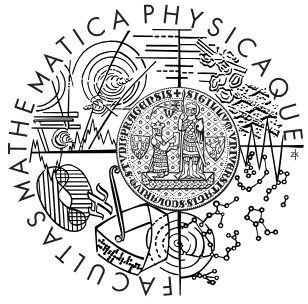
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Martin Popel

Animace algoritmů z teorie automatů

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Rudolf Kryl
Studijní program: Informatika – obecná informatika

2007

Chtěl bych poděkovat vedoucímu své bakalářské práce RNDr. Rudolfu Krylovi a to především za trpělivost.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 30. 7. 2007 Martin Popel

Obsah

Název práce: Animace algoritmů z teorie automatů
Autor: Martin Popel
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Rudolf Kryl
e-mail vedoucího: kryl@ksvi.mff.cuni.cz

Abstrakt: Tato bakalářská práce popisuje softwarový projekt, který umožňuje animovat algoritmy z teorie automatů. Výsledné animace jsou interaktivní a lze je použít jako studijní pomůcku. Automaty mohou mít více grafických reprezentací (tzv. *náhledů*) a uživatel si může zobrazit jen ty, které potřebuje. V projektu jsou implementovány náhledy: tabulka přechodů, seznam přechodů a graf (stavový diagram). Automaty lze upravovat v grafickém editoru, zadávat jim znaky na vstupní pásku a sledovat pak jejich výpočet. Je navržen mechanismus pro simulaci výpočtu libovolných automatů včetně nedeterministických. Algoritmy je možné krokovat v různých úrovních detailu, přičemž se zobrazují vysvětlující komentáře k prováděnému kroku. Projekt obsahuje několik ukázkových algoritmů a lze snadno doplňovat další. Projekt je také možné rozšiřovat o nové náhledy i o další druhy automatů. K tomuto účelu je navržena sada rozhraní, aby již naprogramované algoritmy a náhledy fungovaly i pro nově přidané automaty. Je také připraveno několik pomocných objektů (jako stav, množina stavů, zásobník, či základ obecné přechodové funkce), které usnadní vytváření nových automatů.

Klíčová slova: animace, automaty, algoritmy, konečný automat

Title: Animation of Algorithms from Automata theory
Author: Martin Popel
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Rudolf Kryl
Supervisor's e-mail address: kryl@ksvi.mff.cuni.cz

Abstract: This bachelor's thesis describes the software which animates algorithms from Automata theory. The animations are interactive and can be used for educational purposes. Automata have more graphical representations (*views*) and the user can choose just those he needs. Implemented views are: state transition table, list of transitions, and state diagram. Automata can be edited in a graphical editor, symbols can be inserted on the tape, and then the automaton computation can be watched. All kinds of automata including nondeterministic can be simulated. The user can step through the execution of algorithms in different levels of detail. Comments for every step are shown in a hierarchy tree. The project includes several algorithms, other algorithms can be easily added. It is also easy to add new views and new kinds of automata. A suit of interfaces and utility objects (state, set of states, stack, universal transition function etc.) is prepared to facilitate such additions.

Keywords: animation, automata, algorithms, finite state machine

1 Úvod

1.1 Stručný popis

Softwarový projekt (program), který je součástí této bakalářské práce, umožňuje animaci některých druhů automatů a s nimi souvisejících algoritmů. V následujícím přehledu se pokusím stručně vystihnout jeho základní charakteristiky.

- **Modularita a rozšiřitelnost**
  Základem projektu je grafické prostředí pro animaci algoritmů, nazvané *AnimAlg*. Ostatní části jsou do projektu vloženy ve formě *modulů*, které se načítají až za běhu. Uživatelé mohou do prostředí AnimAlg přidávat nové moduly a tím si ho přizpůsobit svým potřebám. Celý projekt včetně modulů je naprogramován v jazyce Java (verze 5.0).
  Samotné prostředí AnimAlg je navrženo obecněji, aby v něm bylo možné animovat objekty a algoritmy i z jiných oblastí než z teorie automatů. Všechny dosud implementované moduly jsou ovšem zaměřeny právě na teorii automatů.

- **Různé druhy automatů**
  Projekt má propracované zázemí pro široké spektrum automatů. Zázemím se zde myslí celkový návrh, sada rozhraní (interfaces), základní implementace (abstract classes) a pomocné objekty (stav, množina stavů, zásobník, základ obecné přechodové funkce a další).
  Pro ověření funkčnosti jsou v projektu implementovány *konečné automaty* deterministické i nedeterministické a *Turingovy stroje*[1]. Ostatní druhy automatů lze snadno přidat ve formě modulů.

- **Různé náhledy na automaty**
  Moduly obsahující automaty se starají pouze o vnitřní logiku. O vykreslování na obrazovku a interakce s uživatelem se starají jiné moduly, nazývané *náhledy*. Jeden automat může mít více náhledů zobrazených současně. Některé náhledy umožňují uživateli automat upravovat; změny se ihned zobrazují ve všech náhledech.
  V projektu jsou implementovány mimo jiné náhledy: tabulka přechodů, seznam přechodů a graf (stavový diagram). V současné verzi je graf dostupný pouze pro konečné automaty. Není však problém jej rozšířit i pro ostatní druhy automatů, pokud by se zajistilo přehledné zobrazení více přechodů mezi dvěma stavy.

- **Simulace výpočtu automatu**
  Je navržen mechanismus pro simulaci (krokování) výpočtu libovolných automatů včetně nedeterministických. Tato simulace je implementována pomocí speciálního náhledu, který nezobrazuje automat, ale vstupní pásku a další ovládací prvky. Uživatel se může při simulaci výpočtu automatu vracet k již provedeným krokům (tlačítkem *Zpět*).

[1] Formální definice automatů včetně používané terminologie jsou uvedeny v příloze 8.1

- **Jednoduché psaní algoritmů**
  Ve formě modulů se přidávají i algoritmy. Uživatelé tedy napíší zvolený algoritmus v Javě, zkompilují a v nastavení AnimAlgu zadají cestu k adresáři s výsledným `class` souborem. Všechny nalezené algoritmy se pak automaticky načtou do AnimAlgu a zobrazí se v menu.
  Algoritmy z teorie automatů jsou nejčastěji nějaké konstrukce na automatech, tedy převod jednoho nebo více automatů na jiný automat. Často se tyto algoritmy používají pro konstruktivní důkaz nějaké věty. Psaní algoritmů usnadňuje mnoho (již naprogramovaných) metod automatů a dalších objektů.
  Projekt obsahuje několik ukázkových algoritmů, například minimalizaci konečného automatu, převod nedeterministického konečného automatu na deterministický, algoritmy pro důkaz uzavřenosti regulárních jazyků na množinové operace či konstrukci Turingova stroje, který obrací slovo na pásce.

- **Krokování algoritmů**
  Autor algoritmu do kódu explicitně zapisuje popis *událostí*, tedy prováděných kroků. Algoritmus pak jde krokovat po těchto krocích určených autorem algoritmu. Uživatel, který algoritmus spouští, se ale může při krokování rozhodnout některé kroky přeskočit (přesněji řečeno provést naráz, bez zastavování).
  Při krokování náhledy ihned reagují na změny, které algoritmus provedl (např. přidání stavu či přechodu). Uživateli se také zobrazují popisy všech událostí (čili vysvětlující komentáře ke krokům algoritmu) v přehledném hierarchickém zobrazení, které je nazváno *strom událostí* (viz obrázek 6, str. 10).

- **Použitelnost**
  Výsledné animace jsou interaktivní a názorné, lze je tedy použít jako studijní pomůcku. Uživatel má mnoho možností, jak experimentovat s animovanými automaty a algoritmy, a tak lépe porozumět vysvětlované látce. AnimAlg mohou využívat i vyučující pro prezentace na přednáškách a další účely[2]. AnimAlg je možné spustit i jako applet na webových stránkách.
  Podrobná programátorská dokumentace je k dispozici ve formátu JavaDoc. Uživatelská a programátorská příručka jsou součástí této práce (kapitoly 6 a 7).

[2]Vyučující může přidat do AnimAlgu probírané algoritmy (případně jen upravit komentáře existujících algoritmů) a poskytnout pak tyto moduly studentům jako doplněk ke skriptům. AnimAlg lze využít i na cvičeních.

1.2 Představení projektu

Dříve než začnu popisovat jádro projektu, tedy návrh automatů a zvolenou koncepci animování algoritmů, rámcově představím základní schopnosti, které projekt nabízí uživatelům.

Práce s automaty

Uživatel si v prostředí AnimAlg do samostatných oken otevře automaty (případně i jiné objekty), se kterými chce pracovat. Automaty lze načíst ze souboru (ve formátu XML), získat je jako výstup nějakého algoritmu nebo lze vytvořit prázdný automat a ten pak upravit v nějakém náhledu. Všechny dostupné náhledy jednoho automatu se zobrazují jako panely v jednom okně (viz obrázek 1), jde je však oddělit do samostatných oken, aby bylo vidět více náhledů současně (viz obrázek 2).

Náhledy

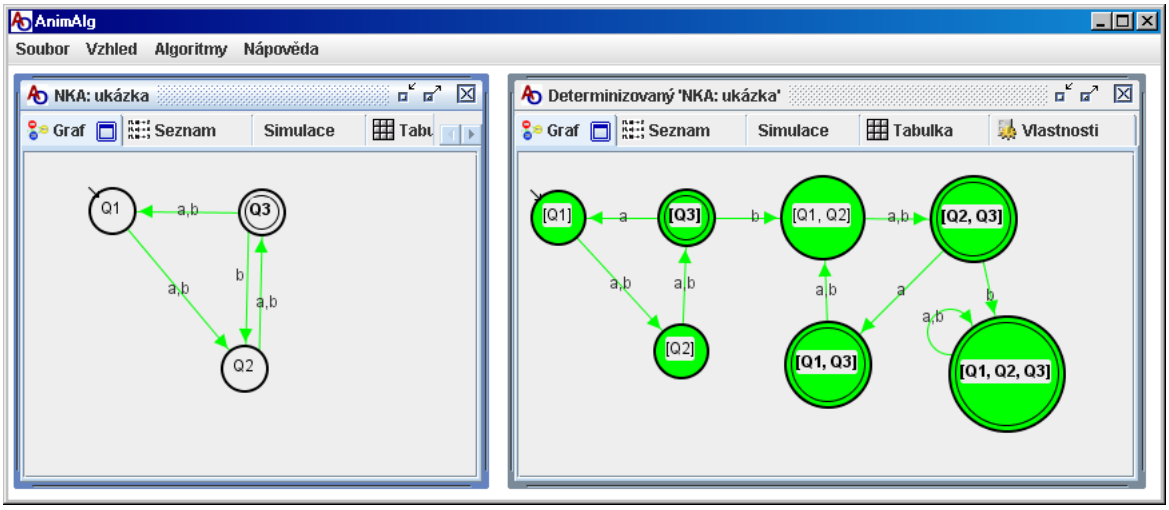
Pro všechny druhy automaty jsou k dispozici jsou následující náhledy:

- **Vlastnosti**
  Zobrazuje typ, název, popis a vstupní abecedu automatu. Krom typu automatu lze vše editovat.
- **Tabulka**
  Zobrazuje přechodovou funkci jako tabulku (stavy = řádky, znaky = sloupce). Názvy koncových stavů jsou tučně, před počátečními se zobrazuje šipka, za aktuálním černá tečka. Při provedení přechodu příslušná buňka zabliká. Pokud automat připouští více přechodů z jednoho stavu přes jeden znak, zobrazí se v buňce všechny tyto přechody oddělené čárkami.
- **Seznam**
  Zobrazuje přechodovou funkci jako seznam přechodů. Přechodová funkce lze v tomto náhledu editovat.
- **Simulace**
  Simuluje výpočet automatu pomocí tlačítek *Krok* (provede další krok výpočtu) a *Zpět* (vrátí výpočet o jeden krok zpět). Zobrazuje se informace o tom, zda automat přijímá (v podobě žárovky) a vstupní páska včetně pozice čtecí hlavy. Při výpočtu nedeterministických automatů se navíc zobrazuje seznam možných přechodů pro příští krok, ze kterého uživatel vybírá ten, který se má provést. Uživatel může zadávat znaky na vstupní pásku pomocí tlačítek nebo může načíst celou pásku ze souboru.

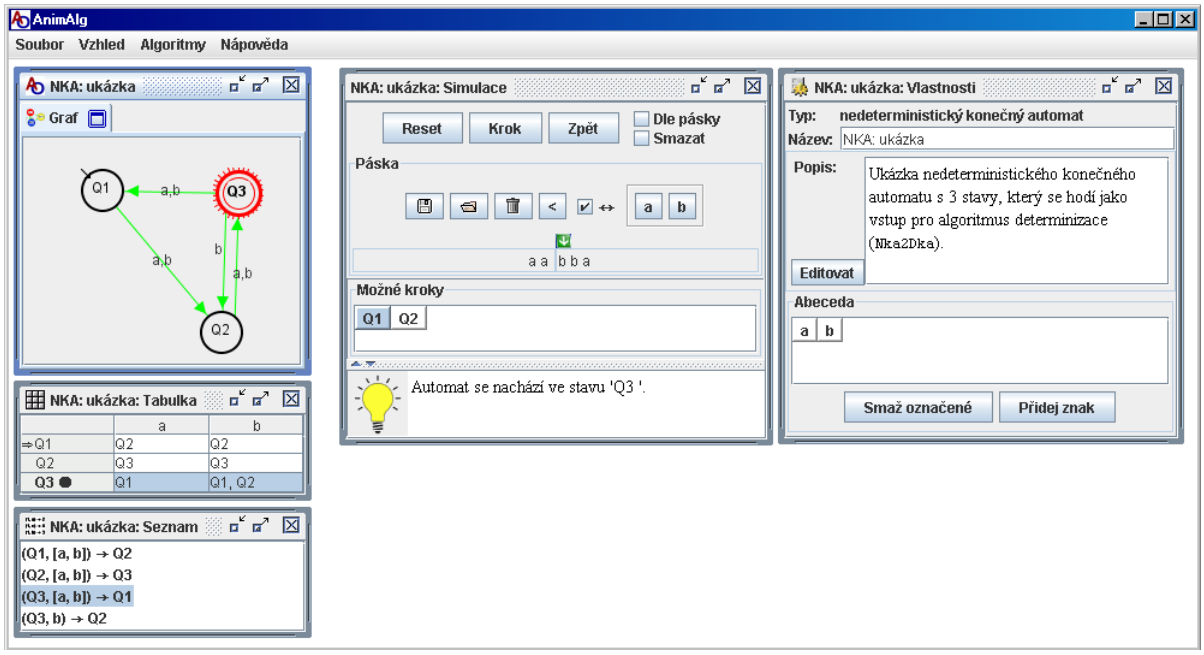
Pro konečné automaty je k dispozici navíc náhled

- **Graf**
  Zobrazuje automat jako graf (stavy = vrcholy, přechody = orientované hrany), neboli stavový diagram. Uživatel může v tomto náhledu vyrobit libovolný konečný automat. Stavy jde přesouvat a měnit jejich název, popis, velikost i barvu. Při provedení přechodu je příslušná hrana mezi dvěma stavy animována (zvýrazní se červeně).

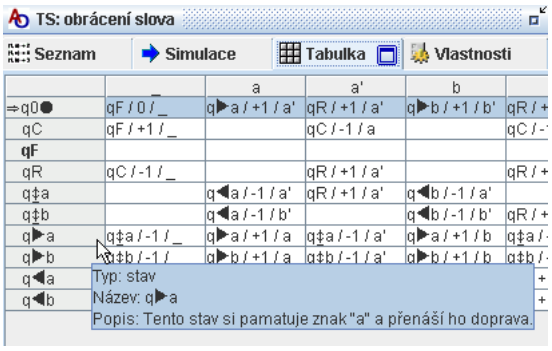
Všech pět popsaných náhledů je vidět na obrázku 2.



Obrázek 1: Na ploše otevřeny 2 automaty, každý má všechny náhledy v jednom okně.



Obrázek 2: Na ploše otevřen jeden automat s náhledy v samostatných oknech.



Obrázek 3: Ukázka zobrazení informací o stavu Turingova stroje

Spouštění algoritmů

V menu je seznam všech algoritmů, které byly načteny ve formě modulů. Před spuštěním algoritmu se zobrazí dialogové okno *Zadání vstupních parametrů* (viz obrázek 4), ve kterém se zobrazuje i podrobný popis algoritmu. Vstupním parametrem algoritmu může být například číslo, řetězec, výběr ze seznamu možností či automat. Jsou-li již na ploše AnimAlgu otevřeny nějaké automaty, které by se daly použít jako vstupní parametr, zobrazí se v nabídce.

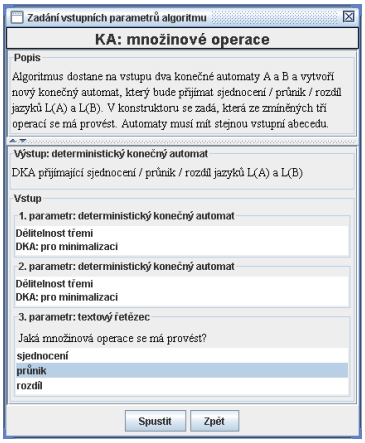
Po zadání vstupních parametrů se zobrazí okno *Krokování algoritmu* (viz obrázek 5), ve kterém jsou tlačítka *Step into*, *Step over* a *Step out*, jimiž se ovládá hloubka zanoření podobně jako v různých ladicích programech (debuggerech). Uživatel může například sledovat jen „hlavní kroky“ algoritmu a ty „detailnější“ přeskakovat.

Kromě podrobného komentáře k aktuálnímu kroku se v okně *Krokování algoritmu* zobrazuje i tzv. *strom událostí* s hierarchickým zobrazením již provedených kroků. Uživatel může jednotlivé uzly ve stromě událostí (tedy jednotlivé události) „rozbalit“ či „sbalit“ a tak dosáhnout toho, aby byl strom přehledný a zároveň zobrazoval ty události, které uživatele zajímají. Pokud se některý krok přeskočí pomocí *Step over* či *Step out*, tak se příslušná událost vykreslí nerozbalená. Uživatel si ji však může dodatečně rozbalit a prohlédnout si komentáře i k přeskočeným krokům.

