, we have:

$$(\oplus a : \mathit{false} : f(a)) = e_\oplus$$

The following table lists some of the most commonly quantified operators, their quantified symbols, and their units:

Operator	$\vee$	$\wedge$	$\cup$	min	max	$+$
Symbol	$\exists$	$\forall$	$\bigcup$	**MIN**	**MAX**	$\sum$
Unit	*false*	*true*	$\emptyset$	$+\infty$	$-\infty$	0

□

Notation 2.3 (Natural numbers). We use the symbol $\mathbb{N}$ to denote the set of all natural numbers. □

Notation 2.4 (Booleans). We use the symbol $\mathbb{B}$ to denote the set of boolean values *true* and *false*. □

Notation 2.5 (Function signatures). For any two sets A and B, we use $f \in A \to B$ to indicate that f is a total function from A to B. Set A is the *domain* of f while B is the *codomain* or *range* of f. □

Notation 2.6 (Function signatures). For any two sets A and B, we use $f \in A \to B$ to indicate that f is a total function from A to B. Set A is the *domain* of f while B is the *codomain* or *range* of f. □

Definition 2.7 (Powerset). For any set A we use $\mathcal{P}(A)$ (the *powerset* of A) to denote the set of all subsets of A. □

Definition 2.8 (Precedence of operators). The set operators in order of decreasing precedence are: $\times$, $\cap$, $\cup$. □

Definition 2.9 (Set of all sequences). To denote the set of all sequences over a set V, we use the suffix star operator: V^*. Note that we will use the word *list* interchangably for sequence. □

Definition 2.10 (Construction of a Sequence). For a set V, we construct a sequence $U \in V^*$ using square brackets as follows:

$$U = [u_0, u_1, \ldots, u_{n-1}]$$

We will write $|U|$ to denote the number of elements in a list. In this case $|U| = n$. The element at index i can be written through a subscript as U_i. □

Definition 2.11 (Sequence Concatenation). For lists over elements of a set V, concatenation $\triangleright \in V \times V^* \to V^*$ prefixes a list with a single item as follows for a $u \in V$ and $U \in V^*$:

$$u \triangleright U = [u, U_0, \ldots, U_{|U|-1}]$$

□

Definition 2.12 (Sequence Operators $\uparrow$, $\downarrow$, $\restriction$, $\llcorner$). For lists over elements of a set V we define four infix operators $\uparrow, \downarrow, \restriction, \llcorner \in V^* \times \mathbb{N} \to V^*$ as follows:

- $U \uparrow n$ is the $n \min |w|$ leftmost symbols of U
- $U \downarrow n$ is the $(|U| - n) \max 0$ rightmost symbols of U
- $U \uparrow n$ is the $n \min |w|$ rightmost symbols of U
- $U \downarrow n$ is the $(|U| - n) \max 0$ leftmost symbols of U

The four operators are pronounced 'left take', 'left drop', 'right take' and 'right drop' respectively. □

2.3 Regular Languages

Definition 2.13 (Alphabet). An alphabet is a non-empty set of *symbols* or *characters*. □

Definition 2.14 (Set of all strings). Given alphabet Σ, we define Σ^* as the set of all strings over Σ. □

Definition 2.15 (Empty string). We use the symbol ε to denote the string of length 0 (the empty string). □

Definition 2.16 (Language). Given alphabet Σ, we define a language L as $L \subseteq \Sigma^*$. □

Definition 2.17 (Concatenation of languages). We define language concatentation, denoted by the "dot" infix operator $\cdot \in \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ as:

$$L_0 \cdot L_1 = \{xy | x \in L_0 \wedge y \in L_1\}$$

□

Definition 2.18 (Language exponentiation). We define exponentiation of language L recursively as follows:

$$L^0 = \varepsilon$$

and for $k > 0$:

$$L^k = L \cdot L^{k-1}$$

□

Definition 2.19 (Closure operator on languages). We define postfix operators $*$ and $+$ over languages of alphabet Σ.

Operator $^* \in \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ (Kleene closure or star closure) is:

$$L^* = (\cup i : 0 \leq i : L^i)$$

Operator $^+ \in \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ (plus closure) is:

$$L^+ = (\cup i : 1 \leq i : L^i)$$

Note that $L^* = L^+ \cup \{\varepsilon\}$. □

Definition 2.20 (Intersection, Relative Difference, Symmetric Difference on Languages). We define intersection, relative difference and symmetric difference of languages L_0, L_1 over alphabet Σ.

Operator $\cap \in \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ (intersection) is:

$$L_0 \cap L_1 = \{x | x \in L_0 \wedge x \in L_1\}$$

Operator $\setminus \in \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ (relative difference) is:

$$L_0 \setminus L_1 = \{x | x \in L_0 \wedge x \notin L_1\}$$

Operator $\triangle \in \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ (symmetric difference) is:

$$L_0 \triangle L_1 = \{x | x \in L_0 \cup L_1 \wedge x \notin L_0 \cap L_1\}$$

Note that relative difference and symmetric difference are not distributive, and when omitting parenthesis, we will interpret from left to right, e.g. $L_1 \setminus L_2 \setminus L_3 = ((L_1 \setminus L_2) \setminus L_3)$ □

Definition 2.21 (Unary language operators $\neg$ and ?). We define prefix operator $\neg$ and postfix operator $^?$ over languages of alphabet Σ.

Operator $\neg \in \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ (negation) is defined as:

$$\neg L = \Sigma^* \setminus L$$

Operator $^? \in \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ (optional) is defined as:

$$L^? = L \cup \{\varepsilon\}$$

□

Definition 2.22 (Regular Languages). We define the set of regular languages REG over alphabet Σ inductively:

- $\emptyset \in REG$
- $\{\varepsilon\} \in REG$
- For $a \in \Sigma, \{a\} \in REG$

For languages $L_0, L_1 \in REG$:

- $L_0^* \in REG$
- $L_0 \cdot L_1 \in REG$
- $L_0 \cup L_1 \in REG$

Nothing else is in REG. □

2.4 Automata

In this section we define automata and several of their properties.

Definition 2.23 (Finite Automaton). A finite automaton (FA) A is a 5-tuple $A = (Q, \Sigma, \delta, S, F)$, where:

- Q is a finite set of states.
- Σ is an alphabet.
- $\delta \in Q \times \Sigma \to \mathcal{P}(Q)$ is a transition function.
- $S \subseteq Q$ is a set of start states.
- $F \subseteq Q$ is a set of final states.

□

Definition 2.24 (Deterministic FA). An automaton $A = (Q, \Sigma, \delta, S, F)$ is deterministic if and only if:

- There is at most one start state: $|S| \leq 1$
- It is ε free
- The transition function maps each state/symbol pair to at most one state: $(\forall q, a : q \in Q \wedge a \in \Sigma : |\delta(q, a)| \leq 1)$

□

Definition 2.25 (Extending the transition relation δ). We extend the transition function $\delta \in Q \times \Sigma \to Q$ to $\delta^* \in Q \times \Sigma^* \to Q$ for $q \in Q, a \in \Sigma, x \in \Sigma^*$ as follows:

$$\delta^*(q, a) = \delta(q, a)$$

$$\delta^*(q, ax) = \delta^*(\delta(q, a), x)$$

□

Definition 2.26 (Language of an FA). The language of a finite automaton $A = (Q, \Sigma, \delta, S, F)$ is given by the function $\mathcal{L}_{FA} \in FA \to \mathcal{P}(\Sigma^*)$ defined as:

$$\mathcal{L}_{FA}(A) = \{x | (\exists s : s \in S : \delta^*(s, x) \in F)\}$$

□

2.5 Graphs

Definition 2.27 (Graphs and Directed Graphs). A graph $G = (V, E)$ consists of a finite set V of vertices, and a finite multiset E of edges, that is, unordered pairs (u, v) of vertices. A graph is *directed* in which the edges in E are ordered. □

Notation 2.28 (Edges of a Graph). We will write $(u, v) \in G$ iff $G = (V, E) \wedge u, v \in V \wedge (u, v) \in E$ □

Definition 2.29 (Paths and Cycles). A (directed) path in a (directed) graph $G = (V, E)$ is a sequence $(v_0, v_2, \ldots, v_n)$ of distinct vertices of V, such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$. If $(v_n, v_0) \in E$, then the (directed) path is a (directed) cycle. A directed graph is acyclic if it has no directed cycles. □

Definition 2.30 (Transitive Closure of a Graph). An edge (u, v) of a directed graph is *transitive* if there is a directed path from u to v that does not contain the edge (u, v). The *transitive closure* of a directed graph G is G^* and has an edge (u, v) for every path from u to v in G. □

Definition 2.31 (Subgraph and Induced Graph). A graph $G' = (V', E')$, such that $V' \subseteq$ and $E' \subset E \cap (V' \times V')$, is a *subgraph* of $G = (V, E)$. If $E' = E \cap (V' \times V')$, then G' is *induced by V'*. □

Definition 2.32 (Connected Component). A graph is *connected* if there is a path between u and v for each pair (u, v) of vertices. A maximal connected subgraph of a graph G is a *connected component*. □

Part II

FIRE Works and FIRE Station Concepts

Chapter 3

FIRE Works & FIRE Station

In this chapter we will introduce the basic concepts behind FIRE Works and FIRE Station.

3.1 Introduction

FIRE Works and FIRE Station are respectively a toolkit and application for working with regular languages. The abbreviation *FIRE* is short for "FInite automata and Regular Expression", which has previously been used for FIRE Engine and FIRE Lite, see [Wat95] for details. FIRE Works is a toolkit consisting of object oriented data structures and algorithms, while FIRE Station is a graphical application that visualizes those data structures using graphs.

Regular languages are those languages that can be generated from a set of symbols (letters) by the basic operations of concatenation, selection (union) and repetition, applied in any order. Due to the repetition operation, regular languages can be infinite, and it is much more convenient to work with them indirectly through *regular expressions* and *finite state automata*, which can symbolically describe such infinite languages in a finite manner.

These symbolic representations are implemented in underlying data structures, and we specifically use graph data structures. For example, for regular expressions we use parse trees, and for finite state automata we use transition graphs. We can visualize these graph data structures, and this is what is done in FIRE Station. For example the parse tree of a regular expression is drawn in Figure 3.1, while Figure 3.2 shows the transition graph of a finite state automaton.

What is unique about FIRE Works is that it shares the same set of nodes for both the parse tree representation of a regular expression, and the transition graph representation of a finite automaton. As a result, we have a parse tree as well as a finite automaton for every node, and we can visualize either as in the last two figures, or we can show both at the same time, as in Figure 3.3. In this figure we show both the edges for the parse tree (solid lines), and the edges for the automaton (dashed lines). To ensure a complete picture both for the transition graph and the parse tree, some additional nodes need to be drawn that were not shown in the previous two figures.

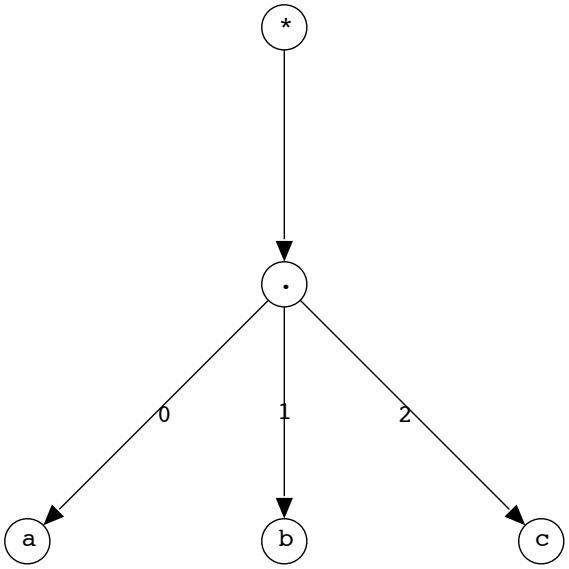

Figure 3.1: Parse Tree of Regular Expression $(\mathsf{abc})^*$

With this new graph rendering, it becomes possible to see relations between regular expressions and automata that were not visible before. For example, we can see if a subexpression of a regular expression also happens to be a state in the finite automaton associated for that regular expression.

How these combined graphs are created and rendered will be discussed in Chapter 11, p. 61.

3.2 Nodes

The graph datastructures in FIRE Works consist of nodes and edges. Each node, along with its properties and its successors in the graph datastructures, represent a regular language. As a matter of shorthand we say that each node represents a regular language. The languages are not directly stored for each node (since they might be infinite), but instead defined by a regular expression or automaton, which in turn are represented by graphs on these nodes. We formalize this by introducing the function

$$\mathcal{L}_{node} \in V \to REG$$

We use V to represent the set of all nodes. It is allowed for two nodes to have the same regular language.

3.3 Layers

Because symbolic representations of regular expressions are more convenient to work with, algorithms on regular languages are commonly described in terms of operations on symbolic

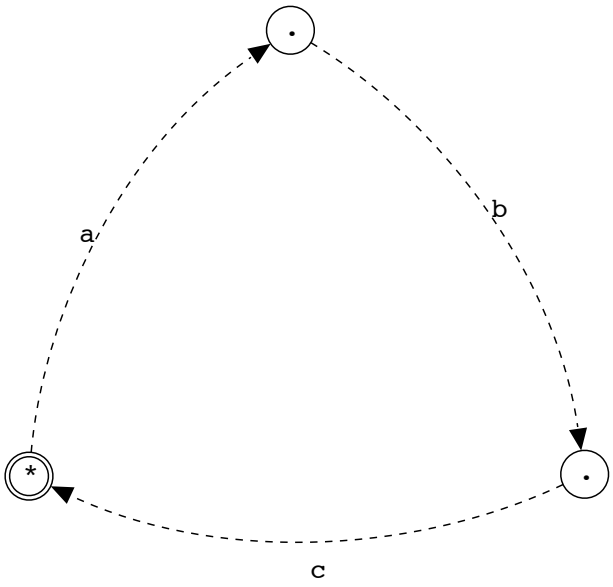

Figure 3.2: Graph Representation of an automaton recognizing (abc)*

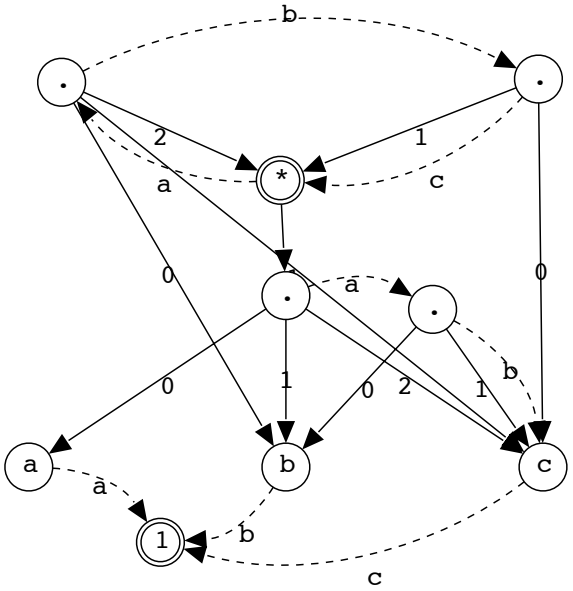

Figure 3.3: Combined graph view of the parse tree and automaton transitions of (abc)*

representations (regular expressions and automata) of those languages. To distinguish between these algorithms, we introduce the concept of *layers*. Each layer corresponds to an algorithm on regular languages (again: the symbolic representation thereof). The name *layer* was chosen because the algorithms incrementally build on top of one another, that is, they are *layered* on top of each other. Additionally, FIRE Station visually represents these layers as graphs, which are drawn on top of each other, in a layered fashion. The graph visualization tool GraphViz [GN00] uses the name layering for this same way of drawing.

Figure 3.4 illustrates how the layers currently available in FIRE Works build on top of one another. We require that any layer can only depend on the layers below it, which means the dependencies are acyclical. This requirement keeps the overall design simple, and seems to correspond well to the natural structure of the algorithms were dealing with: we compute some basic property, and from there we can compute a more complex property; the outcome of which however does not affect the basic property.

