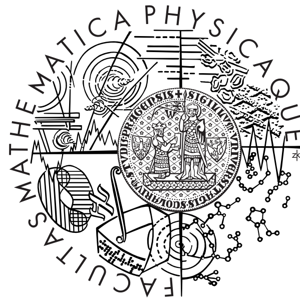
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA



Vojtech Bardiovský

Generovanie seba-replikujúcich celulárnych automatov

Katedra teoretické informatiky a matematické logiky

Vedúci bakalárskej práce: RNDr. Pavel Surynek, Ph.D.
Študijný program: obecná informatika

2010

Ďakujem vedúcemu tejto práce, RNDr. Pavlovi Surynekovi, Ph.D., za jeho pomoc a veľmi podnetné nápady a pripomienky, ktoré usmernili vývoj programu a prácu samotnú.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním.

V Prahe dňa 10.05.2010 Vojtech Bardiovský

Obsah

1 Celulárne automaty - teória **7**
- 1.1 Úvod 7
- 1.2 Celulárny automat 8
- 1.3 Replikácia 10

2 Celulárne automaty - implementácia **14**
- 2.1 Úvod 14
- 2.2 Hash-life 14
- 2.3 Rule Table - prechodové funkcie zadávané tabuľkou 15

3 Generovanie prechodových funkcií **16**
- 3.1 Tempesti 16
- 3.2 Nosičové stavy 19
  - 3.2.1 Počet stavov 20
  - 3.2.2 Beh automatu 20
  - 3.2.3 Pravidlá 21
- 3.3 Komprimácia stavov 28
- 3.4 Implementácia 32

4 Príklady generovaných automatov **34**
- 4.1 Nosičová metóda 34
- 4.2 Nosičová metóda pre viacero cieľových stavov 36
- 4.3 Komprimačná metóda 39
- 4.4 Záver 42

A FCAS - uživateľská dokumentácia **43**
- A.1 Úvod 43
- A.2 Rýchly návod 44
  - A.2.1 Rules - zadávanie prechodových funkcií 44

A.2.2 Editácia prechodovej funkcie Rule Tables 45
A.2.3 Nový automat 45
A.2.4 Načítanie a uloženie automatu 45
A.2.5 Základné ovládacie prvky 46
A.2.6 Knižnica konfigurácií - Pattern Library 46
A.2.7 Zmena stavu - State picker 47
A.2.8 Náhodné rozloženie stavov 47
A.3 Používanie algoritmu Hash-life 48
A.4 Podporované formáty 48
A.4.1 RLE 48
A.4.2 MCL 49
A.4.3 TABLE 50

B FCAS - programová dokumentácia 53
B.1 Návrh - UML 53
B.2 Automaton - automat 56
B.3 RuleComputer - prechodová funkcie 56
B.4 Hash-life 57
B.5 Infonode DockingWindows 58
B.6 Monitor 59
B.7 Ošetrenie chýb 59

Názov práce: Generovanie seba-replikujúcich celulárnych automatov
Autor: Vojtech Bardiovský
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedúci bakalárskej práce: RNDr. Pavel Surynek, Ph.D.
E-mail vedúceho: pavel.surynek@mff.cuni.cz

Abstrakt: Trieda seba-replikujúcich celulárnych automatov je zaujimavá najmä tým, že demonštruje schopnosť jednoduchých prostredí vytvárať štruktúry schopné seba-replikácie. Okrem vytvorenia svojej kópie dokáže vhodne nadefinovaný celulárny automat vytvoriť počas svojho životného cyklu aj dodatočné štruktúry alebo konfigurácie automatu. Cieľom práce je vytvoriť prostredie pre pozorovanie takýchto automatov, teda prostredie dostatočne flexibilné a hlavne schopné počítať veľmi rýchlo, keďže niektoré celulárne automaty potrebujú pre replikáciu rádovo desattisíce až bilióny diskrétnych prechodov. Takéto prostredie umožní návrh a implementáciu všeobecnej Tempestiho slučky, čo je ďalším cieľom práce. Výsledkom bude rozšírenie Tempestiho slučky tak, aby bolo možné automatizovane generovať prechodové funkcie a počiatočné konfigurácie pre danú konfiguráciu, ktorú by mal automat mimo svojej replikácie vytvárať.

Kľučové slová: replikácia celulárne automaty

Title: Generation of self-replicating cellular automata
Author: Vojtech Bardiovský
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: RNDr. Pavel Surynek, Ph.D.
Supervisor's e-mail address: pavel.surynek@mff.cuni.cz

Abstract: The family of self-replicating cellular automata is interesting mainly for being able to demonstrate that even simple environments can make rise to structures capable of self-replication. Besides creating its own copy, a purposedly designed automaton can produce additional side patterns during its lifetime. The aim of the work is to create a cellular automata simulation environment that is flexible and fast, as some cellular automata become interesting only after thousands or millions of steps. The second aim of the work is to design and implement a generalisation of the Tempesti's loop using this environment. The outcome of the work is a generalisation that allows for automatized creation of rules and patterns for a given side pattern.

Keywords: self-replication cellular automata

Kapitola 1

Celulárne automaty - teória

1.1 Úvod

Celulárne automaty sú dlho skúmanou oblasťou informatiky a nachádzajú využitie v mnohých neinformatických vedách, najmä v modelovaní fyzikálnych, chemických alebo sociálnych javov a všeobecne veľkých dynamických systémov. Sú používaným prístupom v skúmaní umelej inteligencie, ale aj napríklad kryptografie.

Zaujímavou triedou celulárnych automatov sú automaty schopné seba-replikácie, teda automaty, ktoré počas svojho behu vytvárajú svoje kópie. Existujú návrhy seba-replikujúcich automatov rôznych veľkostí, periód replikácie a takisto aj samotných princípov replikácie. Tieto automaty boli prvýkrát skúmané John Von Neumannom, ktorý sa snažil o napodobnenie biochemických dejov v genetike. Automaty mali slúžiť ako abstrakcia alebo akési jadro seba-replikácie, ktoré bolo možné skúmať samé o sebe, bez biochemického kontextu.

Špeciálnou kategóriou sú celulárne automaty, ktoré popri svojej replikácii vytvárajú štruktúry rôznych zložitostí. Túto oblasť ako prvý začal skúmať Tempesti[9] a vo svojej práci uverejnil automat, ktorý vytváral počas replikácie jednoduchú štruktúru ako vedľajší produkt behu automatu. Výsledkom jeho práce je automat, Temepestiho slučka, ktorý dokáže datové stavy kopírovať a uchovávať alebo interpretovať podľa potreby. Týmto splnil Von Neumannovu predstavu o seba-replikujúcom automate, ktorý, podobne ako v genetických dejoch, má dva rôzne pohľady na dáta a to transkripčný a translačný.