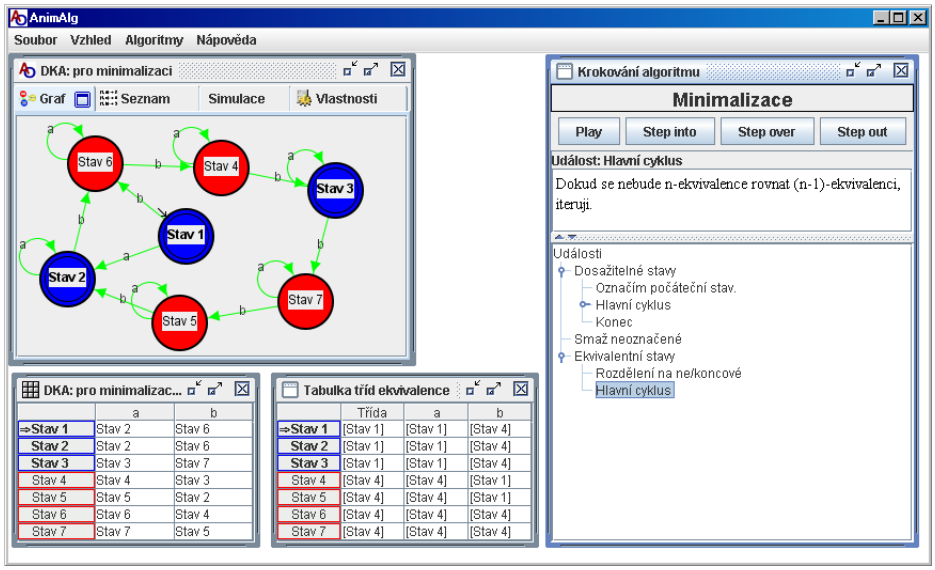
Pokud chce uživatel přehrát celý výpočet algoritmu jako animaci v užším slova smyslu, stiskne tlačítko *Play* a nastaví rychlost animace. Výsledek je pak stejný, jako kdyby uživatel stále mačkal tlačítko *Step into*.

Komentované objekty

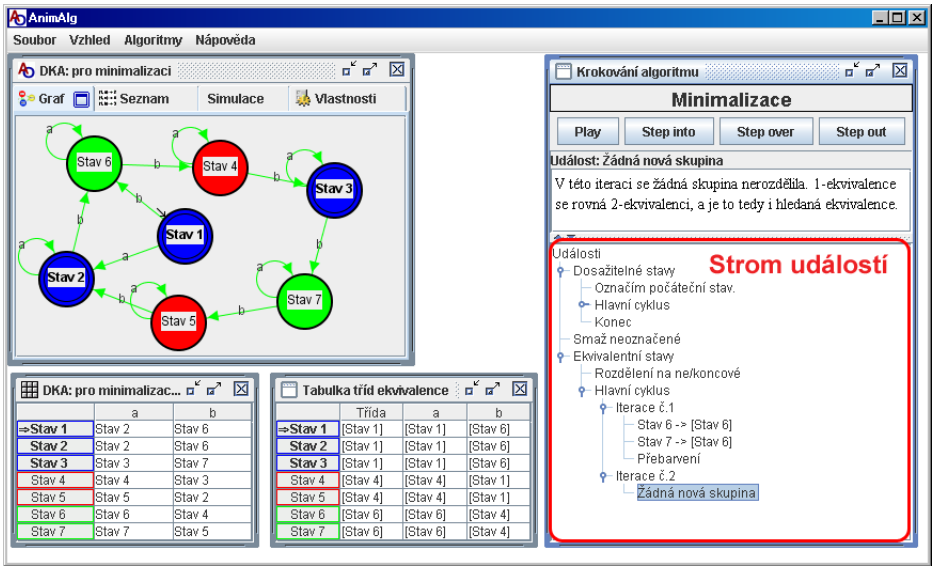
Animace v AnimAlgu by měly být názorné, a proto je mnoho objektů tzv. *komentovaných*. To znamená, že mají přiřazen typ, název a popis (všechny tři atributy jsou textové řetězce). Komentovanými objekty jsou mimo jiné stavy automatů i celé automaty, znaky, algoritmy a události algoritmu. Typ, název a popis se mohou zobrazovat v náhledech různým způsobem, popis se často zobrazuje při najetí myši nad daný objekt jako tooltip (viz obrázek 3). Popis může být libovolně dlouhý a lze v něm používat HTML značky (tedy i indexy, různé typy písma, barvy,...).



Obrázek 4: Okno Zadání vstupních parametrů



Obrázek 5: Ukázka animace algoritmu pro minimalizaci automatu



Obrázek 6: Ukázka téhož algoritmu jako na obrázku 5, ale o několik kroků dál a s vyznačeným stromem událostí

Vytváření algoritmů

Uživatelé mohou programovat nové algoritmy, které půjde animovat v prostředí AnimAlg. Mohou přitom využívat metod již naprogramovaných automatů a pomocných objektů. Podrobněji se algoritmům věnuji v kapitole 3. Zde pouze pro představu uvedu dvě krátké ukázky kódu. První ukázka představuje triviální algoritmus, který smaže z automatu všechny neoznačené stavy:

```
@Nazev ("Smaž neoznačené")
@Popis ("Smaže z automatu všechny <i>neoznačené</i> stavy.")
public class SmazNeoznacene extends AbstractAlgoritmus{
  private I_Automat automat;

  public SmazNeoznacene(I_Automat a){
    automat = a;
  }

  public void vypocet() {
    Stav[] stavy = automat.getMnozinaStavu().getPoleStavu();
    for (Stav stav : stavy){
      if (! stav.oznacen()){
        je("Mažu stav " + stav);
        automat.getMnozinaStavu().remove(stav);
      }
    }
  }
}
```

Každý algoritmus je implementován v samostatné třídě, v konstruktoru dostane všechny potřebné vstupní parametry a v metodě `vypocet` se provede vlastní výpočet. Název a popis algoritmu, které se mají zobrazit uživateli, se zapisují do kódu pomocí anotací. Za povšimnutí stojí volání metody `je`, pomocí které se předává komentář k aktuálnímu kroku. V místech volání této metody půjde algoritmus pozastavit.

Z algoritmu lze také volat jiné algoritmy jako podprogramy. Velmi snadno tak lze vytvořit například algoritmus pro minimalizaci konečného automatu (uvádím už jen metodu `vypocet`):

```
public void vypocet() {
  spust(new OznacDosazitelne(automat));
  spust(new SmazNeoznacene(automat));
  spust(new EkvivalentniStavy(automat));
  spust(new PodilovyAutomat(automat));
  spust(new Normalizace(automat));
}
```

Při spuštění jiného algoritmu pomocí metody `spust` se uživateli zobrazí jako komentář popis daného algoritmu. Zároveň se při vstupu do podprogramu zvýší úroveň zanoření, takže si uživatel bude moci zvolit, zda krokovat daný podprogram, či ho provést celý najednou.

2 Automaty

Teorie automatů má využití v mnoha oblastech a existuje i více přístupů, jak automaty definovat, pojímat, značit, na co dávat důraz apod. Nicméně jádro je stejné: konečný popis nekonečných objektů, abstraktní výpočetní model, Chomského hierarchie. Snažil jsem se, aby i můj projekt byl využitelný ve více oblastech, aby si ho mohli uživatelé přizpůsobovat a zároveň, aby už základní verze byla použitelná pro řadu úloh. Při návrhu a implementaci automatů pro AnimAlg jsem si vytyčil několik požadavků a dospěl k několika rozhodnutím:

Konzistence s teorií

V projektu jsem se snažil držet používané (vyučované) teorie a značení. Pouze výjimečně jsem se odchýlil či zavedl nový pojem, a to vždy s úmyslem, aby se program lépe používal. Například při vytváření konečného automatu se krom obvyklých parametrů (množina koncových stavů, přechodová funkce,...) zadává i název, popis a čtecí hlava, která zprostředkuje automatu přístup k vstupní pásce. Značení jsem volil především podle zdrojů [1] a [2], odkud jsem čerpal i některé ukázkové automaty a komentáře k implementovaným algoritmům. Formální definice používaných automatů včetně zvolené terminologie jsou uvedeny v příloze 8.1.

Při návrhu automatů pro AnimAlg jsem ale musel řešit mnoho otázek, které se v teorii obvykle neřeší. Například pokud mají dva automaty stejnou množinu stavů a do jednoho z automatů je přidán nový stav, má se tento stav objevit i v druhém automatu[1]?

Oddělení náhledů od modelů

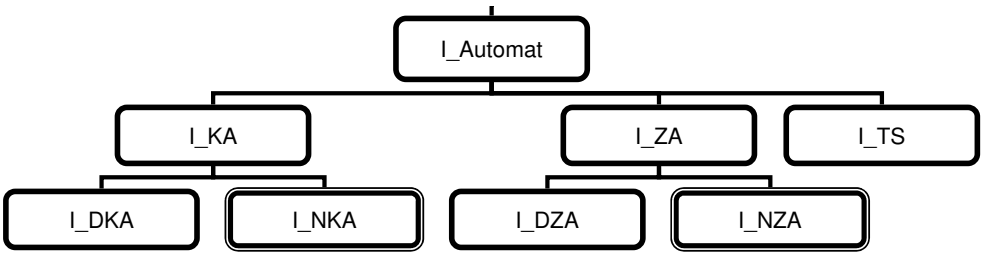
Při návrhu programu jsem se rozhodl striktně oddělit grafické reprezentace, čili náhledy od vnitřní logiky automatů. Částečně jsem se inspiroval návrhovým vzorem Model-view-controller [3], a proto někdy označuji objekty zajišťující vnitřní logiku jako *modely*. Modely nemají přímý odkaz na grafické reprezentace (tedy na část *view*). Grafické reprezentace si registrují u svého modelu tzv. *listener*, pomocí kterého získávají informace o změnách modelu a reagují na ně. Pokud například nějaký algoritmus přidá do automatu nový stav, je všem náhledům (které si registrovaly listener příslušného typu) tohoto automatu oznámena událost `pridanPrvek(Stav)`.

[1] Otázky podobného typu jsou rozebírány později v této kapitole, ale odpověď na tuto otázku uvedu rovnou. Algoritmy, které převádí jeden automat na druhý, mohou být v AnimAlgu implementovány „in situ“ (mění automat, který dostaly na vstupu), nebo tak, že nejdřív vytvoří kopii vstupního automatu a upravují až tu. V druhém případě si tvůrci algoritmů pro AnimAlg mohou vybrat, jak hlubokou kopii vytvoří. Pokud přiřadí množině stavů nového automatu stejný objekt, jako byla původní množina stavů, tak se pozdější přidání stavu do jednoho automatu projeví i v druhém. Implementované ukázkové algoritmy ale vytvářejí vždy hlubokou kopii, takže je nový automat na původním nezávislý.

Hierarchie automatů

Jedním z cílů projektu je, aby šel snadno doplňovat o nové algoritmy, náhledy i nové druhy automatů. Proto jsem navrhl sadu rozhraní pro základní druhy automatů. Názvy těchto rozhraní začínají pro odlišení od implementací písmenem „`I`“ (*interface*). Základní, nejvyšší rozhraní pro všechny automaty je nazváno `I_Automat` a obsahuje schopnosti (tedy metody) obecného automatu. Ostatní názvy jsou odvozeny od zkratek, které používám v celém projektu: KA = konečný automat, ZA = zásobníkový automat, TS = Turingův stroj, D = deterministický, N = nedeterministický.

Rozhraní jsou hierarchicky uspořádána tak, že nové schopnosti jsou přidávány pomocí podrozhraní (*subinterfaces*). Celá hierarchie je znázorněna na obrázku 7. Dvojitým orámováním jsou vyznačena rozhraní nedeterministických automatů. Všechna taková rozhraní nedeterministických automatů rozšiřují rozhraní `I_Nedeterministicky` (jsou jeho podrozhraními), které není na obrázku zakresleno.



Obrázek 7: Hierarchie rozhraní pro základní druhy automatů

V teorii automatů se obvykle označení *konečný automat* a *deterministický konečný automat* považuje za synonymní. (Obdobně je synonymní označení *zásobníkový automat* a *nedeterministický zásobníkový automat* – přesněji řečeno zásobníkový automat je z definice nedeterministický.) Aby nedocházelo k nedorozuměním, rozhodl jsem se v názvech i zkratkách vždy uvádět, zda je automat deterministický, či nikoli. Některé algoritmy mohou pracovat s konečnými automaty nedeterministickými i deterministickými[2]. Proto jsem zavedl rozhraní `I_KA` pro obecný konečný automat, který může, ale nemusí být deterministický, a obdobně `I_ZA` pro obecný zásobníkový automat.

Při některých algoritmech je výhodné pracovat s tzv. zobecněnými NKA (ZNKA), které mohou obsahovat *lambda-přechody*, tj. přechody, při kterých se nečte z pásky žádný znak. Nechtěl jsem hierarchii rozhraní vytvářet zbytečně složitou, a proto neexistuje rozhraní `I_ZNKA`, ale existuje metoda `boolean I_NKA.jeZobecneny()`.

DKA dle striktní definice musí mít přechodovou funkci *úplnou*, tedy $\forall q \in Q, \forall x \in X$ $\exists y \in X$, že $\delta(q, x) = y$. Často je ale výhodné, když přechodová funkce nemusí být úplná. Místo chybějících přechodů si lze představit přechody do nového, nekoncového „garbage“ stavu. Tento stav a přechody lze explicitně doplnit (algoritmem `DoplnNaUplny`), ale z více důvodů je výhodné připustit i DKA s neúplnou přechodovou funkcí. Opět jsem místo dalšího rozhraní přidal metodu `boolean I_DKA.jeUplny()`.

Návrh hierarchie počítá s rozšířením např. o dvoucestné KA, Mooreův a Mealy stroj či nedeterministický Turingův stroj (NTS).

[2]Podle teorie je DKA speciálním případem NKA, tedy by se mohlo zdát, že všechny algoritmy, které pracující s NKA, mohou pracovat i s DKA. Jenže objekt `DKA` například nesmí mít více počátečních stavů. Existuje však triviální algoritmus (v projektu implementován jako `Dka2Nka`), který podle zadaného DKA vytvoří nový NKA.

Obdobně jako hierarchie rozhraní automatů vypadá hierarchie objektů implementujících tato rozhraní. Společný předek všech implementovaných automatů je abstraktní třída `AbstractAutomat`, která má potomky `DKA`, `NKA` a `TS`. Objekt `ZA` v současné verzi implementován není, ale lze ho doplnit, stejně jako lze s využitím `AbstractAutomatu` snadno doplnit dvoucestné KA, Mooreův a Mealyho stroj i NTS.

Obecnost a použitelnost

V mém původním návrhu obsahovalo rozhraní `I_Automat` pouze pár metod a `AbstractAutomat` ještě méně. Tento návrh byl sice dostatečně obecný, aby uživatelé mohli přidávat i různé „exotické“ druhy automatů, ale to byla výhoda jediná a navíc diskutabilní. Přidávání nových druhů automatů (tedy nových rozhraní) i nových implementací automatů bylo poměrně pracné. Velmi špatně se přidávaly i nové náhledy a algoritmy. Například původní verze rozhraní `I_Automat` neposkytovala přístup k přechodové funkci automatu. Nebylo tedy možné vytvořit náhled „Seznam“ či „Tabulka“ pro obecný automat, ale musel se vytvářet náhled pro každý druh automatu zvlášť.

Několikrát jsem přepracoval celou koncepci a dospěl k výsledku, kde nejvíce práce spočívá na `AbstractAutomatu` a objektech, které využívá. Potomci (`DKA`, `NKA` a `TS`) předefinují pouze několik metod. Obdobně i `I_Automat` obsahuje většinu metod a podrozhraní přidávají jen metody, které by pro jiné druhy automatů neměly smysl. Vytváření nových automatů (rozhraní i implementací), náhledů a algoritmů je teď mnohem snazší. Nová koncepce je přitom, doufám, stále dostatečně obecná.

2.1 Rozhraní `I_Automat`

V této kapitole popíši rozhraní `I_Automat` pro obecné automaty a současně i nastíním implementaci třídy `AbstractAutomat`. Pro představu nejdřív uvedu zkrácený kód:

```
public interface I_Automat extends I_AnimAlgObjekt {
  public I_PFce getPFce();
  public I_MnozinaZnaku getAbeceda();
  public I_MnozinaStavu getMnozinaStavu();
  public I_PodmnozinaStavu getMnozinaKoncovych();
  public I_PodmnozinaStavu getMnozinaPocatecnich();
  public Stav getAktualniStav();
  public I_CteciHlava getCteciHlava();

  public boolean krok();
  public boolean jeDalsiKrok();
  public boolean prijima();
  public Action getKrokAction();

  public boolean zpet();
  public boolean jeZpet();
  public Action getZpetAction();
  public boolean reset();

  public String getPopisKonfigurace();
  public I_Automat clone();
}
```

Součásti automatu

Podle teorie je automat definován jako uspořádaná n-tice, jejíž položky jsou různé matematické struktury (např. DKA $= (Q, X, \delta, q_0, F)$, viz 8.1). Automaty implementované v AnimAlgu toto přebírají a jsou zapouzdřením objektů jako přechodová funkce, množina stavů, množina koncových stavů apod. Automaty využívají i objekty, které nejsou součástí n-tice v definici, ale jsou potřeba k fungování automatu, například zásobník či vstupní páska.

Součásti obecného automatu jsou:

- **Přechodová funkce**
  Přechodová funkce je asi nejdůležitější součástí automatu. Některé algoritmy i náhledy dokonce využívají z automatu pouze přechodovou funkci. Obecnou přechodovou funkci definuje rozhraní `I_PFce`, které má několik podrozhraní (`I_PFceDKA` atd.), jež tvoří podobnou hierarchii jako rozhraní automatů. Podrobněji se koncepci přechodových funkcí se věnuje kapitola 2.2.

- **Abeceda**
  Abeceda je množina znaků, které se mohou nacházet na vstupní pásce automatu. Nad abecedou je definována přechodová funkce. Přechodové funkce některých automatů (ZNKA, ZA) ale mohou obsahovat i speciální znak λ reprezentující prázdné slovo. Tento znak se do abecedy nezahrnuje. TS zase pracuje i s prázdným znakem (někdy označovaným též jako *blank* či ϵ, v AnimAlgu je označován podtržítkem). Prázdný znak se dle konvence do abecedy zahrnuje, přestože ho vlastní (přijímané) slovo nesmí obsahovat.

