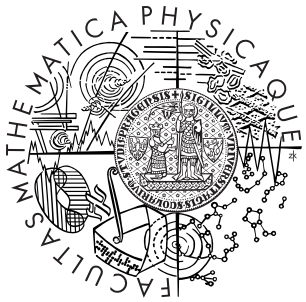
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA



Peter Černo

Prostredie pre reštartovacie automaty

Kabinet software a výuky informatiky

Vedúci bakalárskej práce: RNDr. František Mráz, CSc.
Študijný program: Obecná informatika

Praha, 2008

Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Peter Černo

An environment for restarting automata

Department of Software and Computer Science Education

Supervised by RNDr. František Mráz, CSc.
Study program: General Computer Science

Prague, 2008

Acknowledgements

I would like to thank RNDr. František Mráz, CSc. for his time and all advices and suggestions that led to writing this thesis.

I declare that I have written all of the thesis on my own with the exceptions explicitly mentioned, and that I cited all used sources of information. I agree with public availability and lending of the thesis.

In Prague 2nd November 2008 Peter Černo

Contents

Názov: Prostredie pre reštartovacie automaty
Autor: Peter Černo
Katedra (ústav): Kabinet software a výuky informatiky
Vedúci bakalárskej práce: RNDr. František Mráz, CSc.
e-mail vedúceho: Frantisek.Mraz@mff.cuni.cz

Abstrakt: Reštartovacie automaty sú lingvisticky motivované modely automatov, ktoré môžu byť použité napríklad na kontrolu správnosti viet. Hlavným cieľom tejto práce je vytvoriť špecializovaný program, ktorý umožní jednoduchý interaktívny návrh a testovanie týchto automatov a poskytne špecializované nástroje určené na učenie konečných automatov a definovanie jazykov. Práca prezentuje teoretické základy a uvádza formálnu definíciu reštartovacieho automatu. Ďalej sú v práci diskutované možnosti implementácie takéhoto systému a je popísaná skutočná realizácia systému. K práci je priložená užívateľská príručka.
Kľúčové slová: regulárne jazyky, reštartovací automat, redukčná analýza, gramatická inferencia

Title: An environment for restarting automata
Author: Peter Černo
Department: Department of Software and Computer Science Education
Supervisor: RNDr. František Mráz, CSc.
Supervisor's e-mail address: Frantisek.Mraz@mff.cuni.cz

Abstract: Restarting automata are linguistically motivated models of automata that can be used e.g. in checking correctness of a sentence. The main subject of this work is to design a specialized program which allows an easy design and testing of these automata and provides specialized tools for learning finite automata and defining languages. The thesis presents theoretical background and gives formal definition of restarting automaton. Then the possibilities of implementation of such system are discussed and the actual implementation is described. The user guide is included in the thesis.
Keywords: regular languages, restarting automata, analysis by reduction, grammatical inference

Preface

Restarting automaton was introduced as a model that can be used in analysis by reduction in linguistics. Analysis by reduction consists in stepwise reductions of a given extended sentence until a correct simple sentence is obtained. If this simple sentence is accepted then the whole extended sentence is correct. Each simplification replaces a short part of the sentence by an even shorter one. Restarting automata have been studied for several years now. Few tools for manipulating restarting automata have been developed and several attempts for learning restarting automata by genetic algorithms have been made. Unfortunately the results are far from being applicable. In spite of this we believe that restarting automata, representing a quite new approach in linguistic and grammatical inference, have a bright future and that it is worth to investigate them.

The main goal of this thesis is to develop a specialized program with a simple user-friendly interface enabling to design and to test restarting automata. This program will be used as a tool for exploring properties of these automata. The project is not intended to provide functionalities sufficient to design complex restarting automata that are able to check correctness of real sentences from natural languages. It allows you only to design simple restarting automata recognizing only simple formal languages with small alphabets consisting of few letters. The main purpose of the project is that a user gets a better insight of restarting automata and perhaps one day he or she will join the research of these automata. This application also allows you to verify some simple conjectures or hypotheses about restarting automata and it is also a good framework on which you can build a more complicated and specialized systems.

The whole thesis is divided into four chapters. In the first chapter we introduce all necessary definitions and theorems without proofs mainly from the theory of automata and formal languages. In this chapter we also give a formal definition of a restarting automaton.

The design of restarting automaton can be a complex task. One of the possibilities how to define a restarting automaton is by using meta-instructions. Regular languages play a main role in defining meta-instructions. One of the methods how to obtain regular languages can be machine learning from examples. Therefore the second chapter introduces learning protocols for machine learning of deterministic finite-state automata (DFA). Learning of DFA means identifying a DFA consistent with presentation comprising of a finite non-empty set of positive and negative examples. DFAs are recognizers of regular languages and regular languages play a main role in the meta-instructions of the restarting automaton. In this chapter we describe three learning protocols: Dana Angluin's L* algorithm, RPNI (Regular Positive and Negative Inference) algorithm and the class of SLT (Strictly Locally Testable) languages.

In the third chapter we outline the development of an application for design and testing of restarting automata. First we capture requirements that define what such system should do. Then we make a rough analysis and describe some core classes of the system. After reading this chapter you should be able to modify or extend the system with your own modules.

The last fourth chapter is a user guide to the application. In this chapter we describe how to work with RestartingAutomaton application.

All chapters are quite independent of each other, but we recommend to start with the first chapter to acquire a good theoretical background which is necessary for understanding the rest of the thesis.

Chapter 1

Theoretical background

In this chapter we introduce all necessary definitions and theorems without proofs mainly from the theory of automata and formal languages. The main sources for this chapter are Barták [1], Hopcroft, Motwani, Ullman [2] and Parekh, Honavar [6].

1.1 Basic definitions and notations

Let Σ be a finite nonempty set of symbols called the *alphabet*, Σ^* be the set of all finite strings over Σ, $\Sigma^+ = \Sigma^* - \{\lambda\}$. Let α, β, γ be strings in Σ^*. Then $|\alpha|$ is the *length* of the string α, λ is a special string called the *null* string and has length 0. Given a string $\alpha = \beta\gamma$, β is the *prefix* of α and γ is the *suffix* or *postfix* of α. A *language* L is a subset of Σ^*. If $L_1, L_2 \subseteq \Sigma^*$ are languages, then $L_1.L_2 = \{uv \in \Sigma^* : u \in L_1 \wedge v \in L_2\}$ is the *concatenation* of the languages L_1 and L_2. The *standard order* of strings of the alphabet Σ is denoted by $<$. The standard enumeration of strings over $\Sigma = \{a, b\}$ is λ, a, b, aa, ab, ba, bb, aaa, ...

1.2 Finite state automata

Definition A *deterministic finite automaton* (*DFA*) A is a five-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where:

Q is a finite nonempty set of *states*,

Σ is an alphabet (finite nonempty set of *input symbols*),

$\delta : Q \times \Sigma \to Q$ is a *transition function* that takes as arguments a state and an input symbol and returns a state,

$q_0 \in Q$ is a *start state* and

$F \subseteq Q$ is a set of *final* or *accepting* states.

In informal graph representation, δ is represented by arcs between states and the labels on the arcs. If q is a state, and a is an input symbol, then there is an arc labeled a from q to $\delta(q, a)$.

Extended transition function $\delta^* : Q \times \Sigma^* \to Q$ is defined inductively: $\delta^*(q, \lambda) = q$, $\delta^*(q, wx) = \delta(\delta^*(q, w), x)$, for all $q \in Q$, $x \in \Sigma$ and $w \in \Sigma^*$.

A state $d_0 \in Q$ such that $\forall a \in \Sigma : \delta(d_0, a) = d_0$ is called a *dead* state.

The *language* of a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is denoted $L(A)$, and is defined by $L(A) = \{w | \delta^*(q_0, w) \in F\}$. The language of A is the set of all strings w that take start state q_0 to one of the accepting states.

If for $L \subseteq \Sigma^*$ there exists a DFA A such that $L = L(A)$, then we say L is a *regular language*.

In the Figure 1.1 is an example of DFA with alphabet $\Sigma = \{a, b\}$ that accepts exactly the words with even number of as and even number of bs.

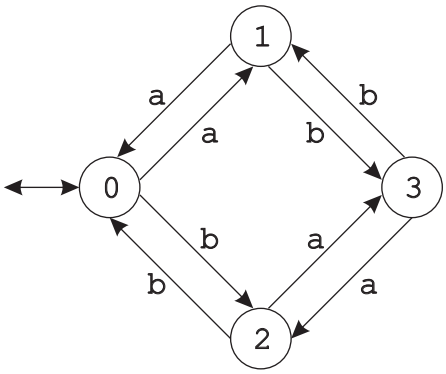


Figure 1.1: Deterministic finite automaton.

This DFA can be represented also in a *tabular form*:

		a	b
<->	0	1	2
	1	0	3
	2	3	0
	3	2	1

Definition A *nondeterministic finite automaton* (*NFA*) A is a five-tuple $A = (Q, \Sigma, \delta, S, F)$, where:

Q is a finite nonempty set of *states*,

Σ is an alphabet (finite nonempty set of *input symbols*),

$\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is a *transition function* that takes as arguments a state and an input symbol and returns a set of states,

$S \subseteq Q$ is a set of *starting states* and

$F \subseteq Q$ is a set of *final* or *accepting* states.

Analogously, in informal graph representation, δ is represented by arcs between states. If p is a state, a is an input symbol and $q \in \delta(p, a)$, then there is an arc labeled a from p to q.

The word $w = x_1 \dots x_n$ from Σ^* is accepted by a nondeterministic finite automaton $A = (Q, \Sigma, \delta, S, F)$ if and only if there exists a sequence of states $q_0, \dots, q_n \in Q$ such that:

$q_0 \in S$,

$q_i \in \delta(q_{i-1}, x_i)$ for all $i \in \{1, \dots, n\}$ and

$q_n \in F$.

λ is accepted if and only if $S \cap F \neq \varnothing$.

We denote $L(A)$ the set of all words from Σ^* that are accepted by A. Apparently every deterministic finite automaton is a special case of nondeterministic finite automaton. But it can be also proved that the reverse implication holds.

Theorem 1.2.1. *If A is a nondeterministic finite automaton then there exists a deterministic finite automaton B such that $L(A) = L(B)$.*

This theorem implies that nondeterministic finite automata recognize exactly regular languages. Deterministic finite automata and nondeterministic finite automata are collectively called *finite state automata*.

Definition Let $A = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. A *labeled example* $(w, c(w))$ for A is such that $w \in \Sigma^*$ and $c(w) = +$ if $w \in L(A)$ (i.e., w is a *positive example*) or $c(w) = -$ if $w \notin L(A)$ (i.e., w is a *negative example*).

Let S^+ and S^- denote some set of *positive* and *negative* examples of A respectively. A is *consistent* with a *sample* $S = S^+ \cup S^-$ if A accepts all positive examples and rejects all negative examples.

A set S^+ is said to be *structurally complete* with respect to DFA A if S^+ covers each transition of A (except the transitions associated with the dead state) and uses every element of the set of final states of A as an accepting state.

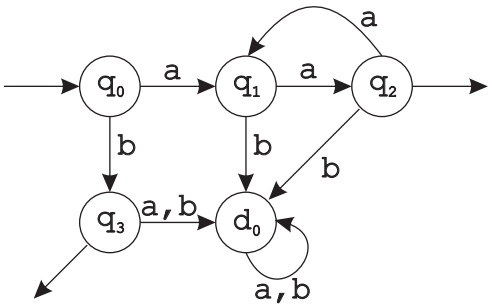


Figure 1.2: Deterministic finite automaton.

It can be verified that the set $S^+ = \{b, aa, aaaa\}$ is structurally complete with respect to the DFA in Figure 1.2. (Here d_0 is a dead state).

Definition Given a set S^+, let PTA(S^+) denote the *prefix tree acceptor* for S^+. PTA(S^+) is a DFA that contains a path from the start state to an accepting state or each string in S^+ modulo common prefixes. Clearly $L(\text{PTA}(S^+)) = S^+$. Learning algorithms such as the RPNI (see the chapter 2) require the states of the PTA to be numbered in standard order. If we consider the set Pref(S^+) of prefixes of the set S^+ then for each state q_i of the PTA there exists exactly one string w_i in the set Pref(S^+) such that $\delta^*(q_0, w_i) = q_i$ and vice-versa. The strings of Pref(S^+) are sorted in standard order and each state q_i is numbered by the position of its corresponding string w_i in the sorted list. The PTA for the set $S^+ = \{b, aa, aaaa\}$ is shown in the following Figure 1.3. Note that its states are numbered in standard order.

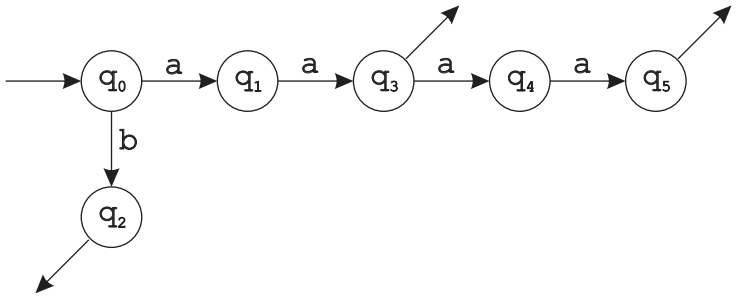


Figure 1.3: PTA.

Definition Let $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be two deterministic finite automata. We say that A_1 and A_2 are *equivalent* if $L(A_1) = L(A_2)$.

Definition A mapping $h : Q_1 \to Q_2$ is called a *homomorphism* if:

1. $h(q_1) = q_2$,
2. $h(\delta_1(q, x)) = \delta_2(h(q), x)$,
3. $q \in F_1 \Leftrightarrow h(q) \in F_2$.

If h is a bijection we call it an *isomorphism* and say that A_1 and A_2 are *isomorphic*.

Theorem 1.2.2. *If there is a homomorphism $h : Q_1 \to Q_2$ of A_1, A_2, then A_1 and A_2 are equivalent.*

Definition Let $A = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. We say that $q \in \Sigma$ is *reachable state* if $\exists w \in \Sigma^* : \delta^*(q_0, w) = q$. Otherwise we say that q is *unreachable*.

Theorem 1.2.3. *If we delete all unreachable states from the DFA A we get an equivalent DFA.*

Definition We say that $p, q \in Q$ are *equivalent* and denote $p \sim q$, if $\forall w \in \Sigma^* : \delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \in F$.

Let us define *i-equivalence* $p \sim^i q$ for all $i \in \{0, 1, \ldots\}$ as $\forall w \in \Sigma^* |w| \leq i : \delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \in F$. Apparently $p \sim q \Leftrightarrow \forall i : p \sim^i q$.

We can construct $\sim^i$ iteratively:

1. $p \sim^0 q$ if and only if $p \in F \Leftrightarrow q \in F$,
2. $p \sim^{i+1} q$ if and only if $p \sim^i q \wedge \forall x \in \Sigma : \delta(p, x) \sim^i \delta(q, x)$.

Theorem 1.2.4. *Let us denote $R_i = Q/_{\sim_i}$ for all $i \in \{0, 1, \ldots\}$. Then the following holds:*

1. *R_{i+1} refines R_i.*
2. *if $R_{i+1} = R_i$ then for all $t > 0 : R_{i+t} = R_i$.*
3. *if $|Q| = n$ then $\exists k : 0 \leq k \leq n-1$ and $R_{k+1} = R_k$.*
4. *if $R_{i+1} = R_i$ then $p \sim q \Leftrightarrow p \sim^i q$.*

This gives us the following algorithm for finding equivalent states:

Algorithm for finding equivalent states

Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: $R = Q/_\sim$ where $\sim$ is state equivalence of DFA A

begin
  construct R_0
  repeat
    construct R_{i+1} from R_i
  until $R_{i+1} = R_i$
  return R_i
end

Definition Let $\equiv$ be an equivalence on Q. We say that $\equiv$ is an *automaton congruency* if $\forall p, q \in Q : p \equiv q \Rightarrow (p \in F \Leftrightarrow q \in F) \wedge \forall x \in \Sigma : \delta(p, x) \equiv \delta(q, x)$.

Theorem 1.2.5. *The equivalence $\sim$ of states is an automaton congruency.*

Definition Let $A = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton and $\equiv$ be an equivalence on Q. We define a *quotient automaton* $A/_{\equiv}$ to be $A/_{\equiv} = (Q/_{\equiv}, \Sigma, \delta_{\equiv}, \{[q_0]_{\equiv}\}, \{[q]_{\equiv} | q \in F\})$, where $\delta_{\equiv}([q]_{\equiv}, x) = \{[\delta(p, x)]_{\equiv} | p \in [q]_{\equiv}\}$.

$A/_{\equiv}$ is obtained by merging states of A that belong to the same equivalence class of $\equiv$.

Sometimes it is useful to consider only a partition of the set of states. If π is a partition of the set of states Q then we define A_π to be $A/_{\equiv}$ where $\equiv$ is an equivalence on the set Q such that $\pi = Q/_{\equiv}$. Analogously if $q \in Q$ then we define $[q]_\pi$ to be $[q]_{\equiv} = \{p \in Q | p \equiv q\}$.

For example, the quotient automaton corresponding to the equivalence $\equiv$ on the set $Q = \{q_0, q_1, q_2, q_3\}$ of the states of the DFA in Figure 1.2 (without considering the dead state d_0) with partition $Q/_{\equiv} = \{\{q_0, q_1\}, \{q_2\}, \{q_3\}\}$ is shown in the Figure 1.4.

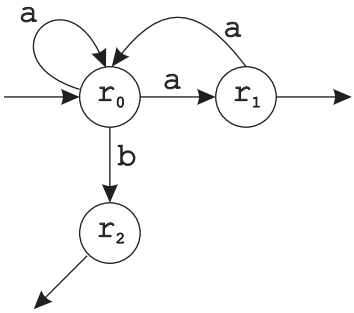


Figure 1.4: Quotient automaton.

Theorem 1.2.6. *If A is a deterministic finite automaton and $\equiv$ is an automaton congruency then $A/_{\equiv}$ is a deterministic finite automaton equivalent with A. In this case we use an alternative definition of quotient automaton: $A/_{\equiv} = (Q/_{\equiv}, \Sigma, \delta_{\equiv}, [q_0]_{\equiv}, \{[q]_{\equiv} | q \in F\})$, where $\delta_{\equiv}([q]_{\equiv}, x) = [\delta(q, x)]_{\equiv}$.*

Theorem 1.2.7. *$A/_{\sim}$ is a deterministic finite automaton equivalent with A with no distinct equivalent states.*

Definition We say that a DFA is *reduced* if it does not have unreachable states and also if it does not have distinct equivalent states.

We say that DFA B is a *reduction* of DFA A if B is reduced and if A and B are equivalent.

Theorem 1.2.8. *For every DFA A there exists its reduction B.*

Theorem 1.2.9. *Two reduced deterministic finite automata are equivalent if and only if they are isomorphic.*

Corollary 1.2.10.

1. *The reductions of two equivalent deterministic finite automata are isomorphic.*
2. *For every deterministic finite automaton there exists exactly one its reduction (except for isomorphism).*

1.3 Grammars

Definition A *production system* is a couple $R = (V, P)$, where:

V is a finite alphabet and

P is a finite set of *production rules*, where production rule (production) is a couple (u, v), where $u, v \in V^*$ (we write $u \to v$).

We say that w can be *directly rewritten* to z (we write $w \Rightarrow z$) if $\exists u, v, x, y \in V^*$ such that $w = xuy$, $z = xvy$ and $(u, v) \in P$.

We say that w can be *rewritten* to z (we write $w \Rightarrow^* z$) if $\exists u_1, \ldots, u_n \in V^*$ such that $w = u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_n = z$.

The sequence $u_1, \ldots, u_n$ is called a *derivation*. If $\forall i \neq j : u_i \neq u_j$, then we call this sequence a *minimal* derivation.

Definition A (generative) *grammar* is a four-couple $G = (V_N, V_T, S, P)$, where:

V_N is a finite nonempty set of *nonterminal symbols*,

V_T is a finite nonempty set of *terminal symbols*,

$V_N \cap V_T = \varnothing$,

$S \in V_N$ is a *starting* nonterminal symbol and

P is a system of production rules $u \to v$, where $u, v \in (V_N \cup V_T)^*$ and u contains at least one nonterminal symbol.