Zaujímalo nás, či by bolo možné navhrnúť automat, ktorý by podobne

ako Tempestiho slučka vytváral danú konfiguráciu, ale bez obmedzení, ktoré Temepestiho spôsob prináša. Návrh automatu, ktorý vytvorí konkrétnu štruktúru je veľmi pracný a to z dvoch dôvodov. Prvý problém je technický a je ním neprehľadný zápis prechodovej funkcie automatu (Tempestiho slučka má 1026 pravidiel pre diskrétne prechody). Druhým problémom je, že neexistuje ucelený princíp tvorby automatu, ktorý by bolo možné rozšíriť na vytvorenie ľubovoľnej štruktúry.

V práci predstavíme možné princípy behu takýchto automatov a tieto princípy aj implementujeme, aby bolo možné vytvoriť pravidlá a konfiguráciu pre ľubovoľný automat.

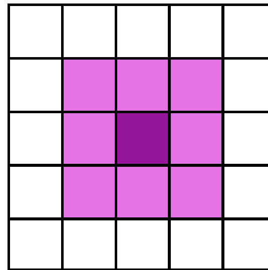
1.2 Celulárny automat

Celulárny automat je množina konečných automatov nazývaných bunkami. Bunky môžu byť usporiadané do rôznych štruktúr, základná teória pracuje s celulárnym automatom usporiadaným do ortogonálnej mriežky. Susedia bunky sú definovaní typom okolia:

Moore definuje susedov bunky ako všetkých osem buniek naokolo v klasickom chápaní mriežky. Počet susedov je daný

$$|N_{Moore,r}| = (r^2 + 1)^2 - 1$$

kde r je polomer (veľkosť) okolia.

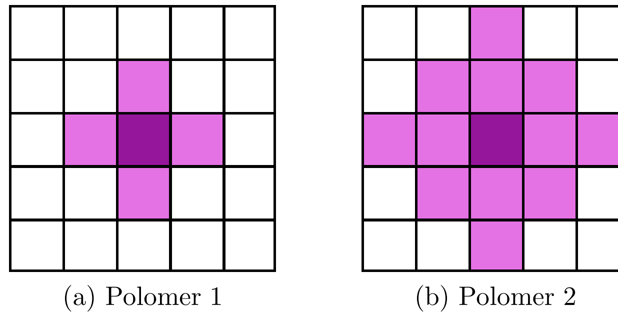


Obr. 1.1: Moorovo okolie s polomerom 1.

Von Neumann definuje susedov ako štyri okolité bunky v základných smeroch a to hore, dole, vľavo a vpravo. Počet susedov je daný

$$|N_{Neumann,r}| = 2r(r + 1)$$

.



Obr. 1.2: Von Neumannovo okolie s polomermi 1 a 2.

Každá bunka x automatu sa nachádza v určitom stave $s(x) \in S \subset \mathbf{N}$. Množina možných stavov S je pre všetky bunky automatu rovnaká.

Činnosť automatu prebieha po diskrétnych krokoch. V každom kroku sa pre každú bunku vypočíta jej nový stav podľa prechodovej funkcie celulárneho automatu. Prechodová funkcia je funkciou stavov susedov bunky a stavu samotnej bunky:

$$f\colon S^{k+1} \to S$$

kde $k = |N_r|$ je počet susedov bunky.

Príklad: Conway's game of life je automat usporiadaný do ortogonálnej mriežky s Moorovým okolím o polomere 1. Automat má dva stavy, bunka môže byť *mŕtva* alebo *živá*. Prechodová funkcia celulárneho automatu je definovaná počtom živých $n_1(x)$ a mŕtvych $n_0(x)$ buniek v susedstve bunky x (platí $n_1(x) + n_0(x) = 8$). Funkcia je definovaná nasledovne:

$$f(N(x)) = \begin{cases} 1 & \text{pre } n_1(x) \in \{2,3\} \text{ a } s(x) = 1 \text{ alebo } n_1(x) = 3 \text{ a } s(x) = 0 \\ 0 & \text{inak} \end{cases}$$

Je zvykom prechodovú funkciu definovať pre lepšiu zrozumiteľnosť slovne. Automat typu *Conway's game of life* by bol takto definovaný nasledovne:

- Ak má živá bunka v okolí dve alebo tri živé bunky, prežíva.
- Ak má mŕtva bunka v okolí tri živé bunky, ožíva.
- Inak bunka zomiera, alebo ostáva mŕtva.

Spôsobov ako definovať prechodovú funkciu automatu je viacero a prechodové funkcie, ktoré majú spoločné parametre sa spájajú do rodín ako

napr. *Life*, *Generations*, *Cyclic CA* a pod.

V texte budeme používať pojem **konfigurácia**, čo je podmnožina buniek automatu spolu s ich stavmi.

1.3 Replikácia

Skúmanie seba-replikujúcich celulárnych automatov začal J. von Neumann [5] v roku 1966. Skúmal možnosť zostrojenia tzv. univerzálneho konštruktoru (angl. universal constructor), ktorý by bol schopný vytvárať ľubovoľné konfigurácie. Výsledkom jeho práce bol abstraktný návrh automatu so všetkými jeho súčasťami. 29-stavový automat bol prvýkrát implementovaný dvojicou Nobili, Pesavento v roku 1995 [7][6].

V roku 1968 predstavil E. Codd[2] vo svojej práci seba-replikujúci automat, ktorý zachovával vlastnosti univerzálneho konštruktoru. Počet stavov sa mu podarilo znížiť na 8, ale veľkosť automatu resp. počet jeho počiatočne aktívnych buniek sa zvýšil rádovo 100 až 1000-násobne.