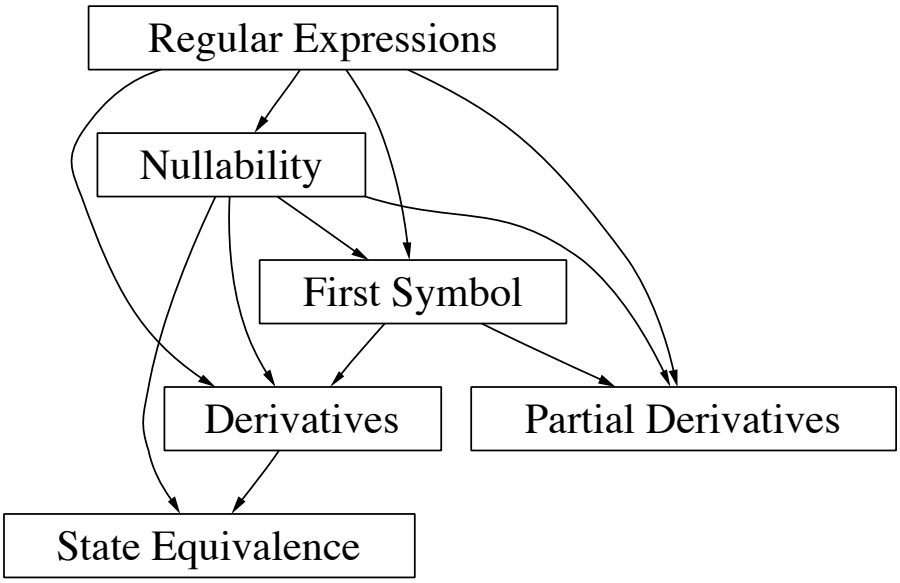

Figure 3.4: The Layers in FIRE Works and their dependencies

The choice of the regular expressions layer as the root layer, rather than an automaton layer, is not arbitrary, but follows from the natural flow of automaton construction: the user creates a regular expression and for this expression a finite automaton is computed. Furthermore, if we choose an automaton layer as the root layer, we would have to compute a regular expression for the automaton that the user creates. The current state-of-the-art in reconstructing regular expressions from finite automata is inadequate for this task. But even if good algorithms were available, they would work in a batch fashion, since a small change to the automaton might result in a major change in the equivalent regular expression. This in contrast with the construction of regular expression, which can be created in a bottom-up fashion, first creating the subexpressions and from these subexpressions creating bigger expressions. From these subexpressions we can already compute partial automata, which will be useful regardless of the way the subexpressions are combined into bigger expressions.

Since we want the system to be consistent, we want every node to have meaning to every layer. This requires the layer dependency graph to have a single root layer, the Regular Expression Layer, and all nodes are "created" via that layer. In this manner, a regular expression is defined for all nodes, at all times. As we will see, in some cases this requires adaptations to existing algorithms.

Note that the regular language of a node can be computed from some layers, but not others; more formally, the function $\mathcal{L}_{node}$ can defined in terms of some layers. For example the Regular Expression Layer, introduced in Chapter 4, p. 31, provides a regular expression for each node, from which we can compute the regular language as per Definition 4.1, p. 31. In other cases a combination of layers can be used to compute the regular language of a node. For example the Nullable Layer combined with the Derivatives Layer (Chapter 5, p. 37 and Chapter 7, p. 45 respectively) can be interpreted as an automaton, which is sufficient to compute the regular language of a node as per Definition 2.26, p. 21.

3.4 Graphs

As mentioned in Section 3.2, the symbolic representations (regular expressions and finite automata) are implemented as graphs on the nodes of V. For the regular expression this graph is the parse tree of the regular expression. For finite automata this is the transition graph on the nodes in V.

In addition to layers that are represented by graphs, there are layers that represent properties of regular languages. In the graph representation, these are mapped into node labels. When not clear from the context, we will refer to layers that define relations between regular languages as *relational layers*, and to the layers defining properties of regular languages as *property layers*.

In subsequent chapters, a mapping from the graph representations, back to the symbolic layer, and ultimately back to regular languages will be given for each layer.

Note that it is the graphs that are ultimately visualized in FIRE Station, as was already illustrated in Figures 3.1, 3.2 and 3.3. In the visualization the edges are drawn with different styles, text labels and colors to differentiate between the layers which they represent. The property layers provide visual attributes to the nodes themselves, for example shape, text labels and color.

3.5 Conclusion

In Table 3.5 we recapture the increasing refinement, from regular languages to graph representations.

Most of the remaining chapters in this part discuss one particular layer. For each layer we provide an informal discussion, followed by a formal description at both the regular language and regular expression level (where appropriate). Then we discuss the implementation in terms of graphs.

Abstraction Level	**Representation**
Regular Languages	Set Theory
Regular Expressions, Automata	Symbolic Representations
Parse Tree, Transition Graphs	High Performance Data Structures

Table 3.1: Abstraction Levels used in FIRE Works and FIRE Station

Chapter 4

Regular Expression Layer

4.1 Introduction

The Regular Expression Layer contains a regular expression for every node. From the regular expression of a node, the regular language of that node can be computed. Most algorithms on regular languages work on the regular expression representation rather than on the language directly, because they can deal with infinite languages. In turn, many algorithms are implemented most easily on a parse tree representation of such regular expressions, instead of directly on the regular expression. Therefore, the regular expressions on the Regular Expression Layer are represented by parse trees.

The Regular Expression Layer is the most fundamental layer in FIRE Works: it does not rely on any other layers, but all other layers rely on it, either directly or indirectly. We require that for any node that is create in FIRE Works, the regular expression is available.

In contrast with the binary parse trees that are often found in the literature, the Regular Expression Layer uses n-ary trees, by which we mean trees whose nodes can have any number of child nodes. The reason for this is that some algorithms can be implemented much more efficiently on n-ary trees (for example common subexpression elimination and rewriting, both of which will be discussed in Chapter 12, p. 67). All existing algorithms that work on binary trees can be made to work with a n-ary tree.

Note that besides the traditional parse tree, the Regular Expression Layer can also contain parse graphs (specifically directed acyclic graphs, or DAGs), also discussed in Chapter 12. Since we interpret these parse graphs as if they were trees, we will keep using the term *tree*.

4.2 Formal Description

We start by giving a formal definition of regular expressions.

Definition 4.1 (Regular Expressions). We define the set of regular expressions RE over alphabet Σ. Note that most of the regular operators have similar appearance to those used for operations on regular languages; we will use a sans-serif font style for regular expressions to distinguish them from regular languages.

- $\emptyset \in RE$ and $\mathcal{L}_{RE}(\emptyset) = \emptyset$
- $\varepsilon \in RE$ and $\mathcal{L}_{RE}(\varepsilon) = \{\varepsilon\}$
- For all $a \in \Sigma, \mathsf{a} \in RE$ and $\mathcal{L}_{RE}(\mathsf{a}) = \{a\}$

For $\mathsf{E}, \mathsf{F} \in RE$

- $\mathsf{E} \cup \mathsf{F} \in RE$ and $\mathcal{L}_{RE}(\mathsf{E} \cup \mathsf{F}) = \mathcal{L}_{RE}(\mathsf{E}) \cup \mathcal{L}_{RE}(\mathsf{F})$
- $\mathsf{E} \cdot \mathsf{F} \in RE$ and $\mathcal{L}_{RE}(\mathsf{E} \cdot \mathsf{F}) = \mathcal{L}_{RE}(\mathsf{E}) \cdot \mathcal{L}_{RE}(\mathsf{F})$
- $\mathsf{E}^* \in RE$ and $\mathcal{L}_{RE}(\mathsf{E}^*) = (\mathcal{L}_{RE}(\mathsf{E}))^*$
- $\mathsf{E}^+ \in RE$ and $\mathcal{L}_{RE}(\mathsf{E}^+) = (\mathcal{L}_{RE}(\mathsf{E}))^+$
- $\mathsf{E}^? \in RE$ and $\mathcal{L}_{RE}(\mathsf{E}^?) = \mathcal{L}_{RE}(\mathsf{E})^?$
- $\mathsf{E} \setminus \mathsf{F} \in RE$ and $\mathcal{L}_{RE}(\mathsf{E} \setminus \mathsf{F}) = \mathcal{L}_{RE}(\mathsf{E}) \setminus \mathcal{L}_{RE}(\mathsf{F})$
- $\neg\mathsf{E} \in RE$ and $\mathcal{L}_{RE}(\neg\mathsf{E}) = \neg\mathcal{L}_{RE}(\mathsf{E})$
- $\mathsf{E} \cap \mathsf{F} \in RE$ and $\mathcal{L}_{RE}(\mathsf{E} \cap \mathsf{F}) = \mathcal{L}_{RE}(\mathsf{E}) \cap \mathcal{L}_{RE}(\mathsf{F})$
- $\mathsf{E} \triangle \mathsf{F} \in RE$ and $\mathcal{L}_{RE}(\mathsf{E} \triangle \mathsf{F}) = \mathcal{L}_{RE}(\mathsf{E}) \triangle \mathcal{L}_{RE}(\mathsf{F})$

□

Our definition of regular expression is often considered the *extended* regular expressions. In contrast, the *basic* regular expressions only contain the Kleene closure, concatenation and union operator, but not the plus closure, optional, negation, intersection, symmetric difference or relative difference operators. In this report we will be using the extended definition, unless stated otherwise.

Union, intersection, symmetrical difference and relative difference can be seen as binary logical operations. In addition to these four, there are 12 other logical operations, for a total of 16. Each of the 16 logical operations is uniquely defined by a truth table mapping two booleans into a single boolean result. We will not be using the other 12 operators, since they are uncommon and equivalent regular expressions can easily be constructed with the operators provided. The symbol ε is used for both the empty string, and for the regular expression representing the regular language containing only the empty string. In similar vein, $\emptyset$ is used both to represent the empty regular language and the regular expression representing that language. In both situations it will be clear from the contexts which interpretation is used.

The regular expression a is equivalent to the regular language $\{a\}$, where $a \in \Sigma$. We will go back and forth between symbol and regular expression simply by changing the font style; the intended use will be clear from the font style. Note that the infix dot operator for concatenation of regular expressions can be omitted, for example we take regular expression abc to mean $\mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c}$.

4.3 Implementation

In this section we describe the representation of regular expressions through their parse trees. We will also provide functions that map back from these parse trees to regular expressions.

First, we associate each node with a label. Each of these labels corresponds either to a regular constant ($\emptyset, \varepsilon$), to symbols from the alphabet Σ, or to a regular operator ($^*, \cup, \cdot, \ldots$). The labels determine how each node has to be interpreted.

Definition 4.2 (Parse Tree Node Labels). The node labels defined by the parse tree:

$$nodelabels = \{[\mathsf{a}]|a \in \Sigma\} \cup \{[\emptyset], [\varepsilon], [?], [*], [+], [\cup], [\neg], [\cdot], [\cap], [\triangle], [\setminus]\}$$

□

Note that in Definition 4.2 we have implicitly defined a conversion from symbol to node label, e.g. for a symbol $a \in \Sigma$, we define a node label $[\mathsf{a}]$. We will also use the reverse: given a node label $[\mathsf{a}]$, we will extract the corresponding symbol a simply by removing the brackets and changing the font style.

Definition 4.3 (Node Label of a Node). The label for a particular node is given by

$$label \in V \rightarrow nodelabels$$

□

For regular operators that operate on arguments, we represent the arguments in the parse tree as child nodes. The regular operators are either unary or binary, while we have said that our parse tree is n-ary.

We distinguish two types of binary regular operators, those that are commutative, and those that are not. For example, the union operator is commutative: $\mathsf{E} \cup \mathsf{F} = \mathsf{F} \cup \mathsf{E}$; the concatenation operator is not: $\mathsf{E} \cdot \mathsf{F} \neq \mathsf{F} \cdot \mathsf{E}$

For the n-ary tree, this means that arguments to a commutative operator can be stored in an unordered set, while arguments to non-commutative operators need to be stored in an ordered list.

In Definition 4.4, we define three classes of node labels, which correspond to the unary, commutative binary, and non-commutative binary regular operators.

Definition 4.4 (Sets of node labels). We define one set of node labels for the symbols and three sets of node-labels for non-leaf nodes based on operator arity and commutativity:

- $symbollabels = \{[\mathsf{a}]|a \in \Sigma\}$
- $unarylabels = \{[?], [*], [+], [\neg]\}$
- $setlabels = \{[\cup], [\cap]\}$
- $listlabels = \{[\cdot], [\setminus]\, [\triangle]\}$

Note that only *symbollabels* describes labels for leaf-nodes; the other sets all describe non-leaf labels. □

Next we define three functions to access the parse tree structure in a convenient manner. Each function describes the structure of nodes belonging to one of the three types of non-leaf node labels from Definition 4.4

Definition 4.5 (Parse tree structure). We define the parse tree structure via three functions:

- $child \in V \to V$ for $label(V) \in unarylabels$
- $childset \in V \to \mathcal{P}(V)$ for $label(V) \in setlabels$
- $childlist \in V \to V^*$ for $label(V) \in listlabels$

□

Using the previous definitions of the parse tree, we can now retrieve the regular expression for any node.

Definition 4.6 (Parse Tree to Regular Expression). For a node $v \in V$, we define a mapping $regex \in V \to RE$, which constructs the regular expression represented by the parse tree rooted at that node:

$label(v)$	$regex(v)$
$[\emptyset]$	$\emptyset$
$[\varepsilon]$	ε
[a]	a
[?]	$regex(child(v))^?$
$[*]$	$regex(child(v))^*$
$[+]$	$regex(child(v))^+$
$[\neg]$	$\neg regex(child(v))$
$[\cup]$	$regex(W_0) \cup \ldots \cup regex(W_{\lvert W\rvert-1})$ where $W = childset(v)$
$[\cdot]$	$regex(U_0) \cdot \ldots \cdot regex(U_{\lvert U\rvert-1})$ where $U = childlist(v)$
$[\cap]$	$regex(W_0) \cap \ldots \cap regex(W_{\lvert W\rvert-1})$ where $W = childset(v)$
$[\triangle]$	$regex(U_0) \triangle \ldots \triangle regex(W_{\lvert W\rvert-1})$ where $U = childlist(v)$
$[\setminus]$	$regex(U_0) \setminus \ldots \setminus regex(U_{\lvert U\rvert-1})$ where $U = childlist(v)$

□

We can now define the language for any particular node, $\mathcal{L}_{node}$, via the function *regex*, which yields a regular expression for each node, from which we can compute a regular language as described in Definition 4.1, p. 31.

Definition 4.7 (Regular Language via Regular Expression Layer). For a node $v \in V$, the regular language represented by that node, $\mathcal{L}_{node} \in V \to \mathcal{P}(\Sigma^*)$ is given by:

$$\mathcal{L}_{node}(v) = \mathcal{L}_{RE}(regex(v))$$

□

For the creation of nodes, we straightforwardly define a function for each node label, taking appropriate paramaters for the particular regular operator.

Definition 4.8 (Creating Regular Expression Nodes). The Regular Expression Layer provides a function to create a node for a given operator and parameter. We first give the signature for each of these functions:

- $mknode_{[\emptyset]}, mknode_{[\varepsilon]} \in V$
- $mknode_{\Sigma} \in \Sigma \to V$
- $mknode_{[?]}, mknode_{[*]}, mknode_{[+]}, mknode_{[\neg]} \in V \to V$
- $mknode_{[\cup]}, mknode_{[\cap]}, mknode_{[\triangle]} \in \mathcal{P}(V) \to V$
- $mknode_{[\cdot]}, mknode_{[\setminus]} \in V^* \to V$

Next we define the functions through the regular expression represented by the node they create:

if v is the result of	then $regex(v)$ equals
$mknode_{[\emptyset]}$	$\emptyset$
$mknode_{[\varepsilon]}$	ε
$mknode_{\Sigma}(a)$	a
$mknode_{[?]}(w)$	$regex(w)^?$
$mknode_{[*]}(w)$	$regex(w)^*$
$mknode_{[+]}(w)$	$regex(w)^+$
$mknode_{[\neg]}(w)$	$\neg regex(w)$
$mknode_{[\cup]}(\{w_0, \ldots, w_n\})$	$regex(w_0) \cup \ldots \cup regex(w_n)$
$mknode_{[\cdot]}([w_0, \ldots, w_n])$	$regex(w_0) \cdot \ldots \cdot regex(w_n)$
$mknode_{[\cap]}(\{w_0, \ldots, w_n\})$	$regex(w_0) \cap \ldots \cap regex(w_n)$
$mknode_{[\triangle]}(\{w_0, \ldots, w_n\})$	$regex(w_0) \triangle \ldots \triangle regex(w_n)$
$mknode_{[\setminus]}([w_0, \ldots, w_n])$	$regex(w_0) \setminus \ldots \setminus regex(w_n)$

□

From the *mknode* functions of Definition 4.8 we observe that parse trees have to be created in a bottom-up fashion, because we have to specify the child node(s) for each new node to be created.