A language $L(G)$ *generated* by a grammar G is defined as $L(G) = \{w \in V_T^* | S \Rightarrow^* w\}$. Grammars G_1 and G_2 are *equivalent* if $L(G_1) = L(G_2)$.

Chomsky hierarchy is a classification of grammars based on the form of the production rules.

1. grammars of type 0 (*recursively enumerable* languages $\mathfrak{L}_0$) - production rules in general form.

2. grammars of type 1 (*context-sensitive* languages $\mathfrak{L}_1$) - only production rules of the form $\alpha X \beta \rightarrow \alpha w \beta$, $X \in V_N$, $\alpha, \beta \in (V_N \cup V_T)^*$, $w \in (V_N \cup V_T)^+$. The only exception is a rule $S \rightarrow \lambda$, but in that case S does not occur in the right side of any other production rule.

3. grammars of type 2 (*context-free* languages $\mathfrak{L}_2$) - only production rules of the form $X \rightarrow w$, $X \in V_N$, $w \in (V_N \cup V_T)^*$.

4. grammars of type 3 (*regular/ right linear* languages $\mathfrak{L}_3$) - only production rules of the form $X \rightarrow wY$, $X \rightarrow w$, $X, Y \in V_N$, $w \in W_T^*$.

It can be proved that the following relations hold: $\mathfrak{L}_0 \supset \mathfrak{L}_1 \supset \mathfrak{L}_2 \supset \mathfrak{L}_3$.

1.4 Restarting automata

The main source for this section is Mráz, Otto, Plátek [4]. Restarting automata can be used to model so called syntactic reduction systems. These reduction systems represent a base for analysis by reduction which is a linguistically motivated method for checking correctness of a sentence. So we are going to start with the definition of reduction systems.

Definition A *syntactic reduction system* is a tuple $R = (\Sigma, \Gamma, \vdash_R, L_S)$, where:

Σ is a finite nonempty *input alphabet*,

Γ is a finite nonempty *working alphabet* containing Σ,

$\vdash_R \subseteq \Gamma^* \times \Gamma^*$ is a *reduction relation* and

$L_S \subseteq \Gamma^*$ is a set of *simple sentential forms*.

Any string from Γ^* is called a *sentential form*. The reflexive and transitive closure of $\vdash_R$ is denoted by $\vdash_R^*$.

With each syntactic reduction system $R = (\Sigma, \Gamma, \vdash_R, L_S)$ we associate the following two languages:

1. the *input language* of R: $L(R) = \{u \in \Sigma^* | \exists v \in L_S : u \vdash_R^* v\}$,
2. the *characteristic language* of R: $L_C(R) = \{u \in \Gamma^* | \exists v \in L_S : u \vdash_R^* v\}$. Trivially, $L(R) = L_C(R) \cap \Sigma^*$.

Definition A syntactic reduction system $R = (\Sigma, \Gamma, \vdash_R, L_S)$ is called:

length-reducing if, for each $u, v \in \Gamma^*$, $u \vdash_R v$ implies $|u| > |v|$,

locally reducing if there exists a constant $k > 0$ such that, for each $u, v \in \Gamma^*$, $u \vdash_R v$ implies that there exist words $u_1, u_2, x, y \in \Gamma^*$ for which $u = u_1 x u_2$ and $v = u_1 y u_2$, and $|x| \leq k$.

We are interested in syntactic reduction systems that are length-reducing and locally reducing. In the case of a natural language, the relation $\vdash_R$ corresponds to a stepwise simplification of (extended) sentences, and L_S corresponds to (correct) simple sentences. Apparently, the analysis by reduction is nondeterministic, but it has so-called *error preserving property*:
if $u \vdash_R^* v$ and $u \notin L_C(R)$, then $v \notin L_C(R)$.
Analysis by reduction can be modeled for instance by the RRWW-automaton. Instead of its formal definition (stated in Jančar and col. [3]) we will use its alternative representation adapted from Niemann, Otto [5].

Definition A *restarting automaton* is a system $M = (\Sigma, \Gamma, I)$, where:

Σ is a finite nonempty *input alphabet*,

Γ is a finite nonempty *working alphabet* containing Σ and

I is a finite set of *meta-instructions* of the following two types:

(a) *rewriting* meta-instruction is of the form $(E_l, x \to y, E_r)$, where $x, y \in \Gamma^*$ such that $|x| > |y|$, and $E_l, E_r \subseteq \Gamma^*$ are regular languages called *left* and *right constraints*.

(b) *accepting* meta-instruction is of the form (E, Accept), where $E \subseteq \Gamma^*$ is a regular language.

A restarting automaton $M = (\Sigma, \Gamma, I)$ induces a length-reducing and locally reducing syntactic reduction system $R(M) = (\Sigma, \Gamma, \vdash_M, S(M))$ as follows:

1. for each $u, v \in \Gamma^*$, $u \vdash_M v$ if and only if there exists an instruction $i = (E_l, x \to y, E_r)$ in I and words $u_1, u_2 \in \Gamma^*$ such that $u = u_1 x u_2$, $v = u_1 y u_2$, $u_1 \in E_l$ and $u_2 \in E_r$, and
2. $S(M) = \bigcup_{(E,\text{Accept}) \in I} E$.

Accordingly, the restarting automaton $M = (\Sigma, \Gamma, I)$ defines an input language $L(M)$ and a characteristic language $L_C(M)$:

$$L(M) = \{w \in \Sigma^* | \exists z \in S(M) : w \vdash_M^* z\} \text{ and}$$

$$L_C(M) = \{w \in \Gamma^* | \exists z \in S(M) : w \vdash_M^* z\}.$$

Thus, an input word (a sentential form) w is accepted by M if and only if w can be reduced to some simple sentential form $z \in S(M)$.

The problem of learning analysis by reduction (a restarting automaton) consists in learning the reduction relation $\vdash_M$ and the set of simple sentential forms $S(M)$. There are many possible approaches.

For learning different meta-instructions we can use different models and algorithms for learning regular languages. We can use several learning protocols like learning from positive and negative examples, learning using membership and equivalence queries, etc.

The learning can be done in an incremental way. First we can learn some basic meta-instructions which define only a subset of the target language. Then we can continue to learn new meta-instructions to improve our approximation of the target language.

Chapter 2

Learning regular languages

The main sources for this chapter are Mráz, Otto, Plátek [4], Parekh, Honavar [6] and Rivest, Baggett [7].

Finding a DFA consistent with a given sample is called *learning of DFA*. Efficient learning of DFA is a challenging research problem in grammatical inference. Gold showed that the problem of identifying the minimum state DFA consistent with a presentation S comprising of a finite non-empty set of positive examples S^+ and possibly a finite nonempty set of negative examples S^- is NP-hard.

Angluin showed that given a live-complete set of examples (that contains a representative string for each live state of the target DFA) and a knowledgeable teacher to answer membership queries it is possible to exactly learn the target DFA. In a later paper, Angluin relaxed the requirement of a live-complete set and has designed a polynomial time inference algorithm using both membership and equivalence queries.

In this chapter we introduce three learning protocols:

1. Dana Angluin's L* Algorithm,

2. The RPNI (Regular Positive and Negative Inference) Algorithm,

3. The class of SLT (Strictly Locally Testable) Languages.

2.1 Dana Angluin's L* Algorithm

Dana Angluin's L algorithm* is an algorithm for learning finite state automata using *membership* and *equivalence* queries.

1. *Membership query* is that a teacher has to decide whether to accept or reject a given word. Membership queries alone do not give efficient learning algorithms for DFAs.

2. *Equivalence query* is that a teacher gets a *conjecture* (deterministic finite automaton – DFA) and he has to decide whether this DFA is a desired DFA or not. If it is not then he also has to provide a *counterexample*.

The L* algorithm runs in time polynomial in the size of the minimal DFA equivalent to the target DFA and in the size of the longest counterexample the teacher provides.

The best way to understand how this algorithm works is by an example. Suppose that we want to find a DFA that recognizes a language $L = \{w \in \{a, b\}^* : |w|_a = 2k \text{ and } |w|_b = 2l \text{ for some integers } k, l \geq 0\}$. This language contains exactly the words that have an even number of as and an even number of bs.

During the algorithm we maintain:

1. A set of words $S \subseteq \Sigma^*$ for prefixes,

2. A set of words $E \subseteq \Sigma^*$ for postfixes,

3. A function T on strings that returns 1 if the string is accepted by the target automaton and 0 otherwise. This function represents our knowledge base and we extend this function using the membership queries.

Let $\Sigma = \{a, b\}$ be our alphabet. At the beginning we set $S := \{\lambda\}$ and $E := \{\lambda\}$ and extend our knowledge base T to $(S \cup S.\Sigma).E = \{\lambda, a, b\}$.

We ask membership queries for λ, a, b and apparently we get: $T(\lambda) = 1$, $T(a) = 0$, $T(b) = 0$.

This gives us the following table T_1:

T_1	λ
λ	1
a	0
b	0

The table has basically two parts. In the first (top) part we have rows corresponding to the strings in the set S and in the second (bottom) part we have rows corresponding to the strings in the set $S.\Sigma$. In the second part we will note only the rows that are not already present in the first part.

Each column of the table corresponds to exactly one postfix in E. Together we have $|S \cup S.\Sigma|$ rows and $|E|$ columns in this table.

For each row $s \in S \cup S.\Sigma$ and each column $e \in E$ the corresponding cell in the table has a value $T(s.e)$.

Some terminology will be useful in subsequent explanation. If s is a string from the alphabet Σ then row(s) denotes a vector $(T(s.e_1), \ldots, T(s.e_m))$ where $E = \{e_1, \ldots, e_m\}$. Apparently, the corresponding table is filled, i.e. all the membership queries are answered, if and only if this function is well defined for all the strings from the set $S \cup S.\Sigma$.

For instance in our table T_1 we have row$(\lambda) = (1)^T$, row$(a) = (0)^T$ etc.

We say that a table is *closed* if and only if every row in the bottom part has a corresponding row in the top part, i.e. $\forall s \in S.\Sigma \; \exists t \in S : \text{row}(s) = \text{row}(t)$.

Our table T_1 is not closed because for instance for a string a from the bottom part of the table we do not have a corresponding string in the top part of the table.

Similarly, we say a table is *consistent* if every pair of rows in the top part of the table with identical experiment results (columns) also has identical experiment results when any alphabet symbol is added, i.e. $\forall s_1, s_2 \in S : \text{row}(s_1) = \text{row}(s_2) \Rightarrow \forall a \in \Sigma : \text{row}(s_1.a) = \text{row}(s_2.a)$.

Our table T_1 is apparently consistent because it has only one row in the top part.

Because our table T_1 is not closed, we move the row a from the bottom part to the top part of the table. Now we get new $S = \{\lambda, a\}$ and extend our knowledge base T to $(S \cup S.\Sigma).E = \{\lambda, a, b, aa, ab\}$, i.e. we ask membership

queries for aa, ab and we get $T(aa) = 1$, $T(ab) = 0$.

The corresponding table is now:

T_2	λ
λ	1
a	0
b	0
aa	1
ab	0

This table is closed and consistent, so we can construct the *conjecture* (DFA) $A = (Q, \Sigma, \delta, q_0, F)$, where:

$Q = \{\text{row}(s), s \in S\} = \{0, 1\}$,

$\Sigma = \{a, b\}$,

$\delta(\text{row}(s), a) = \text{row}(s.a)$ for all $s \in S$ and $a \in \Sigma$,

$q_0 = \text{row}(\lambda) = 1$,

$F = \{\text{row}(s) | T(s) = 1\} = \{1\}$.

This gives us the following automaton:

```
        a b
<->  1  0 0
     0  1 0
```

The teacher responds with the counterexample bb. We incorporate the counterexample into the table by adding bb and all its prefixes to the top of the table. Since λ is already in the S, this means we add b and bb. We have $S = \{\lambda, a, b, bb\}$. Again we extend our knowledge base T to $(S \cup S.\Sigma).E$, i.e. we make membership queries on ba, bba and bbb to get the following table:

T_3	λ
λ	1
a	0
b	0
bb	1
aa	1
ab	0
ba	0
bba	0
bbb	0

This table is closed but not consistent (consider rows a and b, and successors aa and ba), so we add an a column (i.e., we add a to E) and make membership queries on aaa, aba, baa, $bbaa$, and $bbba$ to get the following table:

T_4	λ	a
λ	1	0
a	0	1
b	0	0
bb	1	0
aa	1	0
ab	0	0
ba	0	0
bba	0	1
bbb	0	0

Once again we have a closed and consistent table, and conjecture the machine described as follows:

```
          a  b
<->  10   01 00
     01   10 00
     00   00 10
```

The teacher responds with the counterexample abb. We again add the counterexample and all its prefixes to S, the top of the table – in this case the prefixes ab and abb are the only ones not already in the table. To fill out the augmented table, we make membership queries on $abba$, $abaa$, $abbaa$, $abbb$, and $abbba$, to get table T_5:

T_5	λ	a
λ	1	0
a	0	1
b	0	0
bb	1	0
ab	0	0
abb	0	1
aa	1	0
ba	0	0
bba	0	1
bbb	0	0
aba	0	0
$abba$	1	0
$abbb$	0	0

T_5 is closed but not consistent (consider row b and row ab, and their successors bb and abb), so we add a b column (i.e., add b to E) and query the strings aab, bab, $bbab$, $bbbb$, $abab$, $abbab$, $abbbb$ to get table T_6:

T_6	λ	a	b
λ	1	0	0
a	0	1	0
b	0	0	1
bb	1	0	0
ab	0	0	0
abb	0	1	0
aa	1	0	0
ba	0	0	0
bba	0	1	0
bbb	0	0	1
aba	0	0	1
$abba$	1	0	0
$abbb$	0	0	0

Since this table is closed and consistent, we conjecture the corresponding machine, the teacher confirms this conjecture, so we terminate and output the machine.

The target automaton is described as follows:

```
             a   b
<->   100   010 001
      010   100 000
      001   000 100
      000   001 010
```

2.2 RPNI Algorithm

The *regular positive and negative inference algorithm (RPNI)* is a polynomial time algorithm for identification of a DFA consistent with a given set $S = S^+ \cup S^-$.

A labeled sample $S = S^+ \cup S^-$ is provided as input to the algorithm. It constructs a prefix tree automaton $M = \mathrm{PTA}(S^+)$ and numbers its states in the standard order. Let $\bar{n}$ be the number of states of M, $Q = \{0, \ldots, \bar{n}-1\}$ be the set of states of M and $\pi_0 = \{\{0\}, \ldots, \{\bar{n}-1\}\}$ be the initial partition of states. Then $M_{\pi_0} = \mathrm{PTA}(S^+)$ is consistent with all positive and negative training examples contained in S and is treated as the initial hypothesis.

The current hypothesis is M_π and the corresponding partition is denoted by π. The algorithm is outlined in the following figure.

Algorithm RPNI

Input: A sample $S = S^+ \cup S^-$
Output: A DFA compatible with S

begin
  { *Initialization* }
  $\pi = \{\{0\}, \ldots, \{\bar{n}-1\}\}$
  $M_\pi = \mathrm{PTA}(S^+)$
  { *State merging* }
  for $i = 1$ **to** $\bar{n} - 1$
    for $j = 0$ **to** $i - 1$
      { *Merge block containing state i with block containing state j* }
      $\rho = \pi - \{[i]_\pi, [j]_\pi\} \cup \{[i]_\pi \cup [j]_\pi\}$
      { *Obtain the quotient automaton M_ρ* }
      $M_\rho = \mathrm{derive}(M, \rho)$
      { *Determinize the quotient automaton by state merging* }

```
            σ = deterministic_merge(M_ρ)
            { Does M_σ reject all strings in S⁻? }
            if consistent(M_σ, S⁻) then
                M_π = M_σ; π = σ; break
            end if
        end for
    end for
    return M_π
end
```

The function *derive* obtains the quotient automaton M_ρ, corresponding to the partition ρ. M_ρ might be a NFA in which case the function *deterministic_merge* determinizes it by recursively merging the states that cause non-determinism. For example, if q_i, q_j and q_k are states of M_ρ such that for some $a \in \Sigma : \delta(q_i, a) = \{q_j, q_k\}$ then the states q_j and q_k are merged together. The function *consistent* returns true if M_σ is consistent with all examples in S^- and false otherwise.

We demonstrate the execution of the RPNI algorithm on the task of learning the DFA in Figure 1.2. For convenience we present the target DFA in Figure 2.1 without the dead state d_0.

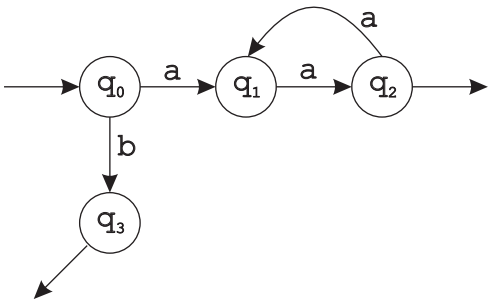


Figure 2.1: Target finite state automaton.

A sample $S = S^+ \cup S^-$ where $S^+ = \{b, aa, aaaa\}$ and $S^- = \{\lambda, a, aaa, baa\}$ is a *characteristic* sample for the target DFA. The exact definition of characteristic sample of a regular language is not necessary. We only note that in this case the RPNI algorithm is guaranteed to return a canonical representation of the target DFA.

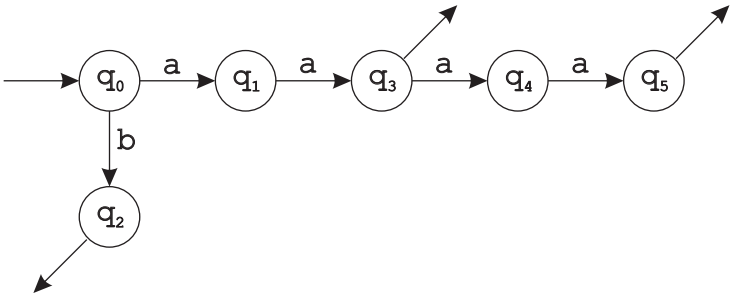


Figure 2.2: PTA.

The DFA $M = \text{PTA}(S^+)$ is depicted in Figure 2.2 where the states are numbered in the standard order. The initial partition is $\pi = \pi_0 = \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$. For simplicity state i always refers to q_i.

The algorithm attempts to merge the blocks containing states 1 and 0 of the partition π. The quotient FSA M_ρ and the DFA M_σ obtained after invoking deterministic_merge are shown in Figure 2.3.

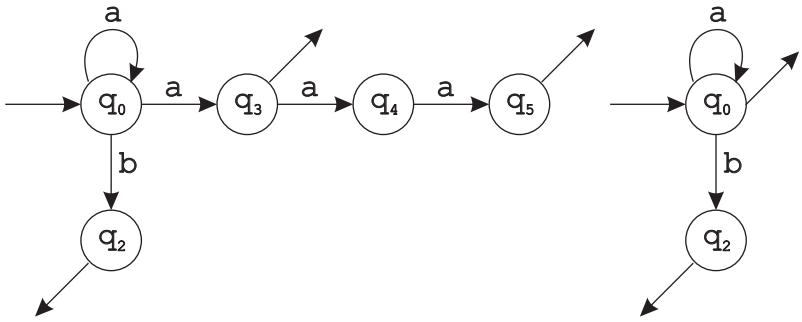


Figure 2.3: Quotient automaton and the corresponding DFA.

The DFA M_σ accepts the negative example $\lambda \in S^-$. Thus, the current partition π remains unchanged.

The following table lists different partitions ρ obtained by fusing the blocks of π_0, the partitions σ obtained by deterministic_merge of ρ, and the negative example (belonging to S^-), if any, that is accepted by the quotient DFA M_σ. The partitions marked * denote the partition π for which M_π is consistent with all examples in S^- and hence is the current

hypothesis. It is easy to see that the DFA corresponding to the partition $\pi = \{\{0\}, \{1, 4\}, \{2\}, \{3, 5\}\}$ is exactly the target DFA we are trying to learn.