- **Množina stavů**
  Všechny automaty mají množinu stavů, která ale může být i jednoprvková či prázdná. Prázdná množina stavů je povolena proto, aby šlo automaty vytvářet *inkrementálně*, tedy nejdříve vytvořit automat prázdný a pak do něj postupně přidávat stavy a přechody. Pro různé účely se hodí speciální stav `NEDEF`, který značí, že stav automatu není definován. Tento stav se do množiny stavů nikdy nezahrnuje.

- **Množina koncových stavů**
  Všechny automaty mají i množinu koncových stavů. Některé automaty ale koncové stavy nevyužívají a metoda `getMnozinaKoncovych` vrátí vždy prázdnou množinu – například ZA přijímající prázdným zásobníkem nebo Mooreův stroj, jehož výstupní funkce může množinu koncových stavů nahradit. Objekt představující množinu koncových stavů musí implementovat rozhraní `I_Podmnozina-Stavu`, které má navíc od rozhraní `I_MnozinaStavu` metodu `getNadmnozina` a požaduje, aby prvky podmnožiny byly vždy i prvkem nadmnožiny.

- **Množina počátečních stavů**
  Většina automatů má jeden počáteční stav. NKA může mít počátečních stavů více. Aby se dalo i k počátečním stavům přistupovat jednotně (a stejně jako ke stavům koncovým), rozhodl jsem se do rozhraní `I_Automat` přidat metodu `getMnozinaPocatecnich`. Ostatní automaty krom NKA vrací vždy množinu s nejvýše jedním prvkem a pro snadné používání mají i metodu `getPocatek`. Dále jsem vytvořil třídu `PodmnozinaStavu1Prvek`, která implementuje rozhraní `I_Podmno-zinaStavu` a navíc zaručuje, že nebude obsahovat více než jeden prvek (přidáním

dalšího prvku se předchozí odebere). Při vytváření implementací automatů tedy není požadavek množiny počátečních stavů žádnou zátěží navíc.

- **Aktuální stav**
  Aktuální stav by měl být jeden z množiny stavů až na jednu výjimku: Pokud z nějakého důvodu není aktuální stav automatu definován, vrátí metoda `getAktualniStav` již zmíněný speciální stav `NEDEF`.

- **Čtecí hlava**
  Každý automat (ve smyslu akceptoru) musí získávat nějakým způsobem vstupní data. Obvykle se tento způsob znázorňuje pomocí čtecí hlavy a pásky. Rozhraní `I_CteciHlava` obsahuje především metody `nactiZnak`, která vrátí znak na aktuální pozici pásky (tedy ten „pod“ hlavou), a `posunHlavu(int)`, která posune hlavu o zadaný počet políček (obvykle se používá jen -1, 0 a +1). TS musí umět na pásku i zapisovat, a proto jsem vytvořil `I_ZapisovaciHlava`, což je podrozhraní rozhraní `I_CteciHlava`. Pokud má automat více čtecích (či zapisovacích) hlav, vrátí metoda `getCteciHlava` tu, která bude zobrazena v náhledech připravených pro automaty s jednou čtecí hlavou (a jednou páskou). Pro ostatní hlavy (a ostatní pásky) by měl mít takový automat definovány jiné přístupové metody.

Výpočet automatu

Z rozhraní `I_Automat` bylo zatím popsáno prvních 7 metod, které zpřístupňují objekty zapouzdřené v automatu. Automaty musí mít ale i metody pro umožnění výpočtu. U každého druhu automatu ovšem výpočet probíhá trochu jinak. Aby bylo možné simulovat (v náhledu „Simulace“) výpočet libovolného automatu, přidal jsem do rozhraní `I_Automat` metodu `krok()`, která provede další krok výpočtu. Co v této metodě dělají jednotlivé druhy automatů? Nejdříve popíšu kroky deterministických automatů:

- **DKA** jen přečte z pásky znak, posune hlavu o +1 a podle své přechodové funkce přejde do příslušného stavu.

- **DZA** navíc čte i symbol z vrcholu zásobníku (který vždy odebere) a poté na zásobník ukládá slovo (zásobníkových symbolů).

- **TS** po přečtení znaku z pásky zapíše na pásku nový znak a posune hlavu (o -1, 0 či +1).

Nedeterministické automaty mohou mít v každém kroku na výběr více možných přechodů, ale provést mohou jen jeden. Navrhl jsem tedy rozhraní pro objekt, jenž určuje, který z možných přechodů se má provést. Tento objekt jsem nazval *orákulum* a rozhraní `I_Orakulum`. Nedeterministické automaty mají orákulum jako svou další součást, tedy rozhraní `I_Nedeterministicky` obsahuje metodu `getOrakulum`. Automat před provedením kroku nabídne orákulu možné přechody, a když je pak zavolána metoda `krok`, tak orákulum vybere jednu z nabízených možností. (Konkrétně metoda `I_Orakulum.vyber()` vrátí číslo možnosti, která se má provést.)

Orákulum může ovládat uživatel či nějaký program (algoritmus). V prvním případě náhled „Simulace“ zobrazuje u nedeterministických automatů grafickou reprezentaci orákula jako seznam možností, ze kterých může uživatel vybírat pomocí myši.

Kroky nedeterministických automatů tedy probíhají podobně jako u jejich deterministických protějšků, pouze se navíc volá orákulum. Aby byl náhled „Simulace“ rozumně použitelný, musí zobrazit možnosti ještě dříve, než uživatel zmáčkne tlačítko *Krok*. Nedeterministické automaty tedy fungují tak, že při provádění jednoho kroku rovnou vypočítají možnosti pro příští krok a oznámí tyto možnosti orákulu. Když je pak znovu zavolána metoda `krok`, tak si automat vyžádá od orákula vybranou možnost a provede ji a opět oznámí možnosti pro příští krok orákulu. Z tohoto principu plyne omezení, že jedno orákulum může ovládat jen jeden automat. Také je vhodné, aby automat orákulu oznamoval seznam možností i tehdy, když se mezi dvěma kroky tento seznam změní (například proto, že uživatel přidal do automatu nový přechod, který vede z aktuálního stavu).

Návratová hodnota metody `krok` určuje, zda se krok podařilo provést. Tato metoda může vrátit `false` z různých důvodů, které záleží na druhu automatu (i na zvoleném pojetí konce výpočtu):

- **DKA** vrací `false`, pokud už byl přečten celý vstup (čili pod čtecí hlavou je na pásce prázdný znak). Tuto definici uplatňuji i pro DKA, které nemají přechodovou funkci úplnou. U těch se totiž může stát, že pro daný stav a znak na pásce není definován žádný přechod. Automat v takovém případě přejde do stavu `NEDEF`, posune čtecí hlavu o +1 a metoda `krok` vrátí `true`. Pokud se už před provedením kroku automat nacházel ve stavu `NEDEF`, tak z definice nemá definovaný žádný přechod dál. I v tomto případě metoda `krok` posune hlavu o +1, zůstane ve stavu `NEDEF` a vrátí `true`. Pouze pokud by už bylo slovo na pásce přečteno, tak se neprovede nic a `krok` vrátí `false`.

  Alternativním řešením by bylo nepřipustit přechod do stavu `NEDEF`. Pokud by nebyl definován přechod, metoda `krok` by vrátila `false`, i když by nebyl přečten celý vstup. Toto řešení jsem zavrhl z toho důvodu, že pokud by se do DKA s neúplnou přechodovou funkcí doplnily (např. algoritmem `DoplnNaUplny`) chybějící přechody, tak by se choval při simulaci jinak než před doplněním. Oba automaty (původní i doplněný) by ale měly být ekvivalentní, protože jsou jen jiným pohledem na totéž.

- **NKA, ZA** a **TS** vrací `false`, pokud není definována žádná *použitelná instrukce* (tedy žádný přechod použitelný v aktuální konfiguraci automatu). U ZA tato situace nastává speciálně tehdy, je-li prázdný zásobník, u TS zase tehdy, je-li automat v koncovém stavu. (TS mohou mít oproti KA a ZA definovány přechody i pro prázdný znak na pásce.)

  Pokud bychom považovali u DKA přechod do stavu `NEDEF` přes neprázdný znak za použitelnou instrukci, tak bychom na ně také mohli použít definici konce výpočtu definovanou v předchozím odstavci (tedy, výpočet je ukončen právě tehdy, když neexistuje žádná použitelná instrukce).

Občas se hodí vědět, zda je výpočet automatu u konce, aniž by se volala metoda `krok`. K tomu slouží metoda `jeDalsiKrok` – pokud vrátí `false`, tak je výpočet u konce. Toho využije například náhled „Simulace“, který učiní tlačítko *Krok* neaktivním (disabled). V původním návrhu vyžadovalo rozhraní `I_Automat` i opačnou implikaci, tedy pokud `jeDalsiKrok` vrátí true, tak výpočet není u konce a následné zavolání metody `krok` musí také vrátit true. Tento požadavek se ukázal jako nepraktický. Uživatel totiž

může automaty i vstupní pásku během simulace upravovat[3], a tím může ovlivnit, zda je automat u konce výpočtu. Některé implementace automatů všechny takovéto změny sledují a mění podle toho hodnotu, kterou pak vrací metoda `jeDalsiKrok`. Vyžadovat toto po všech automatech by ale zkomplikovalo jejich přidávání. Například implementace `TS` vrací metodou `jeDalsiKrok` `true` do té doby, kdy poprvé metoda `krok` vrátí `false`. Při posledním kroku se tedy už nic neprovede.

Aby bylo používání automatů jednodušší, je přidána akce `krokAction`, což je třída implementující standardní rozhraní `javax.swing.Action`. Její spuštění zavolá `krok()` a její metoda `isEnabled` by měla vracet totéž co `jeDalsiKrok`. Díky této akci tlačítko *Krok* reaguje ihned na všechny změny.

Jednou z nejdůležitějších metod rozhraní `I_Automat` je metoda `prijima`, která určuje, zda automatu přijímá slovo na pásce. Některé automaty (ZA, TS) mohou slovo přijmout teprve, až když je výpočet u konce. U KA se ale často používá tzv. sériový výpočet, kdy automat po každém kroku může oznámit, zda přijímá, či ne dosud načtenou část pásky.

Reset automatu a vracení výpočtu zpět

Při simulaci výpočtu automatu je výhodné, může-li se uživatel vracet k předchozím krokům a procházet si tak celý výpočet. K tomu slouží tlačítko *Zpět* v náhledu „Simulace“ a metoda `zpet` v rozhraní `I_Automat`. Jedná se o činnost téměř inverzní k metodě `krok`. Pokud bychom si definovali krok výpočtu algoritmu jako přechod z konfigurace[4] A do konfigurace B, pak následné zavolání metody `zpet` provede vždy přechod z B do A. U deterministických automatů platí i opačné tvrzení: pokud `zpet` provedlo přechod z B do A, pak následný `krok` provede přechod z A do B. Pro nedeterministické automaty toto neplatí, protože orákulum může vybrat jinou možnost, než vybralo při předchozím „průchodu výpočtem“. Tohoto faktu mohou samozřejmě uživatelé při simulaci nedeterministických automatů využívat, aby si vyzkoušeli více výpočtů.

Podobně jako k metodě `krok` existují metody `jeDalsiKrok` a `getKrokAction`, tak k metodě `zpet` existují metody `jeZpet` a `getZpetAction`. Pokud se nelze vrátit zpět nebo automat vůbec tuto funkci neimplementuje (což by mělo být zdokumentováno v jeho popisu), vrátí metoda `zpet` i metoda `jeZpet` `false`.

Další užitečnou funkcí při simulaci je možnost vrátit výpočet na začátek (rovnou, tedy bez nutnosti opakovaného volání `zpet`). K tomu slouží metoda `reset` (a tlačítko

[3]Cílem tohoto rozhodnutí bylo, aby mohl uživatel snadněji s automaty experimentovat, pochopit jejich principy atd. Všechny implementované automaty takovouto editaci „za běhu“ umožňují. Lze ale naprogramovat i implementaci automatu, který podporuje editace pouze s nějakými omezeními, případně je nepodporuje vůbec. Pokud by chtěl automat, respektive jeho tvůrce, zabránit tomu, aby uživatel (či nějaký algoritmus) editoval pásku během výpočtu, může například v prvním kroku výpočtu pásku překopírovat a pak pracovat s kopií.

[4]Konfiguraci automatu zde formálně definovat nebudu. Co vše do ní patří záleží na druhu automatu a někdy i na zvoleném pojetí. V obvyklém pojetí konfiguraci KA tvoří aktuální stav a zbytek čteného slova; ZA mají v konfiguraci navíc zásobník; konfigurace TS (a též dvoucestných KA) lze reprezentovat aktuálním stavem a dvěma zásobníky, které představují pásku včetně pozice hlavy. V náhledu „Simulace“ se ovšem u KA zobrazuje i načtená část slova. V implementacích historie kroků si zase není třeba pamatovat celou konfiguraci – stačí jen provedené přechody.

Reset)[5]. Co se přesně myslí začátkem výpočtu, záleží na konkrétních automatech. Následující odstavec je spíš doporučením, než požadavkem rozhraní `I_Automat`.

Automaty, které mají jediný počáteční stav, by se měly při resetu uvést do tohoto stavu[6]. Historie kroků by se měla vymazat. ZA by měly na zásobník uložit počáteční zásobníkový symbol. Otázka je, co s páskou. Automaty by na ni mohly uložit slovo, které na ní bylo na začátku (to je ale většinou prázdné – uživatel zadává slovo až po vytvoření automatu), nebo by ji mohly celou smazat. Já jsem zvolil pružnější řešení: automat při resetu pásku nemění; případné změny provádí vždy ten, kdo zavolal reset (což je typicky náhled „Simulace“).

Další metody

Metoda `getPopisKonfigurace` vrací slovní popis aktuální konfigurace automatu. Záleží na autorech implementací automatů, co vše do tohoto popisu zahrnou. `Abstract-Automat` vrací informace o tom, v jakém stavu se nachází včetně popisu tohoto stavu, pokud je přítomen. Popis konfigurace se zobrazuje v náhledu „Simulace“.

Metoda `clone` je určena k vytvoření hloubkové kopie (deep copy) automatu, která bude na původním automatu nezávislá. Pokud se po klonování nový automat nějak modifikuje (přidání stavu či přechodu, změna názvu či barvy stavu,...), měl by zůstat původní automat nezměněn.

Dále rozhraní `I_Automat` obsahuje pár pomocných, zde neuvedených, metod a dědí několik metod od svého nadrozhraní `I_AnimAlgObjekt` (viz JavaDoc dokumentaci).

Události automatu

Automat musí nějakým způsobem oznamovat náhledům, když dojde k nějaké změně. Jak již bylo řečeno, využívají se k tomuto účelu *listenery*. Pokud potřebuje náhled reagovat na přidání nového stavu do automatu (či na odebrání stavu z automatu), musí si registrovat příslušný listener na množině stavů. Potřebuje-li reagovat na změny abecedy, registruje si listener na abecedě automatu; obdobně pro přechodovou funkci, čtecí hlavu i množinu počátečních a koncových stavů.

Automat tedy většinu funkcí deleguje na své součásti, a to včetně oznamování událostí při změnách těchto objektů. Přesto zůstalo několik druhů událostí, které oznamuje přímo automat :

- provedení přechodu
  (či změna aktuálního stavu jiným způsobem, např. pomocí akcí zpět či reset)
- reset automatu
- změna hodnoty, kterou vrací metoda `prijima`
- změna hodnoty, kterou vrací metoda `getPopisKonfigurace`

[5]Funkce reset, jakož i zpět, je poskytována „navíc“ – do vlastního automatu, jak je formálně definován, nepatří. Neměla by se zaměňovat s restartovacími automaty.

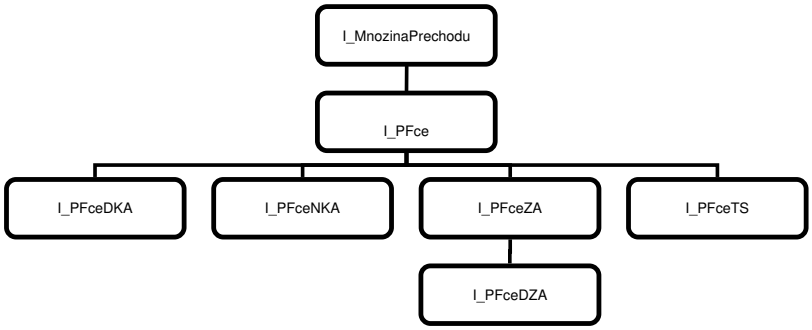
[6]U NKA jsem zvolil následující řešení. Na začátku (po resetu) je automat ve stavu `NEDEF` a jako možnosti pro příští krok se zobrazují myšlené přechody do počátečních stavů automatu.

Co nabízí `AbstractAutomat`

Abstraktní třída `AbstractAutomat` obsahuje základní implementaci téměř všech metod rozhraní `I_Automat`. V konstruktoru dostane název, popis, abecedu, množinu stavů, množinu koncových stavů, množinu počátečních stavů a čtecí hlavu. Potomci musí implementovat pouze metody `krok`, `getPFce` a navíc novou metodu `stavPoResetu`, která vrací stav, do kterého se má automat uvést po resetování. Implementování posledních dvou metod bývá triviální: přechodovou funkci dostávají automaty v konstruktoru a automaty, které mají jediný počátek přejdou po resetu do tohoto stavu, ostatní (NKA) přejdou do stavu `NEDEF`. Při implementování metody `krok` je možné využít pomocné metody `prejdi`, která přidá daný přechod do historie (ta se využívá k tomu, aby fungovala metoda `zpet`), posune čtecí hlavu o daný počet pozic a oznámí listenerům všechny potřebné události. Pokud potomkům nevyhovuje chování `AbstractAutomatu`, mohou některé jeho metody předefinovat. Například `ZA` by musel předefinovat metodu `aktualizujPrijima`, která nastavuje property „přijímá“ a tím i určuje, co bude vracet metoda `prijima`. V implementaci `AbstractAutomatu` se totiž určuje, zda automat přijímá pouze podle toho, jestli je aktuální stav koncový. Zásobníkové automaty nepřijímají, dokud není přečteno celé slovo na vstupu. ZA přijímající prázdným zásobníkem navíc nemají žádné koncové stavy.