4.4 Parse Tree Graphs

Although the parse tree is technically already a graph, we describe here the conversion to a generic graph, which can be used for visualization purposes in FIRE STATION. The nodes of this graph are the same nodes that are in V, so we only have to define the edges on these nodes.

Definition 4.9 (Parse Graph Edges). We define the edges for the parse graph E_{RE}, as follows:

$$\begin{aligned} E_{RE} \quad = \quad & \{(v,w) | label(v) \in \mathit{unarylabels} \wedge w = \mathit{child}(v)\} \cup \\ & \{(v,w) | label(v) \in \mathit{setlabels} \wedge w \in \mathit{childset}(v)\} \cup \\ & \{(v,w) | label(v) \in \mathit{listlabels} \wedge (\exists i : i \in \mathbb{N} : w = \mathit{childlist}(v)_i)\} \end{aligned}$$

□

Chapter 5

Nullable Layer

5.1 Introduction

The nullable layer determines whether the regular language represented by a node contains the empty string, that is the string containing no symbols, represented by ε.

5.2 Formal Description

Definition 5.1 (Nullability of a Regular Language). Nullability of a regular language is defined as $null_{REG} \in REG \to \mathbb{B}$:

$$null_{REG}(L) \equiv \varepsilon \in L$$

□

Definition 5.2 (Nullability of a Regular Expression). Given regular expressions $\mathsf{E}, \mathsf{F} \in RE$, nullability of a regular expression $null_{RE} \in RE \to \mathbb{B}$ is defined as:

- $null_{RE}(\emptyset) \equiv false$
- $null_{RE}(\varepsilon) \equiv true$
- $null_{RE}(\mathsf{a}) \equiv false$ (for $a \in \Sigma$)
- $null_{RE}(\mathsf{E}^?) \equiv true$
- $null_{RE}(\mathsf{E}^*) \equiv true$
- $null_{RE}(\mathsf{E}^+) \equiv null_{RE}(\mathsf{E})$
- $null_{RE}(\neg\mathsf{E}) \equiv \neg null_{RE}(\mathsf{E})$
- $null_{RE}(\mathsf{E} \cup \mathsf{F}) \equiv null_{RE}(\mathsf{E}) \vee null_{RE}(\mathsf{F})$
- $null_{RE}(\mathsf{E} \cdot \mathsf{F}) \equiv null_{RE}(\mathsf{E}) \wedge null_{RE}(\mathsf{F})$

- $null_{RE}(\mathsf{E} \cap \mathsf{F}) \equiv null_{RE}(\mathsf{E}) \wedge null_{RE}(\mathsf{F})$
- $null_{RE}(\mathsf{E} \triangle \mathsf{F}) \equiv null_{RE}(\mathsf{E}) \wedge null_{RE}(\mathsf{F})$
- $null_{RE}(\mathsf{E} \setminus \mathsf{F}) \equiv null_{RE}(\mathsf{E}) \wedge \neg null_{RE}(\mathsf{F})$

□

5.3 Implementation

We refine $null_{RE}$ to compute nullability directly from the parse tree of a node.

Definition 5.3 (Nullability of a node). Nullability of a node $v \in V$ computed via the Regular Expression Layer $null \in V \to \mathbb{B}$ satisfies:

$$null(v) \equiv null_{RE}(regex(v))$$

and is defined for each possible value of $label(v)$ as follows:

$label(v)$	$null(v)$
$[\emptyset]$	$false$
$[\varepsilon]$	$true$
$[\mathsf{a}]$	$false$
$[?]$	$true$
$[*]$	$true$
$[+]$	$null(child(v))$
$[\neg]$	$\neg null(v)$
$[\cup]$	$(\exists w : w \in childset(v) : null(w))$
$[\cdot]$	$(\forall i : 0 \leq i < \lvert W \rvert : null(W_i))$ (where $W = childlist(v)$)
$[\cap]$	$(\forall w : w \in childset(v) : null(w))$
$[\triangle]$	$(\#w : w \in childset(v) : null(w)) = 1$
$[\setminus]$	$null(U_0) \wedge (\forall i : 0 < i < \lvert U \rvert : \neg null(U_i))$ (where $U = childlist(v)$)

□

In finite automata, states that are nullable are the states that are accepting. Since traditionally those states are drawn as two concentric circles, we represent the nullable property in that way also.

We define the auxiliary function *lastnull*, which will be useful for both the First Symbol Layer (Chapter 6 on page 41) and the Derivatives Layer (Chapter 7 on page 45).

Definition 5.4 (Function *lastnull*). Function $lastnull \in V^* \to \mathbb{N}$ returns the greatest index of a list of nodes $U \in V^*$ for which all nodes at earlier indexes are nullable. If all nodes are nullable it returns the index of the last node in the list.

$$\begin{aligned} lastnull(U) \quad = \quad & (\mathbf{MAX} n : n \leq |U| : (\forall i : i < n : null(U_i)) \wedge \\ & (n = |U| - 1 \vee \neg null(U_n))) \end{aligned}$$

□

Chapter 6

First Symbol Layer

6.1 Introduction

The set of first symbols of a (regular) language are those symbols that are a prefix of one or more strings in the language.

In FIRE WORKS, the set of first symbols is used to compute the derivatives and partial derivatives (which will be introduced in Chapter 7, p. 45 and Chapter 8, p. 51 respectively).

Unfortunately there is no known method of computing the exact set of first symbols of a regular language from a regular expression representing that language. The problem can easily be seen for the regular expression $\mathsf{ab} \setminus \mathsf{ab}$. Obviously the language of this regular expression is empty, and therefore the set of first symbols is empty. However if we replace ab by more complex regular expressions that are not identical, but represent identical languages, then it is not possible to determine whether or not a is in the first symbol set directly from the regular expression.

One way to compute the exact set of first symbols is to compute an automaton for the regular expression. From this automaton, remove all dead states. The set of first symbols is the set of symbols for which the initial state has outgoing transitions. However, this defeats the main purpose of the first symbol layer, which is to aid in the construction of an automaton (e.g. via the derivatives layer, Chapter 7, p. 45)

Instead, we will have to satisfy ourselves in computing a slightly larger set of firstsymbols for regular expressions, which contains the first symbols of the regular language, but is better than taking the entire alphabet.

6.2 Formal Description

We first define $\mathit{firstsym}_{REG}$, which computes the first symbol set for a given language.

Definition 6.1 (First symbols of a language). For a language $L \subseteq \Sigma^*$ the set of first symbols for that language, $\mathit{firstsym}_{REG} \in REG \to \mathcal{P}(\Sigma)$, is defined as:

$$\mathit{firstsym}_{REG}(L) = \{a | ax \in L\}$$

□

The definition of first symbols on regular expressions, $\mathit{firstsym}_{RE}$ computes a set of symbols that satisfies the condition:

$$\mathit{firstsym}_{REG}(\mathcal{L}_{RE}(\mathsf{E})) \subseteq \mathit{firstsym}_{RE}(\mathsf{E})$$

i.e. the first symbol set of a regular expression contains at least the set of first symbols of the regular language represented by that regular expression.

Definition 6.2 (First symbols of a regular expression). The first symbol on regular expressions is defined as $\mathit{firstsym}_{RE} \in RE \to \mathcal{P}(\Sigma)$ for any node $v \in V$:

- $\mathit{firstsym}_{RE}(\emptyset) = \emptyset$
- $\mathit{firstsym}_{RE}(\varepsilon) = \emptyset$
- $\mathit{firstsym}_{RE}(\mathsf{a}) = \{a\}$
- $\mathit{firstsym}_{RE}(\mathsf{E}^?) = \mathit{firstsym}_{RE}(\mathsf{E})$
- $\mathit{firstsym}_{RE}(\mathsf{E}^*) = \mathit{firstsym}_{RE}(\mathsf{E})$
- $\mathit{firstsym}_{RE}(\mathsf{E}^+) = \mathit{firstsym}_{RE}(\mathsf{E})$
- $\mathit{firstsym}_{RE}(\neg\mathsf{E}) = \mathit{firstsym}_{RE}(\mathsf{E})$
- $\mathit{firstsym}_{RE}(\mathsf{E} \cup \mathsf{F}) = \mathit{firstsym}_{RE}(\mathsf{E}) \cup \mathit{firstsym}_{RE}(\mathsf{F})$
- $\mathit{firstsym}_{RE}(\mathsf{E} \cdot \mathsf{F}) = \mathit{firstsym}_{RE}(\mathsf{E})$
  (if $\neg \mathit{null}_{RE}(\mathsf{E})$)
- $\mathit{firstsym}_{RE}(\mathsf{E} \cdot \mathsf{F}) = \mathit{firstsym}_{RE}(\mathsf{E}) \cup \mathit{firstsym}_{RE}(\mathsf{F})$
  (if $\mathit{null}_{RE}(\mathsf{E})$)
- $\mathit{firstsym}_{RE}(\mathsf{E} \cap \mathsf{F}) = \mathit{firstsym}_{RE}(\mathsf{E}) \cap \mathit{firstsym}_{RE}(\mathsf{F})$
- $\mathit{firstsym}_{RE}(\mathsf{E} \triangle \mathsf{F}) = \mathit{firstsym}_{RE}(\mathsf{E}) \cup \mathit{firstsym}_{RE}(\mathsf{F})$
- $\mathit{firstsym}_{RE}(\mathsf{E} \setminus \mathsf{F}) = \mathit{firstsym}_{RE}(\mathsf{E})$

□

6.3 Implementation

We refine Definition 6.2 to operate directly on the parse tree representation of the regular expressions as provided by the Regular Expression Layer.

Definition 6.3 (First symbol via Parse Tree). For any node $v \in V$, the function $\mathit{firstsym} \in V \to \mathcal{P}(\Sigma)$ satisfies the following condition:

$$\mathit{firstsym}(v) = \mathit{firstsym}_{RE}(\mathit{regex}(v))$$

and is defined as follows:

$\mathit{label}(v)$	$\mathit{firstsym}(v)$
$[\emptyset]$	$\emptyset$
$[\varepsilon]$	$\emptyset$
$[\mathsf{a}]$	$\{a\}$
$[?], [*], [+], [\neg]$	$\mathit{firstsym}(\mathit{child}(v))$
$[\cup]$	$(\bigcup w : w \in \mathit{childset}(v) : \mathit{firstsym}(w))$
$[\cdot]$	$(\bigcup i : 0 \leq i \leq \mathit{lastnull}(U) : \mathit{firstsym}(U_i))$ where $U = \mathit{childlist}(v)$
$[\triangle]$	$(\bigcup w \in \mathit{childset}(v) : \mathit{firstsym}(w))$
$[\cap]$	$(\bigcap w \in \mathit{childset}(v) : \mathit{firstsym}(w))$
$[\backslash]$	$\mathit{firstsym}(\mathit{childlist}(v)_0)$

□

The set of first symbols is not visualized in FIRE Station. It is used internally by FIRE Works, specifically by the derivatives and partial derivatives layers, as discussed in Chapter 7, p. 45 and Chapter 8, p. 51 respectively.

Chapter 7

Derivatives Layer

7.1 Introduction

The derivative of a regular language with regard to a particular symbol is the result of stripping that symbol from the front of all strings in the language that start with that symbol, and removing all other strings.

Derivatives of regular expressions were first introduced in 1964 by Brzozowski [Brz64], and make it possible to compute the derivatives of a regular language represented by a regular expression, such that the result is again in the form of a regular expression.

As discussed in Chapter 3, p. 25, we want to have a representative regular expression for each node in FIRE Works. Since the output of the derivatives algorithm is a new regular expression, this requirement is straightforwardly met, and makes it ideally suited for FIRE Works,

7.2 Formal Description

First we give a formal definition of derivatives of languages:

Definition 7.1 (Derivatives of languages). For a language $L \in \mathcal{P}(\Sigma^*)$, the derivative with respect to symbol $a \in \Sigma$, $a^{-1} \in \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ is defined as:

$$a^{-1}L = \{x | ax \in L\}$$

□

Next, we provide the Brzozowski definition of derivatives of regular expressions. Note that we adapt the notation slightly from the original in [Brz64]; we also list the extended operators (intersection, symmetric difference and relative difference) explicitly.

Definition 7.2 (Derivatives of Regular Expressions). Derivatives of regular expressions, $a^{-1} \in RE \rightarrow RE$ are recursively defined in the following manner:

- $a^{-1}\emptyset = \emptyset$

- $a^{-1}\varepsilon = \emptyset$
- $a^{-1}\mathsf{a} = \varepsilon$
- $a^{-1}(\mathsf{E}^?) = a^{-1}\mathsf{E}$
- $a^{-1}(\mathsf{E}^*) = a^{-1}\mathsf{E} \cdot \mathsf{E}^*$
- $a^{-1}(\mathsf{E}^+) = a^{-1}\mathsf{E} \cdot \mathsf{E}^*$
- $a^{-1}(\neg\mathsf{E}) = \neg(a^{-1}\mathsf{E})$
- $a^{-1}\mathsf{E} \cup \mathsf{F} = a^{-1}\mathsf{E} \cup a^{-1}\mathsf{F}$
- $a^{-1}(\mathsf{E} \cdot \mathsf{F}) = a^{-1}\mathsf{E} \cdot \mathsf{F} \cup a^{-1}\mathsf{F}$ if $null(\mathsf{E})$
- $a^{-1}(\mathsf{E} \cdot \mathsf{F}) = a^{-1}\mathsf{E} \cdot \mathsf{F}$ if $\neg null(\mathsf{E})$
- $a^{-1}(\mathsf{E} \cap \mathsf{F}) = a^{-1}\mathsf{E} \ \cap a^{-1}\mathsf{F}$
- $a^{-1}(\mathsf{E} \triangle \mathsf{F}) = a^{-1}\mathsf{E} \triangle a^{-1}\mathsf{F}$
- $a^{-1}(\mathsf{E} \setminus \mathsf{F}) = a^{-1}\mathsf{E} \setminus a^{-1}\mathsf{F}$

□

In [Brz64], Brzozowski points out that the computation of derivatives of regular expressions terminates if all *similar* regular expressions are grouped together. Two regular expressions are similar if one can be transformed into the other using only the identities for associativity, commutativity and idempotence of the union operator, as listed in Table 7.2.

$$\mathsf{E} \cup \mathsf{E} \equiv \mathsf{E}$$
$$\mathsf{E} \cup \mathsf{F} \equiv \mathsf{F} \cup \mathsf{E}$$
$$(\mathsf{E} \cup \mathsf{F}) \cup \mathsf{G} \equiv \mathsf{E} \cup (\mathsf{F} \cup \mathsf{G})$$

In the same paper Brzozowski also points out that the regular expressions can be reduced by the identities, e.g. $\mathcal{L}_{RE}(\varepsilon \cdot \mathsf{E}) = \mathcal{L}_{RE}(\mathsf{E})$, which we'll write in shorthand $\varepsilon \cdot \mathsf{E} \equiv \mathsf{E}$. This results in a smaller number of unique regular expressions. The identities given by Brzozowski are listed in Table 7.2.

$$\mathsf{E} \cup \emptyset \equiv \mathsf{E}$$
$$\mathsf{E} \cdot \emptyset \equiv \emptyset$$
$$\emptyset \cdot \mathsf{E} \equiv \emptyset$$
$$\mathsf{E} \cdot \varepsilon \equiv \mathsf{E}$$
$$\varepsilon \cdot \mathsf{E} \equiv \mathsf{E}$$

Table 7.1: Regular Expression Identities needed for Brzozowski's derivatives

7.3 Implementation

We implement derivatives of regular expressions directly on the parse tree representation of regular expression layer, by creating nodes representing the derivatives through the family of *mknode* functions. The derivative of a node $v \in V$ w.r.t. a symbol $a \in \Sigma$ is obtained with $\delta_D \in V \times \Sigma \to V$, which creates the node representating the derivative regular expression. This function satisfies the condition:

$$regex(\delta_D(v, a)) = a^{-1}(regex(v), a)$$

To simplify the notation of the derivative for concatenation, we first define $tail_D$ in Definition 7.3.