Partition ρ	Partition σ	Example
$\{\{0,1\},\{2\},\{3\},\{4\},\{5\}\}$	$\{\{0,1,3,4,5\},\{2\}\}$	a
$\{\{0,2\},\{1\},\{3\},\{4\},\{5\}\}$	$\{\{0,2\},\{1\},\{3\},\{4\},\{5\}\}$	λ
$\{\{0\},\{1,2\},\{3\},\{4\},\{5\}\}$	$\{\{0\},\{1,2\},\{3\},\{4\},\{5\}\}$	a
$\{\{0,3\},\{1\},\{3\},\{4\},\{5\}\}$	$\{\{0,3\},\{1,4\},\{2\},\{5\}\}$	λ
$\{\{0\},\{1,3\},\{2\},\{4\},\{5\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a
$\{\{0\},\{1\},\{2,3\},\{4\},\{5\}\}$	$\{\{0\},\{1\},\{2,3\},\{4\},\{5\}\}$	baa
$\{\{0,4\},\{1\},\{2\},\{3\},\{5\}\}$	$\{\{0,2\},\{1,5\},\{2\},\{3\}\}$	a
$\{\{0\},\{1,4\},\{2\},\{3\},\{5\}\}$	$\{\{0\},\{1,4\},\{2\},\{3,5\}\}^*$	-
$\{\{0,3,5\},\{1,4\},\{2\}\}$	$\{\{0,3,5\},\{1,4\},\{2\}\}$	λ
$\{\{0\},\{1,3,4,5\},\{2\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a
$\{\{0\},\{1,4\},\{2,3,5\}\}$	$\{\{0\},\{1,4\},\{2,3,5\}\}$	baa
$\{\{0\},\{1,4\},\{2\},\{3,5\}\}$	$\{\{0\},\{1,4\},\{2\},\{3,5\}\}^*$	-
$\{\{0\},\{1,3,4,5\},\{2\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a

2.3 SLT Languages

Let Σ be a finite nonempty set of symbols (alphabet), w be a word from Σ^* and k be a positive integer. We define $P_k(w)$ and $S_k(w)$ to be the *prefix* and *suffix* (or *postfix*) of a word w of length k, respectively. Further, let $I_k(w)$ be the set of all substrings of w of length k except the prefix and suffix of w of length k, that is $I_k(w) = \{u \in \Sigma^* |\ |u| = k \wedge \exists x, y \in \Sigma^* : x, y \neq \lambda \wedge w = xuy\}$. These are defined only for $|w| \geq k$.

Let k be a positive integer. A language $L \subseteq \Sigma^*$ is *strictly k-testable* if there exist finite sets $A, B, C \subseteq \Sigma^k$ such that, for all $w \in L$ satisfying $|w| \geq k$ we have: $w \in L \Leftrightarrow P_k(w) \in A \wedge S_k(w) \in B \wedge I_k(w) \subseteq C$. In this case (A, B, C) is called a *triple* for L. For a triple $S = (A, B, C)$ we denote the corresponding language as L_S.

We say that L is *strictly locally testable* if it is strictly k-testable for some $k > 0$. It can be proved that strictly locally testable languages are a special subclass of the class of regular languages, so for every strictly k-testable language L there exists a DFA A, such that $L = L(A)$.

Note that the definition of strictly k-testable says nothing about the

strings of length $k-1$ or less. Hence, L is strictly k-testable if and only if $L \cap \Sigma^k\Sigma^* = (A\Sigma^* \cap \Sigma^* B) - \Sigma^+(\Sigma^k - C)\Sigma^+$.

We will denote the family of strictly k-testable languages by k-SLT and the class of strictly locally testable languages by SLT.

For a language L, a *positive presentation* of L is an infinite sequence $\{w_i\}_{i=1}^{\infty}$ of words from L such that every $w \in L$ occurs at least once in the sequence.

Let us briefly recall a learning algorithm for strictly k-testable languages.

Algorithm for learning k-SLT languages

Input: An integer $k > 0$ and a positive presentation of a target strictly k-testable language L.
Output: A sequence of triples $S_i = (A_i, B_i, C_i)$ for k-testable languages.

begin
  let $S_0 := (\varnothing, \varnothing, \varnothing)$ be the initial triple
  repeat (forever)
    let $S_i = (A_i, B_i, C_i)$ be the current triple
    read the next positive sample w_{i+1}
    if $w_{i+1} \in L_{S_i}$ **then**
      $S_{i+1} := S_i$
    else
      $A_{i+1} := A_i \cup P_k(w_{i+1})$
      $B_{i+1} := B_i \cup S_k(w_{i+1})$
      $C_{i+1} := C_i \cup I_k(w_{i+1})$
      $S_{i+1} := (A_{i+1}, B_{i+1}, C_{i+1})$
    end if
    $i := i + 1$
  end repeat
end

We assume that $|w_i| \geq k$ for all i. Otherwise we can for instance move all the small words w (such that $|w| < k$) to a special set D. This set D is always finite.

It can be proven that there exists an index i_0 such that for all $i \geq i_0$: $S_i = S_{i_0} = S$. Apparently $w_i \in L_S$ for all i.

Chapter 3

Implementation

In this chapter we will outline the development of a system for design and testing of restarting automata.

First we capture requirements that define what such system should do.

Then we make a rough analysis and consider which development tools, language and target platform are most appropriate for developing this system.

After that we make a general overview of the system, describe some core classes and basic design principles behind the system.

We enclose the chapter with description of XML serialization, representation of languages and representation of restarting automata.

This chapter is not intended to be a comprehensive reference guide of the system internals. All necessary details can be found in the source code. After reading this part you should be able to modify or extend the system with your own modules.

3.1 Requirements

These are the basic requirements of the proposed system:

1. The system is intended for interactive work with restarting automata with few meta-instructions. The main goal of the system is to allow an easy investigation of these automata.

2. The system should have a user friendly (graphical) user interface.

3. The system should support working in a noninteractive mode where other processes and applications can communicate with the system.

4. The system should support storing every element of the system (alphabet, DFA, language, meta-instruction, restarting automaton) in XML representation that can be serialized (deserialized) to (from) a file (or clipboard). This XML representation is platform-independent.

5. Incremental design of restarting automata by stepwise design of meta-instructions:

   (a) Accepting meta-instruction is defined by its accepting language.

   (b) Reducing meta-instruction is defined by its left language, right language and two words x and y where $\alpha x \beta$ can be reduced to $\alpha y \beta$ if α is in the left language and β is in the right language.

6. The system should support different ways of design of languages:

   (a) *DFA modeler*: with this tool you can enter a regular language by specifying its underlying deterministic finite automaton.

   (b) *Dana Angluin's L* algorithm*: this is a machine learning algorithm which learns deterministic finite automaton using membership and equivalence queries (Section 2.1).

   (c) *RPNI algorithm*: this is a machine learning algorithm which learns deterministic finite automaton based on a given set of labeled examples (Section 2.2).

   (d) *Regex modeler*: with this tool it is possible to enter a regular language by specifying regular expression.

   (e) *SLT modeler*: with this tool it is possible to design a regular language by specifying a positive integer k and positive examples using the algorithm for learning k-SLT languages (Section 2.3).

7. The system will support testing of developed finite automata and restarting automata. In particular, for a restarting automaton, if defined correctly, it should be possible to:

   (a) Decide whether a given word w is accepted by some of the accepting meta-instruction of the automaton.

(b) Decide whether a given word w_1 can be reduced to another word w_2 and list all possible reduction paths.

(c) For a given word w list all the words that can be reduced from this word w.

(d) Decide whether a given word w is accepted by the automaton, i.e. if there exists a word u that is accepted by some of the accepting meta-instruction of the automaton and w can be reduced to u.

Of course, these procedures can in general generate very long lists of words. This system is proposed as an interactive tool, so we do not optimalize its time complexity, but we concentrate on easy manipulation with languages and their different representations.

3.2 Architecture

Before we start developing a system we have to choose a target architecture, programming language and development tools used to build the system.

Concerning architecture we were considering UNIX-like operating systems and Win32 platform.

For languages there are many possibilities (we mention only the most popular): C, C++, C♯ and Java.

C compiler without doubt produces the fastest executables, but developing larger systems can be tedious without using some advanced technologies such as object-oriented programming.

Because we do not suppose to use the program for long inputs and complex automata we have excluded C language from our consideration.

C++ is an extension of C that allows object-oriented development and still it is one of the most used programming languages in software engineering and also in research. But there are many problems of which we mention only the most important:

1. C++ code is often strongly dependent on the platform and it is very difficult to transfer this code to another platform.

2. Often you have to choose between various external libraries (for instance for GUI in Win32 there are Win32 API, ATL, MFC, etc.)

3. There are many problematic issues connected with globalization, UNICODE characters, COM objects etc.

Because of these problems we have excluded C++ language from our consideration.

C♯ and Java are very similar languages and from the perspective of developing a system they represent the equivalent choices.

We have chosen C♯ language because it is more C++ like, it is richer in syntax, and it has an excellent development environment: Microsoft Visual Studio 2005 Team Suite.

C♯ has a special relationship to its runtime environment, the .NET Framework. The .NET Framework defines an environment that supports the development and execution of highly distributed, component-based applications. There are few popular versions of .NET Framework. We have chosen the .NET Framework 2.0 because it supports generic types, XML serialization and deserialization of classes (with certain restrictions of course) and other useful technologies like .NET Remoting. There are also some newer versions of .NET Framework (like the .NET Framework 3.5). However the .NET Framework 2.0 has the advantage that it is also fully supported on UNIX-like operating systems (for more information see the website of the project Mono: http://www.mono-project.com/).

Note: If we speak about XML serialization (deserialization) of classes in C♯ we mean automatic conversion between C♯ classes and their XML representation. Classes can be of course serialized and deserialized manually, but this is less effective approach. We will return to this topic later in the Section 3.5.

Architecture summary:

Project name: RestartingAutomaton
Platform: Microsoft Windows
Programming language: C♯ 2.0
Runtime environment: .NET Framework 2.0
Developing tools: Microsoft Visual Studio 2005 Team Suite

3.3 General overview

In our system we recognize several layers of abstraction (see Figure 3.1).

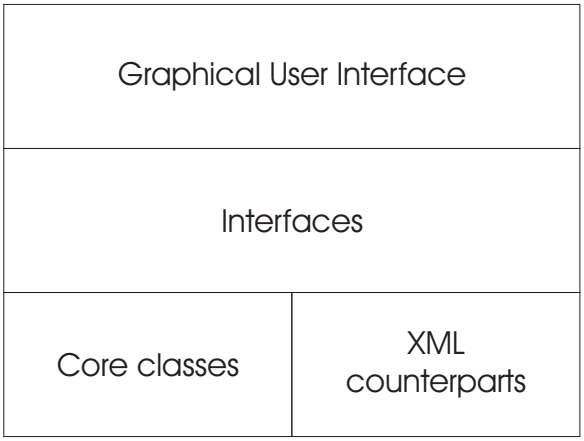


Figure 3.1: Layers of abstraction.

Every system is built up of some basic indivisible atoms. If we exclude basic data types such as integers, strings, then we are left with several core classes that are specific for our system.

Before we start enumerating these classes we have to make our first fundamental design decision. We have to choose how to represent symbols from which bigger objects such as alphabets, words and languages are created.

1. Symbols are chars. In C♯ there exists a data type called **char** that is 2 bytes long and represents a UNICODE character.

2. Symbols are integers. There are several integer data types in C♯, such as **int**, **uint**, **long**, **ulong** etc.

3. Symbols are specific **struct**s or **class**es.

4. Symbols are classes that implement a special interface.

5. Symbols are of general supertype. In C♯ there exists a supertype called **object** which is an ancestor for all classes.

6. Symbols are of unspecified generic type T.

The first choice is easy to understand, easy to program, even there exists a class **string** near to hand that can be used for representing words. But the classes and algorithms used in our system are more general and can work with many other types of symbols than **char**s.

The second choice is nearly the same as the first, it allows many more symbols, but there is no support for words from the programming language.

The third choice is a bad one because we do not know in advance how this specific struct or class for symbols should look like. It can be for instance defined at the beginning of a program, but there are better techniques to achieve this.

The fourth choice looks good but it can be ineffective when working with symbols as classes. The instances of classes are stored in a heap so accessing data members of a class is undoubtedly much more slower than working with value data types stored on stack. It is also ineffective if we have many small instances of these classes on the heap because it is demanding of memory. One solution to this problem could be using Flyweight design pattern. The idea is that we could have something like a pool of symbols common for the whole system and the words could have just references to the symbols of this pool. We have found some problematic issues concerning the serialization with this approach.

The fifth choice can be regarded as a special case of the fourth choice with an empty interface. The problem is that we have no control over the type of the symbols and the serialization can be even more complicated.

The sixth choice means using generic types. It is sufficiently general solution, type safe and it does not have problems with efficiency because we use value types for symbols. Serialization works well, but there are some problems with serialization if we use class inheritance. We will return to this topic later. Of course we put some restrictions to this type T such as it has to be a value type that is linearly ordered, i.e. every symbol can be compared with every other symbol.

It is no surprise that we have chosen the sixth possibility. It can seem strange that in our system we use all generic classes only with the type **char**. Our system is intended for an interactive work and symbols as chars is the most convenient representation for users.

On the other hand many classes and algorithms in our system are quite useful and can be therefore reused by another more complicated system that does not work with **char**s.

Now we know what kind of symbols we are going to work with. The following Code listing 3.1 shows how any class that somehow works with these symbols should look like.

Listing 3.1: A C♯ generic class template

```
1  public class GenericClass<T>
2    where T: struct, IComparable<T>, IEquatable<T>
3  {
4    ...
5  }
```

The second row represents all restrictions required for a generic type T. For T you can substitute all basic ordinal data types, such as **char**, **int** etc. or user defined **struct**s that implement listed interfaces.

3.4 Core classes

In this section we briefly describe the following core classes of our system:

1. *GenericString<T>*: a string of generic symbols.
2. *Alphabet<T>*: an alphabet (a finite nonempty set of symbols).
3. *DFA<T>*: a deterministic finite automaton (Section 1.2)
4. *LStar<T>*: Dana Angluin's L* algorithm (Section 2.1).
5. *RPNI<T>*: RPNI algorithm (Section 2.2).
6. *Regex*: regular expressions (C♯ class Regex).
7. *SLT<T>*: class for k-SLT languages (Section 2.3).

3.4.1 GenericString<T>

In our system we often work with words. A word is a string of symbols. If the symbols were **char**s we could use **string** data type for representing words. The problem is that symbols are of unspecified generic type T.

We have to create a special class called GenericString<T> that behaves like an ordinary **string** class but is not bound with **char**s. This class is quite fundamental because it is used by many classes in our system.

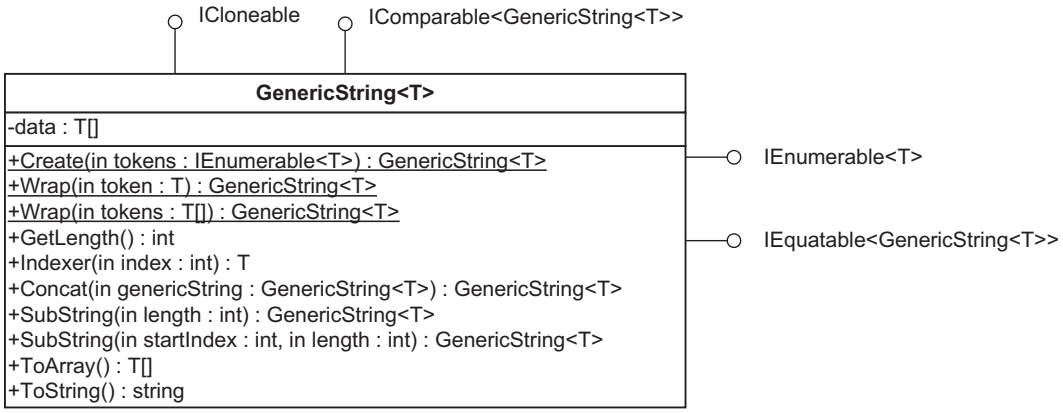


Figure 3.2: GenericString<T> Class Diagram.

There are many ways how to design such class. We have decided to make this class immutable. It means that this class does not have any methods (or properties) that can modify the internal state of the class. Note that ordinary **string** is also immutable.

The question is what is this decision good for? The problem is that in C♯ you do not have **const** modifiers and you cannot have **const** references.

This can be problem in the case when you for instance pass an instance of GenericString<T> class as an argument to a method and you want to prevent modification of this instance inside the method. If the GenericString<T> class is immutable, it cannot be modified inside any method.

Another approach to this problem is to use Proxy design pattern. For explanation imagine you have a collection of **int**s and you want to prevent this collection to be modified outside. The most common solution to this is to encapsulate this collection to a class that behaves just like the collection of **int**s on the outside, but cannot be modified.

Listing 3.2: Readonly collection

```
1    int[] a = {1, 2, 3, 4};
2    ICollection<int> readonly_a =
3      new ReadOnlyCollection<int>(a);
```

In the Code listing 3.2 ReadOnlyCollection<**int**> represents a proxy that encapsulates the collection of **int**s.

If we transfer this approach to our GenericString<T> class it would mean that every time you want to pass the instance of this class as an argument you should encapsulate this instance to a special read-only proxy class. We have decided to avoid this approach.

There is also a third approach. You can pass always the clone of the original string. This works fine but it is a memory-consuming solution.

Although we have chosen an immutable version of GenericString<T> class, there are some problems connected with immutable classes. First, they do not have default constructors and therefore cannot be serialized or deserialized (automatically). Second, every time you ask for instance for a substring or you try to concatenate two generic strings it will create a new instance of GenericString<T> class.

3.4.2 Alphabet<T>

Class Alphabet<T> is an immutable class that represents an alphabet (a finite nonempty set of symbols of generic type T) in our system. It is very similar to GenericString<T> class, but there are some important differences. The most important difference is that it plays a different role in our system. Then the symbols of the alphabet are ordered and no duplicities are allowed. Therefore you can check whether a given symbol is in the alphabet or not in time logarithmic to the number of symbols in the alphabet.

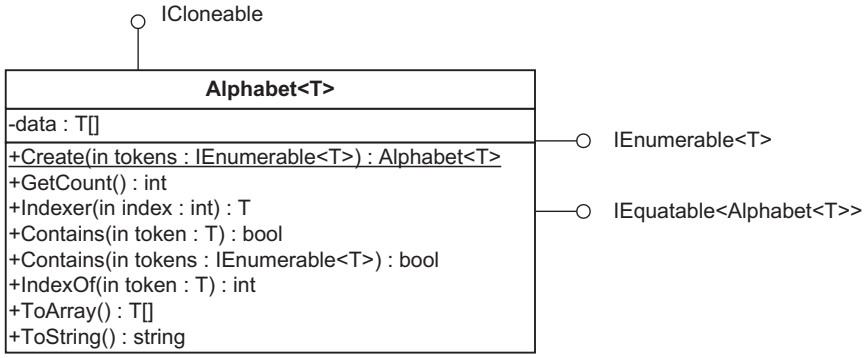


Figure 3.3: Alphabet<T> Class Diagram.

3.4.3 DFA<T>

Class DFA<T> represents a deterministic finite automaton in our system. The instance of DFA<T> class is strongly bound to its alphabet and the number of states which are passed as the arguments to the constructor. The alphabet and the number of states of the DFA cannot be changed after creation of this instance. The states of the DFA are always numbers $0, 1, 2, \ldots$ where 0 always represents the input state.

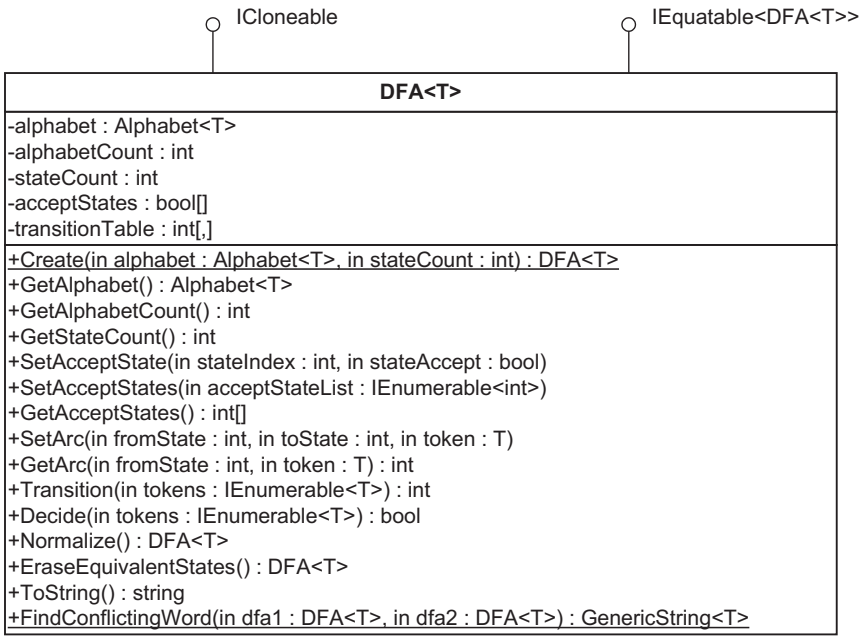


Figure 3.4: DFA<T> Class Diagram.