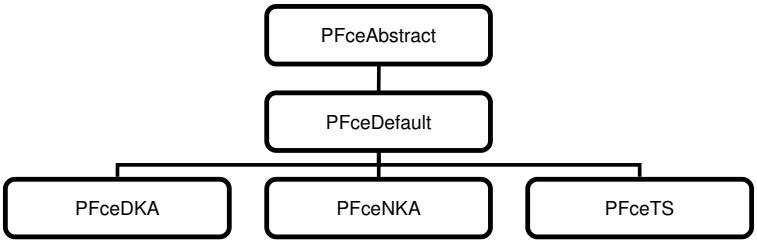
2.2 Přechodová funkce

Většina práce automatů byla delegována na jejich přechodové funkce. Dobrá koncepce přechodových funkcí je tedy klíčová pro splnění většiny cílů, které si tento projekt klade. Během vývoje jsem tuto koncepci několikrát vylepšoval a snažil se přitom skloubit několik požadavků. Hlavními z nich je jednoduché (pokud možno intuitivní) zacházení s přechodovou funkcí v algoritmech a náhledech a také jednoduché přidávání nových druhů přechodových funkcí. Rozhraní přechodových funkcí tvoří podobnou hierarchii, jako rozhraní automatů:



Obrázek 8: Hierarchie rozhraní pro přechodové funkce

Podobně jako u automatů, jsem i u přechodových funkcí postupně dospěl k tomu, že už základní rozhraní `I_PFce` by mělo obsahovat většinu metod a podrozhraní by měla přidávat jen několik málo metod (specifických pro daný druh automatu). Stejně tak důležitým se ukázal být i požadavek, aby existovala základní (univerzální) třída, která bude implementovat většinu metod rozhraní `I_PFce`, aby bylo přidávání potomků (tedy nových druhů přechodových funkcí) co nejsnadnější.



Obrázek 9: Hierarchie implementací přechodových funkcí

Tuto univerzální třídu jsem nazval `PFceDefault`. Krom toho jsem vytvořil i abstraktní třídu `PFceAbstrakt`, která implementuje jen několik metod z `I_PFce` a neobsahuje žádné datové struktury. Tato abstraktní třída může posloužit jako základ někomu, kdo by chtěl vytvořit takovou implementaci, která by nebyla možná jako potomek `PFceDefault` (například kdyby se měla data přechodové funkce načítat z databáze).

Koncepce implementace přechodové funkce v AnimAlgu

Přechodová funkce obecně přiřazuje každému prvku z definičního oboru jeden prvek z oboru hodnot. V první verzi návrhu jsem rozhraní přechodových funkcí navrhoval přímočaře podle této definice. Například `I_PFceKA` měla metodu `Stav dalsi(Stav, Znak)`, `I_PFceTS` měla metodu `StavPosunZapsat dalsi(Stav, Znak)`, která vracela objekt obsahující další stav, posun a znak k zapsání na pásku.

Tento původní návrh se ovšem ukázal vzhledem k cílům projektu jako nevyhovující. Implementace každého druhu přechodové funkce se totiž musela dělat skoro celá znova, společný předek nemohl obsahovat funkci `dalsi`. Také bylo nutné vytvářet novou třídu či rozhraní pro každý druh oboru hodnot, přičemž nedeterministické funkce musely mít jinou třídu než jejich deterministické protějšky (`I_PFceNKA` vracela seznam stavů, `I_PFceNTS` by vracela seznam objektů `StavPosunZapsat` atd.).

V novém návrhu jsem pojal přechodovou funkci jako množinu přechodů. Každý přechod má položky určující definiční obor i obor hodnot. `I_PFce` má mimo jiné metodu `prechody(Stav, Znak)`, která vrátí množinu přechodů, které vedou ze zadaného stavu přes zadaný znak. `I_PFceDKA` může mít pouze jeden takový přechod, a proto přidává metodu `prechod(Stav, Znak)`, která vrátí tento jeden přechod (nebo `null`, pokud není definován).

Třída `Prechod`

Bylo nutné navrhnout, jak budou vypadat jednotlivé přechody, které budou prvky přechodové funkce. Přechody různých druhů automatů mají některé položky stejné, ale v některých se liší. Možným řešením bylo navrhnout hierarchii rozhraní přechodů: `I_Prechod`, `I_PrechodZA`,... Nechtěl jsem však celý návrh více komplikovat a zavádět další hierarchii. Proto jsem se rozhodl vytvořit jedinou třídu `Prechod`, která obsahuje všechny položky, které se využívají v přechodech KA, ZA a TS. Pokaždé tedy zůstanou některé položky nevyužity (budou mít hodnotu `null`).

Položky přechodu (atributy třídy `Prechod`) jsou:

- `odkud` – stav, ze kterého přechod vede
- `znaky` – množina znaků, přes které přechod vede (viz dále)

- `zasobnikovySymbol` – symbol, který se při přechodu odebírá z vrcholu zásobníku
- `kam` – stav, do kterého přechod vede
- `posun` – o kolik políček (znaků) se má posunout čtecí hlava (-1, 0, +1)
- `zapsatZnak` – znak, který se má zapsat na pásku
- `zapsatSlovo` – slovo, které se má uložit na zásobník

Rozhodl jsem se, že se v třídě `Prechod` nebude ukládat jeden znak, ale rovnou celá množina znaků. Přechod lze provést, pokud je přečten z pásky libovolný znak z této množiny. Jedná se tedy o úsporné uložení více přechodů, které mají všechny položky krom znaků stejné, do jednoho objektu třídy `Prechod`[7]. Jednoduše se vytvoří například přechod, který vede přes všechny znaky abecedy: `p = new Prechod(odkud, kam, automat.getAbeceda());`. Existuje ale i konstruktor, kterému se zadá výčet znaků (proměnný počet parametrů), takže lze snadno vytvořit i přechod s jedním znakem.

Krom již zmíněných položek má třída `Prechod` ještě atributy `druh` a `barva`. Atribut `druh` může nabývat hodnot KA, ZA, TS či JINY a využívá se například v metodě `toString`, která vrací řetězec reprezentující ty položky, jež jsou pro daný druh automatu relevantní. Pro JINY se vypíší všechny položky. Tato hodnota je rezervována pro budoucí využití (například pro přechod TS s více páskami by musela být vytvořena nová třída jako potomek `Prechodu`). Atribut `barva` využívají algoritmy, které potřebují označit přechod – např. algoritmus hledající dosažitelné stavy (průchodem do šířky) označuje dosažené stavy a zároveň i přechody, kterými se do těchto stavů dostal; označené přechody pak tvoří kostru grafu. Barva přechodu se zobrazuje v některých náhledech („Graf“ a „Tabulka“).

Nedeterminismus

Oborem hodnot nedeterministických přechodových funkcí je obvykle potenční množina, např. `I_PFceNKA`: $Q \times X \to P(Q)$. Bylo by sice možné vytvořit pro NKA jiný druh přechodu, který místo `kam` bude mít množinu stavů, ale tím by se celý návrh zkomplikoval. Rozhodl jsem se tedy pro jiné řešení. Přechody NKA budou stejného druhu, jako přechody DKA (podobně i NZA bude mít stejné přechody jako DZA), tedy každý přechod bude mít jen jeden cílový stav (`kam`). Nedeterminismus `I_PFceNKA` se projeví tím, že bude existovat více přechodů z jednoho stavu přes jeden znak. Naopak `I_PFceDKA` bude dohlížet na to, aby se mezi přechody odchozími z nějakého stavu vyskytoval každý znak nejvýše v jednom přechodu.

Slučování přechodů

Přechod se do přechodové funkce přidává metodou `add(Prechod)`. Pokud už přechodové funkce obsahovala jiný přechod, který měl všechny položky krom znaků stejné, tak se oba přechody *sloučí*. Sloučení znamená, že se do množiny znaků původního přechodu

[7] Tento přístup se používá i v některých pojetích teorie automatů. Například místo zápisu: $\delta(Q, a) = R,\ \delta(Q, b) = R$ se píše pouze $\delta(Q, \{a, b\}) = R$. V grafu KA je zápis množiny znaků u jedné „šipky“ standardní. Třída `Prechod` nepředstavuje klasický, *jednoznakový* přechod, ale právě takovouto šipku, tedy *víceznakový* přechod. Díky tomu se některé operace s přechody lépe (přehledněji) popisují.

přidají znaky z nového přechodu. Metoda `add` vrátí `false`, pokud nebyla přechodová funkce přidáním změněna (tedy přechod byl již v přechodové funkci obsažen). Pokud `add` vrátí `true`, tak se buď přidávaný přechod sloučil s již existujícím přechodem, nebo se přidal přímo zadaný přechod.

Stejná sémantika se uplatňuje i na metodu `contains(Prechod)`, která zjišťuje, zda přechodová funkce obsahuje zadaný přechod. Volání `contains(x)` vrátí `true` právě tehdy, když přechodová funkce obsahuje přechod `y`, který je *stejný krom znaků*, a množina znaků přechodu `x` je podmnožinou znaků přechodu `y`. Relace *stejný krom znaků* je ekvivalence, která považuje dva přechody za ekvivalentní, pokud mají stejné všechny položky krom množiny znaků. Při implementaci je možné provést optimalizace a např. u KA prohlásit dva přechody za *stejné krom znaků*, pokud mají stejné položky `odkud` a `kam`.

Lambda přechody

U ZA a ZNKA se využívají lambda-přechody, které nečtou z pásky žádný znak, a tedy by položka `posun` u těchto přechodů měla být 0. Položka `posun` se ale u ZA a ZNKA nevyužívá, protože posun je jednoznačně určen znakem přechodu: krom λ znamenají všechny znaky posun +1. Rozhodl jsem se ukládat znak λ do množiny znaků přechodu spolu s ostatními znaky. `AbstractAutomat` dostává v metodě `prejdi(Prechod, Znak)` přechod, který se má provést, a znak, který se čte z pásky. Pokud je tento znak λ, tak se čtecí hlava neposunuje, v opačném případě se hlava posune podle položky `posun` (a ta je pro KA i ZA vždy +1).

Nezávislost na automatu

Přechodová funkce by neměla mít odkaz na automat, který ji využívá. Díky tomuto požadavku může jednu přechodovou funkci (jednu instanci) využívat více automatů (každý může mít třeba jinou množinu koncových stavů). Je také možné, aby například třídu `PFceDKA` využíval DKA i Mooreův či Mealyho stroj.

Synchronizace s abecedou

Přechodová funkce je obecně definována nad vstupní abecedou a množinou stavů, u ZA navíc nad zásobníkovou abecedou. Nemělo by se tedy nikdy stát, aby přechod obsahoval znak, který není v abecedě automatu (výjimkou jsou již zmíněné lambda-přechody). Jak toto zajistit, když přechodová funkce nemá odkaz na automat, který ji využívá?

Přechodová funkce dostane v konstruktoru abecedu a množinu stavů, nad kterými má být definována. Bude mít také metody `getAbeceda` a `getMnozinaStavu`. Díky tomu se zmenší i počet parametrů, které se zadávají v konstruktoru jednotlivým automatům – stačí zadat přechodovou funkci, protože ta už obsahuje odkaz (přesněji referenci) na abecedu a množinu stavů.

Co má přechodová funkce učinit v případě, když se do ní přidá přechod, jehož znaky nejsou prvky abecedy? Možných řešení je několik: vyvolat výjimku, přidat jen ty znaky, které jsou v abecedě, a nebo abecedu rozšířit o nové znaky. `PFceDefault` implementuje poslední zmíněné řešení, ale je možné vytvořit implementace přechodových funkcí s jiným chováním (je také možné vytvořit nemodifikovatelnou abecedu).

Záměrem zvoleného řešení je, aby mohl uživatel jednoduše vytvářet automaty inkrementálním způsobem, tedy postupně přidávat přechody s novými znaky, aniž by musel nejprve každý znak (v náhledu „Vlastnosti“) přidat do abecedy.

Pokud se odebere znak z abecedy, přechodová funkce automaticky odstraní všechny přechody s tímto znakem (tedy odebere daný znak z množiny znaků každého přechodu, a pokud byl poslední, tak smaže celý přechod). Obdobně pokud je odebrán stav z množiny stavů, přechodová funkce odstraní přechody, které vedly z nebo do tohoto stavu.

Při přidávání přechodu se ovšem synchronizace s množinou stavů provádí jinak. V náhledu „Graf“ nejde přidat přechod z nebo do neexistujícího stavu, protože přechody se přidávají z kontextové nabídky stavů a jako cílové stavy se nabízí pouze ty existující. Pokud jsou přechody přidávány algoritmem, který nemůže zaručit, že startovní (odkud) i cílový (kam) stav přechodu je prvkem množiny stavů, musí místo metody `add` použít `xadd`, jak bude vysvětleno v příštím odstavci.

Překlad přechodů

Často se v algoritmech vytváří podle jednoho automatu nový automat. Automaty mají stavy stejně pojmenované, ale jedná se o různé instance stavů (aby byl nový automat nezávislý na původním). Pokud je třeba přidat přechod `x` z původního automatu do nového, nelze použít `add(x)`, protože přechod `x` odkazuje na stavy, které v novém automatu nejsou. Je ale navržena metoda `xadd(Prechod)`, která nejdříve provede tzv. *překlad* přechodu a přidá až přeložený přechod. Překlad znamená, že se podle zadaného přechodu vytvoří nový přechod, jehož stavy `odkud` a `kam` se najdou v množině stavů a budou mít stejný název jako stavy zadaného přechodu. Při překladu se také vytvoří kopie množiny znaků, aby byl nový přechod nezávislý na původním.

Přechodová funkce jako `Set<Prechod>`

Rozhraní `I_PFce` je rozšířením standardního Java rozhraní `Set` s parametrem `Prechod`, tedy lze s přechodovou funkcí zacházet jako s množinou přechodů a používat krom již zmíněných metod i iterování a metody `remove`, `addAll`, `removeAll`, `containsAll` atd. To přináší uživatelům mnoho výhod a usnadnění práce, ale také pár nástrah.

Kvůli popsanému slučování přechodů je přechodová funkce vlastně množinou množin. Uvažujme například přechod `x` s množinou znaků $\{a, b\}$ a přechod `y`, který je *stejný krom znaků* jako `x`, ale v množině znaků má pouze a. Pokud je do přechodové funkce přidán `x`, pak funkce automaticky obsahuje i `y`. Volání `contains(y)` tedy vrátí `true`, i když při iterování přes přechodovou funkci se nenajde žádný přechod, který by byl ekvivalentní (podle metody `equals`) s `y`. Metodu `equals` nelze předefinovat, aby vracela totéž co relace *stejný krom znaků* – nemělo by to smysl.

Rozhraní `Set` používá ve specifikaci mnoha metod právě `equals`. Jak bylo popsáno, přechodová funkce a obecněji i jakákoli množina přechodů se řídí mírně odlišnou specifikací. Aby nevznikaly nejasnosti, vytvořil jsem rozhraní `I_MnozinaPrechodu`, které upřesňuje popis (v JavaDoc dokumentaci) k jednotlivým metodám `Set` a přidává navíc metodu `getStejnyKromZnaku(Prechod)`, která vrátí ten přechod z množiny přechodů, který je *stejný krom znaků* jako parametr metody.

Zdrojový kód `I_PFce`

Popis všech metod je v JavaDoc dokumentaci. Zde uvádím jen seznam metod:

```
public interface I_PFce extends I_MnozinaPrechodu {
  public I_MnozinaStavu getMnozinaStavu();
  public I_MnozinaZnaku getAbeceda();

  public Set<Prechod> odchoziPrechody(Stav odkud);
  public List<Prechod> prichoziPrechody(Stav kam);
  public SortedSet<Znak> odchoziZnaky(Stav stav);
  public Set<Prechod> prechody(Stav odkud, Stav kam);
  public Set<Prechod> prechody(Stav odkud, Znak znak);
  public boolean jePrechod(Stav odkud, Znak znak);
  public boolean jePrechod(Stav odkud);
  public boolean jePrechod(Stav odkud, Stav kam);
  public boolean jePrechod(Znak znak);

  public void zrusPrechody(Stav stav);
  public void zrusPrechody(Znak znak);
  public boolean odeberZnakZPrechodu(Prechod prechod, Znak znak);

  public boolean xcontains(Prechod prechod);
  public boolean xadd(Prechod prechod);
  public boolean xremove(Prechod prechod);
  public boolean xcontainsAll(Collection<? extends Prechod> c);
  public boolean xaddAll(Collection<? extends Prechod> c);
  public boolean xremoveAll(Collection<? extends Prechod> c);

  public I_PFceTableModel getTableModel();
  public ListModel getListModel();

  public void addPFceListener(PFceListener l);
  public void removePFceListener(PFceListener l);
  public void addPropertyListener(PropertyListener listener);
  public void removePropertyListener(PropertyListener listener);
}
```

Jak je vidět, krom metod zděděných od `I_MnozinaPrechodu` obsahuje `I_PFce` i další metody pro získávání informací o přechodové funkci i pro její modifikování. Za zmínku stojí metody, které ulehčují práci náhledům „Tabulka“ a „Seznam“. `getTableModel` vrací objekt, který se použije jako model tabulky přechodů. Udržovat takovýto model aktualizovaný vyžaduje jisté paměťové a hlavně výpočetní zdroje, a proto je obsažen už v přechodové funkci, která může provádět různé optimalizace. Krom toho může být tento model využíván více náhledy a bylo by zbytečné, kdyby ho musel každý náhled vytvářet znovu. Metoda `getListModel` vrací model pro seznam přechodů a platí pro ni obdobná tvrzení. Pokud by některá implementace přechodové funkce tyto metody nepodporovala, může vrátit místo modelu `null` a v náhledu se zobrazí jen informační hláška.

Implementace

Třída `PFceDefault` deleguje většinu práce na objekty třídy `PFceStavDefault`. Každý takový objekt obsahuje všechny přechody, které mají stejný stav `odkud`, je to tedy množina přechodů odchozích z daného stavu. `PFceDefault` podle hashovací tabulky určí, na který z těchto objektů má delegovat volání nějaké metody. Aby byly některé operace proveditelné v rozumném čase, udržuje si `PFceStavDefault` i seznam příchozích přechodů. Modely pro tabulku a seznam přechodů se vytváří „líně“, tedy až když je poprvé zavolána příslušná metoda.

Potomci `PFceDefault` mohou místo `PFceStavDefault` používat `PFceStavZnakMap`, která obsahuje mapování (hashovací tabulku) znaků na odchozí přechody, a tedy zaručuje, že ze stavu bude přes daný znak vycházet nejvýše jeden přechod. Tento postup využívá `PFceDKA` a `PFceTS`.