Definition 7.3 (Tail Derivatives). Tail derivatives $tail_D \in V^* \times \Sigma \to V$, which, from a list of nodes $U \in V^*$, creates a node that represents the derivative of the node at the first node, concatenated with the other nodes in the list:

$$tail_D(U, a) = mknode_{[\cdot]}(\delta_D(U_0, a) \triangleright (U \downarrow 1))$$

□

Now, in Definition 7.4 we describe the way to construct the derivative node of a given node w.r.t. a symbol via the Regular Expression Layer.

Definition 7.4 (Derivatives via the Regular Expression Layer). The construction of a derivative node via $\delta_D \in V \times \Sigma \to V$ of a node $v \in V$ w.r.t. a symbol $a \in \mathit{firstsym}(v)$ is defined for each possible value of $label(v)$ except for $[\emptyset]$ and $[\varepsilon]$ as follows:

$label(v)$	$\delta_D(v, a)$
[a]	$mknode_{[\varepsilon]}$
[?]	$\delta_D(child(v), a)$
[∗]	$mknode_{[\cdot]}([\delta_D(child(v), a), v])$
[+]	$mknode_{[\cdot]}([\delta_D(child(v), a), mknode_{[*]}(v)])$
[¬]	$mknode_{[\neg]}(\delta_D(child(v), a))$
[∪]	$mknode_{[\cup]}(\{\delta_D(u, a) \mid u \in childset(v)\})$
[·]	$mknode_{[\cup]}(\{tail_D(U \downarrow i, a) \mid 0 \leq i \leq lastnull(U)\})$ where $U = childlist(v)$
[∩]	$mknode_{[\cap]}(\{\delta_D(u, a) \mid u \in childset(v)\})$
[△]	$mknode_{[\triangle]}([\delta_D(U_0, a), \ldots, \delta_D(U_{\lvert U\rvert-1}, a)])$ where $U = childlist(v)$
[\]	$mknode_{[\setminus]}([\delta_D(U_0, a), \ldots, \delta_D(U_{\lvert U\rvert-1}, a)])$ where $U = childlist(v)$

Note that we have not defined δ_D for nodes with label $[\emptyset]$ or $[\varepsilon]$, because the first symbol set is empty by definition, and we only compute derivatives for a node's first symbols. □

As Brzozowski points out in [Brz64], to guarantee termination, we need to detect derivatives with similar regular expressions. Since the Regular Expression Layer is in charge of the creation of new nodes, this now becomes an added requirement for that layer. The way we implement this is through *Common Subexpression Elimination*, which will be discussed in depth in Chapter 12, p. 67.

The reductions through identities from Table 7.2 can be implemented in two ways. The first is to implement them directly in the progam logic of the Derivatives Layer. The second approach, known as *regular expression rewriting*, is more flexible. Besides the identities, it allows us to use other equivalence rules. This potentially leads to fewer unique regular expressions, thus fewer nodes and smaller automata. The process of rewriting will be discussed in Chapter 12, p. 67.

Note that for simplification through rewriting, we have to weaken the condition:

$$regex(\delta_D(v, a)) = a^{-1}(regex(v), a)$$

which says that the derivative node represents the derivative regular expression. Instead, we use:

$$\mathcal{L}_{RE}(regex(\delta_D(v, a))) = \mathcal{L}_{RE}(a^{-1}(regex(v), a))$$

which says that the language represented by the derivative node has to be equal to the language of the derivative expression, though the exact regular expression may not be identical to that derivative regular expression.

7.4 Derivatives Automaton and Node Language

Definition 7.5 (Derivative automaton A_D). The derivatives DFA for a regular language defined by node $v \in V$ is defined as:

$$A_D(v) = (V, \Sigma, \delta_D, v, \{w : null(w)\})$$

□

Note that the derivatives automaton is deterministic. This automaton may contain (many) unreachable states. It is possible to reduce the automaton to those nodes reachable on δ_D starting at v without affecting the language recognized.

Definition 7.6 (Regular Language via Derivatives Layer). The regular language represented by a node $v \in V$ on the derivatives layer is defined by the language of the automaton starting at that node:

$$\mathcal{L}_{node}(v) = \mathcal{L}_{A_D(v)}(v)$$

□

7.5 Derivatives Graph

We provide a generic representation of the edges of the derivatives layer, for use in FIRE STATION.

Definition 7.7 (Derivatives Layer Graph). The edges used to represent the derivatives layer as a graph follow straightforward from the derivatives:

$$E_D = \{(v, \delta_D(v, a)) | v \in V \wedge a \in \mathit{firstsym}(v)\}$$

□

Chapter 8

Partial Derivatives Layer

8.1 Introduction

Partial derivatives of regular expressions were first introduced by Antimirov, see [Ant96]. Partial derivatives are similar to Brzozowski's derivatives, discussed in Chapter 7, p. 45, with the difference that the partial derivative that is computed from a regular expression yields a *set* of regular expressions, rather than a single derivative regular expression. Partial derivatives produce a non-deterministic transition function, which can be used to construct an NFA recognizing the regular expression for which they were computed.

Partial derivatives rely on the property that any regular expression can be described as the union of a set of regular expressions. In the Brzozowski derivatives algorithm, this union of regular expressions is actually constructed. From this single, new regular expression the Brzozowski algorithm then computes the next derivatives. In the partial derivatives approach, instead, the set of *partial* derivatives is stored, and for each element in the set the partial derivatives are computed in turn. Unlike the Brzozowski derivatives, partial derivatives do not produce an exponential number of derivatives.

The partial derivatives approach does not yield significant differences from the Brzozowksi derivatives for most of the extended regular operators, specifically: $\neg, \triangle, \backslash$. This can be understood by looking at the Brzozowski derivatives of the extended operators, as described in Definition 7.2, p. 45. If the outermost operator used on the right-hand side of an equation is the union, then obviously we can split the union into a set of expressions. However when the outermost operator is a regular operator other than union, we can't split it into a set of regular expressions, unless we can distribute the outermost operator over the union operator. Operators $\neg, \triangle$ and $\backslash$ do not distribute over union.

The intersection operator $\cap$ does distribute over union, however in the worst case this would lead to more than linear growth in the number of partial derivatives, whereas if we do not distribute intersection over union, we end up with a linear growth.

8.2 Formal Definition

There exists no concept of partial derivatives of regular expressions. Instead we go straight to partial derivatives of regular expressions $\frac{\partial}{\partial a} \in RE \to \mathcal{P}(RE)$, which maps a regular expression E into a set of partial derivative regular expressions w.r.t. a symbol $a \in \Sigma$, such that:

$$(\bigcup \mathsf{F} : \mathsf{F} \in \frac{\partial}{\partial a}\mathsf{E} : \mathcal{L}_{RE}(\mathsf{F})) = a^{-1}\mathcal{L}_{RE}(\mathsf{E})$$

(for a^{-1}, derivative of regular language, see Definition 7.1, p. 45)

Definition 8.1 (Union of a set of Regular Expressions). To simplify our definition of $\frac{\partial}{\partial a}$, we first define function $union_{RE} \in RE \times \Sigma \to RE$ for any $\mathsf{E} \in RE, a \in \Sigma$ as follows:

$$union_{RE}(\mathsf{E}, a) = \mathsf{F}_0 \cup \ldots \cup \mathsf{F}_n, \text{where} \{\mathsf{F}_0, \ldots, \mathsf{F}_n\} = \frac{\partial}{\partial a}\mathsf{E}$$

□

Definition 8.2 (Partial Derivatives of a Regular Expression). For any regular expression represented by node $v \in V$ and symbol $a \in \mathit{firstsym}(v)$, the partial derivative $\frac{\partial}{\partial a} \in RE \to \mathcal{P}(RE)$, produces a set regular expressions and is defined as:

- $\frac{\partial}{\partial a}\mathsf{a} = \{\varepsilon\}$
- $\frac{\partial}{\partial a}\mathsf{E}^? = \frac{\partial}{\partial a}\mathsf{E} \times \{\mathsf{E}^*\}$
- $\frac{\partial}{\partial a}\mathsf{E}^* = \frac{\partial}{\partial a}\mathsf{E} \times \{\mathsf{E}^*\}$
- $\frac{\partial}{\partial a}\neg\mathsf{E} = \{\neg union_{RE}(\mathsf{E}, \mathsf{a})\}$
- $\frac{\partial}{\partial a}\mathsf{E} \cup \mathsf{F} = \frac{\partial}{\partial a}\mathsf{E} \cup \frac{\partial}{\partial a}\mathsf{F}$
- $\frac{\partial}{\partial a}\mathsf{E} \cdot \mathsf{F} = \frac{\partial}{\partial a}\mathsf{E} \times \{\mathsf{F}\} \cup \frac{\partial}{\partial a}\mathsf{F}$ if $null(\mathsf{E})$
- $\frac{\partial}{\partial a}\mathsf{E} \cdot \mathsf{F} = \frac{\partial}{\partial a}\mathsf{E} \times \{\mathsf{F}\}$ if $\neg null(\mathsf{E})$
- $\frac{\partial}{\partial a}\mathsf{E} \cap \mathsf{F} = \{union_{RE}(\mathsf{E}, \mathsf{a}) \cap union_{RE}(\mathsf{F})\}$
- $\frac{\partial}{\partial a}\mathsf{E} \triangle \mathsf{F} = \{union_{RE}(\mathsf{E}, \mathsf{a}) \triangle union_{RE}(\mathsf{F})\}$
- $\frac{\partial}{\partial a}\mathsf{E} \setminus \mathsf{F} = \{union_{RE}(\mathsf{E}, \mathsf{a}) \setminus union_{RE}(\mathsf{F})\}$

In all other cases, the set of partial derivates is empty. □

8.3 Implementation

Creating nodes representing the regular expression of a partial derivative is done with the help of the *mknode* functions. The partial derivates function, $\delta_{PD} \in V \times \Sigma \Rightarrow \mathcal{P}(V)$ is defined in Definition 8.5, and satisfies the condition:

$$\frac{\partial}{\partial a} regex(v) = \{regex(w) | w \in \delta_{PD}(v, a)\}$$

To simplify Definition 8.5 we first give two auxilliary definitions, which rely on the still to be defined δ_{PD}.

Definition 8.3 (Union of Partial Derivatives Nodes). The union of partial derivatives $union_{PD}$ of a node $v \in V$ is a node representing the union of the set of partial derivatives of node v w.r.t. a symbol $a \in \Sigma$:

$$union_{PD}(v, a) = mknode_{[\cup]}(\delta_{PD}(v, a))$$

□

Definition 8.4 (Tail Partial Derivative). The tail partial derivative, $tail_{PD} \in V^* \times \Sigma \rightarrow \mathcal{P}(V)$ of a list of nodes W, is a set of nodes each of which represents the concatenation of a partial derivative of the first element of W and the tail of W:

$$tail_{PD}(W, a) = \{mknode_{[\cdot]}(w \triangleright (W \lfloor 1)) | w \in \delta_{PD}(W_0, a)\}$$

□

Definition 8.5 (Partial Derivatives via the Regular Expression Layer). Partial derivatives $\delta_{PD} \in V \times \Sigma \Rightarrow \mathcal{P}(V)$ of node $v \in V$ and symbol $a \in \mathit{firstsym}(v)$ are defined for each possible value of $label(v)$ (except $[\emptyset]$ and $[\varepsilon]$) as follows:

$label(v)$	$\delta_{PD}(v, a)$
[a]	$\{mknode_{[\varepsilon]}\}$
[?]	$\delta_{PD}(child(v), a)$
[∗]	$\{mknode_{[\cdot]}([w, v]) \mid w \in \delta_{PD}(child(v), a)\}$
[+]	$\{mknode_{[\cdot]}([w, mknode_{[*]}(u)]) \mid w \in \delta_{PD}(u, a)\}$ where $u = child(v)$
[¬]	$\{mknode_{[\neg]}(union_{PD}(child(v), a))\}$
[∪]	$(\bigcup w : w \in childset(v) : \delta_{PD}(w, a))$
[·]	$(\bigcup i : 0 \leq i \leq lastnull(childlist(v)) : tail_{PD}(U \lfloor i, a))$
[∩]	$\{mknode_{[\cap]}(\{union_{PD}(w, a) \mid w \in childset(v)\})\}$
[\]	$\{mknode_{[\triangle]}([union_{PD}(W_0, a), \ldots, union_{PD}(W_{\lvert W-1 \rvert}, a)])\}$ where $W = childlist(v)$
[\]	$\{mknode_{[\backslash]}([union_{PD}(W_0, a), \ldots, union_{PD}(W_{\lvert W-1 \rvert}, a)])\}$ where $W = childlist(v)$

For $label(v) = [\emptyset]$ and $label(v) = [\varepsilon]$ the set of partial derivatives is always the empty set. □

As discussed for the derivatives layer in Chapter 7, p. 45, we use global common subexpression elimination to map identical regular expressions to the same node. This guarantees termination of the algorithm, since there exists only a finite number of partial derivatives (see [Ant96] for details).

Through the use of rewriting we can further reduce the number of partial derivatives by recognizing equivalent regular expressions.

8.4 Partial Derivatives Automaton and Node Language

Definition 8.6 (Partial Derivatives Automaton A_{PD}). The partial derivatives automaton for the regular language represented by a node $v \in V$ is defined as

$$A_{PD}(v) = (V, \Sigma, \delta_{PD}, v, \{w : null(w)\})$$

This automaton can contain (many) unreachable states. It is possible to reduce the automaton to those nodes reachable on δ_{PD} starting at v without affecting the language recognized. □

Definition 8.7 (Regular Language via Partial Derivatives Layer). The regular language represented by a node $v \in V$ on the partial derivatives layer is defined by the right linear language of the automaton starting at that node:

$$\mathcal{L}_{node}(v) = \mathcal{L}_{A_{PD}(v)}(v)$$

□

8.5 Partial Derivatives Graph

The graph used to represent the partial derivatives layer is defined as follows:

Definition 8.8 (Partial Derivatives Layer Graph). The edges for the Partial Derivatives Layer over the nodes V are given by:

$$E_{PD} = \{(v, w) | v \in V \wedge a \in \Sigma \wedge w \in \delta_{PD}(v, a)\}$$

□

Chapter 9

Empty Language Layer

9.1 Introduction

The empty language layer is a node property layer, which depends on the Chapter 7. The empty language layer determines whether the regular language of a node is equal to $\emptyset$.

Knowing whether or not a node's language is empty is important when optimizing automata; empty language nodes are effectively *trap states*: once a transition to an empty language state is made, a final state can no longer be reached. In automaton optimization, the empty language states are removed from the automaton, and instead the automaton has no transitions on the symbols that would otherwise go to an empty language state.

If the automaton is desired to be *complete*, meaning it has a transition for each state/symbol pair, a single empty language state (the *sink state*) is added and all missing transitions, *sink transitions*, go to this state. The sink state usually has a transition to itself on all symbols.

Note that when *minimizing* a DFA, all empty language states will reduce to a single state, however minimization does not add or remove any sink transitions; thus removal of sink states is a separate step in the optimization of automata.

Empty language nodes can also be used to compare string patterns with regular expressions, as described in Section 9.4

9.2 Formal Specification

The empty language property is described in terms of a transition function, which is provided by either a derivative automaton or by a partial derivative automaton discussed in the previous chapters. Given a (deterministic) transition function $\Delta \in V \times \Sigma \to V$, the language of a node is empty if there is no path from that node to a nullable node. This is formally described in Definition 9.3.

Definition 9.1 (Emptyness of Regular Expressions). Emptyness of a regular expression is determined by function $empty_{RE} \in RE \to \mathbb{B}$ and is defined for any $\mathsf{E} \in RE$ as:

$$empty_{RE}(\mathsf{E}) \equiv \mathcal{L}_{RE}(\mathsf{E}) = \emptyset$$

□

9.3 Implementation

Definition 9.2 (Emptyness of a node). The emptyness $empty \in V \to \mathbb{B}$ of a node $v \in V$ is defined as follows:

$$empty(v) \equiv empty_{RE}(regex(v))$$

□

The above specification cannot be easily implemented. Instead we give an alternative definition, in terms of nodes, which can be implemented straightforwardly.