How this class works is best illustrated on an example. Suppose that we want to represent the automaton shown in the Figure 3.5.

In the Code listing 3.3 it is shown how to create such an automaton.

Listing 3.3: How to create deterministic finite automaton.

```
1  Alphabet<char> alphabet = new Alphabet<char>("ab");
2  DFA<char> dfa = new DFA<char>(alphabet, 6);
3  dfa.SetAcceptStates(new int[] { 1, 5, 2 });
4  dfa.SetArc(0, 1, 'a'); dfa.SetArc(0, 0, 'b');
5  dfa.SetArc(1, 3, 'a'); dfa.SetArc(1, 1, 'b');
```

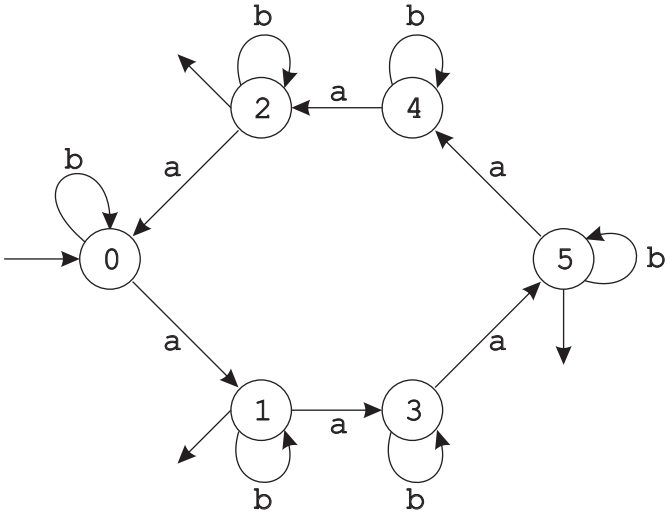


Figure 3.5: Deterministic finite automaton.

```
6    dfa.SetArc(3, 5, 'a'); dfa.SetArc(3, 3, 'b');
7    dfa.SetArc(5, 4, 'a'); dfa.SetArc(5, 5, 'b');
8    dfa.SetArc(4, 2, 'a'); dfa.SetArc(4, 4, 'b');
9    dfa.SetArc(2, 0, 'a'); dfa.SetArc(2, 2, 'b');
```

If we call Console.WriteLine(dfa.ToString()); we get the following output.

```
          a   b
 ->   0   1   0
<-    1   3   1
<-    2   0   2
      3   5   3
      4   2   4
<-    5   4   5
```

As you can see in the Code listing 3.3 in the first row we define an alphabet of the target automaton. In the second row we create the automaton with the alphabet and 6 states. In the third row we set which states are accepting states. The last step (rows 4 – 9) is the definition of the transition table.

There are few useful methods in DFA<T> class. First method is Normalize() which returns the same automaton with states relabeled in an unambiguous

way. It is useful when you want to find out whether two reduced automata are equivalent. You just normalize these automata and then compare whether they are the same.

Another useful method is EraseEquivalentStates() which returns reduced automaton. It uses the algorithm sketched in Section 1.2.

If we normalize our automaton we get the following automaton:

```
              a    b
 ->     0     1    0
<-      1     2    1
        2     3    2
<-      3     4    3
        4     5    4
<-      5     0    5
```

The corresponding reduced automaton is in the following listing:

```
              a    b
 ->     0     1    0
<-      1     0    1
```

Static method FindConflictingWord(dfa1, dfa2) returns the smallest word (with respect to the standard enumeration of strings) that is accepted in one automaton and rejected in the other automaton. It returns null if these automata are equivalent.

3.4.4 LStar<T>

Before we start with LStar<T> class designated for Dana Angluin's L* algorithm we mention an auxiliary class KnowledgeBase<T> that stores labeled samples. Every labeled sample is stored in a structure called WordAccept<T> that encapsulates a word and its acceptance.

We have separated this class from LStar<T> class, because it is quite useful and can be reused by another algorithms that work with labeled samples.

There are two important things to note on KnowledgeBase<T> class. First it remembers the order of the samples. This is important for instance

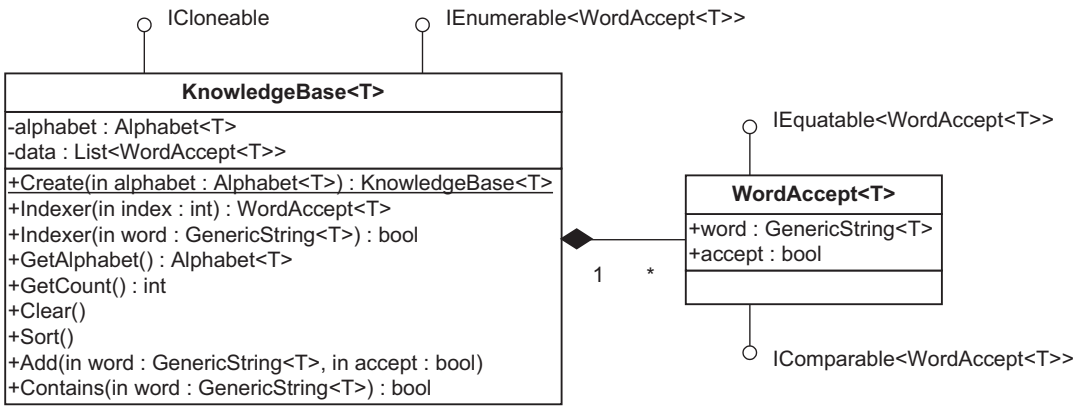


Figure 3.6: KnowledgeBase<T> Class Diagram.

for L* algorithm which is sensitive to the order of the words. Another important feature is that you can register some events with this class that inform you about changes in the class. For instance if someone changes the acceptance of some word in the knowledge base, the L* algorithm must be launched anew on the new samples.

As we have already mentioned, LStar<T> class encapsulates Dana Angluin's L* algorithm. Its main role is to give you a conjecture (DFA) that is consistent with samples stored in its knowledge base. If it is not possible to construct a conjecture it will give you a list of words that are to be added to the knowledge base. This class is quite interactive. You just add words to the knowledge base until you can ask for a conjecture.

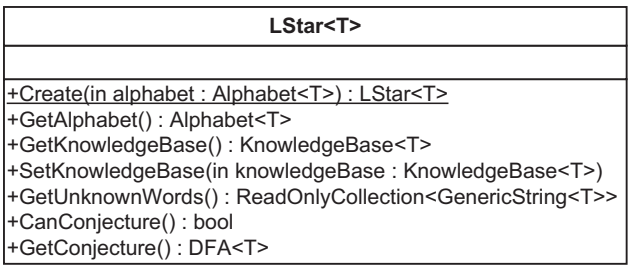


Figure 3.7: LStar<T> Class Diagram.

For explanation how to work with this class we emulate the example

illustrated in the Section 2.1. First we have to specify the alphabet and create an instance of LStar<**char**> class.

Listing 3.4: Dana Angluin's L* algorithm.

```
1  Alphabet<char> alphabet = new Alphabet<char>("ab");
2  LStar<char> lstar = new LStar<char>(alphabet);
```

Now if we look at the collection lstar.UnknownWords we get the following words: λ, a, b. We have to add these words to the knowledge base.

```
3  lstar.KnowledgeBase.Add(new GenericString<char>(""), true);
4  lstar.KnowledgeBase.Add(new GenericString<char>("a"), false);
5  lstar.KnowledgeBase.Add(new GenericString<char>("b"), false);
```

After adding these words the algorithm finds out that it needs also words aa and ab.

```
6  lstar.KnowledgeBase.Add(new GenericString<char>("aa"), true);
7  lstar.KnowledgeBase.Add(new GenericString<char>("ab"), false);
```

Now lstar.CanConjecture == **true** so if we call
Console.WriteLine(lstar.Conjecture.ToString());
we get the following automaton:

```
              a    b
<->     0     1    1
        1     0    1
```

This is not the target automaton, because it rejects bb. We add this word as a counterexample to the knowledge base.

```
8  lstar.KnowledgeBase.Add(new GenericString<char>("bb"), true);
```

The algorithm responds with unknown words: ba, bba, bbb. If we proceed this way we finally get the target automaton as shown in the following output:

```
             a    b
<->     0    1    2
        1    0    3
        2    3    0
        3    2    1
```

This is the list of all words that you need to add to the knowledge base if you want to get our target automaton:

```
accept ''       reject abb
reject a        reject abaa
reject b        accept abba
accept aa       reject abbb
reject ab       reject abbaa
accept bb       reject abbba
reject ba       reject aab
reject bba      reject bab
reject bbb      accept abab
reject aaa      reject bbab
reject aba      accept bbbb
reject baa      reject abbab
accept bbaa     reject abbbb
reject bbba
```

Note that these words are not set in the standard order. What is even more surprising is that if we sorted these words then the algorithm would not be able to give you the conjecture. Instead it would ask you for some additional membership queries. It is now clear that the order of samples is important for Dana Angluin's L* algorithm.

3.4.5 RPNI<T>

The RPNI (regular positive and negative inference algorithm) is a polynomial time algorithm for identification of a DFA consistent with given positive and negative samples. Class PositiveNegativeSamples<T> encapsulates positive and negative samples as two lists of GenericString<T>s. You can register some events with this class that inform you about changes in the class.

The RPNI<T> class encapsulates RPNI algorithm. This class is quite simple. You set the samples and then you can ask for a conjecture consistent with these samples.

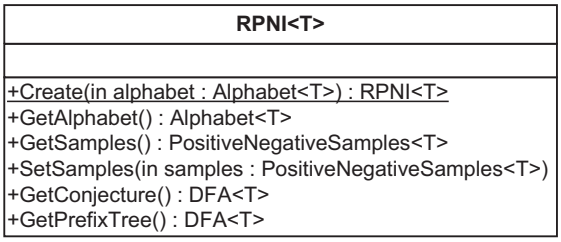


Figure 3.8: RPNI<T> Class Diagram.

Suppose that we want to find a DFA consistent with $S = S^+ \cup S^-$ where $S^+ = \{b, aa, aaaa\}$ and $S^- = \{\lambda, a, aaa, baa\}$ as in the Section 2.2.

First we create an instance of RPNI<T> class.

Listing 3.5: RPNI algorithm.

```
1 Alphabet<char> alphabet = new Alphabet<char>("ab");
2 RPNI<char> rpni = new RPNI<char>(alphabet);
```

Then we add positive and negative samples.

```
3 rpni.Samples.AddPositiveSample(new GenericString<char>("b"));
4 rpni.Samples.AddPositiveSample(new GenericString<char>("aa"));
5 rpni.Samples.AddPositiveSample(new GenericString<char>("aaaa"));
6 rpni.Samples.AddNegativeSample(new GenericString<char>(""));
7 rpni.Samples.AddNegativeSample(new GenericString<char>("a"));
8 rpni.Samples.AddNegativeSample(new GenericString<char>("aaa"));
9 rpni.Samples.AddNegativeSample(new GenericString<char>("baa"));
```

Now if we call Console.WriteLine(rpni.Conjecture.ToString()); we get the following output:

```
          a   b
 ->   0   1   2
      1   3   4
<-    2   4   4
<-    3   1   4
      4   4   4
```

It is easy to see that this automaton is isomorphic to a DFA shown in the Figure 1.2.

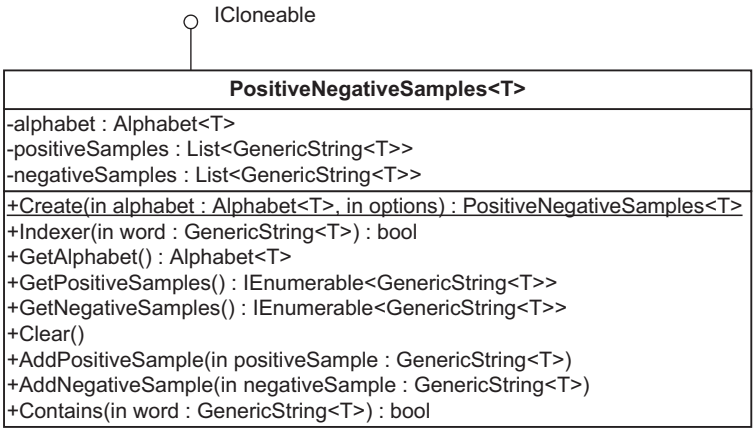


Figure 3.9: PositiveNegativeSamples<T> Class Diagram.

3.4.6 Regex

Regular expressions represent a powerful way how to enter a regular language. In our system we use C♯ class Regex from System.Text.RegularExpressions namespace. We do not implement conversions between regular expressions and deterministic finite automata. Also note that regular expressions can be used only with **char**s.

Few useful examples of regular expressions:

Regex	Description
`a`	matches words containing character a
`^ab`	matches words starting with ab
`ab$`	matches words ending with ab
`^.$`	matches a single character
`\w\w`	matches words with at least two word characters
`^\d\d\d$`	matches three digit numbers
`^[ac]..$`	matches words with three characters, starting with a or c
`^[a-c]..$`	matches words with three characters, starting with a, b or c
`^a*$`	matches words that contain only character a
`^b+$`	matches words that contain only character b (at least one)
`^c{2,5}$`	matches words: cc, ccc, cccc, ccccc
`^(ab){2,3}$`	matches words: abab and ababab

3.4.7 SLT<T>

SLT<T> class is a class designated for strictly locally testable languages. In the constructor of this class you specify an alphabet and a positive integer k for k-SLT language. This integer k can be changed at any time after creation of the instance. You can add positive samples to the instance, but negative samples are forbidden.

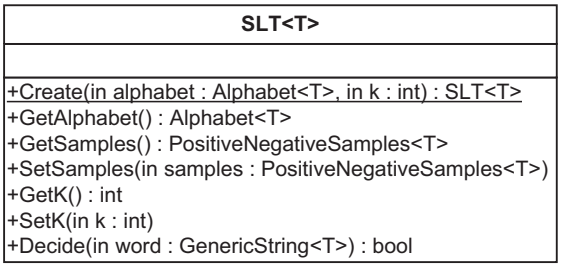


Figure 3.10: SLT<T> Class Diagram.

Although it is possible to construct a DFA equivalent with the instance of this class, we have not implemented this functionality. Instead we provide a method Decide which you can use for deciding whether a given word is accepted or rejected.

We illustrate SLT<T> class on a simple example. First we create an instance of this class with $k = 2$.

Listing 3.6: SLT Languages.

```
1 Alphabet<char> alphabet = new Alphabet<char>("abc");
2 SLT<char> slt = new SLT<char>(alphabet, 2);
```

Then we add samples *abbba* and *abcba*.

```
3 slt.Samples.AddPositiveSample(new GenericString<char>("abbba"));
4 slt.Samples.AddPositiveSample(new GenericString<char>("abcba"));
```

Now slt.Decide(**new** GenericString<**char**>("abbcbba")) returns **true**. But if we change k to slt.K = 3; it will return **false**. Do you see why?

3.5 Introduction to XML Serialization in C♯

The main source for this section is [14]. For more detailed description see http://agiledeveloper.com/articles/XMLSerialization.pdf

There are many classes in .NET framework designated for processing XML documents. Class XmlReader provides a fast, read-only, serial access to an XML document, class XmlWriter can be used for writing XML data.

There are also classes like XmlDocument and XmlNode that are used to represent DOM (Document Object Model) of XML data and allow construction of object structure in main memory.

There exists also a class called SoapFormatter that allows you to serialize (or deserialize) almost any object (or hierarchy of objects) to XML format. But nowadays using of this class is deprecated. It is recommended to use BinaryFormatter instead. Unfortunately BinaryFormatter produces binary data which are unreadable for a human.

While these classes are significant, our focus is on XML Serialization, and we will not discuss these classes further.

The process of transforming the contents of an object into XML format is called serialization, and the reverse process of transforming an XML document into a .NET object is called deserialization.

3.5.1 An example

To create a class that can be serialized by using XML Serialization, you must perform the following tasks:

1. Specify the class as public.

2. Specify all members that must be serialized as public.

3. Create a parameterless constructor.

If there are private or protected members, they will be skipped during the serialization. Public properties are also serialized, but you have to specify both getter and setter methods.

Suppose that we want to serialize the following class. Note that this class is declared public and contains only public members.

Listing 3.7: XML Example Class.

```
1  public class Example
2  {
3      public int intValue1;
4      public int intValue2;
5      public string str;
6      public int[] intArray;
7      public List<string> stringList;
8      public DateTime date;
9  }
```

The first step is to add the following namespaces:

```
1  using System.IO;
2  using System.Xml.Serialization;
```

The first namespace is for IO operations and the second one is for XML Serialization.

Then we create an instance of this class and load it with some data.

```
1  Example example = new Example();
2  example.intValue1 = 5;
3  example.intValue2 = 10;
4  example.str = "Hello World";
5  example.intArray = new int[] { 2, 3, 5, 7, 11 };
6  example.stringList = new List<string>();
7  example.stringList.Add("one");
8  example.stringList.Add("two");
9  example.stringList.Add("three");
10 example.date = DateTime.Now;
```

To serialize this example to an XML file 'Example.xml' we first create an instance of XmlSerializer class and then we use this instance to serialize our example.

Listing 3.8: XML Serialization.

```
1  XmlSerializer serializer =
2    new XmlSerializer(typeof(Example));
```

```
3  FileStream fileSaveSream =
4    new FileStream("Example.xml", FileMode.Create);
5  serializer.Serialize(fileSaveSream, example);
6  fileSaveSream.Close();
```

The resulting file 'Example.xml' looks like this:

```
<?xml version="1.0"?>
<Example xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <intValue1>5</intValue1>
  <intValue2>10</intValue2>
  <str>Hello World</str>
  <intArray>
    <int>2</int>
    <int>3</int>
    <int>5</int>
    <int>7</int>
    <int>11</int>
  </intArray>
  <stringList>
    <string>one</string>
    <string>two</string>
    <string>three</string>
  </stringList>
  <dateTime>2008-05-04T21:34:12.375+02:00</dateTime>
</Example>
```

Our instance serializer of XmlSerializer class can be also used to deserialize object from an XML file.

Listing 3.9: XML Deserialization.

```
1  FileStream fileLoadStream =
2    new FileStream("Example.xml", FileMode.Open);
3  example = serializer.Deserialize(fileLoadStream) as Example;
4  fileLoadStream.Close();
```

3.5.2 Controlling XML Serialization

You can control XML Serialization using attributes. By default, an XML element name is determined by the class or member name. This default behavior can be changed if you want to give the element a new name.

Let us add some attributes to our example class (see Code listing 3.10).

Listing 3.10: XML Example Class with Attributes.

```
1  [XmlRoot("AttrExample")]
2  public class Example
3  {
4      [XmlElement(ElementName = "ImportantInteger")]
5      public int intValue1;
6      [XmlIgnore()]
7      public int intValue2;
8      [XmlAttribute("StringAttribute")]
9      public string str;
10     [XmlArray("ArrayOfInts"),
11      XmlArrayItem("intItem")]
12     public int[] intArray;
13     [XmlArray("ListOfStrings"),
14      XmlArrayItem("stringItem")]
15     public List<string> stringList;
16     [XmlElement("ActualDate")]
17     public DateTime date;
18 }
```

After serialization we get the following XML output:

```
<?xml version="1.0"?>
<AttrExample xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             StringAttribute="Hello World">
  <ImportantInteger>5</ImportantInteger>
  <ArrayOfInts>
    <intItem>2</intItem>
    <intItem>3</intItem>
    <intItem>5</intItem>
    <intItem>7</intItem>
```

```
    <intItem>11</intItem>
  </ArrayOfInts>
  <ListOfStrings>
    <stringItem>one</stringItem>
    <stringItem>two</stringItem>
    <stringItem>three</stringItem>
  </ListOfStrings>
  <ActualDate>2008-05-05T14:03:28.078125+02:00</ActualDate>
</AttrExample>
```

3.5.3 Serializing compositions

Suppose that we have two classes where Master class is composed of Slave classes as in the Code listing 3.11.