2.3 Množiny

Při návrhu rozhraní `I_MnozinaStavu` a `I_MnozinaZnaku` jsem si stanovil několik požadavků, z nichž je většina společná pro obě rozhraní.

- **bez duplicit**
  Množina nesmí obsahovat více prvků se stejným názvem. Název je totiž jednoznačným identifikátorem stavu i znaku jak pro uživatele, tak pro některé algoritmy. Název znaku je neměnný, ale o stavu to neplatí – buďto ho mění uživatel, nebo i některé algoritmy (např. `Normalizace` přejmenovává stavy čísly 1 až n). Množina si tedy musí registrovat na každý svůj stav tzv. *veto-listener*, který zakáže takové změny názvu, kterými by v množině došlo k duplicitě.

- **návaznost na Java Collections**
  Množina implementuje standardní rozhraní `Set` (s parametrem `Stav` resp. `Znak`), aby bylo používání pro uživatele (autory algoritmů) co nejpohodlnější.

- **observable collection**
  Množina oznamuje pomocí listenerů přidání či odebrání prvku. Množina stavů navíc i oznamuje, pokud se změní vlastnost (*bound property*) některého prvku (popis, barva,...). Pro uživatele (zejména programátory nových automatů a náhledů) je pak využívání množiny stavů jednodušší, než kdyby museli registrovat listener na každý prvek a při odebrání prvku z množiny tento listener zase odregistrovat.

- **setřídění**
  Množina by měla být setříděná (obvykle podle názvů). Tento požadavek vychází vstříc hlavně potřebám náhledů. Např. v tabulce přechodů jsou stavy (řádky) i znaky (sloupce) setříděné a toto uspořádání by mělo být zachováno i při přidávání a odebírání stavů či znaků. Bylo by možné při každé změně tabulku načíst celou znova a stavy i znaky setřídit. Jednotlivé náhledy jsou ale moduly, které o sobě nemusí „vědět“, a tak by mohla být stejná práce zbytečně dělána vícekrát. Vzhledem k tomu, že celý projekt je zaměřen na animace a názornost (nikoli na rychlost), předpokládá se, že bude skoro vždy zobrazen nějaký náhled vyžadující setřídění. Proto by měla už sama množina vracet prvky setříděné.

- **přímý přístup**
  Tento požadavek souvisí s předchozím. `TableModel`, který využívá tabulka přechodů, musí implementovat metodu `getValueAt(int r, int c)`, tedy musí vědět, jaký stav je v řádce `r` a jaký znak je ve sloupci `c`. Opět lze uvažovat o řešení, kde si `TableModel` ukládá stavy (i znaky) do pole, které při každé změně znovu načte. Podle mě je výhodnější, když už množina bude mít metodu `getElementAt(int index)`.

- **ListModel**
  Některé náhledy zobrazují stavy (či znaky) jako seznam v komponentě `JList`. Pokud by tento seznam nemusel reagovat na změny (např. v modálním dialogu), je vše snadné. Pro reakci na změny musí mít `JList` model (`ListModel`), který reaguje na události jako např. `intervalAdded(ListDataEvent)`. Opět se mi zdá výhodnější, bude-li už množina implementovat rozhraní `ListModel`.

- **rychlost**
  Projekt není určen pro velké objemy dat (automaty se stovkami stavů apod.), proto není kladen na rychlost přílišný důraz. Vykreslování náhledů zabere typicky mnohem více času než vlastní výpočet vnitřní logiky. Přesto by měla být implementace množiny co nejefektivnější při splnění předchozích požadavků. Otázkou je, pro které operace má být množina optimalizována: zda upřednostnit rychlejší metodu `contains(prvek)` před metodou `getElementAt(index)`; zda mít rychlejší vyhledávání, ale pomalejší přidávání či odebírání prvků. Obecně ale platí, že `contains(prvek)` se volá velmi často (a to nejen v algoritmech, ale i „na pozadí“, například náhled může při každém překreslení kontrolovat, zda stav patří do množiny koncových). Pokud se množina zobrazuje v komponentě `JList`, volá se `getElementAt(index)` také velmi často.

Zavrhnuté možnosti implementace

- **java.util.TreeMap** je implementace množiny pomocí červeno-černých stromů, která tedy poskytuje setřídění. Neumožňuje však přímý přístup, pokaždé by bylo nutné použít iterátor s časovou složitostí *o(N)*.

- **jiné hotové řešení**, konkrétně jsem prozkoumal org.apache.commons.collections [4]: Nenabízí listenery a generika, navíc implementace nejvhodnější k mému účelu (`ListOrderedMap`) uplatňuje *insertion-order*, ale množiny stavů a znaků mají být setříděny dle názvů. Nicméně tento zdroj posloužil jako dobrá inspirace.

Zvolené řešení

Navrhl jsem rozhraní `I_SetridenaMnozina<E>`, které obsahuje všechny výše uvedené požadavky. Naprogramoval jsem tři třídy, které implementují toto rozhraní: `ArraySet` (implementace pomocí pole), `AvlSet` („prošívaný“ AVL-strom umožňující přístup pomocí indexu v *o(log n)*) a `AvlHashSet` (k AVL-stromu přidána hashovací tabulka). Dále jsem vytvořil třídu `SetridenaMnozina<E>`, která slouží jako *wrapper* pro různé implementace, tedy v konstruktoru se jí zadá implementace a ona volání téměř všech metod deleguje na tuto implementaci. V konstruktoru lze zadat kromě zmíněných tří implementací i libovolnou vlastní. Třídy `MnozinaStavu` a `MnozinaZnaku` jsou potomky třídy `SetridenaMnozina`, která je odstíní od konkrétní implementace.

Důsledky a komplikace

V důsledku požadavků, že stavy mohou měnit své názvy a množina nesmí připustit duplicitu, vzniklo při implementaci několik komplikací. Podrobnější rozbor je v programátorské příručce na straně 45. Zde pouze ve zkratce:

Setříděné množiny určují setřídění podle tzv. *komparátoru*, který může být zadán i implicitně, pokud prvky množiny implementují rozhraní `Comparable`. Bohužel třída `Stav` toto rozhraní implementovat nemůže, protože by pak instance musely měnit hashcode při každé změně názvu. Obecně se naráží na problém, že stavy se stejným názvem se považují za ekvivalentní pouze pro některé účely a nelze jim předefinovat metodu `equals`, aby se řídila pouze názvem. Je ale žádoucí, aby se implementace (`AvlHashSet`), která používá hashovací tabulku, chovala identicky, jako ostatní implementace (rozdíl je jen v časové složitosti jednotlivých operací).

Všechny uvedené komplikace se nakonec v projektu podařilo vyřešit. Pro používání i rozšiřování AnimAlgu běžným způsobem, nemusí uživatel tuto problematiku studovat. Pouze pokud by někdo chtěl přidávat vlastní implementace množin nebo se dozvědět více o vnitřním fungování, měl by si prostudovat příslušné části programátorské příručky a JavaDoc dokumentace.

2.4 Pomocné objekty

Nejdůležitější objekty, které automaty využívají, již byly popsány. Následuje popis dalších, pomocných, leč důležitých objektů. Úplný seznam lze nalézt v JavaDoc dokumentaci.

Stav

Třída `Stav` má atributy pro název, popis a barvu. Během vytváření algoritmů a náhledů vyvstala potřeba ukládat do stavu i další vlastnosti, které je vhodné sdílet mezi náhledy a algoritmy. Například souřadnice (pro náhled „Graf“) nebo číslo skupiny (pro předání ekvivalence na stavech mezi algoritmem `EkvivalentniStavy` a `PodilovyAutomat`). Takovýchto vlastností (využitelných třeba jen v jednom algoritmu) lze vymyslet celou řadu. Proto jsem se rozhodl ukládat tyto vlastnosti do stavu pomocí tzv. *client-properties*, jejichž fungování vysvětlím na příkladě:

```
stav.putClientProperty("souřadnice", new Point(5,5));
```

uloží do stavu objekt představující souřadnice a volání

```
Point p = (Point)stav.getClientProperty("souřadnice");
```

tyto souřadnice zase načte.

Složený stav

Třída `StavSlozeny` je potomkem třídy `Stav` a představuje tzv. *složený stav*, který se skládá z několika *vnitřních stavů*. Toho se využije například při determinizaci NKA (kdy je nová množina stavů potenční množinou původní množiny stavů) nebo v algoritmech pro důkaz uzavřenosti regulárních jazyků na množinové operace (kdy je nová množina stavů kartézským součinem původních množin stavů). V prvním příkladě vnitřní stavy představují podmnožinu, tedy nezáleží na jejich pořadí, a proto je složený stav setřídí dle názvů. V druhém případě vnitřní stavy představují uspořádanou

n-tici, a složený stav tedy zachová jejich pořadí (v konstruktoru se musí zadat parametr `zaleziNaPoradi`). Složený stav má metodu `getStavy`, která vrátí jeho vnitřní stavy.

Znak

Třída `Znak` může představovat znak vstupní abecedy nebo zásobníkový symbol. V některých úlohách se používají znaky s indexy či jiným označením, proto znak může být libovolný textový řetězec. Tento řetězec vrací metoda `getNazev`. Doporučuji ale zachovávat znaky (tedy jejich názvy) co nejkratší a případné vysvětlení uložit do popisu znaku. Narozdíl od stavů není možné názvy znaků měnit, díky čemuž mohla být předefinována metoda `equals` třídy `Znak` tak, aby znaky se stejným názvem byly ekvivalentní. `Znak` také implementuje rozhraní `Comparable`.

Přechod

Třída `Prechod` už byla popsána v kapitole 2.2. Zde pouze doplním několik informací. Pro každý druh automatu obsahuje `Prechod` jeden komparátor (`prechodComparatorKA`, `prechodComparatorZA`,...), který definuje lineární uspořádání na přechodech podle položek relevantních pro daný druh automatu kromě položky `znaky`. Když se přechody ukládají do přechodových funkcí použije se právě tento komparátor. `Prechod` také obsahuje sadu metod, které vrací textovou reprezentaci přechodu pro různé účely. Tyto metody se využívají při vypisování přechodu v náhledech „Seznam“ a „Tabulka“ i při ukládání přechodu do souboru.

Páska

Konečné a zásobníkové automaty využívají pásku, na kterou nic nezapisují a může být jednostranná (tedy nepřistupují na záporné indexy). Pro tyto účely je navrženo rozhraní `I_Paska`. TS využívá oboustrannou přepisovatelnou pásku, pro kterou je určeno rozhraní `I_OboustrannaPaska`. Třída `Paska` implementuje obě tato rozhraní a navíc i rozhraní `I_Zasobnik`. Páska pomocí listenerů oznamuje všechny změny, které se na ní provedou, takže může být snadno zobrazena v náhledech.

Čtecí hlava

Třída `CteciHlava` dostává v konstruktoru objekt implementující rozhraní `I_Paska`. Čtecí hlava si pamatuje aktuální pozici na pásce a zprostředkovává automatu čtení z pásky. Čtecí hlava oznamuje listenerům, když se změní její pozice a když se změní znak na aktuální pozici. Zapisovací hlava (třída `ZapisovaciHlava`) funguje stejně, pouze dostává v konstruktoru objekt implementující rozhraní `I_OboustrannaPaska` a umožňuje navíc zápis.

3 Algoritmy

3.1 Jazyk algoritmů

Při návrhu koncepce algoritmů pro AnimAlg jsem musel nejdříve zvolit, v jakém programovacím jazyce se budou algoritmy psát. V úvahu připadala Java, nebo vlastní jazyk navržený speciálně pro účely AnimAlgu. Rozhodl jsem se pro Javu, k čemuž mě vedly následující úvahy.

V jazyce Java (konkrétně JDK 5.0) je napsáno celé prostředí AnimAlg i automaty a náhledy. Algoritmy musí mít zejména k automatům dobrý přístup. Koncepce automatů těží z modularity a objektově orientovaného programování. Ve vlastním jazyce by bylo poměrně obtížné zajistit přístup i k nově přidaným druhům automatů.

Osobně nemám s návrhem programovacích jazyků zkušenosti a při špatném návrhu vlastního jazyka by byl celý projekt nepoužitelný. Naopak i při výběru Javy jako základního jazyka algoritmů je možné později doplnit do AnimAlgu modul, který bude interpretovat nějaký pseudokód či překládat algoritmy z pseudokódu do Javy.

Jazyk Java není vždy zcela přímočarý a nelze v něm programovat „neobjektově“, což může být překážkou pro začínající programátory. Na druhou stranu je relativně rozšířený, univerzální, jednoduše naučitelný a pro dané účely podobný jazyku C/C++. I pokud by se musel uživatel tento jazyk učit, domnívám se, že je to výhodnější, než se učit jazyk, který nemá využití jinde než v AnimAlgu.

Ve výsledné animaci algoritmu se nezobrazuje zdrojový kód, ale strom událostí, což má své výhody i nevýhody. Díky tomu ale může být animace srozumitelná i uživatelům bez znalosti programování, přestože kód algoritmu používá pokročilých programovacích technik specifických pro Javu.

Autoři algoritmů v Javě mohou využívat všech výhod z toho plynoucích: objektově orientované programování, silná nejen typová kontrola při překladu, možnost použít vývojové prostředí s debuggerem, možnost vytvářet v algoritmu pokročilé grafické komponenty, které se zobrazí při animaci,...

3.2 Koncepce algoritmů

Algoritmus určený pro AnimAlg je obyčejná Java třída, která implementuje rozhraní `I_Algoritmus`. Toto rozhraní jsem se snažil navrhnout poměrně obecně, aby umožňovalo různé přístupy ke krokování. Hlavním cílem však bylo, aby se algoritmy daly psát co nejjednodušeji, bez nutnosti starat se o technické detaily krokování. Vytvořil jsem tedy třídu `AbstractAlgoritmus`, která zajišťuje většinu pomocných metod rozhraní `I_Algoritmus`. Algoritmy, které jsou potomky této třídy, musí implementovat pouze metodu `vypocet`, která provede vlastní výpočet.

Nejprve vysvětlím, jak funguje `AbstractAlgoritmus` a okno *Krokování algoritmu*. Teprve poté rozliším, co z toho vyžaduje rozhraní `I_Algoritmus` a co je konkrétní

řešení zvolené v třídě `AbstractAlgoritmus`. V následujícím textu tedy předpokládám, že popisované algoritmy jsou potomky právě této třídy.

Prostředí AnimAlg načte pomocí Java Reflection API informace o všech dostupných algoritmech. Když si uživatel vybere nějaký algoritmus, tak se v okně *Zadání vstupních parametrů* zobrazí název a popis algoritmu a hlavně seznam parametrů, které musí uživatel zadat. Typy parametrů se určí podle parametrů konstruktoru algoritmu (popis, možné hodnoty a implicitní hodnota se načtou z anotací, viz str. 33). Instance algoritmu se vytvoří, až když uživatel vyplní všechny parametry a stiskne tlačítko *Spustit*.

V té chvíli se také vytvoří nové vlákno, ve kterém se bude provádět metoda `vypocet`. Při každém stisku tlačítka *Step in*, *Step over* nebo *Step out* se zavolá metoda `krok` (definovaná v rozhraní `I_Algoritmus`), která probudí (či „nastartuje“) vlákno výpočtu. To provede nejbližší krok a pak se opět uspí, aby si mohl uživatel prohlédnout, jak krok dopadl.

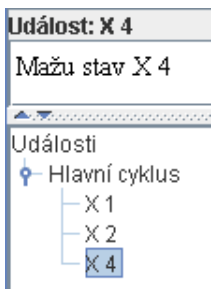
Jak ale algoritmus pozná, kdy se má zase zastavit, tedy kdy se má vlákno výpočtu uspat? K tomu jsou určeny tzv. *události* algoritmu. Autor algoritmu na příhodná místa v kódu vloží metodu `je`, které předá informace o tom, jaká událost se má vyvolat (především název a popis události). Tato událost je pomocí `UdalostListener`u oznámena oknu *Krokování algoritmu*, které rozhodne, zda se má pokračovat ve výpočtu, či nikoli.

Okno si pamatuje, jaké tlačítko bylo naposledy stisknuto. Pokud *Step in*, tak se vlákno zastaví při nejbližší události. Pokud *Step over*, tak se vlákno zastaví při nejbližší události s *hloubkou zanoření* menší nebo rovnou hloubce zanoření před stisknutím tlačítka. Při *Step out* musí být hloubka zanoření ostře menší. Z každé události tedy musí jít vyčíst její hloubka zanoření.

Předně je ale třeba definovat hloubku zanoření. V ladicích programech se hloubka zanoření počítá ve zdrojovém kódu obvykle podle aktuálního počtu do sebe zanořených podprogramů, někdy je ale možné pomocí step over provést naráz i cyklus, který není umístěn v samostatném podprogramu. V AnimAlgu si hloubku zanoření jednotlivých událostí volí autor algoritmu dle vlastního uvážení. Obvykle ale hloubka zanoření události nějakým způsobem koresponduje s hloubkou zanoření v kódu.

Každá událost má krom názvu a popisu i odkaz na tzv. *vnější událost*. Pokud je tento odkaz `null`, tak je hloubka zanoření události rovna nule, jinak je hloubka zanoření rovna hloubce zanoření vnější události + 1. Metoda `je` dostává v prvním parametru název a v druhém popis události, která se má vyvolat. Jako třetí parametr lze zadat vnější událost nově vyvolané události. Pokud tento parametr není zadán, bude vnější událost `null`. Metoda `je` vrací vyvolanou událost. Pro názornost uvedu příklad kódu a výřez z okna *Krokování algoritmu*:

```
// automat má neoznačené stavy X1, X2 a X4
I_MnozinaStavu ms = automat.getMnozinaStavu();
Stav[] stavy = ms.poleStavu();
Udalost u = je("Hlavní cyklus",
               "Mažu neoznačené stavy.");
for (Stav stav : stavy){
  if (! stav.oznacen()){
    je(stav.getNazev(), "Mažu stav " + stav, u);
    ms.remove(stav);
  }
}
```



Můžeme říci, že událost `x` je *otevřená*, pokud dosud nebyla vyvolána událost, která by nebyla potomkem `x`. Přičemž potomkem se v této definici nemyslí objektovou dědičnost, ale hierarchii zanoření ve stromě událostí.

Když algoritmus vyvolá novou událost s vnější událostí `y`, tak okno *Krokování algoritmu* zkouší najít `y` mezi právě otevřenými událostmi (které si pamatuje pomocí stromu událostí). Pokud takovou najde, tak přidá novou událost jako jejího dalšího potomka. Pokud takovou nenajde, tak přidá novu událost na úroveň 0.