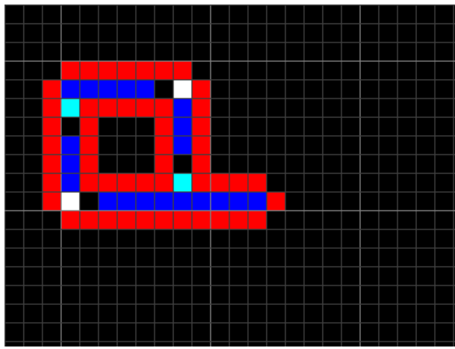
V roku 1984 predstavil prvý jednoduchý seba-replikujúci automat C. Langton[4]. Nadväzuje na prácu Codda, ale upúšťa od univerzálneho konštruktoru a cieľom jeho práce je implementácia netriviálneho, ale jednoduchého seba-replikujúceho automatu. Langton ukázal, že pri seba-replikujúcich automatoch existuje istý definičný problém. Pred uverejnením jeho práce bola seba-replikácia tesne spojená s schopnosťou univerzálnej konštrukcie, aby sa vyhlo triviálnym prípadom. Napríklad jednorozmerný dvojstavový automat s jednou bunkou v stave 1 a prechodovou funkciou

$$f(s_l, s_p) = (s_l + s_p) \bmod 2 \quad \text{kde } s_l \text{ a } s_p \text{ sú susedia bunky.}$$

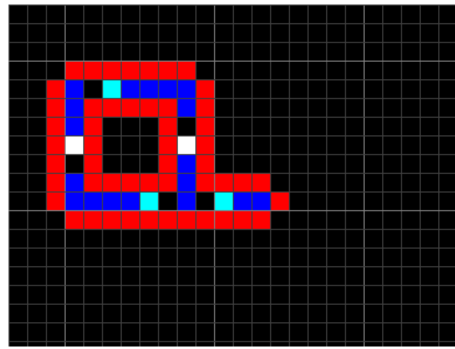
V takomto automate iste nastáva replikácia pôvodnej konfigurácie, ale bolo by nezmyselné nazývať ho seba-replikujúcim automatom, pretože od takého automatu očakávame, laicky povedané, zložitejší chod. Langton sa snaží o vysvetlenie tohto javu a novú definíciu napríklad argumentom, že replikácia u tohoto automatu vychádza priamo z vlastností prechodovej funkcie a nie z konfigurácie, ktorá by mala byť "aktívne zapojená" do svojej replikácie. Táto neúplnosť definície seba-replikujúceho automatu pravdepodobne prispela k všeobecnej snahe skúmať len automaty, ktoré majú schopnosť univerzálnej konštrukcie. V skutočnosti však aj automaty, ktoré nie sú univerzálnymi konštruktormi môžu byť schopné netriviálnej replikácie. Langton uvádza von

Neumannov pokus o kritérium takéhoto automatu ako schopnosť narábať s kódom (postupnosťou stavov) podobne ako pri biologickej reprodukcii a to tak, že kód môže byť interpretovaný (v biol. translácia) alebo prepísaný, teda neinterpretovaný (transkripcia).

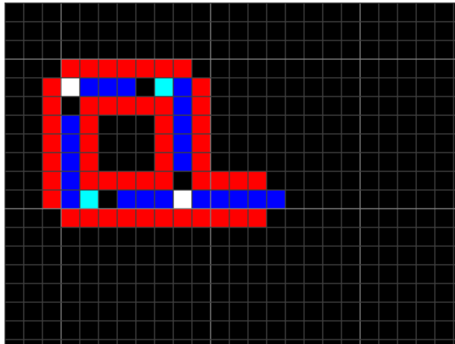
Langtonov automat je založený na Coddovej slučke, ktorá dokáže uchovávať informáciu a postupne ju "vysielať". Replikácia je riadená sadou inštrukcií, ktoré sú kódované postupnosťami stavov. Inštrukcie cirkulujú vo vnútri a tak slučka slúži ako pamäť obsahujúca program. Inštrukcie sa pri prechode do výsuvnej ruky (angl. retracting arm) kopírujú a iniciujú zmeny ako napríklad: *Predĺženie ruky*, *Predĺženie ruky vľavo*, *Predĺženie ruky vpravo*, *Predĺženie obalu* a pod.



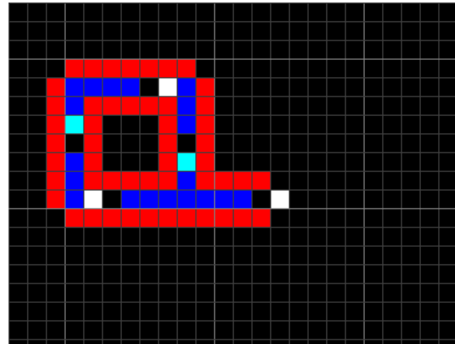
(a) Začiatok



(b) Skopírovanie inštrukcie



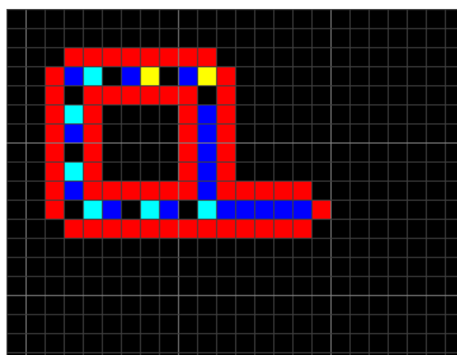
(c) Predĺženie ruky

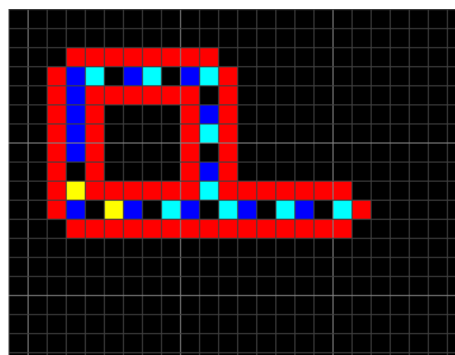


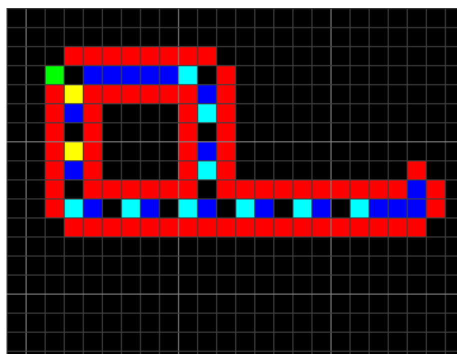
(d) Predĺženie obalu slučky

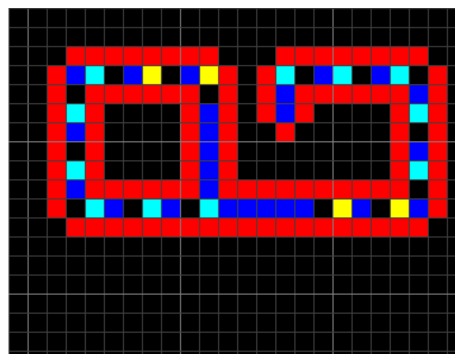
Obr. 1.3: Coddova slučka

Po Langtonovi sa viacero autorov snažilo o zmenšenie počtu stavov a veľkosti automatu, najznámejší Byl[1] a Reggia a kol.[8].