Listing 3.11: XML Composition Example.

```
1  public class Slave
2  {
3      public int slaveID;
4      public string slaveName;
5  }
6  public class Master
7  {
8      public string masterName;
9      public Slave mainSlave;
10     public Slave[] slaveCollection;
11 }
```

Let us create some instances.

```
1  Master master = new Master();
2  master.masterName = "Master Object";
3  Slave chiefSlave = new Slave();
4  chiefSlave.slaveID = 1;
5  chiefSlave.slaveName = "Chief Slave";
6  master.mainSlave = chiefSlave;
7  Slave slave1 = new Slave();
8  slave1.slaveID = 5;
9  slave1.slaveName = "Obedient Slave";
```

```
10  Slave slave2 = new Slave();
11  slave2.slaveID = 10;
12  slave2.slaveName = null;
13  master.slaveCollection = new Slave[] { slave1, slave2 };
```

If we serialize the master instance, we get the following XML output:

```
<?xml version="1.0"?>
<Master xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <masterName>Master Object</masterName>
  <mainSlave>
    <slaveID>1</slaveID>
    <slaveName>Chief Slave</slaveName>
  </mainSlave>
  <slaveCollection>
    <Slave>
      <slaveID>5</slaveID>
      <slaveName>Obedient Slave</slaveName>
    </Slave>
    <Slave>
      <slaveID>10</slaveID>
    </Slave>
  </slaveCollection>
</Master>
```

Note that null values are skipped during the serialization.

3.5.4 Serializing derived classes

Now suppose that we want to use a DerivedSlave class that is derived from a Slave class as is shown in the Code listing 3.12.

Listing 3.12: XML Derived Class Example.

```
1  public class DerivedSlave : Slave
2  {
3      public int specialParameter;
4  }
```

Let us modify for instance the slave2 instance to be of DerivedSlave type.

```
14 DerivedSlave slave2 = new DerivedSlave();
15 slave2.slaveID = 10;
16 slave2.slaveName = "Derived Slave";
17 slave2.specialParameter = 666;
```

If we try to serialize our master instance we get InvalidOperationException exception. This is because the serialization process does not deal with inheritance hierarchy in a smooth way. For this to work, we will have to indicate that the slave reference may refer to an object of Slave or DerivedSlave class as follows in the Code listing 3.13.

Listing 3.13: XML Derived Class Correction.

```
1  public class Master
2  {
3      public string masterName;
4      [XmlElement(Type = typeof(Slave)),
5       XmlElement(Type = typeof(DerivedSlave))]
6      public Slave mainSlave;
7      [XmlArrayItem(Type = typeof(Slave)),
8       XmlArrayItem(Type = typeof(DerivedSlave))]
9      public Slave[] slaveCollection;
10 }
```

After serialization we get the following XML output:

```
<?xml version="1.0"?>
<Master xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <masterName>Master Object</masterName>
  <Slave>
    <slaveID>1</slaveID>
    <slaveName>Chief Slave</slaveName>
  </Slave>
  <slaveCollection>
    <Slave>
      <slaveID>5</slaveID>
```

```
      <slaveName>Obedient Slave</slaveName>
    </Slave>
    <DerivedSlave>
      <slaveID>10</slaveID>
      <slaveName>Derived Slave</slaveName>
      <specialParameter>666</specialParameter>
    </DerivedSlave>
  </slaveCollection>
</Master>
```

While the above fix works, we have completely violated the Open-Closed Principle. The code is not extensible to adding new types of Slaves. If we decide to add another class which inherits from Slave or DerivedSlave, we will have to modify the Master class. This seems to be the only significant limitation of the XML serialization mechanism.

3.5.5 XML Serialization summary

Now we know that if we want to create a class intended for XML Serialization it must be declared public and it must have a parameterless constructor. We also know that only public data members and properties are serialized and that we can control the serialization by using attributes.

But there are some problems. One of them is that not every data type can be serialized. Only few chosen are supported. For instance two-dimensional arrays (**int** [,]) can not be serialized this way.

Collections and compositions are supported but if we use class inheritance we have to specify for each data member or property exactly what classes can it refer to. This completely violates the Open-Closed Principle. It means that if we add a new derived class we have to update all the places in the code that can hold a reference to this new class. This significantly reduces modularity of our system. In other words it is not possible for instance to add new modules to our system only by copying some dlls to some directory. You have to change the code.

We have not mentioned generic classes yet. The XML Serialization works fine with generic classes too, but there is one problem. If we work with derived classes there is no way how to specify what generic classes can a data member or a property refer to. It is possible to specify this only if we fix the unknown parameter T in these classes. This is not a big problem in our system, because we have restricted us only to **char**s.

3.6 XML counterparts of core classes

In this section we describe how the XML serialization is used in our system to serialize and (or) deserialize core classes, languages and restarting automata. We suppose that the reader has basic understanding of XML serialization in C♯ 2.0 (otherwise see the Section 3.5).

The first idea is to make all mentioned classes serializable.

This approach has some significant flaws. First the serializable class must be specified as public and it must have a parameterless constructor. This is a problem especially for immutable classes. An immutable class should not have a parameterless constructor, but what is more important, it must not have setter methods (in properties) and public attributes, because otherwise we could modify the internal state of the class. In other words it is impossible to have an immutable class that is serializable.

Then there are classes such as DFA<T> that are not immutable, but also can be serialized (deserialized) only with difficulties. First problem is that some data types cannot be serialized (for instance two-dimensional arrays). The second problem is that this approach violates one of the basic design principles called KISS (Keep It Simple Stupid). Classes such as DFA<T> are designed to fulfill one specific purpose or task. If we made such a class serializable, the complexity of the class would necessarily grow. It would be more difficult to modify or extend such a class.

We have decided to separate XML serialization from core classes in our system by creating XML counterparts of core classes.

The basic idea is illustrated in the Figure 3.11.

Suppose that we want to serialize (or deserialize) a class Widget. First step is to specify a special IWidgetData interface. This interface contains all necessary methods that give us all data sufficient to create an instance of the Widget class. (This explains the postfix Data in the name of the interface). The Widget class must implement this interface and also there must be a constructor in the Widget class that allows us to create an instance of Widget class from IWidgetData interface.

Second step is to define an XML counterpart WidgetXML class. This is a simple public serializable class which only purpose is to serialize and deserialize all data provided by IWidgetData interface. Again this class must

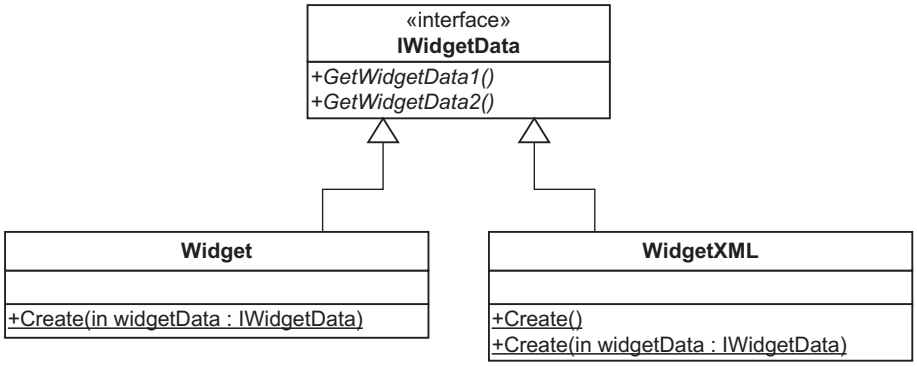


Figure 3.11: Widget Class Diagram.

implement IWidgetData interface and also there must be a constructor in this class that allows us to create an instance of this class from IWidgetData interface.

Now we illustrate how this all works. Suppose that we have an instance widget of Widget class and we want to serialize this instance. This can be done only with few lines of code as is shown in the Code listing 3.14.

Listing 3.14: Widget class serialization.

```
1 XmlSerializer widgetSerializer =
2   new XmlSerializer(typeof(WidgetXML));
3 WidgetXML widgetXML = new WidgetXML(widget);
4 widgetSerializer.Serialize(stream, widgetXML);
5 stream.Close();
```

The most important is the third line where we transform the original Widget class into its XML counterpart WidgetXML class.

The deserialization is as easy as the serialization because the transformation can be done also in the other way, i.e. from the WidgetXML class to the Widget class (see Code listing 3.15).

Listing 3.15: Widget class deserialization.

```
1 XmlSerializer widgetSerializer =
2   new XmlSerializer(typeof(WidgetXML));
```

```
3  WidgetXML widgetXML =
4    widgetSerializer.Deserialize(stream) as WidgetXML;
5  Widget widget = new Widget(widgetXML);
6  stream.Close();
```

There are few important things to note:

First it is recommended to choose the return data types of methods of IWidgetData interface as simple as possible (the best are the data types that can be easily serialized/deserialized).

Second if we implement IWidgetData interface it is advisable not to return references to internal data of the class. It is recommended to return deep copies of internal data. For instance suppose that Widget class contains an array of **int**s and suppose that GetWidgetData1() method of IWidgetData interface returns a reference to this array. Then it would be possible to change internal data of Widget class through this method.

In spite of these two limitations this approach works great in practice. It allows us to change an XML representation of classes without touching the code of these classes. We only modify the code of their XML counterparts.

Also inheritance in XML serialization is implemented easier with simple XML classes rather than with complex multipurpose serializable classes.

Another useful feature of this approach is that we can create the whole hierarchy of IData interfaces. For instance in our system IDFAData<T> interface extends IAlphabetData<T> interface. It means that whenever we have an instance of a DFA<T> class we can create (for example) an instance of AlphabetXML class. This feature is also widely used in our system.

For better explanation of this idea we give two examples. The first example is an AlphabetXML<T> class that represents an XML counterpart of the Alphabet<T> class and the second example is a DFAXML<T> class that represents an XML counterpart of the DFA<T> class.

3.6.1 AlphabetXML<T>

Class AlphabetXML<T> represents an XML counterpart of the Alphabet<T> class. Both of these classes implement IAlphabetData<T> interface as is shown in the Figure 3.12.

Note that in the method GetAlphabetData() of the IAlphabetData<T>

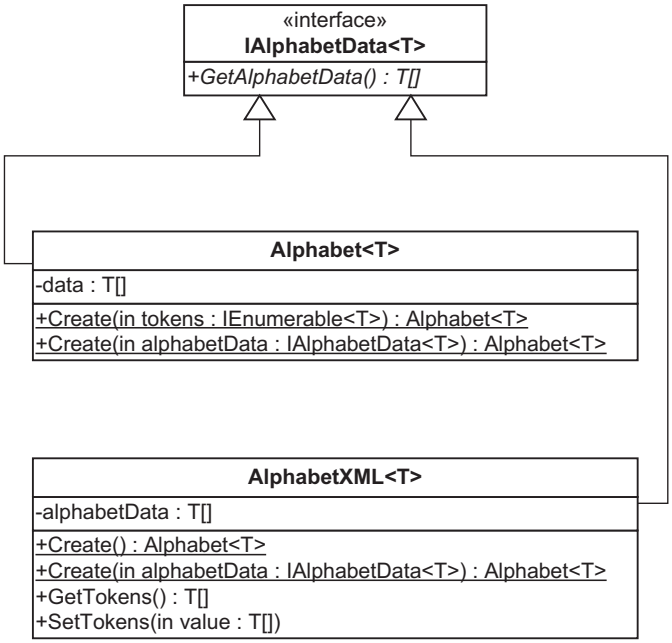


Figure 3.12: AlphabetXML<T> Class Diagram.

interface we always return a copy of the internal array of tokens. The implementation of this method in the AlphabetXML<T> class is shown in the Code listing 3.16.

Listing 3.16: GetAlphabetData().

```
1 public T[] GetAlphabetData()
2 {
3         return this.alphabetData.Clone() as T[];
4 }
```

3.6.2 DFAXML<T>

Class DFAXML<T> represents an XML counterpart of the DFA<T> class. Both of these classes implement IDFAData<T> interface as is shown in the Figure 3.13.

First note that the IDFAData<T> interface extends the IAlphabetData<T> interface. This allows us to create alphabetXML private member in the

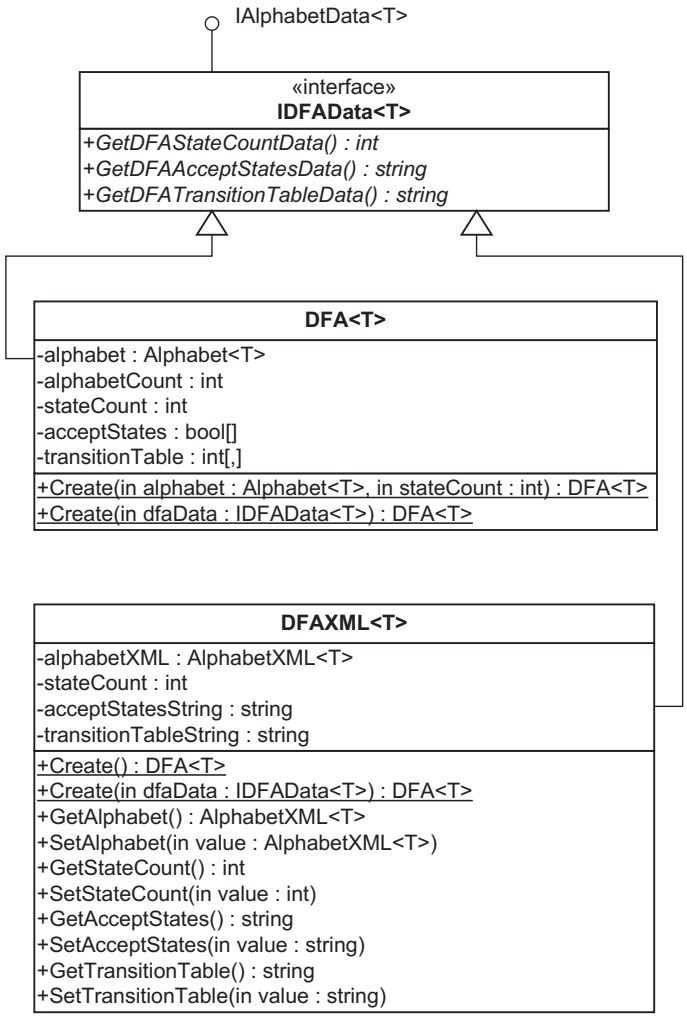


Figure 3.13: DFAXML<T> Class Diagram.

DFAXML<T> constructor by using IDFAData<T> interface dfaData as is shown in the Code listing 3.17:

Listing 3.17: DFAXML<T> constructor.

```
1 public DFAXML(IDFAData<T> dfaData)
2 {
3     this.alphabetXML = new AlphabetXML<T>(dfaData);
4     this.stateCount = dfaData.GetDFAStateCountData();
5     this.acceptStatesString =
6       dfaData.GetDFAAcceptStatesData();
7     this.transitionTableString =
8       dfaData.GetDFATransitionTableData();
9 }
```

The same holds also for the DFA<T> class.

Also note that the return value data types of Get...Data() methods of the IDFAData<T> interface are simple data types that can be serialized (or deserialized).

For instance GetDFATransitionTableData() returns the transition table as a **string**. This is because we are not able to serialize (deserialize) two-dimensional arrays. The transformation to and from the string representation are of course implemented only in the DFA<T> class.

3.7 Restarting automata

In this section we describe one of the most important classes of our system, the RestartingAutomaton<T> class. We use representation of these automata adapted from the Section 1.4. In this representation the restarting automaton consists of the set of accepting meta-instructions and the set of reducing (or rewriting) meta-instructions.

Every accepting meta-instruction is defined by its accepting language and every reducing meta-instruction is defined by its left language, right language and two words: from-word and to-word.

The first part of this section is devoted to languages. The second part describes accepting and reducing meta-instructions and the last third part describes the RestartingAutomaton<T> class.

3.7.1 Languages

A language L is a subset of Σ^* where Σ is a finite nonempty set of symbols called the alphabet (see Section 1.1). In our system a language is an immutable class that implements ILanguage<T> interface as is shown in the Figure 3.14.

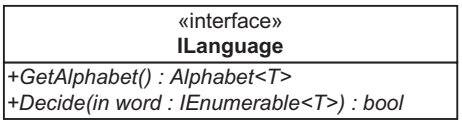


Figure 3.14: ILanguage<T> Class Diagram.

In other words a language is an immutable class that can decide whether a given word is accepted or rejected and that can give you its own alphabet.

There are some core classes in our system (such as DFA<T> or SLT<T>) that can decide whether a given word is accepted or rejected and also can give you the alphabet, however none of these classes implements the ILanguage<T> interface. This is because these classes are not languages. They are designed for other purposes.

On the other hand behind every language class in our system there is one of these core classes hidden that actually realizes these decisions whether to accept or reject a given word. For instance behind LanguageDFA<T> class there is a DFA<T> class.

You can easily recognize the Adapter design pattern behind this schema. Language classes are adapters that adapts core classes to ILanguage<T> interface. Core classes are in the role of adaptees.

These language classes also have their own XML counterparts and the important thing to note is that every such XML language class is inherited from the ILanguageXML<T> base class. This allows us to serialize and deserialize language classes without knowing the exact type of the language. It is not as easy as it seems at first glance. For this to work we have created two static classes. The first one is LanguageFactory<T> class and the second one is LanguageXMLFactory<T> class (see Figure 3.15).

The ILanguageData<T> interface is the common ancestor of all interfaces connecting language classes with their XML counterparts.

If we have an instance that implements ILanguageData<T> interface then by using LanguageFactory<T> we can create the corresponding language and

by using LanguageXMLFactory<T> we can create the corresponding XML counterpart to this language.

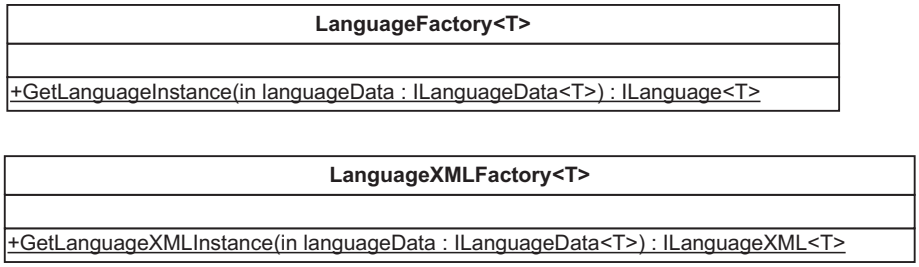


Figure 3.15: Language Factory.

In the Code listing 3.18 you can see the LanguageFactory<T> static class implementation. (LanguageXMLFactory<T> looks similarly).

Listing 3.18: LanguageFactory<T> implementation.

```
1  public static ILanguage<T> GetLanguageInstance(
2    ILanguageData<T> languageData)
3  {
4    if (languageData == null)
5      return null;
6    else if (languageData is ILanguageDFALStarData<T>)
7      return new LanguageDFALStar<T>(
8        languageData as ILanguageDFALStarData<T>);
9    else if (languageData is ILanguageDFARPNIData<T>)
10     return new LanguageDFARPNI<T>(
11       languageData as ILanguageDFARPNIData<T>);
12   else if (languageData is ILanguageDFAData<T>)
13     return new LanguageDFA<T>(
14       languageData as ILanguageDFAData<T>);
15   else if (languageData is ILanguageRegexData<T>)
16     return new LanguageRegex<T>(
17       languageData as ILanguageRegexData<T>);
18   else if (languageData is ILanguageSLTData<T>)
19     return new LanguageSLT<T>(
20       languageData as ILanguageSLTData<T>);
21   else throw new InvalidProgramException(
22     "LanguageFactory: Unrecognized ILanguageData interface.");
23 }
```

3.7.2 Metainstructions

In the Figure 3.16 you can see the class diagrams of accepting and reducing meta-instructions. Note that their members are immutable classes.