Tímto postupem se také zjistí hloubka zanoření nové události. Podle toho, které krokovací tlačítko bylo stisknuto naposledy a jaká při tom byla hloubka zanoření se vypočte hloubka, při níž se má algoritmus pozastavit. Pokud je hloubka aktuální události větší než hloubka pro zastavení, tak se nechá algoritmus pracovat dál (vlákno se neuspává).

Podalgoritmy

V algoritmu lze z metody `vypocet` spouštět libovolné jiné metody a lze také spouštět jiné algoritmy (tedy *podalgoritmy*). Pokud by byl podalgoritmus ihned po vytvoření spuštěn (`new NejakyAlgoritmus.vypocet();`), tak se provede celý naráz a jeho události se nezobrazí ve stromě událostí, protože nebude mít žádný odkaz na okno *Krokování algoritmu*. Proto se podalgoritmy spouštějí pomocí metody `spust`, která zajistí vše potřebné. Krom vlastního podalgoritmu lze této metodě předat událost, která se má nastavit jako vnější událost pro všechny události podalgoritmu, které by jinak měly hloubku zanoření 0. Tím se dosáhne toho, že se události podalgoritmu zobrazí ve stromě událostí „na správném místě“.

Rozhraní `I_Algoritmus`

```
public interface I_Algoritmus extends I_Komentovany{
  public void vypocet();
  public void krok();
  public void ukonci();
  public Object vysledek();
  public void zobrazKomponenty(I_Grafika grafika);
  public Udalost getVnejsi();
  public void setVnejsi(Udalost vnejsi);
  public void setUdalostListener(UdalostListener udalostListener);
  public UdalostListener getUdalostListener();
}
```

První dvě metody již byly popsány. Metoda `ukonci` způsobí co nejrychlejší ukončení algoritmu a je volána například, když uživatel zavře okno *Krokování algoritmu*.

Některé algoritmy pracují „in situ“ a nepotřebují vracet žádný výsledek – jejich výstupem je upravení automatu, který získaly na vstupu. Ostatní algoritmy vrací svůj výstup pomocí metody `vysledek`. Pokud by bylo výstupních objektů víc, vrátí algoritmus touto metodou jejich pole či kolekci.

Většina algoritmů nepotřebuje zobrazovat žádné grafické komponenty – pro pochopení algoritmu stačí strom událostí a náhledy, které zobrazují, jak se mění automaty, se kterými algoritmus pracuje. Občas je ale vhodné uživateli ukázat ještě nějaké údaje navíc, například algoritmus `EkvivalentniStavy` zobrazuje tabulku ekvivalencí (viz

str. 10). Pomocí metody `zobrazKomponenty` se algoritmu předá objekt `grafika`, který umí vykreslit grafické komponenty na plochu. Tato metoda se obvykle volá ihned po vytvoření instance algoritmu, aby algoritmus mohl vykreslovat komponenty kdykoli během výpočtu. Někdy je však nežádoucí, aby algoritmus cokoli zobrazoval, například pokud je spuštěn z jiného algoritmu jen jako pomocný výpočet. V tom případě se metoda `zobrazKomponenty` jednoduše nezavolá (potomci `AbstractAutomatu` to poznají tak, že atribut `grafika` je `null`). Algoritmy, které vrací jako výsledek automat (nebo i jiný animAlg-objekt), použijí k jeho zobrazení též objekt `grafika`.

Anotování algoritmů

Informace, které se uživateli o algoritmu zobrazí ještě před spuštěním se zadávají pomocí Java anotací. Krom celého algoritmu jde takto anotovat i konstruktor a jeho jednotlivé parametry. Vše nejlépe vysvětlím na příkladu:

```
@Nazev ("KA: množinové operace")
@Popis ("Algoritmus dostane na vstupu dva KA: A a B, které...")
@Vysledek ("DKA přijímající sjednocení / průnik / rozdíl jazyků L(A) a L(B)")
public class KA_MnOperace extends AbstractAlgoritmus {

  @Konstruktor
  public KA_MnOperace(
        @Popis    ("1. automat")
      I_DKA automat1,
        @Popis    ("2. automat")
      I_DKA automat2,
        @Popis    ("Jaká množinová operace se má provést?")
        @Moznosti ({"sjednocení", "průnik", "rozdíl"})
        @Implicit ("sjednocení")
      String operace){
...
```

Anotaci `Vysledek` mají pouze ty, algoritmy, které vrací výsledek (různý od `null`). Algoritmus může mít více konstruktorů, ale pouze jeden lze používat pro spouštění přímo z AnimAlgu – ostatní mohu být využívány jinými algoritmy. Pokud má algoritmus více konstruktorů, musí mít právě jeden z nich anotaci `Konstruktor`, aby AnimAlg věděl, který má používat. Anotace `Moznosti` určuje seznam možných hodnot; `Implicit`, jaká hodnota parametru má být nastavena jako výchozí. Java připouští v anotacích pouze `String`. Pokud je ovšem typ parametru `Boolean`, `Integer` nebo `Color`, tak AnimAlg převede anotaci `Implicit` na příslušný objekt. Lze tedy mít například anotace `Implicit("true")`, `Implicit("7")` či `Implicit("RED")`.

Co nabízí `AbstractAlgoritmus`

Většina již byla zmíněna. `AbstractAlgoritmus` se stará o spouštění a krokování vlákna (metodou `krok`). Nabízí potomkům několik verzí metody `je` a `spust`. Při ukončení algoritmu (metodou `ukonci`) nejdříve rekurzivně ukončí všechny podalgoritmy. Implementuje metody `getNazev` a `getPopis` tak, že přečte příslušné informace z anotace algoritmu.

4 Prostředí AnimAlg

Některé části grafického prostředí AnimAlg už byly popsány v minulých kapitolách (zejména 1.2). Také již bylo zmíněno, že samotné prostředí není nijak vázáno na teorii automatů a umožňuje animovat algoritmy i z jiných oblastí. Vše specifické pro danou oblast se dodává ve formě modulů. Doposud byly explicitně zmíněny pouze dva druhy modulů: algoritmus a náhled. V této kapitole popíšu i zbývající druhy modulů a také, jak se s nimi zachází.

4.1 Moduly

V AnimAlgu existuje pět druhů modulů, pro každý z nich je vytvořeno rozhraní.

- **AnimAlg-objekty**
  To jsou objekty, které lze otevírat do oken na ploše AnimAlgu. V projektu zatím nejsou implementovány jiné animAlg-objekty než automaty. Proto jsem se dosud termínu *animAlg-objekt* vyhýbal a používal raději *automat* (abych nezaváděl nový pojem dříve, než to bude nutné). Do projektu je ovšem možné doplnit i jiné objekty, které se budou otvírat v oknech s více náhledy, například gramatiky, HMM (Hidden Markov Model) či grafy (z teorie grafů). AnimAlg-objekt je společný název pro všechny tyto objekty zajišťující vnitřní logiku.

  Rozhraní `I_AnimAlgObjekt` je podrozhraním rozhraní `I_Komentovany`, které má metody pro přístup k atributům typ, název a popis. `I_AnimAlgObjekt` požaduje navíc možnost zaregistrovat si listener pro změny těchto atributů (krom typu, který měnit nelze).

- **Náhledy**
  Náhled je modul, který zobrazuje nějaký animAlg-objekt. Vlastní grafickou komponentu (potomka třídy `JComponent`) vrátí metoda `getNahled` z rozhraní `I_Nahled`. Dále mají náhledy název, volitelnou ikonu (ta se zobrazuje vedle názvu) a popis (ten se zobrazuje jako tooltip; název by měl být totiž co nejkratší). Náhled musí mít konstruktor s právě jedním parametrem, kterým je některé podrozhraní `I_AnimAlgObjektu`. Náhled tím deklaruje, že umí zobrazovat všechny objekty implementující toto podrozhraní.

- **Algoritmy**
  Algoritmy (včetně rozhraní `I_Algoritmus`) už byly popsány v kapitole 3.

- **IO-moduly**
  IO-moduly (Input Output moduly) zajišťují načítání a ukládání animAlg-objektů ve formátu XML (eXtensible Markup Language) a jsou využívány v menu *Soubor* v příkazech *Otevřít* a *Uložit*. Každý IO-modul má anotaci `Typy`, ve které specifikuje, které animAlg-objekty umí načítat pomocí statické metody `otevri`. To, které animAlg-objekty umí ukládat, se pozná podle prvního parametru statické

metody `uloz`. Samozřejmě by měl každý IO-modul umět načítat ty animAlg-objekty, které umí uložit. Rozhraní `I_IO` je prázdné (pouze značkovací), protože Java interface nesmí obsahovat statické metody. IO-moduly se ovšem načítají pomocí Java reflection API (podobně jako ostatní moduly, krom animAlg-objektů), kde se vše potřebné ověří.

- **Továrny**
  Továrna je modul, který vytváří nové animAlg-objekty. Továrny se používají především v menu *Soubor*, kde se pomocí nich vytváří prázdné animAlg-objekty. (Pro každý druh animAlg-objektu by tedy měla existovat továrna, která vyrobí prázdný animAlg-objekt daného druhu.) Továrna může ale vyrábět i animAlg-objekt s již nějakým obsahem, čímž může částečně nahradit IO-modul. Toho se využívá zejména tehdy, když je AnimAlg spuštěn jako applet a nemůže číst ze souborů (uživatel má přístup aspoň k několika hotovým ukázkovým animAlg-objektům).

  Rozhraní `I_Tovarna` je pouze rozšířením rozhraní `I_Algoritmus`: metoda `getVysledek` musí vrátit animAlg-objekt. Továrna může vyrábět i více animAlg-objektů – jejich seznam uvede pomocí anotace `Moznosti` u jediného parametru svého konstruktoru.

Výběr modulů

Všechny zmíněné moduly jsou Java třídy a mohou být doplňovány uživateli do již zkompilovaného prostředí AnimAlg. To, které moduly se mají načíst, si určuje uživatel v souboru `nastaveni.xml`. Pouze animAlg-objekty se do `nastaveni.xml` neuvádějí. Není to totiž třeba – animAlg-objekt se totiž může na ploše otevřít pouze jedním z následujících tří způsobů: načtením ze souboru pomocí IO-modulu, vytvořením pomocí nějakého modulu továrna nebo vytvořením pomocí nějakého algoritmu.

Při spuštění AnimAlgu se načtou informace o všech modulech zadané v souboru `nastaveni.xml`. Ukázka tohoto souboru je v příloze 8.2, kde se také nachází ukázka souboru s uloženým DKA (tak jak ho ukládá a načítá IO-modul `IO_Automat`). Pokud se soubor `nastaveni.xml` nenajde nebo není povoleno z něj číst (což je případ appletu), načte se výchozí nastavení. Existují dva způsoby, jak v `nastaveni.xml` zadávat moduly:

- **jednotlivě**
  Použije se tag `trida`, v jehož atributu `nazev` bude plně kvalifikovaný (tedy včetně balíku) název třídy, ve které je uložen příslušný modul.

- **po celých adresářích**
  Použije se tag `adresar` s atributy `cesta` a `balik` a načtou se všechny třídy v adresáři se zadanou cestou. Přesněji řečeno: AnimAlg načte názvy všech `class` souborů v zadaném adresáři, odstraní z jejich názvů příponu „`.class`“, před každý název připojí jméno zadaného balíku a pokusí se načíst příslušnou třídu. Tento způsob se obzvlášť hodí, pokud uživatel přidává nové algoritmy, aby nemusel pokaždé měnit soubor `nastaveni.xml`.

U algoritmů lze navíc v tagu `trida` i `adresar` zadat atribut `kategorie`, který určuje, do jaké kategorie se má algoritmus zařadit. Algoritmy zařazené v nějaké kategorii

se pak zobrazují v podmenu s názvem této kategorie. Menu *Algoritmy* tak zůstane přehledné, i pokud se přidá větší množství algoritmů.

Moduly se udržují ve stejném pořadí, v jakém byly načteny z `nastaveni.xml`. To je důležité zejména pro náhledy, protože při otevření nového animAlg-objektu je vidět jen panel prvního náhledu (z ostatních náhledů je vidět jen název a ikona). Pokud se v `nastaveni.xml` zadá jeden náhled (či jiný modul) vícekrát, bude načten pouze jednou. Této vlastnosti se využívá v ukázkovém souboru v příloze (8.2) právě k tomu, aby se na prvním místě zobrazil náhled „Graf“ a až pak ostatní náhledy (které jsou zadány v tagu `adresar`).

Správci modulů

O seznam načtených modulů daného druhu se starají *správci modulů*, tedy třídy: `SpravceNahledu`, `SpravceAlgoritmu`, `SpravceIO` a `SpravceTovaren`. Tyto třídy mají statické metody pro přidání modulů (či celých adresářů), které se využívají při načítání souboru `nastaveni.xml`. Mají také statické metody pro přístup k načteným modulům. Například `SpravceNahledu` obsahuje metodu `vytvorNahled` a metodu `nahledyProTyp`, která vrátí všechny náhledy, které mohou zobrazit zadaný typ animAlg-objektu. `SpravceAlgoritmu` má zase metodu `vytvorAlgoritmus`, které okno *Zadání parametrů* předává vybraný algoritmus a jeho parametry. `SpravceIO` má metodu `uloz`, které se pouze zadá animAlg-objekt a cesta, kam se má uložit; správce pak sám vybere vhodný IO-modul, který provede vlastní uložení. Načítání animAlg-objektů ze souborů funguje obdobně.

Nahrazování náhledů

Samotné prostředí AnimAlg (bez modulů specifických pro teorii automatů) obsahuje jediný náhled (nazvaný „Vlastnosti“), který zobrazuje to, co je všem animAlg-objektům společné, tedy typ, název a popis. To zaručuje, že se v *okně animAlg-objektu* zobrazí vždy aspoň jeden náhled. Pro automaty je ovšem vytvořen náhled (taktéž nazvaný „Vlastnosti“), který krom typu, názvu a popisu zobrazuje také vstupní abecedu automatu. Je zbytečné, aby se uživateli zobrazovaly oba tyto náhledy. Pokud by bylo do AnimAlgu přidáno více náhledů, které by nahrazovaly nějaké jiné, a zobrazovaly by se všechny, bylo by ovládání prostředí nepřehledné. Bylo třeba najít nějaké řešení.

Kdyby uživatel nepracoval s jinými animAlg-objekty než s automaty, mohl by z `nastaveni.xml` vymazat řádek `<trida nazev="animalg.gui.AO_Vlastnosti"/>`, který způsobil načtení prvního náhledu, a nechal by jen `<trida nazev="animalg.automaty.gui.A_Vlastnosti"/>`. Já jsem však zvolil jiné řešení: u náhledu `A_Vlastnosti` je anotace `@Nahrazuje("animalg.gui.AO_Vlastnosti")`. Pokud je zobrazovaný animAlg-objekt automatem, tak `SpravceNahledu` vynechá ze seznamu vhodných náhledů `AO_Vlastnosti`.

5 Závěr

V zadání projektu byl pouze požadavek, aby program animoval algoritmy z teorie automatů a aby bylo možné tyto algoritmy pohodlně doplňovat. Původně jsem předpokládal, že hlavní práce bude spočívat v implementaci základních typů automatů (KA, ZA, TS) a naprogramování algoritmů, které budou „hezky vypadat“. Postupně jsem začal zjišťovat, že by bylo vhodné navrhnout projekt obecněji, modulárně, aby šel lépe přizpůsobit různým účelům.

Pokud by byl projekt zaměřen na užší oblast, než je teorie automatů, byla by jeho implementace v mnohém jednodušší. Například u algoritmů vnitřního třídění je vstupem i výstupem vždy řada čísel a pracuje se s omezeným množstvím objektů (pole, halda). Algoritmy teorie automatů jsou však rozmanitější a vytvoření kvalitního prostředí pro jejich animaci je v tomto ohledu náročnější.

Navrhl jsem tedy grafické prostředí AnimAlg tak, aby umožňovalo animaci algoritmů nejen z teorie automatů, ale z libovolné oblasti. Vše specifické pro danou oblast (algoritmy, náhledy, objekty, se kterými se pracuje,...) se dodává ve formě modulů. Takto si mohou AnimAlg přizpůsobit svým potřebám i uživatelé s různými požadavky.

Jedním z význačných rysů AnimAlgu je, že se při animaci nezobrazuje zdrojový kód algoritmu, ale tzv. *strom událostí*. To může být pro některé animace velmi výhodné: strom událostí je pochopitelný a názorný i bez hlubších znalostí programování; nezobrazují se v něm „technické kroky“, které znesnadňují pochopení algoritmu, ale jsou nutné pro konkrétní implementaci algoritmu nebo pro ovládání doplňkových grafických komponent. Je však nutno dodat, že pro některé jiné animace se strom událostí příliš nehodí. Například u již zmíněného vnitřního třídění by bylo vhodnější zobrazovat zdrojový kód.

Jednotlivé druhy automatů, náhledy i algoritmy z teorie automatů (tedy jádro projektu), jsou naprogramovány v modulech AnimAlgu. I tuto část jsem se snažil navrhnout a implementovat dostatečně obecně a rozšiřitelně. Naprogramoval jsem základ obecného automatu včetně obecné přechodové funkce, věnoval jsem se též otázce nedeterminismu. Nové druhy automatů se do projektu snadno přidávají pomocí objektové dědičnosti.

Nejvíc práce na celém projektu představovalo dosažení vytyčené míry obecnosti a modularity, a to jak v modulech pro teorii automatů, tak v samotném prostředí. Projekt nyní nabízí mnohem více možností, než bylo plánováno v původním návrhu. Zároveň ale obsahuje i některé části, které zůstaly nedodělány nebo jsou implementovány jen jako ukázka. Většinu ovšem mohou doplnit uživatelé ve formě modulů – projekt je na to připraven. Jedná se zejména o doplnění dalších automatů (ZA, 2KA,...) a algoritmů, případně i náhledů. Konkrétně by mohla být přidána do náhledu „Tabulka“ možnost editace, náhled „Graf“ by mohl být rozšířen i pro jiné automaty než konečné a editace by mohla být více „user friendly“ (např. rychlejší zadávání přechodů pomocí myši).

Doufám však, že projekt půjde dobře využívat i v současné podobě (bez nutnosti přidávat nové moduly) a bude užitečný zejména pro studenty teorie automatů.