(a) Začiatok


(b) Predĺženie ruky


(c) Zabočenie vľavo


(d) Slučka takmer skopírovaná

Obr. 1.4: Langtonova slučka

Konečne v roku 1995 G. Tempesti uverejňuje prácu na seba-replikujúce automaty[9]. Jeho automat používa model Langtonovej slučky, ale podobne ako Byl odstraňuje jeden z obalov (tu je to však vonkajší). Slučka sa pomocou výsuvných rúk replikuje do štyroch strán naraz. Princíp replikácie je podobný Langtonovej slučke, ale na rozdiel od všetkých predošlých slučiek má Tempestiho slučka schopnosť uchovávať prídavnú informáciu, ktorá môže byť využitá na vybudovanie konfigurácie. Tempesti ukazuje slučku s prázdnou informáciou, ale aj slučku, ktorá obsahuje informáciu o konkrétnej konfigurácii, ktorú je schopná vybudovať. Konkrétne ide o konfiguráciu v tvare písmen LSL (skratka pre Logic Systems Laboratory, Tempestiho pôsobisko).

Tempestiho slučka je schopná počas svojej sebareplikácie zanechávať na automate konfigurácie ako vedľajší produkt behu. Pre vytváranie ľubovoľných

konfigurácií je Tempestiho slučka nevhodná, pretože je ušitá na mieru pre konkrétnu konfiguráciu. Spôsob akým konfiguráciu tvorí je nesystematický a takmer neprenosný. Ušité na mieru sú aj pravidlá, a preto sa snažíme o ich zovšeobecnenie. Približný opis behu Tempestiho slučky nasleduje v sekcii 3.1.

Kapitola 2

Celulárne automaty - implementácia

2.1 Úvod

Pre tento typ úlohy sme vychádzali z toho, že prostredie, v ktorom bude možné navrhnúť a pracovať so seba-replikujúcimi automatmi musí byť dostatočne flexibilné, teda napríklad musí umožňovať prácu s viacerými automatmi a kopírovanie konfigurácií medzi nimi. Ďalej musí byť dostatočne rýchle, pretože naivné spôsoby simulácie automatov sú nepostačujúce pre veľké konfigurácie. Možno najdôležitejším kritériom je sprehľadnenie práce s prechodovými funkciami, ktoré sú v textovom zápise nečitateľné.

Tieto úlohy sa nám podarilo splniť a simulačné prostredie pre celulárne automaty *fcas (skr. pre fast cellular automata simulator)* je opísané v sekciách užívateľská dokumentácia (A) a programová dokumentácia (B).

2.2 Hash-life

Prvé a tretie kritérium pre naše prostredie sú technické záležitosti, druhé vyžaduje zmenu prístupu. Vychádzame z algoritmu Hash-life opísaného W. Gosperom v roku 1984[3]. Tento algoritmus dokáže zabezpečiť plynulosť simulácie a nízke využitie pamäte aj pri obrovských automatoch. Využíva k tomu rozdelenie nenulovej plochy automatu na *quad-tree*, teda strom so štyrma nasledovníkmi a vhodné hašovanie jeho uzlov. Týmto dokáže efektívne využiť pamäť tak, aby sa žiaden výpočet neopakoval dvakrát.

Okrem toho, pre ilustrácu, napríklad aj prázdny automat o veľkosti $2^{32} \times 2^{32}$ zaberie v pamäti len miesto zodpovedajúce 33 uzlom stromu. Naša implementácia tohto algoritmu je opísaná v sekcii B.4.

2.3 Rule Table - prechodové funkcie zadávané tabuľkou

Ako sme už uviedli v definícii celulárneho automatu, existuje viacero možností ako definovať prechodovú funkciu. Niektoré prechodové funkcie sú závislé iba od množstva žijúcich buniek v okolí bunky. Pre zložitejšie automaty je ale potrebné používať pravidlá vo všeobecnom formáte, kde prechodová funkcia je závislá na stavoch všetkých susedov s presne danými pozíciami. Takéto prechodové funkcie je zvykom definovať vo formáte *Rule Table*[11] a jedna funkčná hodnota sa bežne nazýva pravidlo (angl. rule). Pravidlá sa zadávajú pomocou premenných pre uľahčenie práce a takisto využívajú symetrie, ktoré v niektorých prípadoch značne redukujú počet pravidiel. Tento formát je podrobne popísaný v sekcii A.4.3.

Počet pravidiel môže byť veľký, maximálne až n^N, čo je napríklad pre Moorovo okolie s desiatimi stavmi (konfigurácia Tempestiho slučky) maximálne 10^9 pravidiel. Aj napriek tomu, že v skutočnosti je počet pravidiel omnoho menší (Tempestiho slučka má 1026 pravidiel), je nutné výpočet automatu optimalizovať. Dve triviálne riešenia ako lineárny zoznam a hašovacia tabuľka sa ukázali ako nepoužiteľné. Lineárny zoznam z triviálnych dôvodov (rýchlosť výpočtu nemôže byť závislá od počtu pravidiel) a hašovacia tabuľka preto, lebo pre jej zostrojenie potrebujeme dosadiť za všetky premenné a teda narábať so všetkými pravidlami naraz, čo veľmi predĺži čas potrebný na vytvorenie automatu.

V našej implementácii používame pre výpočet jedného prechodu konečný automat nad abecedou stavov, ktorý pre daný počet krokov vždy dosiahne konečný stav, teda nový stav bunky, pre ktorú bol výpočet prevádzaný. Tento automat pracuje v koštantnom čase a používa oveľa menej pamäte, ako by bolo nutné pre pravidlá uchovávané v zozname. Upresnenie v odstavci B.3.