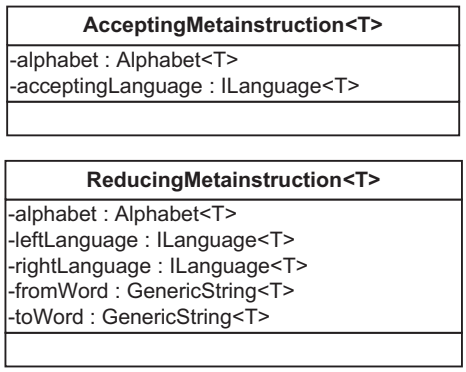


Figure 3.16: Accepting and reducing meta-instructions.

These classes do not have any special functionalities. They serve only as a package that encapsulates their own attributes.

These classes also have their XML counterparts. The only interesting thing to note is that if we want to serialize (or deserialize) languages we have to specify all descendants of ILanguageXML<T> base class as is shown in the Code listing 3.19.

Listing 3.19: AcceptingMetainstructionXML<T> language property.

```
1  [XmlElement("AccLngFA", typeof(LanguageDFAXML<char>)),
2  XmlElement("AccLngDFALStar", typeof(LanguageDFALStarXML<char>)),
3  XmlElement("AccLngDFARPNI", typeof(LanguageDFARPNIXML<char>)),
4  XmlElement("AccLngRegex", typeof(LanguageRegexXML<char>)),
5  XmlElement("AccLngSLT", typeof(LanguageSLTXML<char>))]
6  public ILanguageXML<T> AcceptingLanguage
7  {
8          get { return this.acceptingLanguageXML; }
9          set { this.acceptingLanguageXML = value; }
10 }
```

3.7.3 RestartingAutomaton<T>

In the Figure 3.17 you can see the class diagram of RestartingAutomaton<T> class. This class represents a package that contains the list of accepting meta-instructions and the list of reducing meta-instructions.

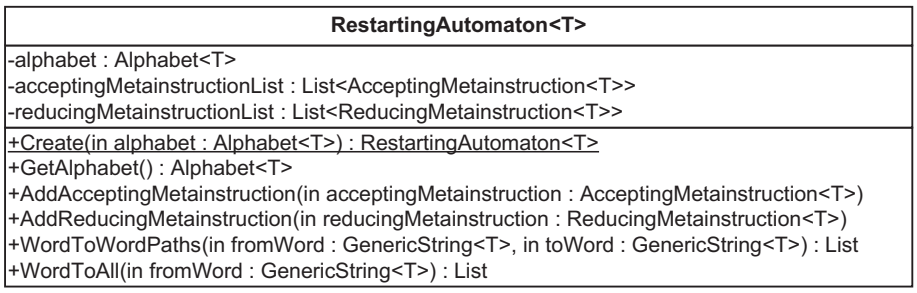


Figure 3.17: RestartingAutomaton<T> Class Diagram.

The most important methods are WordToWordPaths and WordToAll methods. The first one returns the list of all reducing paths from the fromWord to the toWord and the second one returns the list of all words that can be reduced from the fromWord.

This class is the place where it is possible to add new functionalities of the restarting automaton.

3.8 How to add new learning protocols

In this section we describe how to add new learning protocols to the system. Learning protocols are protocols that are used to produce languages. The best way how to describe this process is by using an example. We have decided to describe adding k-SLT languages to the system.

1. We define SLT<T> class that encapsulates the learning protocol.

2. Sometimes this step is unnecessary but in this case we define an XML counterpart: SLTXML<T> class and the corresponding ISLTData<T> interface that connects these two classes.

   In the Code listing 3.20 you can see a small fraction of the implementation of these SLT core classes.

Listing 3.20: SLT core classes.

```
1  public interface ISLTData<T> :
2      IPositiveNegativeSamplesData<T>
3      where T : struct, IComparable<T>, IEquatable<T>
4  {
5      /// Returns k for k-SLT languages.
6      int GetSLTK();
7  }
8
9  public sealed class SLT<T> :
10     ISLTData<T>,
11     ICloneable
12     where T : struct, IComparable<T>, IEquatable<T>
13 {
14     #region Constructors
15     public SLT(Alphabet<T> alphabet, int k) { ... }
16     public SLT(ISLTData<T> sltData) { ... }
17     #endregion
18
19     #region ISLTData Interface
20     /// Returns data describing the alphabet.
21     public T[] GetAlphabetData() { ... }
22     ...
23     /// Returns k for k-SLT languages.
24     public int GetSLTK() { ... }
25     #endregion
26
27     #region ICloneable Interface
28     #region Properties
29     #region Event Handlers
30     #region Methods
31     #region Private Methods
32     #region Private Data Members
33 }
34
35 [XmlRoot("SLT")]
36 public sealed class SLTXML<T> :
37     ISLTData<T>
38     where T : struct, IComparable<T>, IEquatable<T>
39 {
```

```
40      #region Constructors
41      public SLTXML() { ... }
42      public SLTXML(ISLTData<T> sltData) { ... }
43      #endregion
44
45      #region ISLTData Interface
46      #region Properties
47      [XmlElement("PositiveNegativeSamples")]
48      public PositiveNegativeSamplesXML<T>
49          PositiveNegativeSamples { get; set; }
50      [XmlElement("K")]
51      public int K { get; set; }
52      #endregion
53
54      #region Private Data Members
55  }
```

As you can see in the Code listing 3.20 the ISLTData<T> interface extends the IPositiveNegativeSamplesData<T> interface (which extends the IAlphabetData<T> interface).

It means that the positive and negative samples (including the alphabet) together with the positive integer k represent all data necessary to initialize an instance of the SLT<T> class (or the SLTXML<T> class).

3. We define an immutable LanguageSLT<T> class that adapts SLT<T> class to the ILanguage<T> interface.

4. We define an XML counterpart: LanguageSLTXML<T> class to the LanguageSLT<T> class and the corresponding ILanguageSLTData<T> interconnecting interface. The LanguageSLTXML<T> class must be inherited from ILanguageXML<T> class and the ILanguageSLTData<T> interface must extend ILanguageData<T> interface.

   In the Code listing 3.21 you can see a small fraction of the implementation of these SLT language classes.

Listing 3.21: SLT language classes.

```
1  public interface ILanguageSLTData<T> :
```

```
2       ISLTData<T>,
3       ILanguageData<T>
4       where T : struct, IComparable<T>, IEquatable<T>
5   {
6
7   }
8
9   public sealed class LanguageSLT<T> :
10      ILanguage<T>,
11      ILanguageSLTData<T>
12      where T : struct, IComparable<T>, IEquatable<T>
13  {
14      #region Constructors
15      public LanguageSLT(SLT<T> slt) { ... }
16      public LanguageSLT(ILanguageSLTData<T>
17          languageSLTData) { ... }
18      #endregion
19
20      #region ILanguage Interface
21      public Alphabet<T> Alphabet { get; }
22      public bool Decide(IEnumerable<T> word) { ... }
23      #endregion
24
25      #region ILanguageSLTData Interface
26      #region Methods
27      #region Private Data Members
28  }
29
30  [XmlRoot("LanguageSLT")]
31  public class LanguageSLTXML<T> :
32      ILanguageXML<T>,
33      ILanguageSLTData<T>
34      where T : struct, IComparable<T>, IEquatable<T>
35  {
36      #region Constructors
37      public LanguageSLTXML() { ... }
38      public LanguageSLTXML(ILanguageSLTData<T>
39          languageSLTData) { ... }
40      #endregion
41
```

```
42      #region ILanguageSLTData Interface
43
44      #region Properties
45      [XmlElement("SLT")]
46      public SLTXML<T> SLT { get; set; }
47      #endregion
48
49      #region Private Data Members
50  }
```

5. We extend the LanguageFactory<T> and the LanguageXMLFactory<T> to be able to recognize ILanguageSLTData<T> interface. You can see the extension of the LanguageFactory<T> in the Code listing 3.18 (the rows 18, 19 and 20).

6. As we have seen in the Section 3.7.2 we must add some code to the accepting and reducing meta-instruction XML classes so that they are able to recognize this new language during the process of XML serialization and deserialization. You can see the extension for the accepting meta-instruction in the Code listing 3.19 (the row 5).

7. The last step is to install this learning protocol into the GUI of the application. This step is out of scope of this thesis so we are not going to describe the process.

3.9 How to add new functionality

In this section we describe how to add a new functionality of the restarting automaton to the system.

1. Add a new method (or methods) with the desired functionality into the RestartingAutomaton<T> class.

2. Make this new functionality accessible from the GUI of the application.

Chapter 4

User guide

In this chapter we describe how to work with RestartingAutomaton application that is attached to this thesis on a CD (see Figure 4.1).

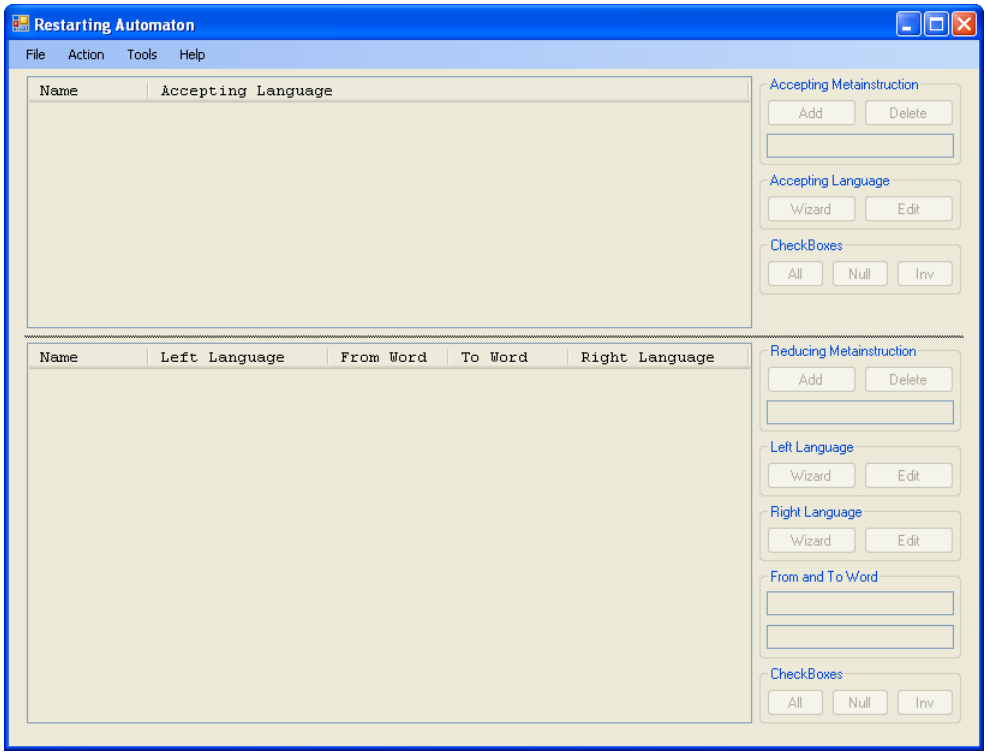


Figure 4.1: RestartingAutomaton application.

We first introduce the main purpose of the RestartingAutomaton appli-

cation and mention all the things you can do with the application.

Then we describe the installation process of the application and show how to run this application on your own computer.

Finally we introduce language tools that are used to define formal languages and after this we show how to design and test restarting automaton.

4.1 Purpose

RestartingAutomaton application allows you to do the following things:

1. Design a restarting automaton. The design of restarting automaton consists of stepwise design of accepting and reducing meta-instructions. You can save (load) restarting automaton to (from) an XML file.

2. Test correctly defined restarting automaton.

   (a) The system is able to give you a list of all words that can be obtained by reductions from a given word w. It also gives you a list of all accepting meta-instructions to every reduced word that accept this word. The input word w is accepted by a restarting automaton if and only if a word w can be reduced to a word that is accepted by at least one accepting meta-instruction.

   (b) The system is able to give you a list of all reduction paths from one given word to another given word.

3. Start a server if the restarting automaton is defined correctly. In the server mode the client applications can use services provided by the server application.

4. You can use specialized tools to define formal languages. Each tool enables you to test the correctly defined language. You can also save, load, copy, paste and view an XML representation of the actual state of the tool.

Note that the application allows you only to design simple restarting automata recognizing only simple formal languages with small alphabets consisting of few letters. On large inputs the computation can take a long time and it can produce a huge output.

4.2 Installation

For Win32 platform you need to have .NET Framework 2.0 installed on your computer. You can download this framework from the Microsoft web site: http://www.microsoft.com/. If the .NET Framework is installed correctly, just copy the file:
<CD Drive>\RestartingAutomaton\RestartingAutomaton.exe
from a CD attached to this thesis to a local drive. To run the application just double-click on the RestartingAutomaton.exe file.

For UNIX platform you need to have Mono project installed on your computer. For more information see the web page: http://www.mono-project.com/. If the Mono project is installed correctly, just copy the file:
<CD Drive>\RestartingAutomaton\RestartingAutomaton.exe
from a CD attached to this thesis to a local drive. To run the application just enter the command:

```
> mono RestartingAutomaton.exe
```

The source code of the application is packed in the file:
<CD Drive>\RestartingAutomaton\RestartingAutomaton.zip
on a CD attached to this thesis. If you want to do modifications to the code we recommend to use Microsoft Visual Studio 2005 as a development environment.

4.3 Language tools

In this section we introduce language tools that are used to define formal languages. In the menu Tools there are following language tools:

1. DFA Modeler
2. LStar Algorithm
3. RPNI Algorithm
4. Regular Expression
5. SLT Language

These tools are graphical user tools used to work with learning protocols as defined in the requirements of the system.

You can use them to try how these learning protocols work and of course you can save the results of your work for later use either to a file or to a clipboard.

You can also use these tools for defining languages of meta-instructions during the design of restarting automaton.

All these tools have a lot of functionalities in common. The work with every tool usually starts with defining an alphabet. If you use these tools during the process of creation of the restarting automaton the alphabet is automatically inherited from the restarting automaton.

The alphabet in our system always consists of finite number of UNICODE characters. All characters are supported but we recommend to avoid strange characters like whitespaces or quotation marks.

4.3.1 DFA Modeler Tool

DFA Modeler tool allows you to enter a regular language by specifying its underlying deterministic finite automaton. To open this tool just click on the DFA Modeler menu item in the Tools menu.

You can see the corresponding window in the Figure 4.2.

The largest text box is a place where you can enter the transition table of the DFA. The transition table contains all necessary information about the DFA including the alphabet, the number of states and the transitions between the states.

Here is an example of a transition table of a DFA that accepts the words with even number of as and even number of bs:

```
              a    b
<->      0    1    2
         1    0    3
         2    3    0
         3    2    1
```

The first row of the transition table is the alphabet. In our case the alphabet consists of two symbols: a and b.

The first column contains the numbers 0, 1, ... , $n-1$ (in this order), where n is the total number of states of the DFA.

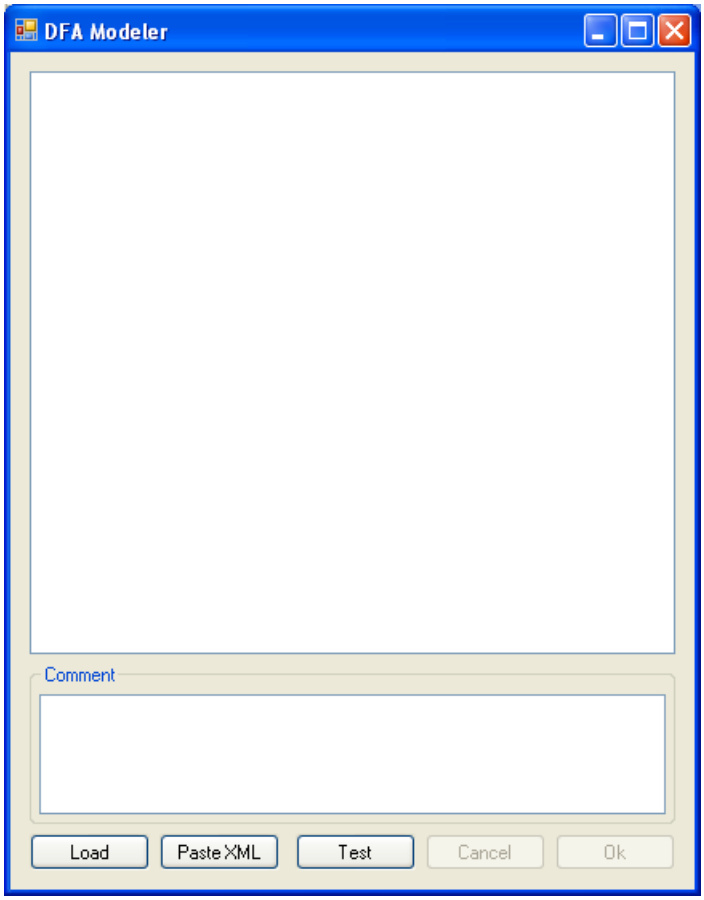


Figure 4.2: DFA Modeler tool.

The start state of the DFA is always the state with the number 0. We express this fact with an arrow -> to the left of this state. In our case the start state is also an output state, so we write an arrow <-> to the left of this state.

If we want to express that a state (other than the start state) is an output state we write an arrow <- to the left of this state.

Transitions are written to the right side of the state. For instance in our example if we are in the state with the number 2 and we read a symbol a we get to the state with the number 3. If we read a symbol b we get to the state with the number 0.

There is also a Comment TextBox where you can enter a comment to the DFA. The comment is an auxiliary information that does not have an influence on the DFA.

After entering the transition table you can click on the Test button to test the corresponding DFA. If something is wrong with the transition table the message box will tell you what you have to fix. If the transition table is correct the DFA Preview form will show up as in the Figure 4.3.

To test the DFA just enter some word to the white TextBox and click on the Decide button. You will get a message telling you whether the entered word is accepted or rejected by the DFA. If you enter a word that is not from the alphabet of the DFA you will get a message telling you that all letters of the word must be from the alphabet.

You can also save (copy) an XML representation of the DFA into a file (clipboard) by clicking on the Save (Copy XML) button. You can later load (paste) this XML representation from a file (clipboard) in the DFA Modeler tool.

4.3.2 LStar Algorithm Tool

LStar Algorithm tool encapsulates Dana Angluin's L* algorithm that is a machine learning algorithm which learns deterministic finite automaton using membership and equivalence queries (see Section 2.1). To open this tool just click on the LStar Algorithm menu item in the Tools menu.

You can see the corresponding window in the Figure 4.4.

First thing you have to do is to define an alphabet. Just click on the Alphabet button and then enter the alphabet into the white TextBox of the Alphabet form (see Figure 4.5).

As we have said the alphabet always consists of finite number of UNI-

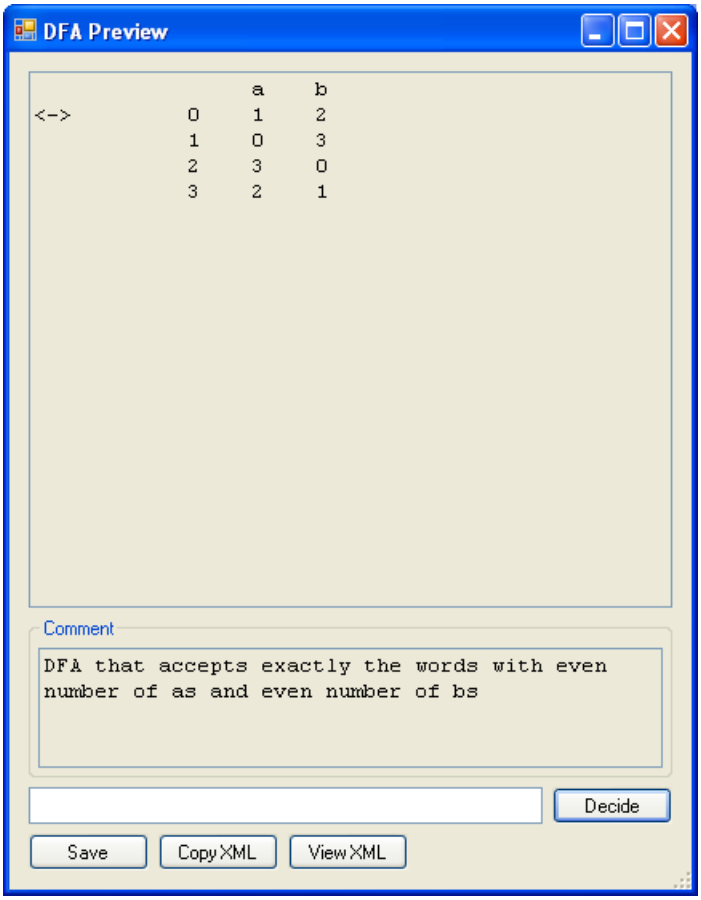


Figure 4.3: DFA Preview form.