6 Příloha: Uživatelská příručka

Tato příručka má posloužit jako manuál ovládání prostředí AnimAlg. Doporučuji však přečíst si nejprve kapitolu 1.

6.1 Prostředí AnimAlg

Požadavky

Prostředí AnimAlg je napsáno v jazyce Java (konkrétně JDK 5.0)) a jde používat jako applet i jako samostatný program.

Pro běžné používání AnimAlgu stačí mít na počítači Java virtual machine – JRE (Java 2 Platform Standard Edition Runtime Environment) verze 5.0 či vyšší. Pro rozšiřování AnimAlgu (programování nových modulů) je třeba mít nainstalováno JDK (Java 2 Platform Standard Edition Development Kit), které je stejně jako JRE poskytováno pro všechny systémy zdarma (http://java.sun.com). Doporučené rozlišení obrazovky je 1024×768 a vyšší.

Způsoby spuštění

Prostředí AnimAlg lze spustit několika způsoby. Pro běžné uživatele jsou určeny první dva zmíněné. Všechny zmiňované soubory se nachází na přiloženém CD.

- **Applet**
  Po otevření souboru `index.html` v libovolném prohlížeči podporujícím Java applety se zobrazí prostředí AnimAlg. V této variantě nelze kvůli bezpečnostním opatřením prohlížeče ukládat a načítat soubory. Výchozí velikost appletu je nastavena na 800×600 bodů. Tuto velikost lze změnit editací souboru `index.html`.

- **Program jako `jar`**
  Celý program je v distribuován ve spustitelném archivu `AnimAlg.jar`. Na některých počítačích lze spustit přímo tento soubor. Obecný postup je napsat na příkazovou řádku `java -jar AnimAlg.jar`. K tomu musí být program `java` (interpret JRE) správně nainstalován (obsažen v CLASSPATH).

- **Program z `class` souborů**
  V adresáři `classes` jsou uloženy přeložené třídy (`class` soubory), ze kterých se skládá výše zmíněný archiv. Z tohoto adresáře lze program spustit příkazem `java animalg.Main`. To se hodí zejména, pokud uživatelé přidávají do AnimAlgu nové moduly (aby nemuseli pokaždé vytvářet `jar` archiv).

- **`ant` build**
  Celý program lze také nejprve přeložit ze zdrojových kódů uložených v adresáři `src`. Pro usnadnění je přiložen soubor `build.xml`, takže pokud je v počítači nainstalován program `ant` (http://ant.apache.org), stačí v hlavním adresáři napsat příkaz `ant`.

Nastavení AnimAlgu

Prostředí AnimAlg si mohou uživatelé přizpůsobovat různými způsoby, zejména tím, které všechny moduly do AnimAlgu přidají. Toto nastavení se načítá ze souboru `nastaveni.xml` v adresáři, odkud je program spuštěn. Pokud se tento soubor nenajde nebo není povoleno z něj číst (což je případ appletu), načte se výchozí nastavení.

Jako parametr lze programu AnimAlg při spuštění zadat cestu k souboru s nastavením. Soubor se pak nemusí jmenovat `nastaveni.xml` a může se nacházet v libovolném adresáři.

V příloze 8.2 je uveden ukázkový soubor `nastaveni.xml`, který je vhodné si prohlédnout pro lepší pochopení následujících řádků. V kapitole 4 je popsáno, k čemu slouží jednotlivé druhy modulů (animAlg-objekty, náhledy, algoritmy, IO-moduly a továrny). Zde uvedu pouze několik doplňujících informací a tipů:

- Do kteréhokoliv tagu v konfiguračním souboru lze zadat atribut `basedir`, který udává cestu k *výchozímu adresáři*. Od tohoto tagu dále až do konce souboru už stačí zadávat všechny cesty pouze z tohoto výchozího adresáře, čímž se ušetří zbytečné opisování. Atribut `basedir` lze uvést v `nastaveni.xml` i vícekrát – nový výskyt zruší platnost předchozího.

- Atribut `cesta` v tagu `soubory` udává cestu k adresáři, který se má otevřít, když se v menu AnimAlgu zvolí *Soubor – Otevřít*. V tomto adresáři by tedy měly být uloženy animAlg-objekty a další data. Pokud se atribut `cesta` neuvede, tak se jako výchozí adresář použije domovský adresář uživatele.

- V atributu `otevrit_pri_startu` tagu `soubory` lze zadat název souboru, který se má po spuštění AnimAlgu automaticky načíst na plochu. Tento soubor se hledá v adresáři, který byl uveden v atributu `cesta`.

- Každý druh modulů se zapisuje ve zvláštní sekci, tagy těchto sekcí jsou: `algoritmy`, `nahledy`, `io` a `tovarny`. Všechny moduly lze zadávat dvěma způsoby: jednotlivě či po celých adresářích.

- Pro jednotlivé zadávání modulů je určen tag `trida`, v jehož atributu `nazev` je plně kvalifikovaný název třídy, ve které je uložen příslušný modul. Například `<trida nazev="animalg.automaty.gui.KA_Graf" />`. Cesta ke `class` souboru s daným modulem se v této variantě zadávat nemusí.

- Pro zadávání modulů po celých adresářích je určen tag `adresar`. V atributu `cesta` se zadá cesta k adresáři, který obsahuje vybrané moduly ve formě `class` souborů. V atributu `balik` se zadá balík (package), do kterého všechny přidávané moduly patří. Pokud se v zadaném adresáři nachází i jiné soubory nebo moduly jiného druhu, tak to nevadí – nenačtou se. AnimAlg při načítání modulu totiž vždy kontroluje, zda modul má všechny předepsané vlastnosti.
  Tento způsob se obzvlášť hodí, pokud uživatel přidává nové algoritmy, aby nemusel pokaždé měnit soubor `nastaveni.xml`. Stačí například mít v nastavení řádek `<adresar cesta="animalg\algoritmy" balik="animalg.algoritmy" />` a všechny nově naprogramované algoritmy ukládat do tohoto adresáře (a balíku). Algoritmy se pak automaticky zobrazí v menu AnimAlgu a není třeba žádná další registrace.

- U algoritmů lze navíc v tagu `trida` i `adresar` zadat atribut `kategorie`, který určuje, do jaké kategorie se má algoritmus zařadit. Algoritmy zařazené v nějaké kategorii se pak zobrazují v podmenu s názvem této kategorie. Menu *Algoritmy* tak zůstane přehledné, i pokud se přidá větší množství algoritmů.

Okna AnimAlgu

Prostředí AnimAlg se skládá z menu a z plochy, na kterou lze otevírat různá okna. Základní druhy oken jsou *okno zadání vstupních parametrů*, *okno krokování algoritmu* a *okno animAlg-objektu*. První dvě zmíněná okna jsou popsána v kapitole 1.2, kde jsou zobrazena na obrázcích 4 a 5. *Okno animAlg-objektu* bude popsáno podrobněji nyní.

AnimAlg-objekt je objekt určený pro animaci v AnimAlgu. Všechny dosud naprogramované animAlg-objekty jsou automaty (z teorie automatů). *Okno animAlg-objektu* má nahoře lištu, pomocí níž lze vybírat, který z nabízených náhledů se má v okně zobrazit. Náhledy jsou při otevření okna v panelech (kartách) *okna animAlg-objektu*, ale je možné je oddělit do samostatných oken. K tomu slouží tlačítko, které se zobrazuje napravo od názvu náhledu. Alternativně lze kliknout na název náhledu pravým tlačítkem myši a vybrat *oddělit*.

Menu AnimAlgu

- **Soubor**
  Obsahuje možnosti *Otevřít*, *Uložit* a pak položky, které byly načteny z modulů továren a slouží k vytvoření nových animAlg-objektů.
  Pomocí *Otevřít* se otevírají na plochu animAlg-objekty uložené ve formátu XML. Pomocí *Uložit* se ukládá do souboru animAlg-objekt v aktuálním okně. Pokud je položka *Uložit* neaktivní, je třeba nejprve kliknout na okno animAlg-objektu, který se má uložit. (Pozor: pokud má animAlg-objekt některé náhledy oddělené do vlastních oken, nestačí kliknout na okno náhledu, ale vždy se musí kliknout na okno animAlg-objektu.)
  Pokud je AnimAlg spuštěn jako applet, tak se položky *Otevřít* a *Uložit* nezobrazují.

- **Vzhled**
  V menu *Vzhled* si uživatel může vybrat vhled (tzv. *look and feel*) celého programu: oken, tlačítek, dialogů,... Konkrétní nabídka závisí na JRE – AnimAlg nabídne všechny dostupné (nainstalované) vzhledy. Typicky je k dispozici *Metal*, *Motif* a *Windows* či *GTK*. AnimAlg tedy může vypadat tak jako aplikace na operačním systému, na který je uživatel zvyklý.
  Dále jsou v menu *Vzhled* položky *zobrazit ikony* a *zobrazit názvy*, které určují, co vše se má zobrazovat v liště náhledů *okna animAlg-objektu*. K čemu je to dobré? Pokud se uživateli nevejdou všechny panely náhledů do okna, zobrazí se za posledním náhledem šipky, pomocí kterých lze náhledy posouvat. Pokud si ale uživatel pamatuje ikony jednotlivých náhledů, může se rozhodnout zobrazovat pouze tyto ikony bez názvů, čímž se ušetří místo a lišta s náhledy je tak přehlednější (a vejde se do okna).

- **Algoritmy**
  Položkami v tomto menu jsou jednotlivé algoritmy načtené ve formě modulů.

6.2 Automaty

Náhled Graf

Zobrazuje automat jako orientovaný graf a někdy bývá též označován jako stavový diagram. Současná verze je určena pouze pro konečné automaty deterministické (DKA) i nedeterministické (NKA). Značení bylo vybráno následující: koncové stavy mají dvojité orámování a název tučně; u počátečního stavu se zobrazuje malá šipka; aktuální stav má okraj červený a vychází z něho krátké paprsky[1]. Při provedení přechodu se spojnice daných stavů animuje (ve směru přechodu) pomocí řady červených koleček.

Náhled „Graf“ může zobrazit dva druhy kontextového menu, a to v závislosti na tom, zda se klikne (pravým tlačítkem myši) na některý stav, nebo někam mimo stav. V prvním případě se zobrazí *kontextové menu stavu*, v druhém *kontextové menu automatu*.
Následující přehled uvádí, jak provádět jednotlivé činnosti. Uvádím zde ke každé činnosti jen jeden postup (ten nejjednodušší), ale někdy existují i alternativy pomocí zmíněných kontextových menu.

- **Přesunutí stavu**
  Stav se jednoduše přetáhne myší na zvolené souřadnice.

- **Přidání nového stavu**
  Provede se dvojklik na zvoleném místě na ploše grafu. Automaticky se vygeneruje stav s unikátním názvem („Stav “ a pořadové číslo) a prázdným popisem.

- **Změna vlastností stavu: název, popis, barva, koncový, počáteční**
  Dvojklikem na stav se vyvolá dialog, kde lze všechny uvedené vlastnosti nastavit. Pro editaci popisu je nutné stisknout tlačítko *Editovat*, pro ukončení editace *Uložit* nebo *Storno*. V popisu lze používat HTML značky (tagy). Tyto značky se zobrazují pouze při editaci, jindy se rovnou vykreslí výsledek (např. „ tučně“ se zobrazí jako „**tučně**“).

- **Vymazání stavu**
  V *kontextovém menu stavu* se zvolí *Vymazat stav*.

- **Přidání přechodu**
  Nad stavem, ze kterého má přechod vést, se vyvolá *kontextové menu stavu* a zvolí se *Přidat přechod*. V dialogovém okně *Přidání nového přechodu* se pak z nabídky vybere, do kterého stavu má přechod vést a přes které znaky. Pokud dosud nebyly přidány žádné znaky (nebo obecněji, pokud nebyl přidán ten znak, přes který má přechod vést) musí se nový znak zapsat do kolonky nový znak. K přidávání znaků do abecedy automatu lze ovšem využít i náhled „Vlastnosti“. U DKA se v nabídce znaků zobrazují jen ty, přes které ještě nevede z daného stavu žádný přechod. U NKA se zobrazují v nabídce vždy všechny znaky abecedy a navíc znak λ, který reprezentuje prázdné slovo a používá se v tzv. lambda-přechodech.

[1] Paprsky byly dodány, proto, aby byl aktuální stav vždy snadno nalezitelný. Navíc v některých algoritmech se používá označování stavů různými barvami. Pokud by byl aktuální stav označen červeně, nebylo by bez paprsků poznat, že je aktuální. Délku paprsků si může uživatel nastavit – pokud zadá 0, nebudou se zobrazovat.

- **Vymazání přechodu**
  V *kontextovém menu stavu*, ze kterého daný přechod vede, se zvolí *Vymazat přechod*.

- **Přizpůsobení velikosti**
  Pokud se nevejde celý graf do okna (protože uživatel zmenšil okno), zobrazí se dole či napravo scroll-bar pro posouvání. Pokud si uživatel přesune stavy blíže k sobě (aby byly všechny vidět i ve zmenšeném okně), scroll-bar nezmizí automaticky, ale je třeba v *kontextovém menu automatu* zvolit *Přizpůsobit velikost*.

- **Nastavení zobrazení**
  Když se v *kontextovém menu automatu* zvolí *Nastavení zobrazení*, ukáže se stejně pojmenované dialogové okno. V současné verzi lze v tomto okně zadat maximální velikost stavů v bodech a délku paprsků v bodech. Velikost stavu se vždy přizpůsobuje délce názvu. Některé algoritmy ovšem generují názvy stavů, které mohou být poměrně dlouhé (např. u tzv. složených stavů). Proto je výhodné nastavit maximální velikost stavu, aby byl graf stále přehledný. Když se název do maximální velikosti nevejde, zobrazí se jen jeho část a tři tečky. Celý název i popis stavu se zobrazuje při najetí myši nad stav jako tooltip.

Náhled Seznam

Zobrazuje seznam přechodů libovolného automatu a zároveň umožňuje i základní editaci automatu. Všechny editace se provádí přes kontextové menu (nad zvoleným přechodem se klikne na pravé tlačítko myši). Tato editace se hodí především pro úpravy Turingových strojů, protože pro konečné automaty je uživatelsky příjemnější editace v náhledu „Graf“.

Náhled Tabulka

Je dostupný pro všechny automaty, ale neumožňuje editaci. Ve sloupcích se zobrazují znaky abecedy, v řádcích stavy a v buňkách přechody, které vedou z daného stavu přes daný znak. V prvním sloupci jsou názvy stavů, přičemž před počátečním stavem se vykresluje šipka, koncové stavy jsou zobrazeny tučně a za aktuálním stavem je černé kolečko. Pokud je stav označen, orámuje se jeho název příslušnou barvou. Pokud je označen přechod, zobrazí se v příslušné buňce tabulky čtvereček v barvě označení. Při provedení přechodu se příslušná buňka na půl sekundy červeně orámuje.

Náhled Vlastnosti

Umožňuje editovat název, popis a abecedu automatu. Zobrazuje se i typ automatu, ale ten editovat nelze. Editace názvu automatu se potvrdí stisknutím klávesy Enter. Název automatu se zobrazuje na různých místech, např. u náhledů, které jsou otevřeny v samostatných oknech, takže je vhodné volit ho stručný a podrobnosti umístit do popisu. Popis automatu se edituje stejným způsobem jako popis stavu v náhledu „Graf“. Tlačítkem *Přidej znak* se přidá do abecedy nový znak, ke kterému je možné připojit i popis. Pokud je stisknuto Ctrl či Shift, lze myší vybrat více znaků najednou a ty pak smazat tlačítkem *Smaž označené*.

Náhled Simulace

Slouží k simulaci (krokování) výpočtu libovolného automatu. Náhled je graficky rozdělen na tři oblasti: horní, střední a spodní.

Ve spodní oblasti je obrázek žárovky, která se rozsvítí vždy, když automat přijímá. Také se zde nachází popis aktuální konfigurace automatu.

Ve střední oblasti je vstupní páska. Uprostřed nad páskou je schematicky nakreslena čtecí (případně i zapisovací) hlava. Nad čtecí hlavou je sada tlačítek. V odděleném rámečku jsou tlačítka, pomocí kterých lze zapisovat znaky abecedy na konec pásky. Ostatní tlačítka jsou označeny ikonami s významy: uložit pásku do souboru, načíst pásku ze souboru, smazat celou pásku, smazat poslední znak na pásce. Také je zde umístěno tlačítko, kterým si uživatel může zvolit, zda se mají mezi znaky na pásce zobrazovat mezery.

V horní oblasti jsou tlačítka pro vlastní krokování: *Reset*, *Krok* a *Zpět* (podrobný popis jejich funkce je uveden na stranách 16 - 19). Pokud si uživatel zaškrtne volbu *Dle pásky*, tak se při přidání znaku na pásku automaticky provede krok. Tato volba se hodí obzvlášť pro DKA. Pokud je zaškrtnuta volba *Smazat*, tak se při každém resetu automatu smaže páska. Tyto dvě volby tedy slouží pouze pro zpříjemnění práce s náhledem, aby uživatel nemusel zbytečně klikat.

Pokud je simulovaný automat nedeterministický, tak se mezi střední a spodní oblastí ještě objeví seznam možných přechodů (grafická reprezentace orákula). Z těchto přechodů si uživatel vybírá ten, který se má provést v příštím kroku.

7 Příloha: Programátorská příručka

Podrobná programátorská dokumentace je k dispozici ve formátu JavaDoc na přiloženém CD. Zdrojové kódy již naprogramovaných modulů (obzvláště algoritmů) obsahují komentáře a jsou vhodnou inspirací k psaní vlastních modulů. Mnoho informací důležitých pro programátory již bylo uvedeno v kapitolách 2 a 3. Tyto informace zde nebudu opakovat a v této příručce uvedu pouze odpovědi na několik otázek, které by mohly vzniknout při přidávání modulů nových modul do AnimAlgu.

Jak iterovat?