Definition 9.3 (Recursive definition for emptyness of a node). Recursive definition of $empty$, with $\Delta \in V \times \Sigma \to V$ a deterministic transition function:

$$empty(v) \equiv \neg null(v) \wedge (\forall a : a \in \Sigma : empty(\Delta(v, a)))$$

□

This specification can easily be implemented using the transition function δ_D from the Derivatives Layer, Chapter 7, p. 45. The implementation can perform a breadth-first traversal of the transition function to determine if there is a path from node $v \in V$ to any node that is nullable. A similar approach works for non-deterministic transition functions.

We visualize empty language nodes by a drawing a diagonal line through them, giving a similar shape as the empty language symbol: $\emptyset$.

9.4 Rudimentary Pattern Matching

We can check whether a string pattern matches a particular regular expression by building the derivatives for the string pattern as well for the regular expression, and then computing the intersection of these two nodes. If the intersection is empty, the pattern is not recognized by the regular expression; if it is not empty, then it matches.

9.5 Testing for Overlap of Regular Languages

Another application is to determine whether the intersection of two regular languages is empty. We do this by building the automaton equivalent to the intersection of the two languages. If the intersection node is an empty language node, it means that the regular languages of those nodes have no strings in common. However if the intersection node is not empty, then the derivatives layer provides an automaton that can recognize strings from the language.

Chapter 10

Equivalent States Layer

10.1 Introduction

The equivalent states layer determines which nodes represent identical regular languages. The process of finding equivalent states is the first step in the minimization of DFA. When referring to DFAs the term *states* is used instead of nodes, hence the name Equivalent States Layer. The minimization algorithm relies on the relations between nodes computed on the derivatives layer. The minimization process results in sets of states, each such set containing nodes describing the same regular language.

10.2 Formal Definition

Definition 10.1 (Equivalence of Regular Languages). Regular languages equivalence $equiv_{REG} \in REG \times REG \to \mathbb{B}$. Two languages $L_0, L_1 \in REG$ are equivalent if they contain the same set of strings:

$$equiv_{REG}(L_0, L_1) \equiv L_0 = L_1$$

□

From Definition 10.1, using $\mathcal{L}_{node}$, we define equivalence of regular languages represented by nodes:

Definition 10.2 (Equivalence of Regular Languages of Nodes). Equivalence of two nodes is defined through the equivalence of the languages they represent. We define $equiv \in V \times V \to \mathbb{B}$ of a pair of nodes $v, w \in V$ as:

$$equiv(v, w) \equiv \mathcal{L}_{node}(v) = \mathcal{L}_{node}(w)$$

□

We have seen several functions that are equal to $\mathcal{L}_{node}$. Specifically: Definition 4.7, p. 35, Definition 7.6, p. 48, and Definition 8.7, p. 54, which we repeat here in compact form:

$$\mathcal{L}_{node}(v) = \mathcal{L}_{RE}(regex(v)) = \mathcal{L}_{A_D(v)}(v) = \mathcal{L}_{A_{PD}(v)}(v)$$

Which of these definitions will be used is left to the implementation, discussed in the next section.

10.3 Implementation

Computation of equivalence directly from regular expressions is hard (see also Chapter 12 on regular expression rewriting), while determining the language equivalence of states in a deterministic automaton is easy, and there is an abundance of algorithms to do so. We therefore choose this approach for the implementation of the Equivalent States Layer. We refine *equiv* as follows:

Definition 10.3 (Equivalence of States). Given an automaton A on the nodes (states) V, equivalence *equiv* of nodes $v, w \in V$ is defined as:

$$equiv(v, w) \equiv \mathcal{L}_A(v) = \mathcal{L}_A(w)$$

□

The automaton defined by the derivatives layer, A_D, is an excellent candidate for the computation of *equiv*. In the next two sections we will discuss a couple of these algorithms.

10.3.1 Traditional Algorithm

The traditional algorithms used to find sets of nodes that are indistinguishable iteratively partition the set of nodes. The initial partitioning is that into two sets of nodes: those that are in *null*, and those that are not in *null*. From there we can iteratively partition into smaller subsets of indistinguishable nodes, until a fixed-point is reached. For further details see [HU79] and [Hop71].

10.3.2 An Incremental Algorithm

Although the traditional algorithm works well as a post-processing step in the construction of automata, in FIRE Works we are dealing with a somewhat different situation. New nodes are added and removed all the time. Since the addition of nodes does not modify the regular language represented by existing nodes, it does not affect the equivalence relation between those existing nodes. It is wasteful to compute the, already known, equivalence relation between the existing nodes over and over. We give an alternative algorithm, which incrementally computes equivalence of nodes, without reprocessing old nodes.

The only known algorithm that can solve this problem is that provided by Watson [Wat01]. The algorithm relies on computing the equivalence of nodes in a pair-wise fashion. This has

the effect that the algorithm can be terminated at any point and the intermediate results be used. We are also interested in the mirror property of the algorithm: it can be resumed at any point, even after we add new states, as long as those states do not affect the equivalence between pre-existing nodes. This condition is satisfied as long as we do not compute the equivalence of states until the transition functions computed by the derivatives layer are complete, i.e. all derivatives of existing nodes have been computed exhaustively.

Some smart optimizations discussed by Watson and Daciuk in [WD03] allow the incremental algorithm to run efficiently enough to make it competetive with other known minimization algorithms. The only added requirement over those of the traditional algorithm of the previous section is that for the incremental algorithm to work, *useless* nodes (i.e. those recognizing the empty language) have been removed. Therefore this algorithm also relies on the Empty Language Layer discussed in Chapter 9, p. 55

10.4 Equivalent States of Non-Deterministic Automata

The algorithms discussed in the previous two sections rely on *deterministic* transition functions. There exist algorithms for the computation of equivalence of states in non-deterministic automata, see for example [IY03].

The goal of computing equivalent states is (usually) the minimization of the automaton. For a deterministic automaton, merging equivalent states yields the minimal automaton. However merging states that are equivalent in a non-deterministic automaton is not sufficient to obtain a minimal automaton. A better technique involves *preorders* of languages and is discussed in [CC04]. Note that even that technique does not yield the minimal automaton.

Computation of state equivalence or preorders for non-deterministic automata are currently not part of FIRE Works and FIRE Station, but an layer may be added in the future to support these.

10.5 Equivalent States Graph

To visualize the equivalent state property, we create an edge between every two states in the same equivalence class. FIRE Station works on directed graphs, however our relation is symmetrical. We want to add an edge in one direction or the other, but not both, since the second edge does not add any information. We choose the direction based on some (arbitrary) partial ordering $\prec \in V \times V$ on the nodes in V. The edges for the Equivalent States Layer is then defined as:

$$E_{ESL} = \{(v, w) | v, w \in V \wedge v \prec w \wedge equiv(v, w)\}$$

Chapter 11

Graph Layout

11.1 Introduction

Graph layout is a part of the graph visualization process and a pre-requisite for graph drawing. The graph layout algorithm takes a graph as input, and computes a set of vertex positions, and (possibly) the placement and routing of edges. FIRE STATION provides a graph layout framework geared towards two-dimensional node-and-edge graph drawings.

Each (relational) layer provides its own set of edges. We define the set of all such sets of edges as: $layeredges \in \mathcal{P}(V \times V)$. From the layers discussed in this report, $layeredges = \{E_{RE}, E_D, E_{PD}, E_{ESL}\}$

11.2 Node and Layer Visibility

FIRE STATION allows the user to manipulate graph visualization in several ways. One thing that will always be made visible is a set of root nodes, V_{roots}, which will be made more concrete at a later point. The first element the user can control is *nodevisiblegraphs*, the set of "node visible graphs", which determine what nodes will be visible besides the root nodes. We define $E_{nodevisible}$ as the union of the edge sets in *nodevisiblegraphs*:

$$E_{nodevisible} = (\bigcup e : e \in nodevisiblegraphs : e)$$

Now the set of visible nodes, $V_{visible}$, is determined by their reachability from the root nodes via the edges in the node visible graphs $nodevisiblegraphs \subseteq layeredges$. We use the transitive closure to define $V_{visible}$:

$$V_{visible} = \{w | (\exists v : v \in V_{roots} : (v, w) \in (V, E_{nodevisible})^*)\}$$

Oftentimes, the set of graphs for node visibility is identical to the set of visible graphs. For example, if we want to show the graph for the Regular Expression Layer, then all nodes reachable via the Regular Expression Layer should be visible. If we want to show a derivatives automaton, then all nodes reachable by the start-state of that automaton need to be visible,

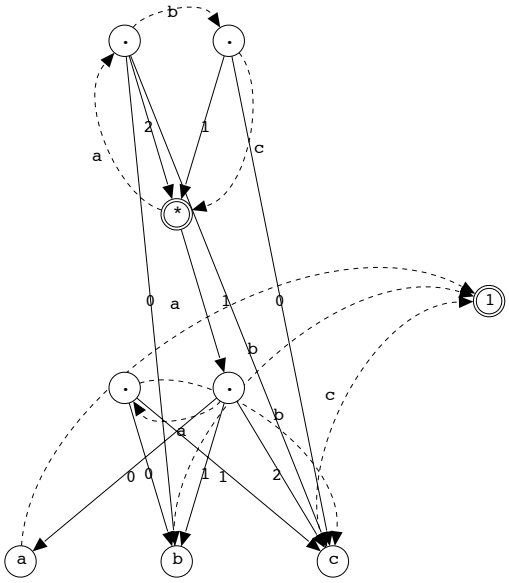

Figure 11.1: Visualization of the parse graph and derivatives graph, with the parse graph driving the layout

otherwise we have an incomplete picture of the automaton. We give two examples where the distinction between node visibility and visible graphs is useful:

First example: we have computed the derivatives of a regular expression, and want to show the nodes within the automaton that represent the same language. Now, suppose the derivative of some subexpression happens to have the same language as a node in the automaton. If we make the derivative of the subexpression visible, the entire automaton reachable from that node will also become visible. Instead, we only want to show the equivalent states relation between nodes that happen to be visible, but not have it affect the visibility of nodes.

In the second example, we look at a partial derivatives automaton. At the same time, the graph for the Regular Expression Layer is visible, but not contributing to node visibility. The result is an incomplete visualization of the parse tree graph, however, we now have an indication of which states in the partial derivatives automaton are subexpressions of other states.

11.3 Layout Driving Graphs

The layout algorithms under consideration in this thesis are based on the given set of edges between nodes. By changing the provided edges, we have (some) control over the resulting layout. For example, if we consider only the edges from the parse layer, the layout algorithm produces Figure 11.1. When using the edges in the derivatives layer, the layout in Figure 11.2 is the result. In Figure 11.3 the layout using edges from both parse and derivatives layer is shown.

If a layer contributes its edges to the layout algorithm, we call the layer “layout driving”, and

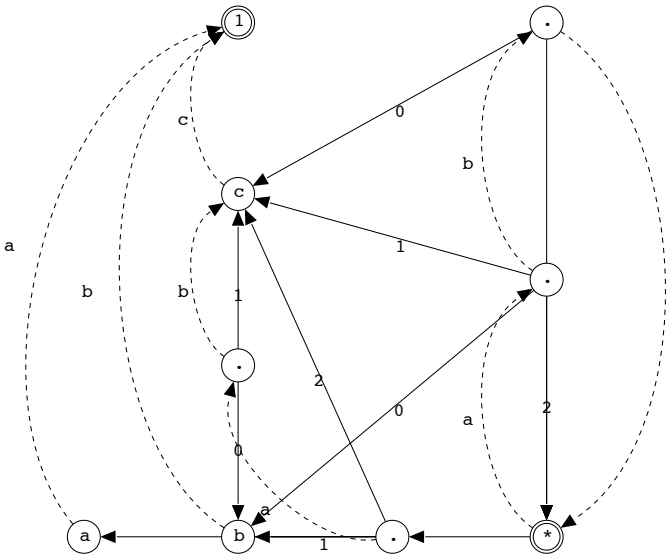

Figure 11.2: Visualization of the parse graph and derivatives graph, with the derivatives graph driving the layout

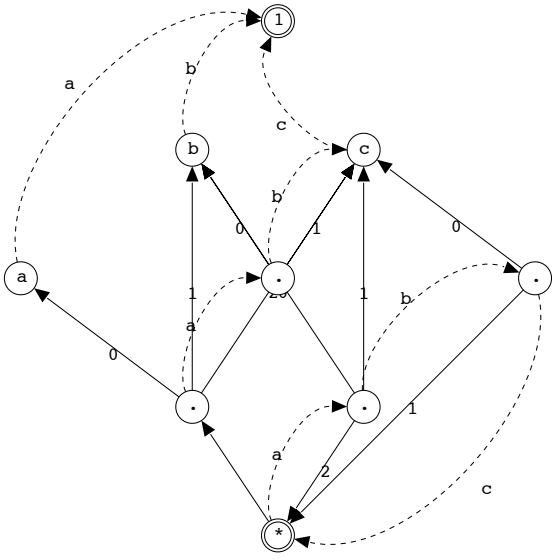

Figure 11.3: Visualization of the parse graph and derivatives graph, with both graphs driving the layout

it is part of the set *drivinggraphs*. Note that it is possible for layers to be it invisible, but still be driving the layout.

11.4 Layout Algorithm

Layout is performed on the graph G_{layout}, which is induced on the graph $(V, E_{nodevisible})$ by $V_{visible}$. The layout algorithm takes the graph G_{layout} and partitions it into its maximally connected components (note: but not *strongly* connected components), as if the graph were interpreted as an undirected graph. To each of these components, the selected layout algorithm is then applied. The resulting layouts are then positioned next to each other (horizontally or vertically).

The layout algorithm currently used in FIRE Station is throroughly discussed in Chapter 9 of [BETT99], and we will outline it here briefly.

- Reverse certain edges so that the graph becomes a directed acyclic graph (DAG).
- Pick a root node for the DAG.
- Determine the shortest path from the root for each node in the DAG and assign this as its row.
- For each edge from a node v on row n to a node w on row m, such that $n + 1 < m$, remove the edge (v,w) and create dummy nodes on each of the rows $n + 1 \dots m - 1$, along with dummy edges $\{(n, n + 1) | n \leq i \wedge i < m\}$
- Sort the edges on each row so that it minimizes the number of edges crossings. This is hard and usually approximated using heuristic approaches.
- Remove the dummy nodes and restore the original edges.

(To avoid confusion we have replaced the term *layer* in the original algorithm description by the term *row*.) Note that we can reverse the graph before the second step takes place, which leads to a graph whose leaves rather than its roots are placed on the same layer, see Figure 11.4. Note how in the normal rendering, all nodes are pulled as far to the bottom as possible, while in the rendering using the reversal layout, all nodes are floated to the top as much as possible.

11.5 Graph Rendering

When layout has been completed, each of the graphs in the set *visiblegraphs* is drawn, one on top of the other. Different colors and line styles, as well as curving of edges ensure that edges that exist in more than one of the visible graphs are distinguishable. Edge routing algorithms are currently not implemented in FIRE Station. They may improve the overall visualization by attempting to avoid overlapping nodes and edges.