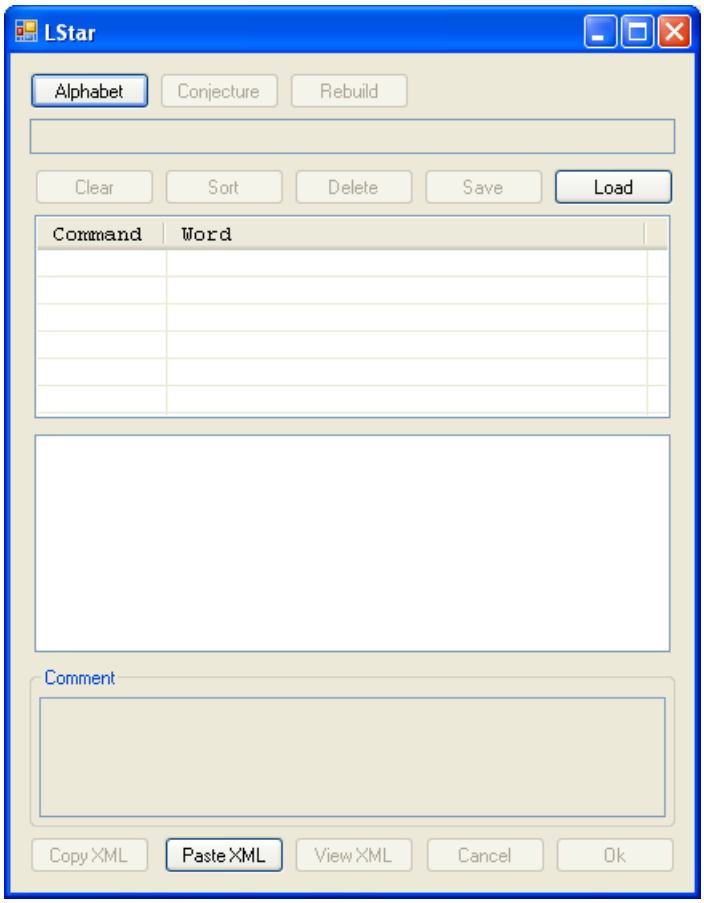


Figure 4.4: LStar Algorithm tool.

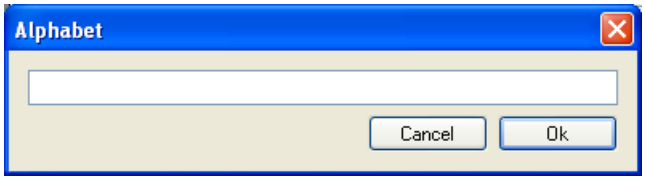


Figure 4.5: Alphabet form.

CODE characters. We recommend to avoid strange characters like whitespaces or quotation marks although these characters are also supported.

For example enter the alphabet ab consisting of two symbols: a and b, and then click on the Ok button.

After defining the alphabet some of the controls of the LStar Algorithm tool will be enabled. The TextBox under the Alphabet button is a command TextBox. Here you can enter commands to express what words should be accepted and what words should be rejected. Each command has one of the following forms:

1. `accept word`
2. `acc word`
3. `reject word`
4. `rej word`

where `word` is a word from the entered alphabet (in our case from the alphabet $\{a, b\}$).

If the command starts with `accept` or `acc` then the entered word is considered to be accepted. On the other hand if the command starts with `reject` or `rej` then the entered word is considered to be rejected.

If the alphabet contains special symbols like whitespaces or quotation marks you can use single or double quotation marks to enclose the word. These enclosing quotation marks are of course ignored. For instance `aabb` is the same as `'aabb'` or `"aabb"`, and `"abcd 'bb"` is the same as `'abcd ''bb'`.

As a consequence if you want to enter a command to accept (or reject) an empty word just enter the command `accept ''` (or `reject ''`).

The ListView under the command TextBox is a command ListView. It shows you the list of all entered commands. This list of commands is also called a knowledge base. You can delete or rearrange the items in this ListView, but after doing this you have to click on the Rebuild button. This Rebuild button causes that the LStar algorithm is run anew on this new modified knowledge base.

In the second ListView under the command ListView you can see the set of unknown words. You have to decide whether to accept or reject these words by entering the corresponding command into the command TextBox. If this ListView does not contain any unknown words then the L* algorithm has enough information to produce a conjecture. It means that you can click

on the Conjecture button to see a DFA that is consistent with the knowledge base. If this conjecture is your target DFA then you are done. If it is not you have to enter the counterexample command into the command TextBox. After doing this new unknown words will arise.

As in the DFA Modeler tool (see Section 4.3.1) there is a Comment TextBox where you can enter a comment to the knowledge base.

The Save and Load button are used to save or load the XML representation of the knowledge base. The Copy XML button copies the XML representation of the knowledge base into the clipboard, the Paste XML button pastes the knowledge base from the clipboard into the form (if it is possible) and the View XML button shows you the dialog with the XML representation of the knowledge base.

4.3.3 RPNI Algorithm Tool

RPNI Algorithm tool encapsulates a machine learning algorithm which learns deterministic finite automaton based on a given set of labeled examples (see Section 2.2). To open this tool just click on the RPNI Algorithm menu item in the Tools menu.

You can see the corresponding window in the Figure 4.6.

First click on the Alphabet button and then enter the alphabet into the white TextBox of the Alphabet form (see Figure 4.5). After defining the alphabet some of the controls of the RPNI Algorithm tool will be enabled.

Into the Accept TextBox enter all the words that should be accepted and into the Reject TextBox enter all the words that should be rejected. These two sets of words must be disjunctive.

If the alphabet contains special symbols like whitespaces or quotation marks you can use single or double quotation marks to enclose words as was described in the Section 4.3.2.

After defining all positive and negative samples you can click on the Conjecture button to get a DFA consistent with all these samples.

The Save, Load, Copy XML, Paste XML and View XML buttons work in the same way as the corresponding buttons of the LStar Algorithm tool described in the Section 4.3.2. The only difference is that you can load (or paste) not only the XML representation of positive and negative samples from the RPNI Algorithm tool, but also the XML representation of the knowledge base from the LStar Algorithm tool.

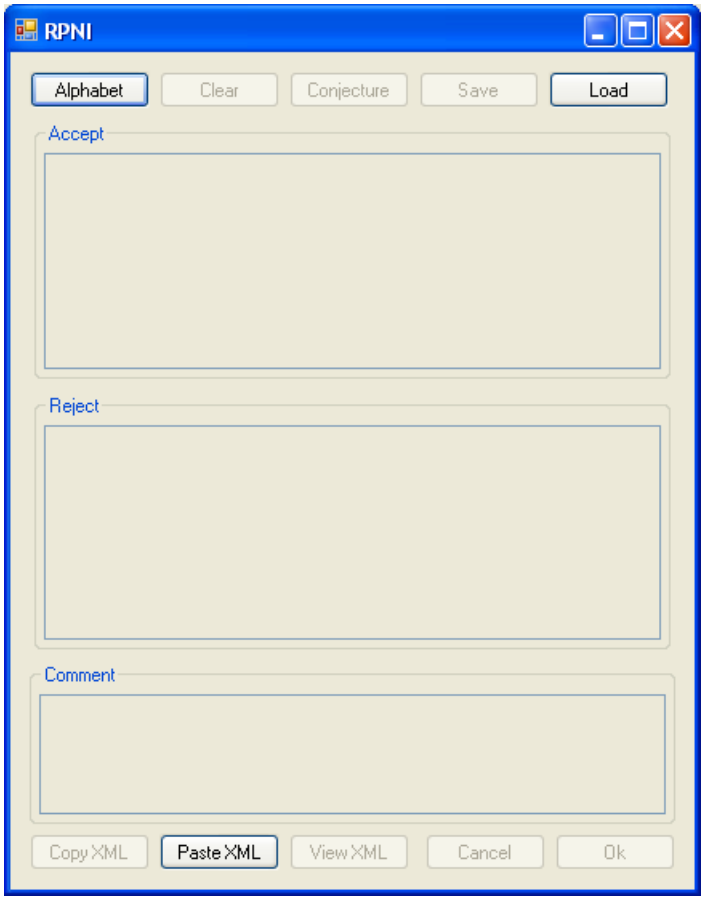


Figure 4.6: RPNI Algorithm tool.

4.3.4 Regular Expression Tool

With Regular Expression tool it is possible to enter a regular language by specifying the regular expression (extended regular expressions are supported). To open this tool just click on the Regular Expression menu item in the Tools menu.

You can see the corresponding window in the Figure 4.7.

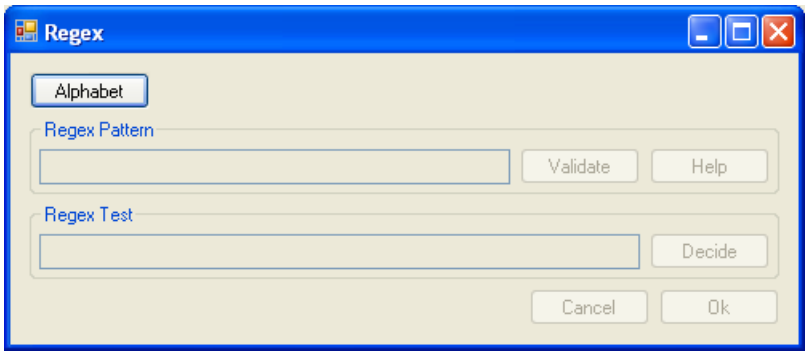


Figure 4.7: Regular Expression tool.

First click on the Alphabet button and then enter the alphabet into the white TextBox of the Alphabet form (see Figure 4.5). After defining the alphabet some of the controls of the RPNI Algorithm tool will be enabled.

Into the first Regex Pattern TextBox enter the regular expressions. If you want to see some examples of regular expressions just click on the Help button. After entering the regular expression you can check the correctness of the regex pattern by clicking on the Validate button.

To test the regular expression you can use the second Regex Test TextBox. Here you can enter a word and then test whether this word is accepted or rejected by the regular expression by clicking on the Decide button.

As you can see there are no buttons for saving or loading (copying or pasting) an XML representation of the regular expression. This is because regular expressions are defined only by their pattern so it is useless to have a special XML representation only for the regex pattern.

4.3.5 SLT Language Tool

With SLT Language tool it is possible to design a regular language by specifying a positive integer k and positive examples using the algorithm for

learning k-SLT languages (see Section 2.3) To open this tool just click on the SLT Language menu item in the Tools menu.

You can see the corresponding window in the Figure 4.8.

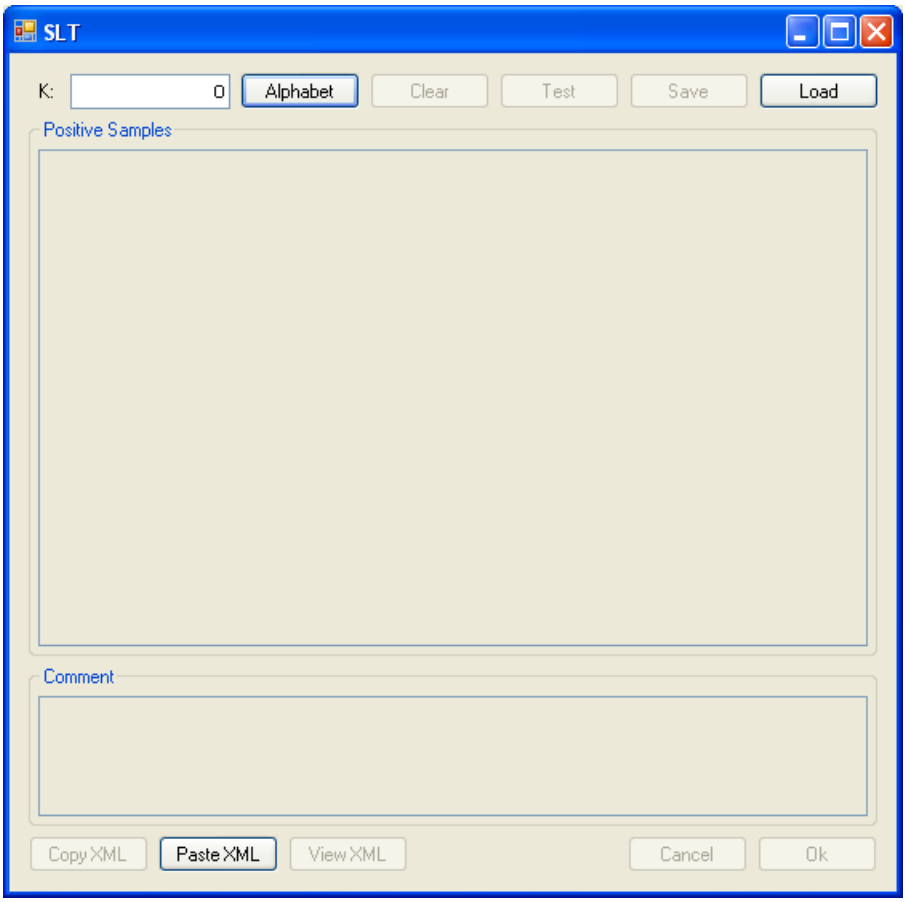


Figure 4.8: SLT Language tool.

First enter a positive integer k into the K TextBox and then click on the Alphabet button and enter the alphabet into the white TextBox of the Alphabet form (see Figure 4.5).

For example enter the positive integer $k = 3$ and then enter e.g. the alphabet ab consisting of two symbols: a and b.

After defining the positive integer k and the alphabet some of the controls of the SLT Language tool will be enabled.

Into the largest Positive Samples TextBox you can enter positive samples. In our case enter e.g. the samples: `ab`, `ba`, `aabbbbbb` and `abababab`.

To test the inferred k-SLT language just click on the Test button and the SLT Preview form will show up as in the Figure 4.9.

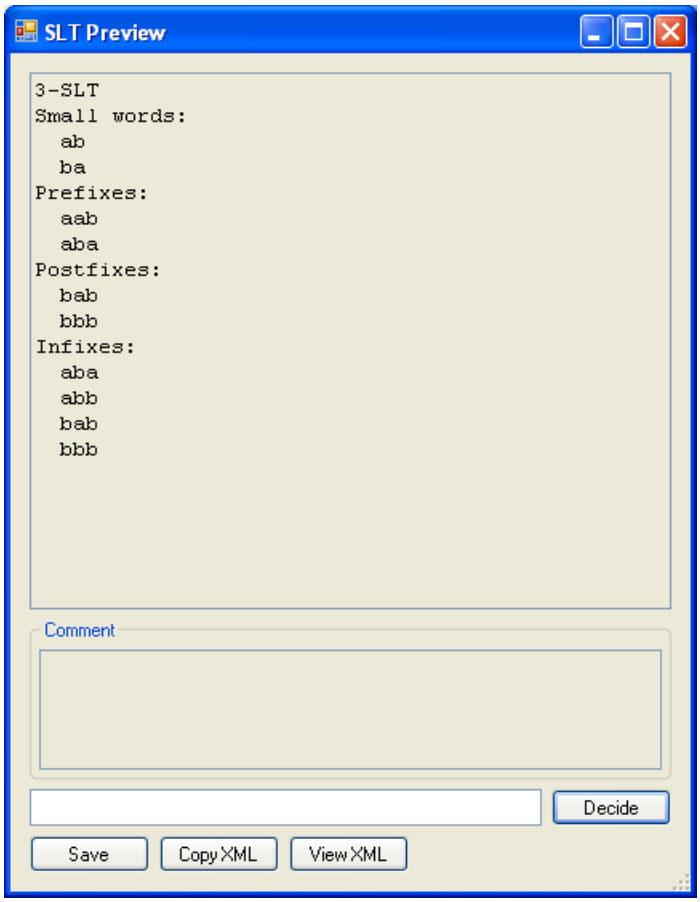


Figure 4.9: SLT Preview form.

For testing just enter some word to the white TextBox and then click on the Decide button.You will get a message telling you whether the entered word is accepted or rejected by the k-SLT language. If you enter a word that is not from the alphabet you will get a message telling you that all letters of the word must be from the alphabet.

4.4 Construction of restarting automaton

The incremental design of restarting automaton consists of stepwise design of meta-instructions. There are two kinds of meta-instructions: accepting and reducing meta-instructions.

Accepting meta-instruction is defined by its accepting language. Reducing meta-instruction is defined by its left language, right language and two words x and y where $\alpha x \beta$ can be reduced to $\alpha y \beta$ if α is in the left language and β is in the right language.

For more information on restarting automata see the Section 1.4.

In this section we describe how to create a restarting automaton that recognizes the language $L = \{a^i b^i c^j d^j | i, j > 0\}$.

For simplicity we use regular expressions to define languages.

This automaton will have only one accepting meta-instruction A00 with the following accepting language:

Name	Accepting Language
A00	`^abcd$`

and two reducing meta-instructions R00 and R01:

Name	Left Language	From Word	To Word	Right Language
R00	`^a*$`	ab	λ	`^b*c*d*$`
R01	`^a*b*c*$`	cd	λ	`^d*$`

You can easily see that this restarting automaton recognizes exactly the language L.

To transfer this restarting automaton into our system you have to start the RestartingAutomaton application. You can see the corresponding window in the Figure 4.1. Click on the New menu item in the File menu. The Alphabet form will show up as in the Figure 4.5. Into the white TextBox of the Alphabet form enter the alphabet for the restarting automaton.

In our case enter e.g. the alphabet *abcd* consisting of letters: a, b, c and d.

After defining the alphabet you can add accepting and reducing meta-instructions into this automaton.

Click on the Add button in the Accepting Metainstruction GroupBox to add one accepting meta-instruction.

Click twice on the Add button in the Reducing Metainstruction GroupBox to add two reducing meta-instructions.

Now you have to define languages for these meta-instructions. For the accepting meta-instruction just double-click on the subitem under the column Accepting Language and the Language Wizard will show up as in the Figure 4.10.

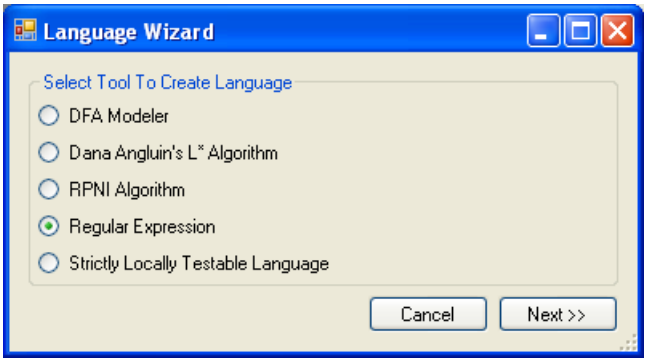


Figure 4.10: Language Wizard.

You can use any of the language tools described in the Section 4.3 to define a language. In our case select the Regular Expression RadioButton and then click on the Next button. The Regular Expression tool for entering regular expressions will show up as in the Figure 4.11.

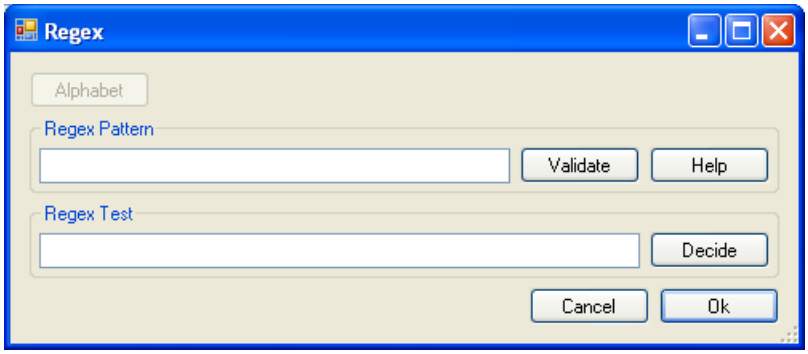


Figure 4.11: Regular Expression tool.