Přes mnoho objektů v AnimAlgu lze iterovat (např. množina stavů či znaků, odchozí přechody ze stavu atd.). Například pro vypsání všech stavů automatu stačí následující kód.

```
I_MnozinaStavu mnozinaStavu = automat.getMnozinaStavu();
for (Stav stav : mnozinaStavu) System.out.println(stav);
```

Při iterování přes množinu stavů (ale i přes ostatní datové struktury implementující rozhraní `Collection`) se nesmí množina měnit jinak než pomocí metod iterátoru, jinak může dojít k výjimce `ConcurrentModificationException`. Co to znamená vysvětlím na příkladech. Pro smazání jednoho prvku z množiny slouží metoda `remove(Object)`, pro smazání všech prvků metoda `clear()`, pokud ale chceme smazat pouze prvky s nějakou vlastností, musíme přes množinu iterovat. Například algoritmus „Smaž neoznačené“ smaže z automatu stavy, které nejsou označeny nějakou barvou (metoda `Stav.oznacen()` vrátí `false`). Následující kód je chybný, neboť se v něm volá `remove(stav)` během iterování, a tak dochází k uvedené výjimce.

```
for (Stav stav : mnozinaStavu)
  if (!stav.oznacen()) mnozinaStavu.remove(stav);
```

Pokud potřebujeme při iterování pouze odebírat z množiny prvek, přes který právě iterujeme, je nejlepší použít metodu `Iterator.remove()`:

```
for (Iterator<Stav> it = mnozinaStavu.iterator(); it.hasNext(); )
   if (!it.next().oznacen()) it.remove();
```

Lze také mazat při iterování přes kopii množiny stavů. V této variantě by šlo množinu měnit i jinak, třeba do ní přidávat prvky.

```
Stav[] kopie = mnozinaStavu.poleStavu();
for (Stav stav : kopie)
  if (!stav.oznacen()) mnozinaStavu.remove(stav);
```

Vše uvedené platí i pro téměř všechny kolekce z balíku `java.util`. Pouze místo metody `poleStavu()` se použije `toArray(new Stav[0])`.

Jak jsou implementovány množiny?

Množinám je věnována kapitola 2.3. Pro rozhraní `I_SetridenaMnozina` jsou k dispozici tři implementace:

- `ArraySet<E>` je implementačně nejjednodušší. Využívá `ArrayList<E>` jako delegáta, takže přímý přístup (`getElementAt(index)`) je maximálně rychlý, ale ostatní operace spíše pomalé: `contains` má *o(log n)* (binary search), při `add` a `remove` se kopírují části pole (tedy *o(n)*, ale s relativně nízkou multiplikativní konstantou).

- `AvlSet<E>` je AVL-strom, který je „prošívaný“, tedy iterování je rychlé (jako u spojového seznamu). Uzly si místo indexu pamatují tzv. *relativní index*, tedy o kolik se jejich index liší od indexu rodičovského uzlu. Tak je zajištěno indexování (přímý přístup) při zachování operací `contains`, `getElementAt`, `add` i `remove` v *o(log n)*. Oproti červeno-černým stromům potřebuje `add` a `remove` v AVL více rotací (*o(log n)* oproti 3), ale vyhledávání je pak většinou o něco rychlejší (1.4*log(n) oproti 2*log(n)).

- `AvlHashSet<E>` je potomkem `AvlSet<E>` a přidává k AVL-stromu hashovací tabulku. Prvky se ukládají do tabulky i do AVL-stromu (v Javě se ukládá pouze reference na objekt, takže se neplýtvá zbytečně pamětí). AVL-strom zajišťuje operace `getElementAt`, iterování a `toArray`. Tabulka zajišťuje operaci `contains`.

Při vytváření nových modulů (především nových automatů) si mohou programátoři vybrat jednu z uvedených tří implementací nebo mohou naprogramovat vlastní implementaci a zadat ji v konstruktoru třídě `SetridenaMnozina`.

Proč Stav neimplementuje rozhraní `Comparable`?

Bylo by určitě výhodné mít stavy porovnatelné dle jejich názvů, a to přímo pomocí rozhraní `Comparable`, které je k tomu určeno. Zejména `MnozinaStavu`, respektive `AVLSet<Stav>`, potřebuje stavy takto porovnávat a pokud by už stavy implementovaly `Comparable`, tak se v konstruktoru nemusí zadávat zvláštní `Comparator`. V Java specifikaci se ovšem důrazně doporučuje, aby relace daná implementací `Comparable` byla konzistentní s `equals`, tedy `(a.compareTo(b) == 0)` $\Longleftrightarrow$ `(a.equals(b))`. Pokud by se ale předefinovala metoda `equals` tak, aby stavy se stejnými názvy byly ekvivalentní, pak by se musela předefinovat i metoda `hashCode` tak, aby stavy se stejnými názvy měly stejný hash-code (to už není doporučení, ale nutnost). A to je problém, protože název stavu se může měnit a mnoho kódu používá `HashSet<Stav>` či `HashMap<Stav, cosi>` s předpokladem, že hash-code stavů za běhu se nemění. Všechny takové tabulky by musely poslouchat listener, zda se nezměnil název stavu, i když je jinak vůbec název stavu nezajímá. Psaní algoritmů využívajících `HashSet<Stav>` by tak bylo o hodně složitější.

Stav tedy neimplementuje rozhraní `Comparable` a místo toho je poskytnut komparátor `Stav.nazevComparator`. Díky tomu hashování stavů funguje jednoduše (nemusí se měnit se změnou názvu), jen je při psaní algoritmů třeba rozlišovat, zda jsou stavy totožné (`==`), nebo mají jen stejný název.

Co to jsou *bound properties*?

Bound properties je termín z architektury JavaBeans. Ve zkratce: jsou to pojmenované atributy, jejichž změna se oznamuje registrovaným `PropertyChangeListener`ům. Rozhraní `PropertyChangeListener` má metodu `propertyChange(PropertyChangeEvent)`, která dostane v parametru objekt, jenž má metody pro získání staré i nové hodnoty atributu (`getOldValue` a `getNewValue`).

Jak se užívají *bound properties* v AnimAlgu?

Tato technika se v AnimAlgu používá například pro oznamování událostí při změně názvu či popisu. Dobře se hodí i pro oznamování některých událostí u automatů: property „přijímá“ (`boolean`) a „popis konfigurace“ (`String`). Pro události „přechod“ a „reset“ jsem se rozhodl také použít tuto techniku, byť se nejedná o pravé properties – v JavaDoc dokumentaci je proto nazývám pseudo-properties. Při události „reset“ se nepředávají žádné parametry, ale při událost „přechod“ je třeba předat informace o tom, jaký přechod se použil. Tuto informaci vrací metoda `getNewValue`.

Jak se používá rozhraní `PropertyListener`?

Na některé objekty v AnimAlgu lze registrovat `PropertyListener`. Rozhraní `animalg.util.PropertyListener` je rozšířením `java.beans.PropertyChangeListener` a obsahuje navíc dvě metody, díky kterým je použití v mnohém jednodušší:

- metoda `getPropertyName` vrátí název property, jejíž změna se má oznamovat. Pokud se má oznamovat změna všech properties, tak musí tato metoda vrátit `null`. Díky této metodě je oznamování událostí rychlejší, protože se oznamují jen těm listenerům, které o ně opravdu mají zájem.

- metoda `oznamovatVEventDT` vrátí `true`, pokud se má oznámení události přeplánovat do tzv. *event-dispatching thread*, což je vlákno, které jako jediné smí přistupovat k metodám knihovny Swing (GUI). AnimAlg používá více vláken (krokování algoritmu probíhá v samostatném vlákně), a tak je třeba dbát na zásady thread-safety.
  Pokud například algoritmus změní název stavu, tak se vyvolá událost, která je oznámena všem listenerům registrovaným na daném stavu pro bound property „název“. Mezi těmito listenery jsou i náhledy, které potřebují překreslit název na obrazovce. Pokud by se však použil normální `PropertyChangeListener`, bude událost oznámena ve vlákně algoritmu a náhledy by musely kreslení na obrazovku přeplánovat do *event-dispatching thread*. Programování takových náhledů by pak bylo zbytečně zdlouhavé a snadno by se mohlo na přeplánování zapomenout. Při použití `PropertyListener`u stačí pouze, aby metoda `oznamovatVEventDT` vrátila `true` a přeplánování se provede automaticky. Kód náhledů je tak přehlednější.

Jak ošetřovat nestandardní situace v algoritmech?

Mějme algoritmus `KA_Zretezeni`, který dostane na vstupu dva konečné automaty (*A1* a *A2*) a na výstupu vytvoří automat, který je jejich zřetězením (přijímá jazyk *L(A1).L(A2)*). Aby bylo možné algoritmus provést, musí mít vstupní automaty stejnou abecedu. Tato informace je napsána v popisu algoritmu, ale není implementován

žádný mechanismus, který by zabránil spuštění algoritmu s automaty, které mají rozdílné abecedy. Algoritmus tedy po spuštění nejdříve zkontroluje, zda jsou abecedy stejné, a pokud nikoli, tak vyvolá událost „Konec – Chyba: rozdílné abecedy“ a skončí.

Pokud by byl tento algoritmus spuštěn přímo uživatelem, tak je popsané řešení dostatečné – událost se vypíše v okně *Krokování algoritmu* a nezobrazí se žádný výsledný automat. Pokud je ale tento algoritmus spuštěn jako podprogram jiného algoritmu, který má s výsledným zřetězeným automatem dále pracovat (předat ho jako vstup dalšímu podalgoritmu), je popsané řešení nedostačující. Výpočet pokračuje i poté, co byla vypsána událost "Konec - Chyba: rozdílné abecedy" a obvykle dochází k dalším chybám. Proto je doporučeno v případě chyby vyvolat v algoritmu vhodnou výjimku. Dále je doporučeno před vyvoláním výjimky vyvolat událost s popisem té chyby (událost by měla mít nastaven příznak „nezastavovat“, tedy měla by být vyvolána metodou `jen`).

Vyvolaná výjimka může být odchycena (vnějším algoritmem, tedy tím, který podprogram spustil), nebo bude zobrazena uživateli v chybovém hlášení a zároveň vypsána na chybovou konzoli spolu s dalšími údaji o výjimce.

Programátor algoritmů při „nestandardních situacích“, tedy musí zvážit, zda vyvolat výjimku, nebo jen vyvolat událost s příslušným popisem a ukončit algoritmus. Například algoritmus `OznacDosazitelne` označí všechny stavy dosažitelné z počátečního stavu (v jiných variantách může hledat i stavy dosažitelné ze zadaných stavů, ale to je teď nepodstatné). Pokud automat nemá žádný počáteční stav, považuji za nejlepší řešení nevyvolávat výjimku, ale vyvolat událost, která uživateli oznámí tuto nestandardní situaci. Pokud totiž není žádný počáteční stav, tak nejsou žádné stavy dosažitelné, a algoritmus dává korektní výsledek.

Jak vyvolávat události?

Není třeba se bát přidat do algoritmu více událostí. Pokud jsou události správně (logicky) strukturované, tedy zanořené do sebe, a rozumně okomentované, nelze jejich přidáním nic zkazit. Pokud uživateli algoritmu přijde nějaký krok jasný (a komentáře k podkrokům zbytečně podrobné), může ho celý přeskočit pomocí *Step over*, a pokud už je uvnitř takového kroku, může se dostat o úroveň výš pomocí *Step out*.

Není třeba se bát psát popisy událostí podrobné. Název události by měl být hlavně stručný, aby byl strom událostí přehledný. Naopak popis by měl být co nejpodrobnější (samozřejmě v rozumné míře). Místo, kde se popis zobrazuje si uživatel může zvětšit i v něm scrollovat. Události se často vypisují v cyklech, kdy jeden druh události (jedno místo v kódu) se zobrazuje ve stromě vícekrát. Většinou si uživatel přečte celý popis pouze poprvé a při dalších výskytech sleduje už je název, případně, co se v popisu změnilo.

S tím souvisí další doporučení. Autor algoritmů by měl do popisu události (případně i do názvu, pokud to bude srozumitelné) dát co nejkonkrétnější údaje. Tedy místo `je("Nový stav","Byl přidán nový stav.")` psát `je("Nový stav", "Do automatu byl přidán stav " + stav + ", protože ...")`. Pokud se například v cyklu přidávají nové stavy, může být vhodnější dát jako název události rovnou název stavu, tedy `je(stav.getNazev(), "...")`. Přesto, že více událostí je vyvoláno stejným místem v kódu, každá může mít jiný popis.

8 Příloha: Dodatky

8.1 Definice automatů

zkratka	**det.**	**název**
DKA	D	(deterministický) konečný automat
MOORE	D	Mooreův stroj (výstup ve stavu)
MEALY	D	Mealyho stroj (výstup při přechodu)
2KA	D	dvousměrný (dvoucestný) konečný automat
NKA	N	nedeterministický konečný automat
ZNKA	N	zobecněný NKA (má i lambda přechody)
NZA	N	(nedeterministický) zásobníkový automat
DZA	D	deterministický zásobníkový automat
TS	D	Turingův stroj

zkratka	**definice**	přechodová funkce δ
DKA	(Q, X, δ, q_0, F)	$Q \times X \to Q$
MOORE	$(Q, X, Y, \delta, Q \to Y, q_0, F)$	$Q \times X \to Q$
MEALY	$(Q, X, Y, \delta, Q \times X \to Y, q_0, F)$	$Q \times X \to Q$
2KA	(Q, X, δ, q_0, F)	$Q \times X \to Q \times \{-1, 0, +1\}$
NKA	(Q, X, δ, Q_0, F)	$Q \times X \to P(Q)$
ZNKA	(Q, X, δ, q_0, F)	$Q \times (X \cup \{\lambda\}) \to P(Q)$
NZA	$(Q, X, Z, \delta, q_0, z_0, F)$	$Q \times (X \cup \{\lambda\}) \cup Y \to P_{FIN}(Q \times Y^*)$
DZA	$(Q, X, Z, \delta, q_0, z_0, F)$	$Q \times (X \cup \{\lambda\}) \cup Y \to P_{FIN}(Q \times Y^*)$
TS	(Q, X, δ, q_0, F)	$(Q \setminus F) \times X \to Q \times X \times \{-1, 0, +1\}$

Q – *množina stavů*
X – *(vstupní) abeceda*
δ – *přechodová funkce*
Y – *výstupní abeceda*
Z – *množina zásobníkových symbolů*
$P()$ – *potenční množina (mn. všech podmnožin)*
P_{FIN} – *množina všech konečných podmnožin*
* – *operace obecné iterace jazyků (Kleene star)*

$F \subseteq Q$ – *množina koncových stavů*
$q_0 \in Q$ – *počáteční stav*
$Q_0 \subseteq Q$ – *množina počátečních stavů*
λ – *prázdné slovo*
$z_0 \in Z$ – *počáteční zásobníkový symbol*

DZA je definován jako NZA, ale s omezením přechodové funkce tak, aby byla vždy nanejvýš jedna možnost, jak pokračovat ve výpočtu. Tedy musí být splněny podmínky:

- $\forall q \in Q, \forall z \ \in Z, \forall x \in (X \cup \{\lambda\})$ platí $|\delta(q, x, z)| \leq 1$
- $\forall q \in Q, \forall z \ \in Z$ platí $\delta(q, \lambda\, z) \neq \emptyset \Rightarrow (\forall x \in X : \delta(q, x, z) = \emptyset)$

8.2 Ukázka formátu souborů

automat.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<animAlgObjekt typ="deterministický konečný automat" nazev="Dělitelnost třemi"
       popis="Automat přijímá, je-li součet vhozených mincí dělitelný třemi.">
  <abeceda>
    <znak nazev="1" popis="1 Kč"/>
    <znak nazev="2" popis="2 Kč"/>
    <znak nazev="3" popis="3 Kč"/>
  </abeceda>
  <stavy>
    <stav nazev="Z 0" popis="Zbytek po dělení 3 je 0." x="126" y="11"
          pocatecni="ano" koncovy="ano"  />
    <stav nazev="Z 1" popis="Zbytek po dělení 3 je 1." x="4"   y="123" />
    <stav nazev="Z 2" popis="Zbytek po dělení 3 je 2." x="138" y="134" />
  </stavy>
  <funkce>
    <prechod odkud="Z 0" znaky="1"    kam="Z 1" />
    <prechod odkud="Z 0" znaky="2, 3" kam="Z 2" />
    <prechod odkud="Z 1" znaky="1"    kam="Z 2" />
    <prechod odkud="Z 1" znaky="2, 3" kam="Z 0" />
    <prechod odkud="Z 2" znaky="2, 3" kam="Z 1" />
    <prechod odkud="Z 2" znaky="1"    kam="Z 0" />
  </funkce>
</animAlgObjekt>
```

nastaveni.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<nastaveni verze="1.0">
  <soubory cesta="data" />
  <moduly basedir="classes">
    <algoritmy>
      <adresar cesta="animalg\automaty\algoritmy"
               balik="animalg.automaty.algoritmy" />
    </algoritmy>
    <nahledy>
      <trida nazev="animalg.automaty.gui.KA_Graf" />
      <adresar cesta="animalg\automaty\gui" balik="animalg.automaty.gui" />
      <trida nazev="animalg.gui.AO_Vlastnosti" />
    </nahledy>
    <io>
      <trida nazev="animalg.automaty.modely.AutomatIO" />
    </io>
    <tovarny>
      <trida nazev="animalg.automaty.modely.TovarnaPrazdne" />
    </tovarny>
  </moduly>
</nastaveni>
```

8.3 Obsah CD

Na kompaktním disku, který je součástí této bakalářské práce, jsou následující adresáře a soubory.

`data`	adresář s ukázkovými automaty a páskami
`classes`	adresář s přeloženými zdrojovými kódy (`*.class`)
`javadoc`	adresář s vygenerovanou programátorskou dokumentací
`src`	adresář obsahující všechny zdrojové kódy
`AnimAlg.jar`	spustitelný program (prostředí AnimAlg včetně modulů)
`bakalarska.pdf`	tato bakalářská práce
`build.xml`	build skript pro program `ant`
`index.html`	HTML soubor obsahující AnimAlg jako applet
`nastaveni.xml`	konfigurační soubor pro `AnimAlg.jar`
`spust.bat`	soubor s příkazem `java -jar AnimAlg.jar`

Podrobné pokyny ke spuštění a obsluze programu jsou uvedeny v uživatelské příručce (kapitola 6).

Literatura

[1] Barták R.: *Automaty a gramatiky*, [online, cit. 29. dubna 2007]. Dostupné na `http://ktiml.ms.mff.cuni.cz/~bartak/automaty`.

[2] Chytil M.: *Automaty a gramatiky*, SNTL, Praha, 1984.

[3] Wikipedia: *Model-view-controller* [online, cit. 24. května 2007]. Dostupné na `http://en.wikipedia.org/wiki/Model-view-controller`.

[4] Apache Commons Collections [online, cit. 24. května 2007]. Dostupné na `http://commons.apache.org/collections/`.