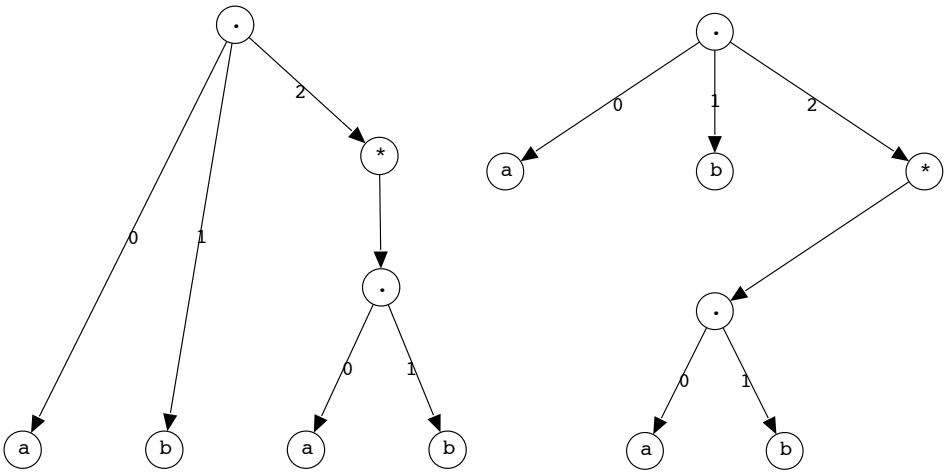

Figure 11.4: Layout of $\mathsf{ab}(\mathsf{ab})^*$: (a) Normal (b) Edge Reversal

Chapter 12

Advanced Parse Tree Manipulation

In this chapter we will discuss three enhancements to the Regular Expression Layer, all of which have the potential of reducing the total number of nodes. The enhancements are:

- *Tree Flattening* ensures that all nodes representing the same regular expressions have parse trees with the same structure.
- *Global Common Subexpression Elimination* identifies identical parse trees and replaces them by a single instance
- *Rewriting* transforms equivalent parse trees into identical ones.

12.1 Tree Flattening

Tree flattening is the process of transforming one parse tree into another by absorbing child nodes into their parents, if the operators of the child and parent node are the same. For example, the parse tree in Figure 12.1(a), can be flattened into that of Figure 12.1(b).

Tree flattening ensures that the parse tree is unique, making it easier to test for equivalence between multiple parse trees. Note that the flattened representation is also the most compact in terms of number of nodes. Flattening can be applied to nodes that have commutative or associative operators, which means the symmetric and relative difference operator are excluded.

12.2 Common Subexpression Elimination

12.2.1 Introduction

It is not uncommon for two nodes in the parse tree to represent identical regular expressions. By finding and eliminating these *common subexpressions*, we can reduce the number of nodes needed to represent the parse tree significantly. This process is commonly referred to as *global common subexpression elimination* (or GCSE), and has been used in the field of compilers for a long time; see for example [Coc70], [Hop94] and [BJKO00].

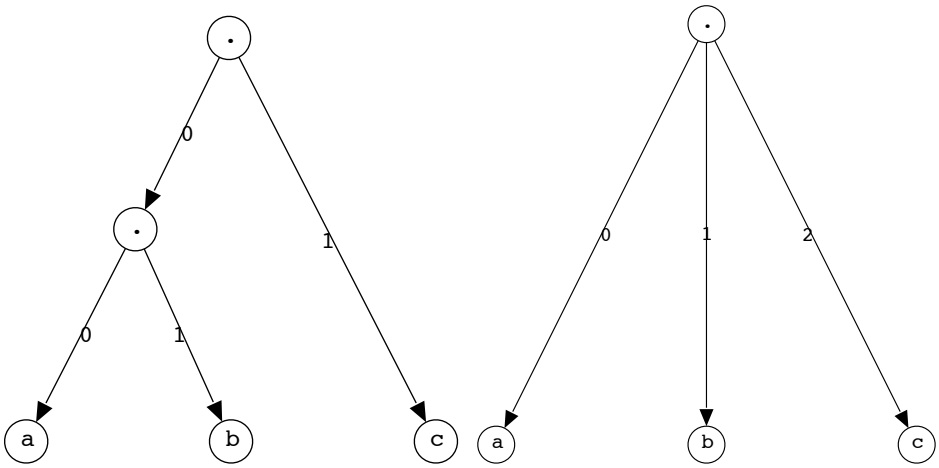

Figure 12.1: Parse Tree of abc. (a) without flattening (b) with flattening

12.2.2 Formal Description

We want to merge two subexpressions when they are equal. We first define what it means for two subexpressions to be equal as follows $cse_{RE} \in RE \times RE \to \mathbb{B}$:

$$cse_{RE}(\mathsf{E}, \mathsf{F}) \equiv \mathsf{E} = \mathsf{F}$$

We adapt this to our node-based system, in the following manner.

Definition 12.1 (Common Subexpression Equivalence). We define $cse \in V \times V \to \mathbb{B}$ on a pair of nodes $v, w \in V$ as:

$$cse(v, w) \equiv regex(v) = regex(w)$$

□

12.2.3 Implementation

If we implement GCSE according to Definition 12.1, we end up with at most one node for each regular expression. This is theoretically the most efficient representation we can obtain in terms of number of nodes, if we don't modify the structure of the regular expressions. However, in practice, there are algorithms that need to create more than one instance of the same regular expression. Therefore we simplify our requirement:

Two nodes are said to be equivalent if they have the same operator and the same child node(s). If we create a new regular expression node with this definition of CSE, a node will be created if there isn't already a node in V that has the specified regular operator and child node(s). If there is a node in V that already represents the regular expression created for the new node, but that contains different child nodes, then the new node will be instantiated regardless.

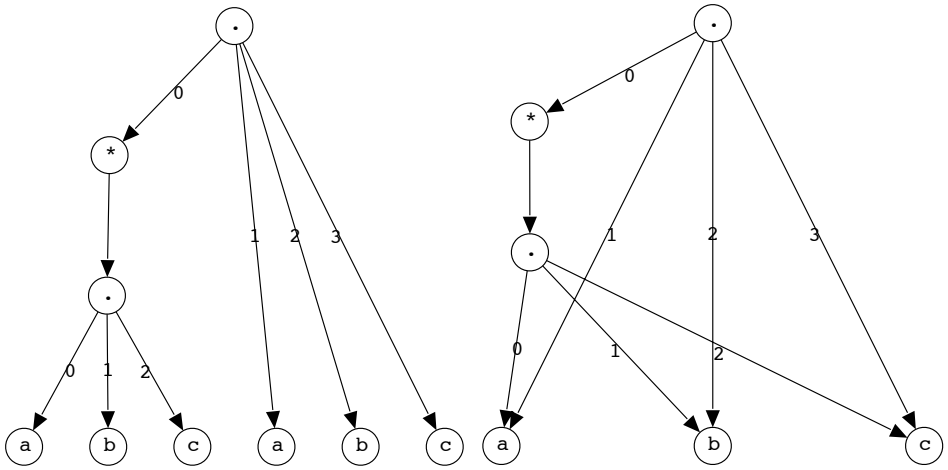

Figure 12.2: (a) (abc)*abc before GCSE (b) after GCSE

Definition 12.2 (Limited CSE). We introduce a limited version of CSE, $cselite \in V \times V \rightarrow \mathbb{B}$, which is defined for any $v, w \in V$ as: $cselite(v, w) \equiv true$ if and only if $label(v) = label(w)$, and one of the following holds:

- $label(v) \in \{[\emptyset], [\varepsilon]\}$
- $label(v) \in symbollabels$
- $label(v) \in unarylabels \wedge child(v) = child(w)$
- $label(v) \in setlabels \wedge childset(v) = childset(w)$
- $label(v) \in listlabels \wedge childlist(v) = childlist(w)$

□

The limited version of CSE as defined in Definition 12.2 can be implemented efficiently by storing a list of parent nodes for each node. When a new node is created we only need to look at the intersection of the parent lists of the child nodes, to see if one of them has the same operator. This approach can be implemented in near-constant time using hashing.

Proper functioning of GCSE requires tree flattening; otherwise it is possible to have different parse trees representing the same regular expression, as we've already seen in Section 12.1.

Note that GCSE changes the parse tree into a directed acyclic graph (DAG). However it does not affect the correctness of anything discussed so far; we can still interpret the DAG as a parse tree for any given node. As an example, the regular expression (abc)*abc results in the parse tree in Figure 12.2(a). The symbol nodes for a, b and c each occur twice. We can replace these by a single instance, as in Figure 12.2(b). Although we end up with a parse DAG, we continue to use the term parse tree, because our interpretation of the parse DAG is as if it were a tree (or set of trees).

12.3 Rewriting

12.3.1 Introduction

For any regular expression, there is an infinite number of distinct regular expressions that represent the same regular language. For example, a and $\mathsf{a} \cdot \varepsilon$ both represent the language $\{a\}$. If we can recognize such equivalent regular expression equivalences, we can combine their nodes, which results in more compact representation and smaller automata.

One way to recognize equivalence of the languages represented by regular expressions is discussed in Chapter 10, p. 57. However, in that approach we only discover equivalence, *after* we have generated a (potentially large) number of equivalent nodes. Here we discuss a different approach called *Regular Expression Rewriting*.

Looking at our earlier example, $\mathsf{a} \equiv \mathsf{a} \cdot \varepsilon$, we could have a rule "remove extraneous ε symbols from the back of a list of concatenated nodes". We can write this down symbolically as: $\mathsf{E} \cdot \varepsilon \Rightarrow \mathsf{E}$, where E is considered a wildcard, i.e. it matches any regular expression. If our initial expression has multiple ε symbols, we can apply the rule more than once until no more symbols can be removed.

If the rewritten regular expression was already present in the Regular Expression Layer, it can be found through Global Common Subexpression Elimination, discussed earlier in this chapter. Since rewriting and GCSE are both performed *before* an actual node is created on the Regular Expression Layer, there is potential for a reduction in the total number of nodes being created.

12.3.2 Formal Description

Regular expression rewriting is defined through rewrite rules. Since the goal is to have a dynamic system, in which rewrite rules can be added and removed freely, it is not possible to give an exact definition of rewriting. Instead we give a set of requirements that rewrite rules have to adhere to. Function $rewrite_{RE} \in RE \rightarrow RE$ encompasses all active rules, and has to satisfy the following condition for any regular expression E:

$$\mathcal{L}_{RE}(\mathsf{E}) = \mathcal{L}_{RE}(rewrite_{RE}(\mathsf{E}))$$

In words: the language of the rewritten expression is the same as that of the original expression.

Rewrite rules take the form *pattern* $\Rightarrow$ *target*. In such a rule, *pattern* describes a regular expression containing wildcards, which can match to any subexpression and is denoted by a symbol directly preceeded by a \$. The *target* part of the rule describes how to create the rewritten expression, possibly using the wildcards from *pattern*. Obviously *target* cannot contain wildcards that were not in *pattern*, however it may remove wildcards, as in the rule $\$\mathsf{E} \cdot \emptyset \Rightarrow \emptyset$. Note that if a wildcard occurs in two or more places in *pattern*, then both instances have to match identical regular expressions. For example, rule $\$\mathsf{E} \cdot \$\mathsf{E}^* \Rightarrow \$\mathsf{E}^+$ can be applied to $\mathsf{abc}(\mathsf{abc})^*$, where $\$\mathsf{E} = \mathsf{abc}$ in both occurrences in the rewrite pattern.

Termination

An important aspect of rewriting is *termination*. A set of rewrite rules is said to be terminating, if, in the process of rewriting, the same expression cannot occur twice. If this were possible, then the rewriting system could enter an infinite loop, performing rewriting $\mathsf{E} \to \mathsf{F} \to \mathsf{E} \to \mathsf{F} \to \ldots$ etc. Termination is also something that we cannot guarantee, because we have no control over the rule-sets provided by the user.

Confluence

The second important aspect of any rewriting system is *confluence*. Confluence means that regardless of the order in which the rewrite rules are applied to a particular regular expression, after rewriting terminates, the resulting expression is always the same. Since the rewrite rules are determined by the user, we cannot guarantee confluence for the application of rewrite rules. However, in our implementation we can determine a specific order in which rules are applied, making the results consistent and predictable.

12.3.3 Implementation

The implementation of regular expression rewriting operates on the structure of the parse tree rather than that of the symbolical notation of the regular expression itself. Although the result is the same, there are some practical caveats.

There are different parse trees that represent the same regular expression. We need some "normal form", so that identical expressions will be represented in the same way. Node flattening, see Section 12.1 provides us with such a normal form.

Secondly, in the example rule of $\$\mathsf{E} \cdot \$\mathsf{E}^* \Rightarrow \$\mathsf{E}^+$, we need a way to test whether the two subtrees matching the two occurrences of wildcard $\$\mathsf{E}$ are equivalent. GCSE provides an elegant solution: two subtrees are equivalent if and only if they are one and the same node.

Rewriting Process

The rewrite process consists of two steps. The first step is to match the regular expression parse tree to the pattern side of one of the rewrite rules. The second step is to construct the parse tree of the rewritten expression. We will now look at each step in more detail.

Pattern Matching

To match a rewrite pattern to a *subject* (the parse tree we're rewriting), we traverse the parse tree of the subject and the parse tree of rewrite pattern in parallel and attempt to match nodes one to one. As soon as we can detect a mismatch, we back up and try an alternative match. Since there are many potential ways to match one tree to another, it is important to detect mismatches as soon as possible, rather than blindly trying all permutations.

Both the pattern and the subject have a single root node, and so we first compare the node labels for those root nodes. If they are different, the nodes don't match. If they are the same,

and they have the same number of child nodes, we try to find a pairing of child nodes such that each pair matches. This process continues recursively down the tree, until a match is found.

Note that for every node, while recursing down the tree, the matching process is independent of the parent nodes and we therefore treat those nodes exactly the same way as the root node is treated (with the exception of wildcards, which will be discussed later).

If the (root) nodes of the pattern and subject have a *set* of child nodes, then there are many ways to pair up those child nodes. It is possible that more than one of those pairings results in a match. Since we cannot predict the right pairing, we have to exhaustively search all pairings for a match. We can limit the number of searched pairings through bounding rules: if any pair of nodes in a pairing don't match, there is no need to try and match the rest and we can move on to the next pairing.

The number of potential matches is complicated significantly through the use of it wildcard nodes. Wildcard nodes are used to match entire subtrees, and if the same wildcard occurs in more than one place, the subtrees matched for each instances have to be identical. For example, the rule $\emptyset \cup \$\mathsf{E} \Rightarrow \$\mathsf{E}$, can be matched to $\emptyset \cup \mathsf{a}$, $\emptyset \cup \mathsf{a} \cup \mathsf{b}$, etc. Wildcard $\$\mathsf{E}$ can effectively match any number of child nodes, which we will call a grouping rather than a pairing.

The process of checking all groupings isn't complicated, however, the number of possible groupings can be significant, especially when we have more than one wildcard that we can group nodes under. For example, rewrite rule $\emptyset \cup \$\mathsf{E} \cup \$\mathsf{F} \Rightarrow \$\mathsf{E} \cup \$\mathsf{F}$ can be matched to $\emptyset \cup \mathsf{a} \cup \mathsf{b} \cup \mathsf{c}$ in 6 different ways (if we allow the sets for the wildcards groupings to be empty).

Our solution to bound the growth of multiple groupings is by restricting the rewrite rules as follows: a node in a rewrite pattern can have at most one wildcard in its set of child nodes. We do allow multiple wildcards in lists, for example in the case of $\$\mathsf{E} \cdot \varepsilon \cdot \$\mathsf{F} \Rightarrow \$\mathsf{E} \cdot \$\mathsf{F}$. However we disallow the *consecutive* occurrence of two wildcards in a list of nodes, as in $\$\mathsf{E} \cdot \$\mathsf{F} \cdot \varepsilon \Rightarrow \$\mathsf{E} \cdot \$\mathsf{F}$

Since wildcards can occur multiple times in a rewrite pattern, every time we match against a wildcard, we also have to check against all earlier matches of that wildcard. This effectively requires a comparison of the subtrees of each match of the wildcard. The operator of a wildcard node is usually determined by the node it matches, but in the case of set and lists, it is possible that a wildcard only matches a part of the set or list.

As an example, we take the rule $\$\mathsf{E} \cdot (\$\mathsf{E})* \Rightarrow \$\mathsf{E}^+$. If this rule is applied to $\mathsf{abc}(\mathsf{abc})*$, then the first instance of $\$\mathsf{E}$ matches the a sublist of the root node, and thus assumes the concatenation operator. The second instance of $\$\mathsf{E}$ matches the concatenation within the Kleene closure, and thus also has concatenation as it's operator, and a match is found.

If we apply the rule to $\mathsf{aa}*$, we would also expect to find a match, however in this case the first occurence of $\$\mathsf{E}$ would match against the first a node, and thus be a symbol node, rather than a concat node, even though it matches a sublist of the root node. The second occurence of $\$\mathsf{E}$ matches the a within the Kleene closure, and again we have a match.