Kapitola 3

Generovanie prechodových funkcií

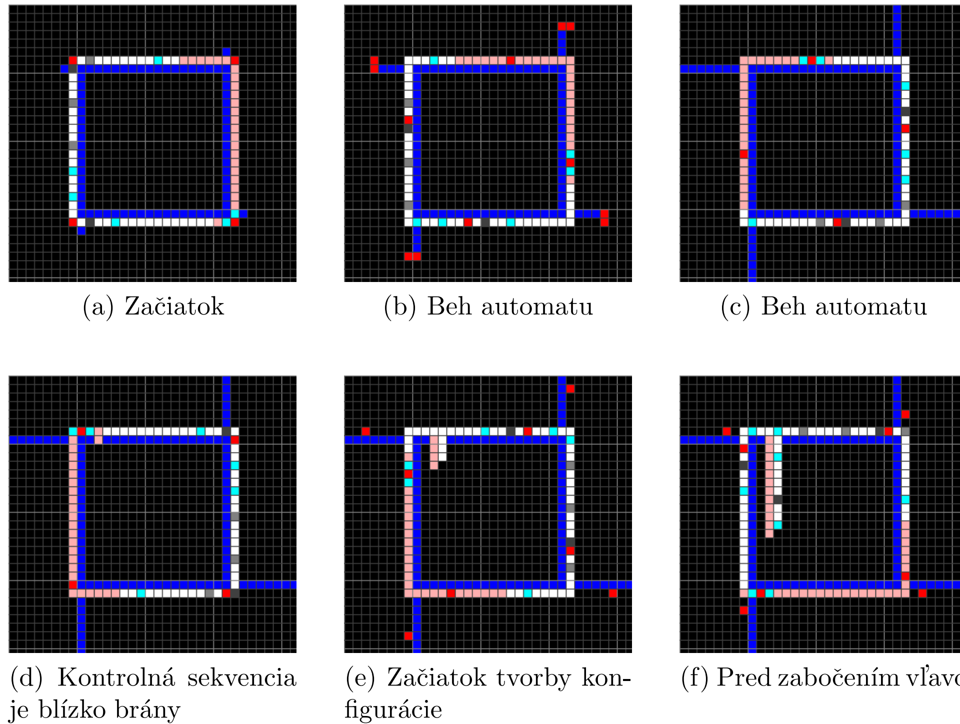
3.1 Tempesti

Tempestiho slučka je automat s konfiguráciou štvorcového tvaru s vnútornou stenou. Z vonkajšej strany okolo steny cirkulujú datové a niektoré základné stavy. Základných stavov vykonávajúcich replikáciu je päť. Tempesti vo svojej práci uvádza príklad automatu so siedmymi stavmi, piatimi základnými a dvoma datovými. Tento automat netvorí žiadnu vedľajšiu konfiguráciu. Replikácia prebieha nasledovne:

- Slučka má v rohoch štyri tzv. brány, ktoré signalizujú, či tadiaľto bola v minulosti vykonávaná replikácia.
- V prípade, že je brána otvorená, z automatu sa vysunie tzv. výsuvná ruka (angl. retracting arm), ktorá postupne buduje súvislý pás steny.
- Bez ďalšej interferencie by výsuvná ruka nikdy neskončila svoju činnosť. Jej kolmé zabočenie sa zabezpečuje pravidelným vysielaním signálov z pôvodnej slučky.
- Keď výsuvná ruka dokončí stenu novej slučky, vyšle posledný signál a z pôvodnej slučky sa začnú kopírovať všetky cirkulujúce stavy. Po skončení sa uzavrie príslušná brána.
- Ak výsuvná ruka narazí na už existujúcu slučku, tak sa stiahne a uzavrie svoju bránu. Takto sa postupne pokrýva celý priestor novými

slučkami v štyroch základných smeroch.

- Po skopírovaní cirkulujúcich stavov na slučku sa začne jej sekundárna činnosť a to je tvorenie vedľajšej konfigurácie.



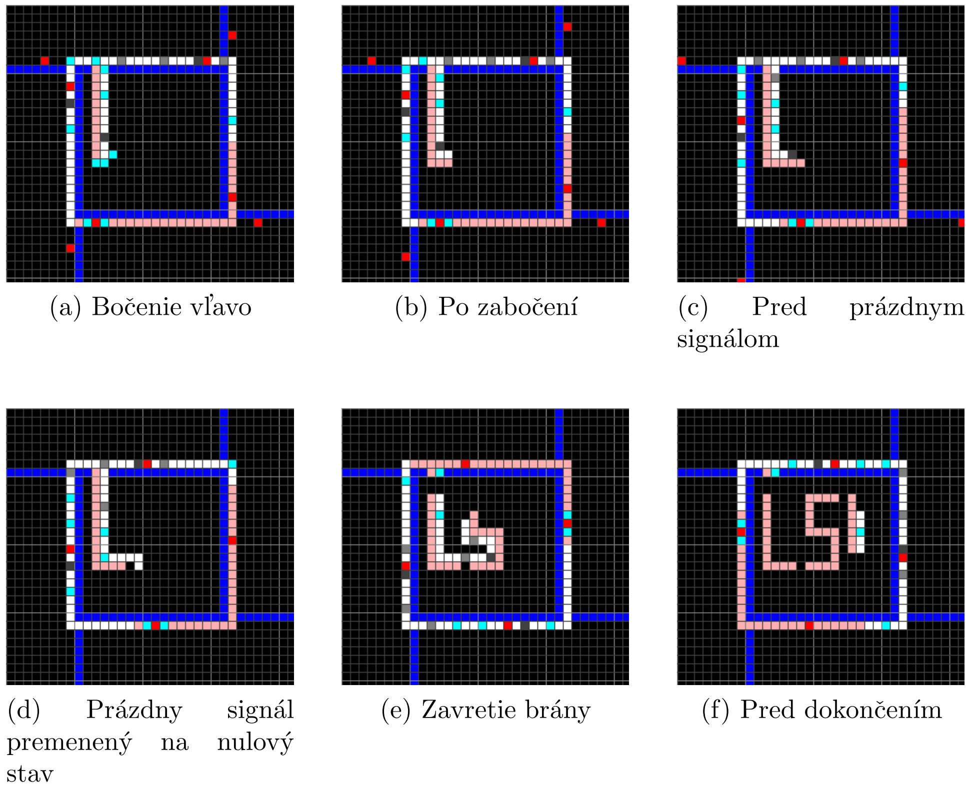
(a) Začiatok (b) Beh automatu (c) Beh automatu

(d) Kontrolná sekvencia je blízko brány (e) Začiatok tvorby konfigurácie (f) Pred zabočením vľavo

Obr. 3.1: Na horných obrázkoch sledujeme postupnú cirkuláciu datových signálov proti smeru hodinových ručičiek okolo modrého obalu slučky a postupné vysúvanie rúk do štyroch strán. Na obrázku (d) sa kontrolná sekvencia, trojica troch stavov (farby tyrkysová, červená, tyrkysová), dostala do horného ľavého rohu a následne začínajú stavy prúdiť dovnútra slučky cez práve otvorenú bránu. Biele stavy sú signálom pre vytvorenie pevného (ružového) stavu cieľovej konfigurácie. Tyrkysové stavy sú signálom pre zabočenie vľavo. Na obrázku (f) je už vybudovaná ľavá časť písmena 'L' a nasleduje tyrkysový signál pre zabočenie vľavo.