As you can see the Alphabet button is disabled. This is because the alphabet is automatically inherited from the restarting automaton and you

cannot change it. Also note that the Ok and Cancel buttons are enabled. You can use these buttons to confirm or reject entered language.

In our case just enter the pattern `^abcd$` and click on the Ok button.

The same procedure can be also applied for languages of the reducing meta-instructions.

To enter a from-word or a to-word just double-click on the proper subitem and the focus will be automatically redirected to the TextBox where you can enter the word. You confirm the text by pressing the enter key.

After defining all languages you should get restarting automaton as in the Figure 4.12.

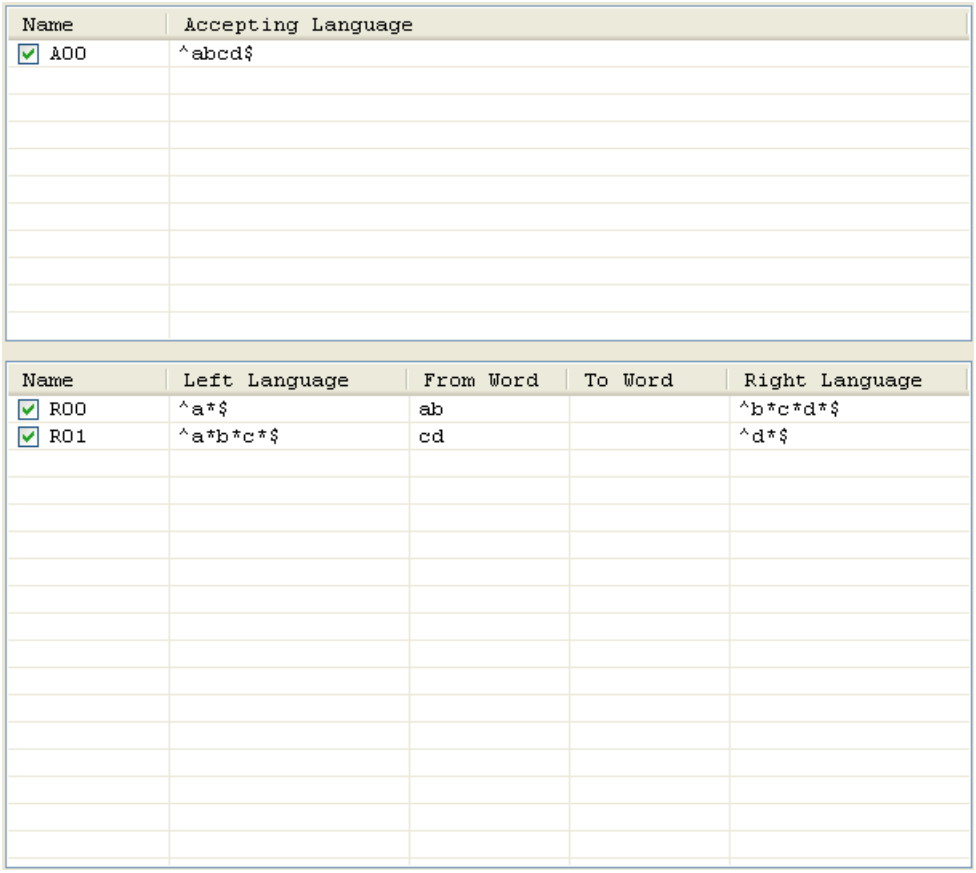


Figure 4.12: Restarting Automaton.

Now you can save the XML representation of this restarting automaton into a file. Just click on the Save menu item in the File menu. Later you can

load this automaton by clicking on the Load menu item in the File menu.

4.5 Using restarting automaton

If the restarting automaton is defined correctly you can do the following things:

1. Compute the list of all words that can be reduced from a given word. This computation also gives you an information about whether the entered word is accepted or rejected by the restarting automaton.

2. Compute the list of all reducing paths from one given word to the other given word.

In this section we use the restarting automaton defined in the previous Section 4.4.

Note that in the main form of the RestartingAutomaton application there is a CheckBox next to every meta-instruction (see Figure 4.12). This CheckBox decides whether the corresponding meta-instruction will be considered during the computation. These CheckBoxes are useful when you want to observe what happens if some meta-instructions are not allowed.

4.5.1 Word To All

If you click on the Word To All menu item in the Action menu then the Word To All dialog will show up as in the Figure 4.13.

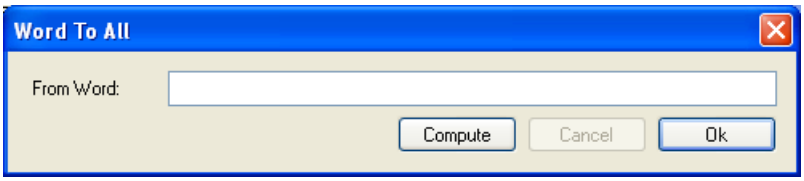


Figure 4.13: Word To All dialog.

Into the From Word TextBox enter a word from which you want to compute all its reductions.

In this case enter e.g. the word `aaabbbccdd` and click on the Compute button. The computation itself is done in another thread so if this computation takes a long time you can cancel the computation with the Cancel

button. The output will contain only the words computed until the cancellation.

After finishing of the computation you should get the Word To All Filter dialog shown in the Figure 4.14.

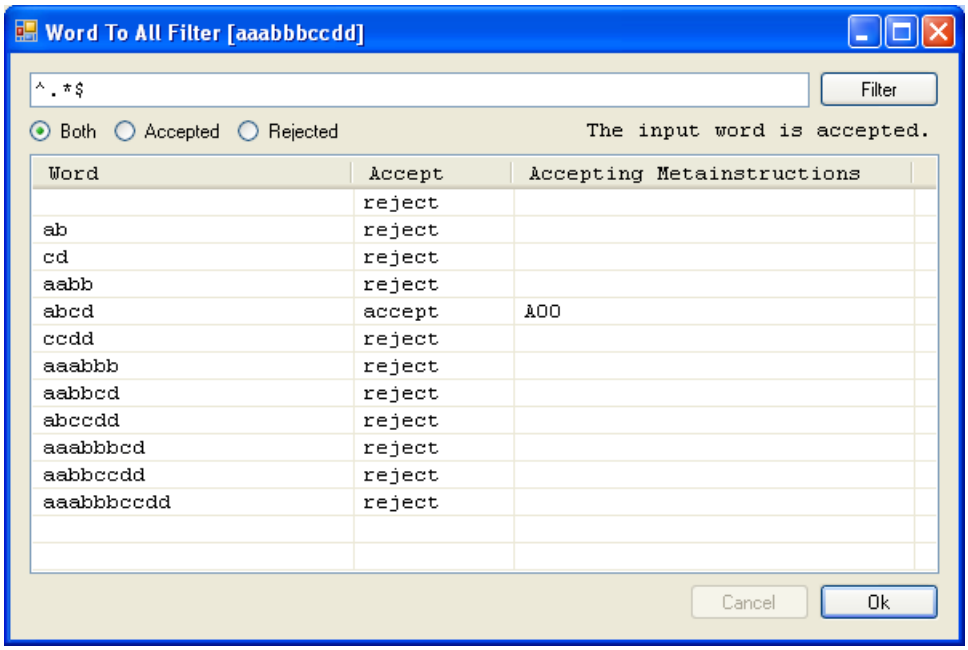


Figure 4.14: Word To All Filter dialog.

If you look at the top right part of the dialog you can see that the dialog tells you that the input word is accepted. This is because there exists a reduction of the entered word that is accepted by at least one accepting metainstruction.

The top part of the dialog is used to filter the output. You can use regular expression to filter the words matching this regular expression and also you can use RadioButtons to filter only the accepted (or rejected) words.

If you double-click on any row in the ListView in the Word To All Filter dialog you will get a dialog that shows you all reducing paths to the word on this clicked row. In the Figure 4.15 you can see this dialog after double-clicking on the word `abcd`.

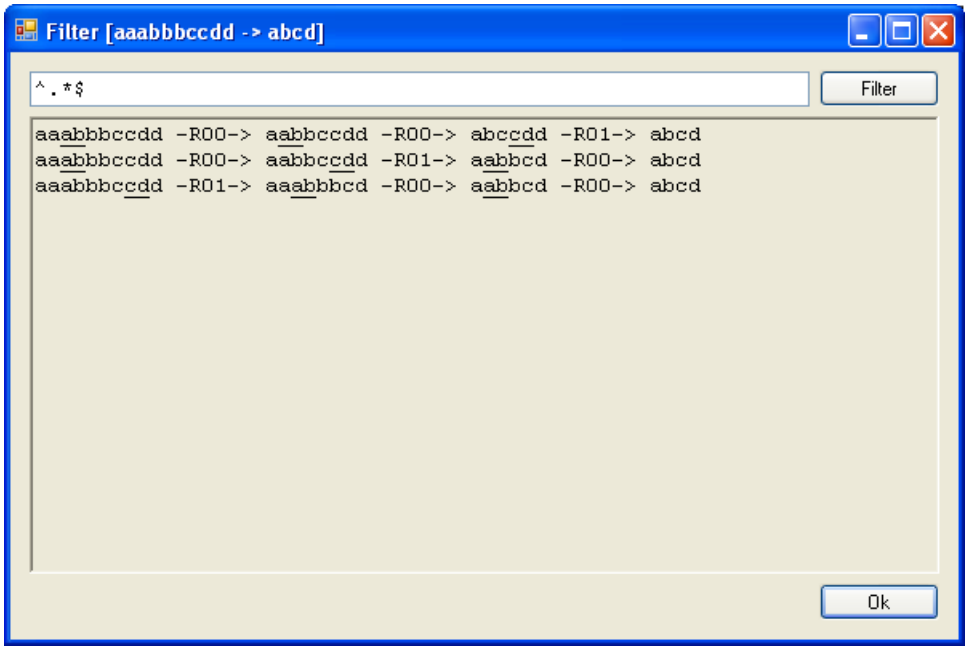


Figure 4.15: Filter dialog.

4.5.2 Word To Word

If you click on the Word To Word menu item in the Action menu then the Word To Word dialog will show up as in the Figure 4.16.

Into the From Word TextBox you enter a word from which every reduction path will start and into the To Word TextBox you enter a word to which every reduction path will end.

For instance if we enter the word `aaabbbccdd` as a from-word and `abcd` as a to-word then after clicking on the Compute button we get the Filter dialog as in the Figure 4.15.

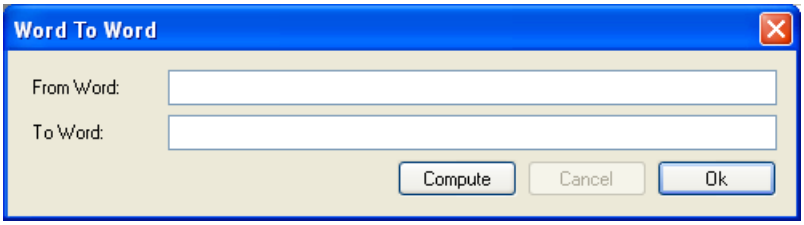


Figure 4.16: Word To Word dialog.

4.6 Remoting

RestartingAutomaton application also supports a server mode where other applications (in the role of clients) can use services provided by the server application. The communication between the client and the server application is based on the .NET Remoting technology. The description of this technology is out of scope of this thesis. On the other hand the using of this technology is quite easy and it is not difficult to build applications that use the services provided by the RestartingAutomaton application.

Suppose that we have the restarting automaton defined in the previous Section 4.4.

To start the server mode click on the Start Server menu item in the Action menu. The Server dialog will show as in the Figure 4.17.

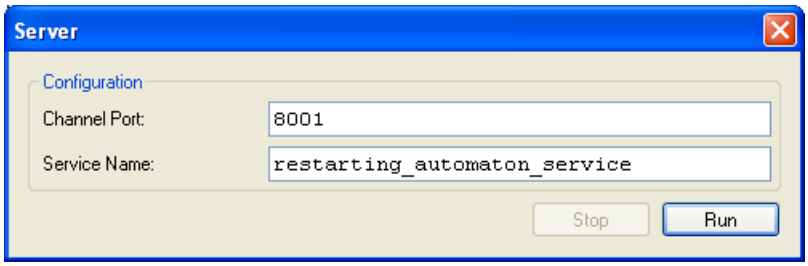


Figure 4.17: Server Dialog.

If you click on the Run button then the server will start. It is recommended not to change the channel port and the service name. If you want to change the channel port then enter the number between 8001 and 8999. Service name should contain only simple letters with no whitespaces or any other special characters.

After starting the server look at the web page:
http://localhost:8001/restarting_automaton_service?WSDL
there you will see an XML description of the provided service.

For demonstration how the remoting works we have created a special client application, the RestratingAutomatonClient application. This application is distributed together with the RestartingAutomaton application. If you run this application you should get a form as in the Figure 4.18.

If you click on the Run button while the server is running you can use the services provided by the server application.

If you want to make your own client application that uses the services of

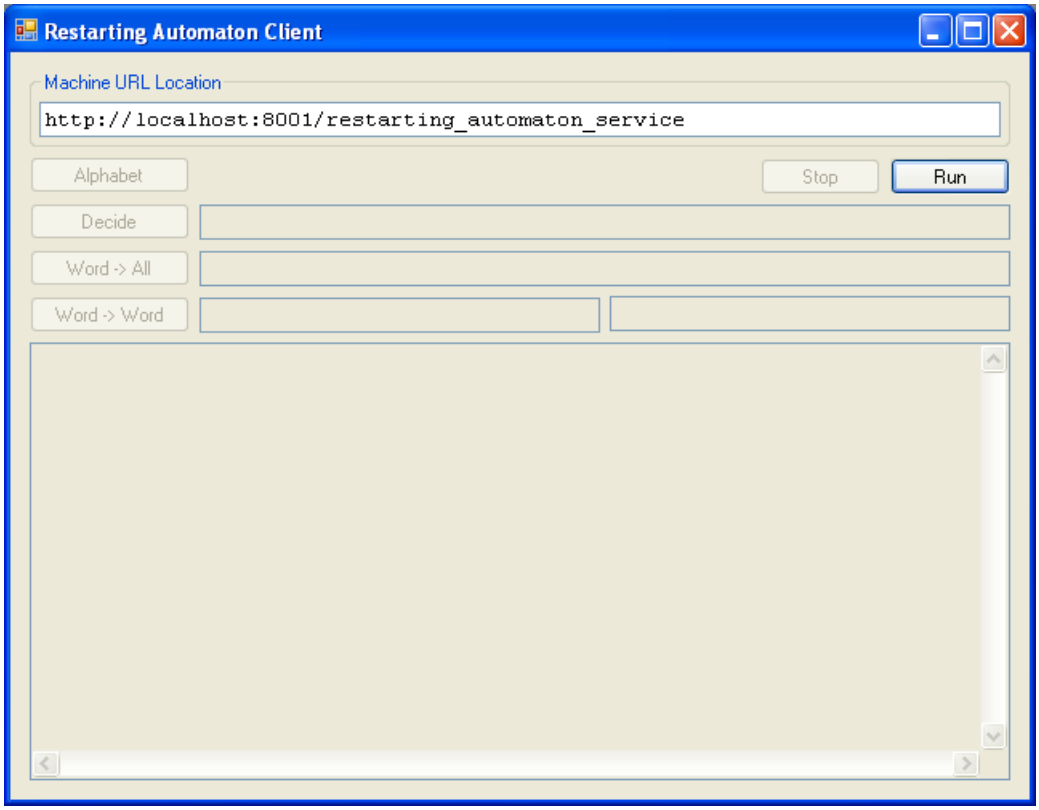


Figure 4.18: RestartingAutomatonClient application.

the RestartingAutomaton application then you need to add the RAServer.dll file into the references of your project. This assembly provides you with the interface through which you can invoke the services of the RestartingAutomaton application. The RAServer.dll file is distributed together with the RestartingAutomaton application. All details can be found in the source code of the RestratingAutomatonClient application.

Conclusion

The main goal of this thesis was to develop a specialized program with a simple user-friendly interface enabling to design and to test restarting automata. Up to now there was no such system for design restarting automata. The goal of the thesis was achieved and the system satisfies all of the proposed requirements.

The system allows interactive work with restarting automata and by using the system it is easy to do experiments, verify hypotheses or even discover new properties of restarting automata. The system also offers you some useful tools with simple user-friendly interface that you can use to investigate properties of some learning protocols like Dana Angluin's L* algorithm, RPNI algorithm etc.

The application is able to run in server mode where other client applications can use services provided by the server. This is profitable when you want to work with restarting automata, but you do not want to bother with all trivialities that can be solved on the server side.

The project is written in C♯ and runs in .NET Framework 2.0 runtime environment, so the source code is platform independent. The .NET Framework 2.0 is also supported on UNIX-like operating systems.

The system is open to adding new modules, learning protocols or new functionalities of restarting automata. The source code of the application is well readable so it is not difficult to do modifications. The whole application is build out of some useful components and classes that can be easily reused in other systems.

There are many ways how to extend and improve the existing project. It is possible to introduce more effective algorithms into the system. The whole system could be rewritten to a system used for batch processing large inputs within the frame of large experiments. An interesting possibility is to use genetic algorithms together with new learning protocols for automated design of restarting automata based on examples.

We hope that this thesis will give an impulse to broaden the notion of restarting automata in the community of linguistic researchers and that some new interesting projects will reuse the components of this project.

Bibliography

[1] R. Barták: *Presentations and lectures* from: *http://kti.ms.mff.cuni.cz/∼bartak/automaty/prednaska.html*, 2007.

[2] J. E. Hopcroft, R. Motwani, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2006.

[3] P. Jančar, F. Mráz, M. Plátek, J. Vogel: *On Monotonic Automata with a Restart Operation.* Journal of Automata, Languages and Combinatorics, 1999, 4(4):287–311.

[4] F. Mráz, F. Otto, M. Plátek: *Learning Analysis by Reduction from Positive Data.* In: Y. Sakakibara, S. Kobayashi, K. Sato, T. Nishino, E. Tomita (Eds.): Proceedings ICGI 2006, LNCS 4201, Springer, Berlin, 2006, 125–136.

[5] G. Niemann and F. Otto: *On the power of RRWW-automata.* In: M. Ito, G. Paun, and S. Yu, eds., Words, Semigroups, and Transductions. World Scientific, Singapore, 341–355, 2001.

[6] R. G. Parekh and V. G. Honavar: *Learning dfa from simple examples.* In: Proceedings of the Eighth International Workshop on Algorithmic Learning Theory (ALT'97), Lecture Notes in Artificial Intelligence 1316, pages 116–131, Sendai, Japan, 1997. Springer.

[7] Ron Rivest, David Baggett: *Machine Learning*, Lecture 23: December 5, 1994 (1994), University of Wollongong.

[8] Sinan Si Alhir: *Learning UML*, O'Reilly Media, Inc., 2003.

[9] Jay Hilyard, Stephen Teilhet: *C♯ Cookbook*, 2nd Edition, O'Reilly Media, Inc., 2006.

[10] Jesse Liberty: *Programming C♯*, 4th Edition, O'Reilly Media, Inc., 2005.

[11] Herbert Schildt: *C♯ 2.0: The Complete Reference*, 2nd Edition, O'Reilly Media, Inc., 2006.

[12] V. Subramaniam: *.NET Gotchas.*, O'Reilly Media, Inc., 2005.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison Wesley Longman, Inc., 1994.

[14] Articles and presentations from: http://agiledeveloper.com/download.aspx.

Other resources

I have used articles and presentations from the following web-sites:

http://agiledeveloper.com/download.aspx
http://www.c-sharpcorner.com/
http://www.codeproject.com/
http://www.developerfusion.co.uk/
http://msdn.microsoft.com/en-us/default.aspx
http://www.mono-project.com/

DragAndDropListView component (made by Matt Hawley) from:
http://www.codeproject.com/KB/list/DragAndDropListView.aspx

Used software

Corel Graphics Suite 11
Microsoft Office Visio 2007
Microsoft Visual Studio 2005 Team Suite
TeXnicCenter 1 Beta 7.50 with MikTeX 2.7

Register