We generalize this to the following: if a wildcard node matches two or more nodes in a list or set, it assumes the same operator as the parent node of the list or set which it (partially) matches. If only a single node is matched, then the wildcard node assumes the operator of that single node.

Comparing of subtrees can be done very efficiently if global common subexpression elimination is used, since identical parse trees are represented by a single node. If GCSE is not used, we have to compare the subtrees directly.

Constructing the Rewritten Expression

Once a rewrite pattern has been matched to a subject, creating the rewritten parse tree is a matter of substituting the wildcards in the rewrite target by those that were matched in the pattern. If GCSE is enabled, in the subject nodes that matched the wildcards can simply be reused, otherwise they first have to be duplicated.

When the nodes of the rewritten parse tree are created, they are again subjected to the rewrite process, since they may in turn match to a (different) rewrite rule. This process goes on until no more rewrite rules can be applied. Note that in this process, some intermediate nodes may be constructed that are not actually part of the rewritten parse tree.

12.3.4 Applying Rewriting

Parse trees are constructed in a bottom-up fashion, using the *mknode* functions. Rewriting is patched into this process, requiring the specification of *mknode* to be more liberal:

Definition 12.3 (Creating Rewritten Regular Expression Nodes). We modify the definition of the *mknode* functions from Definition 4.8, p. 35 to allow for rewriting. Note the change from $regex(v)$ to $\mathcal{L}_{node}(v)$ in the second column.

if v is the result of	then $\mathcal{L}_{node}(v)$ equals
$mknode_{[\emptyset]}$	$\mathcal{L}_{RE}(\emptyset)$
$mknode_{[\varepsilon]}$	$\mathcal{L}_{RE}(\varepsilon)$
$mknode_{\Sigma}(a)$	$\mathcal{L}_{RE}(\mathsf{a})$
$mknode_{[?]}(w)$	$\mathcal{L}_{RE}(regex(w)^{?})$
$mknode_{[*]}(w)$	$\mathcal{L}_{RE}(regex(w)^{*})$
$mknode_{[+]}(w)$	$\mathcal{L}_{RE}(regex(w)^{+})$
$mknode_{[\neg]}(w)$	$\mathcal{L}_{RE}(\neg regex(w)$
$mknode_{[\cup]}(\{w_0, \ldots, w_n\})$	$\mathcal{L}_{RE}(regex(w_0) \cup \ldots \cup regex(w_n))$
$mknode_{[\cdot]}([w_0, \ldots, w_n])$	$\mathcal{L}_{RE}(regex(w_0) \cdot \ldots \cdot regex(w_n))$
$mknode_{[\cap]}(\{w_0, \ldots, w_n\})$	$\mathcal{L}_{RE}(regex(w_0) \cap \ldots \cap regex(w_n))$
$mknode_{[\triangle]}(\{w_0, \ldots, w_n\})$	$\mathcal{L}_{RE}(regex(w_0) \triangle \ldots \triangle regex(w_n))$
$mknode_{[\setminus]}([w_0, \ldots, w_n])$	$\mathcal{L}_{RE}(regex(w_0) \setminus \ldots \setminus regex(w_n))$

□

12.3.5 Final Thoughts on Rewriting

In Section 12.3.2 we already touched on confluence. As we pointed out, rewriting is performed in a very specific order and thus predictable. The exact order in which rewriting is peformed is dictated by the Regular Expression Layer, specifically by the bottom-up fashion in which parse trees are constructed.

Because rewriting is integrated into *mknode*, it is applied at the earliest time possible. Thus if we create a new node using some child nodes, we know that those child nodes have already been exhaustively rewritten.

A potential enhancement to FIRE Works could add other kinds of rewriting algorithms. It is worth noting that, by design, rewriting peformed on the Regular Expression Layer is limited. For example, [SR03] shows that "coinductive rewriting" can be very efficient, however it relies on information from the Brzozowski derivatives. Since we want to keep the Regular Expression Layer independent of other layers, this type of rewriting would therefore have to be implemented on a different layer. For example, it could be implemented on the Derivatives Layer itself, and applied just before it creates new derivative nodes.

Finding a complete set of rewrite rules is very hard, simply because there are too many. A process for automatic rewrite rule discovery would therefore be very helpful. An algorithm for the automatic discovery of rewrite rules could work as follows: create a deterministic automaton (e.g. from the Derivatives Layer) for a given regular expression. Next, compute the equivalent states in this automaton (from the Equivalent States Layer). From any pair of equivalent nodes, we can create a rewrite rule by substituting all symbol nodes by wildcards, where all same symbol nodes are replaced by instances of the same wildcard.

Part III

Software Design and Implementation

Chapter 13

Architectural Design

This chapter provides a high level architectural overview of FIRE Works and FIRE Station.

13.1 Introduction

FIRE Works is a software toolkit, in this case implemented as a library of object-oriented interfaces and classes. FIRE Works is not an application in itself, rather it is to be used as a component in bigger applications.

FIRE Station is an application that uses FIRE Works. The other major components making up FIRE Station are the graph toolkit and a Graphical User Interface (GUI).

The graph toolkit is implemented in the form of a toolkit: it provides generic graph data structures, and associated algorithms, such as algorithms for cycle detection, componentization of graphs and support for graph layout algorithms. The graph toolkit was created as part of this thesis and therefore implemented with FIRE Station in mind.

The GUI is very specific to FIRE Station, and glues the other components together. It provides the user with means of manipulating FIRE Works, for example by creating new regular expressions, layers, and rewrite rules. It also allows manipulation of the graph layout algorithms, for example by determining node visiblity and layout direction.

13.1.1 Object Oriented Design

Both FIRE Works and FIRE Station were designed from the ground up in an object-oriented fashion. [Boo94], [Bud97], and [Mey98] provide information on object-oriented software design and implementation. Additionally we use some design patterns, to which a good introduction is given in [GHJV95].

13.2 Potential Users

To better understand the design of FIRE Works and FIRE Station, it is important to know what kinds of users each serves. Since FIRE Works is provided as a toolkit, typical

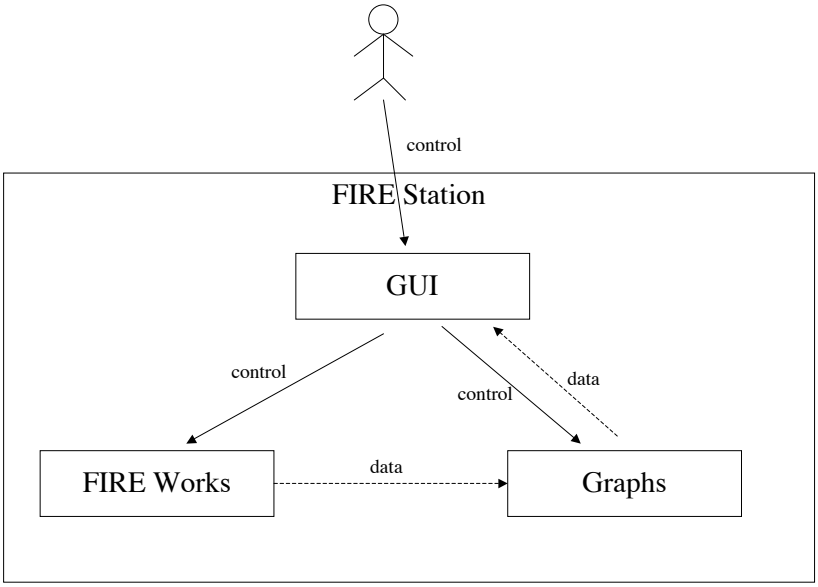

Figure 13.1: Control Flow in FIRE Station

users of FIRE Works are programmers. Specifically, we distinguish scientific programmers, who add new layers to FIRE Works, and application programmers, who integrate the toolkit into an application. We refer to applications that use the FIRE Works toolkit as *client applications*. FIRE Station is an example of a client application of FIRE Works.

FIRE Station has a wider potential audience. This audience includes the scientific programmers interested in FIRE Works, who might want to see their new algorithm visualized (to generate images for their publications), or to simulate new automata. But it also includes designers of regular expressions and automata, who do not necessarily design algorithms, but who want a tool to visually manipulate and debug their regular expressions.

13.3 Control Flow

Interactions between the FIRE Station sub-systems are designed according to the Model-View-Controller (MVC) design pattern (see [SSJ01]). FIRE Works and the graph toolkit are kept completely isolated from each other, meaning that FIRE Works does not have any special provisions for generating graphs, and that the graph toolkit does not know how to create graphs from layers.

In this case FIRE Works is the Model, FIRE Station is the Controller, and the graphs are Views. FIRE Station is responsible for performing operations on FIRE Works when instructed to do so through the GUI. After modification of FIRE Works, FIRE Station is responsible for updating the graphs, as well as updating the visualization of the graphs as they are shown to the user in the GUI. This is illustrated in Figure 13.1.

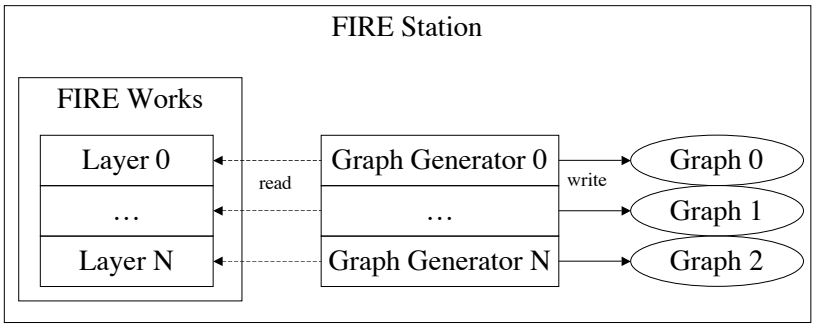

Figure 13.2: Graph Generators

13.4 Extendibility

Because we want FIRE Works and FIRE Station to be easily extendible, layers are created as "plug-ins", which are registered into FIRE Works. When adding a new (relational) layer to FIRE Works, the programmer of that layer also needs to provide a so-called *graph generator* to FIRE Station. Graph generators are small pieces of glue code that know how to read from a (specific) layer, and transform write this data into a graph from the graph toolkit. By adding a new graph generator for each layer, FIRE Station is able to generate the corresponding graph for that new layer. This modular set-up is illustrated in Figure 13.2. If the graph generator is not provided, FIRE Works and FIRE Station will work correctly, however there is no way to make the layer visible.

It is not strictly necessary to have a one-to-one correspondence between layers and graphs. It is, for example, possible to create a graph that represents the *minimized DFA*, which can be constructed by combining the derivatives layer and the equivalent states layer, without an actual minized DFA layer. Note that in that case the minimized DFA is not available in FIRE Works for further manipulation, and care should be taken not to shift to much logic from FIRE Works into the graph generators.

Though graph generators deal with both graphs and layers, they belong to neither FIRE Works nor the graph subsystem. Instead they are glue code that is part of FIRE Station. A future enhancement would allow layers with common characteristics, to implement an well-defined interface that provides access to the layer. For example, all layers that compute a transition function could implement an interface that allows access to this transition function. It is then possible to make a generic graph generator that knows how to construct a graph from any transition function layer. Besides graph generators, there are also node-property generators (NPGs), which are nearly identical to graph generators, except that they operate on property layers, and generate visual attributes for nodes (colors, text labels, shapes and other markup). NPGs are also glue code that is part of FIRE Station, and need to be provided for each new property layer that needs to be visualized.

13.5 Graph Layout

FIRE Station currently supports two-dimensional "node and edge" drawings. There exist many algorithms to perform graph layout for this type of graph drawing. New layout algorithms can be added to FIRE Station by wrapping the layout algorithms into a functional class (functor), satisfying a common interface. The graph layout algorithms are therefore not part of the graph toolkit, since they are very much unique to the FIRE Station approach. They do rely heavily on the functionality provided by the graph toolkit.

As described in Chapter 11, p. 61, node and edge visibility detection requires a set of root nodes. These root nodes are provided by FIRE Works, in the form of *user nodes*, which are those nodes that were explicitly created by the user. There are three ways to create user nodes:

- The user types in a new regular expression which is parsed into a parse tree; the root of this tree is a user node.
- The user explicitly applies an operator to one or more node. The resulting nodes are user nodes.
- The user turns an existing node into a user node explicitly, through the GUI.

13.6 Graphical User Interface Requirements

The GUI ties together FIRE Works and the graph sub-system (the graph toolkit and layout algorithms). Here we present the functionality that the GUI provides for the manipulation of each sub-system.

13.6.1 Manipulation of FIRE Works

It is important to point out that the actual layers in FIRE Works are hidden from the user, or rather, they are converted into graphs and node properties. The user can instantiate graph generators and node-property generators by the name under which they were registered to FIRE Station (see Section 13.4). When the same layer(s) have multiple abstractions, those abstractions can be registered under the same name, which results in them being created and deleted simultaneously. For example the Regular Expression Layer abstractions includes both a graph and a node property, while the derivatives layer abstraction includes both a graph and an automaton.

There are two ways for the user to create new regular languages. The first method is by typing a regular expression into FIRE Station, which is then parsed into parse tree. The second method is to apply an operator to a selection of nodes.

Other aspects of FIRE Works that need to be accessible through the GUI are Global Common Subexpression Elimination (GCSE) and rewriting, as discussed in Chapter 12, p. 67. Specifically, the user needs to be able to switch GCSE and rewriting on and off, and add, delete and modify rewrite rules.

13.6.2 Graph Layout Manipulation

The GUI allows the user to influence the layout and visibility of the graphs, by deciding whether a graph should be visible or invisible, whether it should influence node visibility, and whether it should be layout driving. How each of these elements influences graph layout has already been discussed in Chapter 11. The GUI is responsible for modifying these parameters, but the graph toolkit implements the actual layout algorithm.

Graphs are rendered using the node-and-edge style drawing. The user can exert some control over the way this drawing is done. Depending on the implementation platform, properties such as the edge thickness, color, dash-style, text label font and font size may be modified for each graph individually.

In the final picture of the graphs, the user can select one or more nodes through a point-and-click interface. For the selected nodes, the regular expression represented by each of those nodes is shown. As discussed in the previous section, the user can delete the selected nodes, or apply a regular operator to them, which will result in a change of the graphs, and in turn in an update of the graph layout.

Chapter 14

Detailed Design

This chapter discusses the detailed design of FIRE Works, the graph toolkit and FIRE Station.

14.1 Notation

FIRE Works and the graph components are implemented in the C++ language, while FIRE Station is implemented using a combination of C++ for the platform independent part and Objective-C for the platform specific user interface code. We will therefore specify interfaces in the C++ syntax. We adhere to the following conventions in naming classes and interfaces: classes are prefixed with two letter abbreviations; classes that are part of FIRE Works start with `FW`, while those that are part of FIRE Station start with `FS`. Classes that merely provide a set of virtual methods, but no actual implementation are referred to as interfaces, and will have the capital letter `I` as an additional prefix, e.g. `IFWLayer`.

The names of variables are capitalized so that each word in the name has the first letter capitalized, except for the first word: `parseTree`, unless the first word is an abbrevation, in which case it remains uppercase, as in `FIREWorks`. For pointer variables we prefix a lower-case `p`, and make the letter succeeding it a capital: `pLayer`. For object member variables we use the prefix `m_`, as in `m_parseTree`, or if the member is a pointer `m_pParseTree`.

14.2 FIRE Works

14.2.1 Node Management

Nodes are the most fundamental concept in FIRE Works, and we need to potentially represent billions of nodes. Each layer has particular information associated with each node. For these reasons we have chosen to represent nodes as simply as possible, i.e. not by objects, but by integers:

```
typedef unsigned int node_t;
```

Data related to nodes can then be stored with each layer, using data structures that associate an integer with the particular data. Data structures associating integers with data can easily be implemented using maps or arrays. Maps are quite efficient when storing information for some of the nodes (sparse usage), however arrays are better when data is stored for every single node. To make array usage practical, we ensure that node identifiers are generated in consecutive numerical order, starting at 0. This way, we can store store node information consecutively in an array without leaving gaps in the array. We can then retrieve information about a particular node by indexing such arrays with the node identifier.