Tvorenie vedľajšej konfigurácie:

- Tempesti nadefinoval slučku tak, aby vedľajšou konfiguráciou boli tri písmená "LSL". Datové stavy sú v tomto prípade signálmi, ktoré udávajú smer stavby konfigurácie. Týchto signálov je päť: *Posun vpred*,



(a) Bočenie vľavo

(b) Po zabočení

(c) Pred prázdnym signálom

(d) Prázdny signál premenený na nulový stav

(e) Zavretie brány

(f) Pred dokončením

Obr. 3.2: Na obrázku (a) prebieha proces bočenia signálového prúdu vľavo. Na (b) je už pravouhlá trojica pevných (ružových) stavov vytvorená a prúd postupuje podľa tvaru písmena 'L'. Na obrázku (c) je na rade tmavosivý stav, ktorý je signálom pre prázdne miesto, alebo vynechanie pevného stavu. Na (d) je už tento pevný stav vynechaný a pokračuje sa s budovaním konfigruácie. Na (e) došlo k zavretiu brány vľavo hore a žiaden ďalší signál sa už do slučky nedostane. Ostávajúce stavy sú signálmi pre posun vpred (biele), zabočenie vpravo (svetlosivé), prázdne miesto (tmavosivý) a zabočenie vľavo (tyrkysový).

Prázdne miesto, *Zabočiť doľava*, *Zabočiť doprava* a *Nulový signál* bez významu.

- Najprv sa musí kontrolná sekvencia dostať k bráne v ľavom hornom rohu.

- Po otvorení a vstupe do brány signály postupujú v prúde popri budovanej konfigurácii a každý z nich prispeje k vybudovaniu časti konfigurácie.

- Po vstupe posledného signálu sa brána zatvára.

Hlavný problém Tempestiho slučky a jej nemožnosti na využitie pre ľubovoľné konfigurácie spočíva práve v spôsobe, akým sa konfigurácia tvorí a to pomocou podporného signálového prúdu, ktorý sa posúva popri budovanej konfigurácii. Všetky pravidlá pre signály vychádzajú z tvaru písmen "LSL". Takýto podporný prúd pre prenos signálov je maximálne možné využiť pre konfigurácie, ktoré majú nenulové stavy v každom druhom stĺpci. Tejto špecifickosti sa snažíme vyhnúť, a preto navrhujeme všeobecný princíp budovania vedľajšej konfigurácie, ktorý sme nazvali *Nosičové Stavy*.

3.2 Nosičové stavy

Ak predpokladáme, že signál nesie informáciu o stave jednej bunky, tak je nutné, aby sa dokázal presunúť ku všetkým cieľovým bunkám. Na toto nie je možné využiť Tempestiho datový prúd a to z toho dôvodu, že ním nie je možné žiadnym spôsobom pokryť celú plochu konfigurácie. Ak teda nie je možné mať podporný prúd, konfigurácia musí vznikať priamo v signálovom prúde. Nech je pre zjednodušenie prúd len vertikálny a vytvárame konfiguráciu $m \times 1$ v stĺpci.

Tu sú možné dva prístupy. Buď pevný cieľový stav na danom mieste prúdu určí prvý alebo posledný signál prúdu. Druhý prístup ale nie je možný, lebo by došlo k prepísaniu nasledujúcich stavov posledným signálom. Teda prvý signál, ktorý dorazí k prvej bunke cieľovej konfigurácie, určí jej cieľový stav, resp. prepíše sa na cieľový stav. Po tomto je nutný prepis zvyšku signálov. Musíme zaručiť, aby bunka dokázala transportovať signály a zároveň mala informáciu o jej cieľovom stave. Keď signál tzv. vstúpi do bunky s pevným stavom, zmení sa jej stav na *nosičový stav*, ktorý má informáciu o cieľovom stave aj o transportovanom signále. Keď posledný signál opustí bunku s nosičovým stavom, jej stav sa zmení na cieľový. Teda všetky signály sa po $2m - 1$ krokoch prepíšu na cieľové stavy. Tento postup teraz opíšeme podrobne.

Budeme používať niekoľko typov stavov:

Základnými stavmi budeme myslieť päť základných stavov Tempestiho slučky, ktoré zabezpečujú replikáciu.

Pevné stavy sú stavmi cieľovej konfigurácie, ktorú bude automat vytvárať. Jeden z pevných stavov je nulový stav, takže požadujeme minimálne

dva pevné stavy. Nulový stav je takisto základným stavom, a preto ho pri počítaní celkového počtu stavov musíme odčítať.

Signály sú datovými stavmi, ktoré sa budú prenášať medzi slučkami a obsahujú kompletnú informáciu o generovanej konfigurácii. Pre každý pevný stav existuje jeden signál, ktorý ho reprezentuje a navyše jeden signál pre riadenie prúdu, tzv. *riadiaci signál*.

Nosiče sú stavy, ktoré reprezentujú jeden pevný stav a jeden signál, ktorý cez nich prechádza. Za nosiče sa dajú považovať aj signály, ich pevným stavom je nulový stav. Definované ako nosiče ale nie sú. Je nutné definovať takzvaný *prázdny nosič*, ktorý zastupuje nulový stav, ako bude vysvetlené neskôr. Nie je potrebné, aby bol nosený riadiaci signál, a preto je počet nosičov $n_{\text{nosiče}} = (n_{\text{pevné}} - 1)(n_{\text{signály}} - 1) + 1$.

Pevný riadiaci stav je statickou podobou *riadiaceho signálu*. Zabezpečuje riadenie pohybu signálov vo vnútri slučky.

3.2.1 Počet stavov

Ku generovaniu budeme potrebovať 5 základných stavov z Tempestiho slučky a dodatočné stavy, ktorých počet závisí od počtu stavov generovanej konfigurácie. Keďže každý nosičový stav má v sebe informáciu o cieľovom stave aj o signále, ktorý ním prechádza, počet nosičových stavov je kvadratický v počte pevných stavov. Ich počet je presne:

$$\begin{aligned} n_{\text{spolu}} &= n_{\text{základné}} - 1 + n_{\text{pevné}} + n_{\text{signály}} + n_{\text{nosiče}} + 1 \\ &= n_{\text{základné}} + n_{\text{pevné}} + n_{\text{signály}} + (n_{\text{pevné}} - 1)(n_{\text{signály}} - 1) + 1 \\ &= n_{\text{základné}} + 2n_{\text{pevné}} + 1 + (n_{\text{pevné}} - 1)n_{\text{pevné}} + 1 \\ &= n_{\text{základné}} + n_{\text{pevné}}^2 + n_{\text{pevné}} + 2 \\ &= n_{\text{pevné}}^2 + n_{\text{pevné}} + 7 \end{aligned}$$

3.2.2 Beh automatu

Medzi stavmi obiehajúcimi okolo slučky sú signály a jeden zo základných stavov Tempestiho slučky, tzv. kontrolný stav a prvý nenulový pevný stav. Dáta sú za sebou uložené v danom poradí, zvyšok je vyplnený pevným stavom. Na začiatku dát je kontrolná sekvencia, ktorá otvorí vnútornú bránu a spustí tvorbu vedľajšej konfigurácie. Vo východzej pozícii sú v štyroch

rohoch kontrolné stavy, ktoré pri dotyku s výsuvnou rukou vysielajú signály pre správnu tvorbu novej slučky.

Automat funguje na rovnakom princípe ako Tempestiho slučka až do otvorenia brány kontrolnou sekvenciou. Od tohto momentu všetky signály začínajú prúdiť smerom do slučky.

Interakciou signálu a nulového stavu je zodpovedajúci pevný stav. V prípade, že do pevného stavu zostupuje ďalší signál, mení sa tento pevný stav na nosič. Odklon prúdu do vedľajšieho stĺpca zabezpečuje *riadiaci signál* a *pevný riadiaci stav*.

3.2.3 Pravidlá

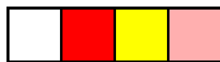
Pravidlá sú logicky rozdelené do sekcií a my ich budeme prezentovať pre lepšiu čitateľnosť graficky. Originálne pravidlá Tempestiho slučky sa až na premenné nemenia a pre ich veľké množstvo ich neuvádzame. Uvádzame len pravidlá, ktoré vzniknú pri cieľovej konfigurácii o dvoch stavoch.

Predtým, než definujeme samotné pravidlá, predstavíme premenné, ktoré sa v nich využívajú. Tie su nutnosťou pre prehľadnosť zápisu a bližšie informácie o nich budú uvedené v zozname pravidiel.

Naše prostredie prezentuje premenné a pravidlá aj v grafickej podobe a tie pre lepšiu čitateľnosť uvádzame. Použité farby sú vytvorené triedou programu *CarrierTLColorMapBuilder* pre dva pevné stavy. Pevné stavy majú čiernu a bielu farbu, signály zodpovedajúce pevným stavom majú červenú a žltú farbu, ružový a oranžový je riadiaci stav v signálovej a pevnej podobe. Nosiče majú všeobecne farby vzniknuté zmiešaním farby noseného signálu a pevného základného stavu. Prázdny nosič je tyrkysový. Obal slučky je modrej farby.

`a-d`

Originálne premenné z Tempestiho slučky predstavujúce datové stavy, teda stavy cirkulujúce okolo slučky, konkrétne ide o signály a prvý nenulový pevný stav.

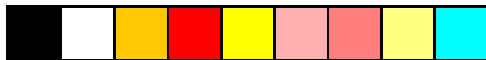


`Ta1-3 - transported all`

Všetky transportované stavy a rohový kontrolný stav.

`a1-2 - all`

Všetky stavy, ktoré nie sú základnými z Tempestiho slučky.



`ss1-4 - solid state`

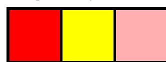
Pevné stavy cieľovej konfigurácie.

`sd1-3 - solid zero & driver`

Nulový a riadiaci pevný stav.

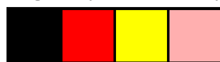
`s1-4 - signal`

Signály.



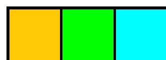
`s01-03 - signal & zero`

Signály a nulový stav.



`cc - control counter`

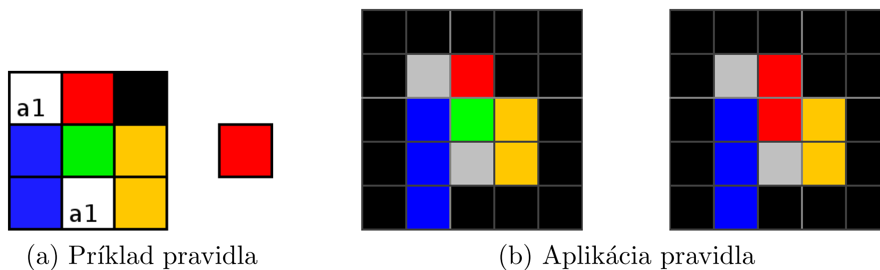
Počítadlo pre prepis kontrolných stavov. Obsahuje tri nezávislé stavy a to pevný riadiaci stav, prázdny nosič a jeden zo základných stavov. Tieto boli vyberané tak, aby bolo zaručené, že u nich nebude vznikať interferencia s inými pravidlami. Vybrali sme ich z už existujúcich stavov preto, aby sme ušetrili prípadné tri nové stavy.



Nasleduje zoznam pravidiel nezahŕňajúci pôvodné pravidlá z Tempestiho slučky. Pravidlá majú formu diagramov tak, ako je uvedené na obrázku 3.3.

Otvorenie slučky

Pravidlá riadiacie proces vstupovania stavov do slučky od momentu, keď sa kontrolná sekvencia (svetlo-ružová, ružová, svetlo-ružová) dostane do ľavého horného rohu. Používajú sa premené `Ta1-3` reprezentujúce stavy, ktoré sú transportované a cirkulujú okolo slučky. Ďalej sa



(a) Príklad pravidla (b) Aplikácia pravidla

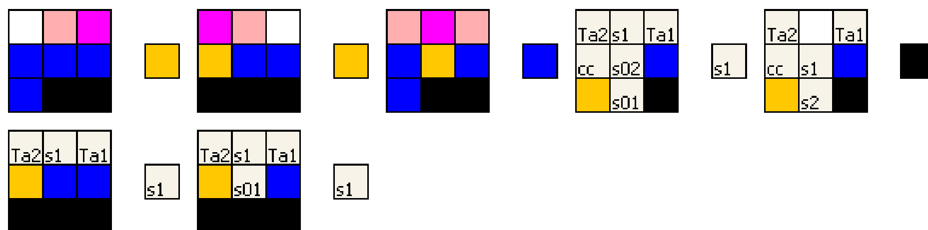
Obr. 3.3: Grafická reprezentácia pravidla automatu. Naľavo je okolie bunky spolu s bunkou (so zeleným stavom) v strede pred diskrétnym prechodom automatu. Napravo je výsledný stav bunky. Diagram využíva, podobne ako textová reprezentácia, aj premenné, v tomto prípade `a1`. Jedno pravidlo teda reprezentuje viac pravidiel, ktoré vzniknú po dosadení všetkých premenných. Na obrázku (b) je automat pred a po aplikácii tohto pravidla za predpokladu, že premenná `a1` obsahuje aj stav označený sivou farbou a nie sú definované žiadne iné pravidlá, ktoré by spĺňali situáciu v automate.

používajú premenné `s1-3` reprezentujúce všetky signály a `s01-s03` pre signály a nulový stav. Transportované stavy v pravidlách používame, aby sme zaručili ich plynulý obeh okolo obalu slučky (modrá farba). Signálové premenné sú potrebné na pohyb signálov vo vnútri slučky a okolo pevného riadiaceho stavu (oranžová farba). Tento stav riadi pohyb signálov a slúži ako stena vo vnútri slučky, okolo ktorej signály postupujú. Používaná je aj premenná `cc`, ktorej stavy strieda jedna konkrétna bunka v slučke a to bunka naľavo od brány. Je nutná pre prepis kontrolných stavov, ako je vysvetlené v nasledujúcej skupine pravidiel.