Potential issues arise with array usage when nodes are deleted. It is not possible to remove individual elements from an array in the way it is possible with a map; at best we can assign them an "undefined" value. Our solution is to keep a running list of node IDs of nodes that have been deleted, and through garbage collection, we recycle the identifiers for those nodes, and thus the spaces in the array they occupied. We will discuss the process for garbage collection in Section 14.2.4.

14.2.2 Layer Classes

Layers build on top of one-another, and therefore need to know when their underlying layers change. In an early version of FIRE Works, the observer pattern was used: each layer was an observable and every layer observes the layers on which it relies. This turned out to be quite inefficient; the observing layer can receive several observer events for each new node. Another approach, also too inefficient for our purposes, is to do the computations "on demand": every time a layer is accessed for information on a particular node, a check has to be made to make sure that this information has been computed, and if not, perform this computation.

Instead of the observer pattern or computation on demand, we have chosen to have a more centrally managed architecture, which updates each of the layers in turn. Because the layers form a directed acyclic graph, there is a partial ordering of the layers such that each layer comes after all the layers on which it depends. The central manager updates the layers in this order, providing the set of new nodes to be processed by the layer. Layer updates are initiated by the class `FWProject`, which will be discussed in the next section. Each layer is implemented as a class, which has to perform the updates described above, as well as several house-keeping tasks such as serialization and garbage collection. To provide the *FWProject* class access to these methods, all layers implement the interface `IFWLayer`:

```
class IFWLayer
{
  virtual void update( const nodeset_t& nodes ) = 0;
  virtual nodeset_t getRequiredNodes(node_t node) const = 0;
  virtual void removeNodes( const nodeset_t& garbage ) = 0;
  virtual void serialize( Serializer& out ) const = 0;
  virtual void deserialize( Deserializer& out ) = 0;
}
```

14.2.3 FIRE Works Projects

As common to many software applications, we want FIRE Works to support project based (or document based) applications. Although within each project we have at most one instance of each layer type (e.g. only one regular expression layer), each project may need to have its own instance. To manage such environments, we use the class `FWProject`. This class is responsible for storing the set of layers within the current environment, as well as the logic for updating the layers (in the right order), serialization and node compaction.

Given the requirement for extendibility of FIRE Works with new layers, without modifying existing FIRE Works code, `FWProject` needs a good way to deal with new layer types. Class `FWProject` does not know the details of any of the layer classes, except what is provided through the `IFWLayer` interface.

Client applications of FIRE Works have to treat layers as read-only objects, and make modifications through an `FWProject` object. This is the only way to guarantee consistency between layers. For example, problems would arise if an application would create derivative edges between a pair of nodes directly. The derivatives automaton would now represent a different regular language than the regular expressions for those nodes. Instead, a regular expression yielding the desired derivatives should be created via the FIRE Works component interface, which in turn, through the Regular Expression Layer, would update the Derivatives Layer.

14.2.4 Root Nodes and Garbage Collection

Class `FWProject` stores a set of nodes which contain all nodes representing regular expressions that were explicitly created by the user (e.g. the root of a parse tree for a user provided regular expression). These nodes are called `user nodes` and form the root nodes when determining reachability of other nodes. Nodes that are not reachable from the roots through the closure of the `getRequiredNodes` method are not serving any purpose. Such nodes are called *dead nodes* and, their IDs can be recycled next time garbage collection occurs.

Garbage collection is a two step process. In the first step, a set of dead nodes is determined, via the `FWProject` method `markDeadNodes`. The set of marked dead nodes are made available to FIRE Works's client application. This allows the client application to remove any references to these dead nodes (in FIRE Station we remove edges to and from dead nodes in each graph). After the client application is done, it notifies `FWProject` via method `recycleDeadNodes`, which instructs all layers to remove all references to the dead nodes, and finally adds the dead nodes to the recycled nodes. Recycled nodes can then used as fresh, new, node IDs.

```
const nodeset_t& FWProject::markDeadNodes() const;
void FWProject::recycleDeadNodes() const;
```

14.3 Graph Subsystem

Graphs representations are implemented in a separate code base, with the goal of eventually using an existing graph tool kit. Although graphs representing layers are often straightforwardly extracted from those layers, a real-time conversion from layer to graph is too time

consuming, since graphs often need to be accessed frequently and in a random-access fashion, for example during the graph layout process. Therefore, graphs are generated incrementally, by adding nodes and edges as they are added to FIRE Works.

Graphs are generated from layers by a variety of functor classes which construct edges, and stored in a generic graph class. These graph functor classes implement the interface `IFSGraphGenerator`, so that all graph generators can be managed centrally and all graphs can be updated automatically. When adding a new (relational) layer to FIRE Works, a corresponding graph generator needs to be written and registered for the layer to become visible as a graph in FIRE Station.

```
class IFSGraphGenerator
{
  virtual void updateGraph( FWProject& project,
                FSGraph& graph ) const;
}
```

When a new layout has to be performed, the process managing the graph generators updates all graphs. It then combines the visible and driving graphs into single graphs (see 11), and calls a layout function to perform the actual layout. Property layers are represented through node properties (color, labels and shape), rather than graphs. In contrast with graphs, node properties *are* generated in real-time. Functor classes are used to generate node properties as new nodes are created, and implement the `IFSNodePropertyGenerator` interface. When a new (property) layer is added to FIRE Works, a corresponding node property generator needs to be added to FIRE Station for the layer to be visible.

```
class IFSNodePropertyGenerator
{
  virtual void setProperty( FWProject& project,
    node_t node, FSNodeProperty& nodeProperty ) const;
}
```

Graph rendering (drawing) is not part of the graph subsystem, because drawing functions are mostly platform specific; therefore graph rendering is part of the platform specific FIRE Station GUI component.

Chapter 15

Implementation

15.1 Introduction

In this chapter we discuss some details of the implementation of the FIRE Works toolkit and FIRE Station application. The information is a great starting place for anyone who wants to dive into the FIRE Station and FIRE Works code, whether for bug fixing or adding new features.

15.1.1 Choice of Languages

Because FIRE Works and FIRE Station are new applications, without legacy code, we were not constrained in the choice of an implementation language.

Our choice of C++ for FIRE Works was made for the following reasons:

- Object-orientation: C++ supports inheritence and virtual functions
- Portability: the features of C++ are well documented, therefore C++ code can be compiled on different compilers and different platforms
- Availability: C++ is available on every platform of interest (Windows, Linux, and Mac OS X)
- Efficiency: if FIRE Works is to be competitive with other toolkits, its compiled code needs to be efficient.

In contrast, our choice of Objective-C for the FIRE Station GUI was for opposite reasons. Although the Objective-C language is available on many platforms through the GNU Compiler Collection, we used it in conjunction with Apple's Cocoa GUI framework, which is only available for Mac OS X. This has resulted in the GUI part of the code being platform dependent.

Instead, a separate GUI implementation will be required for other platforms. Initially we have tried several cross-platform GUI framework, and in fact implemented FIRE Station almost completely within TrollTech's Qt [Tro] GUI Toolkit. However, we found that these

frameworks are no match for platform specific GUI frameworks, in terms of graphics quality, efficiency and ease of programming.

Although Objective-C program compiles into slower executables than C++ programs, the language and available GUI design applications made the incremental development of the FIRE Station GUI go much faster than it would have in C++. This was important because the exact GUI requirements were not pinned down before the project was started, but rather they evolved as more knowledge of potential and limitations became available.

Additionally, we have made use of the so-called Objective-C++ language, which is a hybrid of the C++ and Objective-C language. This language permits access to both C++ and Objective-C classes in the same source code, thereby allowing us to use the classes of the FIRE Works toolkit in the FIRE Station code.

15.1.2 Useful References

We refer the reader to a number of books on the programming languages that are relevant to the source code:

- [Lip99] and [Str97] are (basic respectively thorough) introductory books on C++
- [Cop92], [Cop98] and [SE90] discuss various advanced C++ topics
- [Koc03] and [Dav02] discuss Objective-C in the context of Mac OS X and the Cocoa framework
- [Com01] discusses the Objective-C++ hybrid language

15.2 Code Structure

As is common when writing C++ code, we have split each class implementation into a class declaration file, ending in `.h` and a class definition file, ending in `.cpp`. For the Objective-C code, the class definition file ends in `.m`, and for the hybrid Objective-C++ files, the class definition files end in `.mm`.

15.3 Platforms

All of FIRE Works and the graph toolkit have been implemented in the C++ programming language, and tested with the GNU Compiler Collection (gcc), version 3.3 on Apple Mac OS X 10.3. Most of the layers were straighforward adaptations to C++ of the implementation definition of the layers of Part II.

The FIRE Station GUI is written in the Objective-C programming language using Apple's Cocoa framework, and runs under Mac OS X 10.3. The integration between the Objective-C and C++ code was made possible thanks to the hybrid Objective-C++ language.

15.4 Source Code Documentation

Documentation for the implementation is contained in the source code in the form of comments conforming to the JavaDoc standard ([Mic04]) .

15.5 Obtaining FIRE Station

FIRE Station is currently not yet available to the public. When it becomes available (either in source code or binary form), it will be posted on the FASTAR Group website: `http://www.fastar.org`.

Part IV

Epilogue

Chapter 16

Conclusions

We will conclude this report with a summary of the work that has been done and take a look at directions for future development.

16.1 Summary

The goal of this thesis research was to design and implement (a prototype for) a software toolkit for the coherent, consistent and "loss-less" implementation of regular language algorithms, and a software application for the visualization and manipulation of the underlying graph data structures in the implementation of that toolkit.

We discuss the steps taken to meet these goals by chapter:

Chapter 3 discussed the outline for the FIRE Works framework, and introduced basic concepts such as nodes and layers. We designed the framework so that layers build on top of one-another, using a single root layer.

Chapter 4 through 10 each discussed a specific regular language aspect or algorithm in the form of a layer. In the implementation of each layer, all new nodes are created through a set of functions provided by the Regular Expressions Layer (the root layer). This ensures that a regular expression is associated with each new node, thus achieving loss-less operations. It is up to the developer of a new layer to ensure that the regular expressions for newly created nodes are correct within the context of that layer; this is outside of the control of the core framework for FIRE Works.

Chapter 11 described a method of computing a node layout from multiple graphs sharing a single set of nodes. The process allows the user control through three different sets of graphs: the first set determines node visibility, the second set drives the node layout, and the third set are the graphs whose edges are actually rendered.

Chapter 12 discussed three potential methods of reducing the number of nodes created through the Regular Expression Layer. First, parse tree flattening transforms parse trees into a normal form that ensures all identical regular expressions are represented by identical parse trees. Next, global common subexpression elimination reduces parse trees that are identical (after normalization) to single nodes. And finally, rewriting uses a set of rules to transform

regular expressions from one form into another, potentially increasing the number of identical expression that can be removed through subexpression elimination.

The implementation of FIRE Works and FIRE Station were approached in an object-oriented manner. In Chapter 13 we discussed the Architectural Design, specifically how the FIRE Station ties the FIRE Works and graph subsystem together. The Detailed Design in Chapter 14 discussed issues internal to the various software components specific to their implementation in the Objective-C and C++ programming languages.

Finally, Chapter 15 touched on various details of the Implementation, including programming languages, software platforms, and source code documentation.

As with the construction of any piece of software, a number of new questions, ideas and unforeseen issues have come up. We summarize the most interesting ones in the next section.

16.2 Future Work

The potential extensions and improvements to FIRE Works and FIRE Station are virtually unlimited. Throughout this report we have already mentioned some of these additionas and we will summarize them here:

1. Support for additional regular operators, the missing boolean operators as well as character classes and repeat ranges.
2. Support for regular relations and transducers.
3. Import of existing automata, which requires the recovery of a regular expression for the automaton.
4. Additional layers, for example for the position automaton [Glu61], language preorder relations [CC04], automaton reversal, automaton minimization and automaton determinization.
5. Automatic discovery of rewrite rules as discussed in Chapter 12.
6. Additional means for graph manipulation, for example hiding of subgraphs. Additional graph layout algorithms and more control over the layout.
7. Graph visualization in three dimensions.

Bibliography

[Ant96] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[BETT99] Guiseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[BJKO00] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software - Practice & Experience*, 30:259–291, 2000.

[Boo94] Grady Booch. *Object oriented analysis and design, with applications*. Benjamin/Cummings, 2nd edition, 1994.

[Brz64] Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11:481–494, 1964.

[Bud97] Timothy A. Budd. *An introduction to object-oriented programming*. Addison-Wesley, 2nd edition, 1997.

[CC04] Jean-Marc Champarnaud and Fabien Coulon. Nfa reduction algorithms by means of regular inequalities. *Theor. Comput. Sci.*, 327(3):241–253, 2004.

[CFR04] Ryan Cavalcante, Thomas Finley, and Susan H. Rodger. A visual and interactive automata theory course with jflap 4.0. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 140–144. ACM Press, 2004.

[Coc70] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[Com01] Apple Computer. Objective-c++ release notes, 2001. `http://developer.apple.com/releasenotes/Cocoa/Objective-C++.html`.

[Cop92] James O. Coplien. *Advanced C++: programming styles and idioms*. Addison-Wesley, 1992.

[Cop98] James O. Coplien. *Multi-Paradigm DESIGN for C++*. Addison-Wesley, 1998.

[CZ00] Pascal Caron and Djelloul Ziadi. Characterization of glushkov automata. *Theor. Comput. Sci.*, 233(1-2):75–90, 2000.

[Dav02] James Duncan Davidson. *Learning Cocoa with Objective-C*. O'Reilly, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Glu61] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw., Pract. Exper.*, 30(11):1203–1233, 2000.

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computation*, pages 189–196. Academic Press, New York, 1971.

[Hop94] Mark Hopkins. Regular expression software. comp.compilers newsgroup, 1994. `ftp://iecc.com/pub/file/regex.tar.gz`.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Authomata Theory, Languages and Computation.* Addison-Wesley, 1979.

[IY03] Lucian Ilie and Sheng Yu. Reducing nfas by invariant equivalences. *Theor. Comput. Sci.*, 306(1-3):373–390, 2003.

[Koc03] Stephan Kochan. *Programming in Objective-C*. Sams, 2003.

[Lip99] Stanley B. Lippman. *Essential C++*. Addison-Wesley, 1999.

[LPRGS03] Sylvain Lombardy, Raphaël Poss, Yann Régis-Gianas, and Jacques Sakarovitch. Introducing vaucanson. In *CIAA*, pages 96–107, 2003.

[LS02] Sylvain Lombardy and Jacques Sakarovitch. *Vaucanson. A package for drawing automata. Presentation and user's manual*, 0.1 edition, May 2002.

[Mey98] Bertrand Meyer. *Object-Oriented Software Construction*. Addison-Wesley, 2nd edition, 1998.

[Mic04] Sun Microsystems. Javadoc Tool Home Page, 2004. `http://java.sun.com/j2se/javadoc/`.

[MPR98] Mehryar Mohri, Fernando C.N. Pereira, and Michael Riley. A rational design for a weighted finite-state transducer library. In *Automata Implementation. Second International Workshop on Implementing Automata, WIA '97*. Springer Verlag, 1998. Lecture Notes in Computer Science 1436.

[SE90] Bjarne Stroustrup and M. Ellis. *The annotated C++ reference manual*. Addison-Wesley, 1990.

[SR03] K. Sen and G. Rosu. Generating optimal monitors for extended regular expressions. In *Proceedings of Runtime Verification (RV03)*, volume 89-2. Elsevier, 2003.

[SSJ01] I. Singh, B. Stearns, and M. Johnson. *Designing Enterprise Applications with the J2EE Platform, Second Edition.* Addison-Wesley, 2001.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[Tro] Trolltech. Qt product overview. object-`http://www.trolltech.com/products/qt/index.html`.

[vN97] Gertjan van Noord. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation.* Springer Verlag, 1997. Lecture Notes in Computer Science 1260.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Technische Universiteit Eindhoven, 1995.

[Wat01] Bruce W. Watson. An incremental DFA minimization algorithm. In Lauri Karttunen, Kimmo Koskenniemi, and Gertjan van Noord, editors, *Proceedings of FSMNLP 2001, ESSLLI workshop*, Helsinki, August 2001.

[WD03] Bruce W. Watson and Jan Daciuk. An efficient incremental DFA minimization algorithm. *Natural Language Enineering*, 9(1):49–64, March 2003.