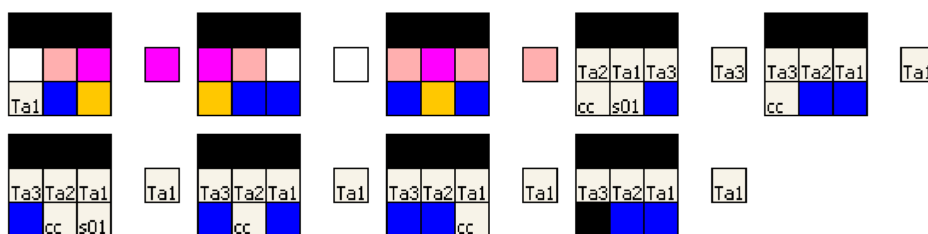
Transformácia kontrolných stavov

Pravidlá, ktoré zaručujú, že tri rohové kontrolné stavy nevyhnutné pre beh Tempestiho slučky sa správne prepíšu na zodpovedajúce stavy. K tomu slúži aj premenná `cc`. Predstavuje tri rôzne stavy a slúži ako počítadlo, bez ktorého by sme nemohli rozlíšiť jednotlivé pravidlá pre prepis rohových kontrolných stavov. Na obrázku prvé tri pravidlá určujú prepis kontrolných stavov na červený, nulový a nulový signál. Po prepise kontrolného stavu sa počítadlo zvýši o jedna, teda bunka zmení svoj stav na ďalší v premennej `cc`.

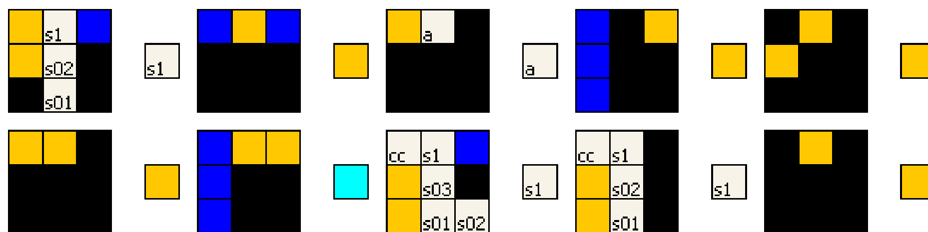
Zmena signálu na pevný stav



(a) Otvorenie brány a vstup stavov do brány

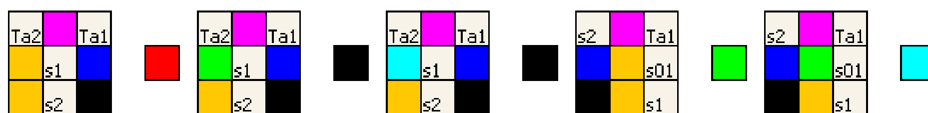


(b) Prechod cirkulujúcich stavov okolo otvorenej brány



(c) Prúdenie stavov dovnútra slučky

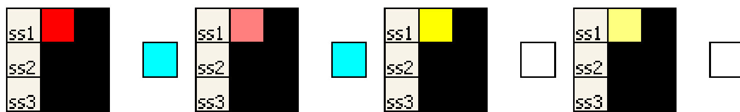
Obr. 3.4: Otvorenie brány a vstup stavov



Obr. 3.5: Transformácia kontrolných stavov

Signály a nosiče obsahujúce signály sa pri strete s nulovým stavom prepisujú na zodpovedajúci pevný stav (biela farba) a signály reprezentujúce nulový pevný stav sa prepisujú na prázdny nosič (tyrkysová farba). Tento slúži ako náhrada za nulový stav, do ktorého by sa signály inak prepisovali na im zodpovedajúce pevné stavy. Pri strete

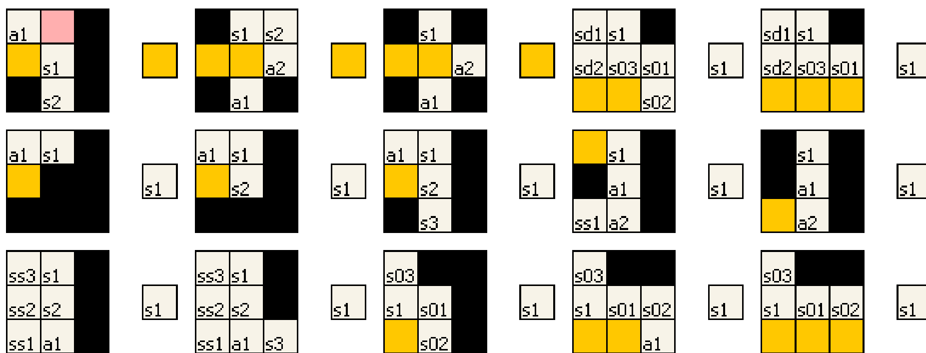
s prázdnym nosičom sa neprepisujú, ale ostávajú signálmi. Premenné `ss1-3` reprezentujú pevné cieľové stavy, teda stavy už hotovej konfigurácie. Vľavo sú umiestnené preto, lebo konfigurácia sa stavia zľava doprava.



Obr. 3.6: Zmena signálu na pevný stav

Postup signálu a interakcia s pevným riadiacim stavom

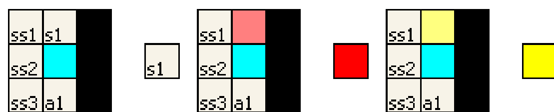
Pravidlá zabezpečujúce vertikálny zostup signálov a odklon prúdu signálov po stĺpcoch. Takisto zabezpečujú zmenu riadiaceho signálu (ružová farba) na pevný riadiaci stav (oranžová farba). Používajú sa premenné `s1-3` pre signály a ich postup, `a1-2` pre všetky stavy v slučke, `ss1-3` pre pevné cieľové stavy, ktoré môžu byť naľavo od práve budovanej konfigurácie a nakoniec premenné `sd1-3` pre nulový stav alebo pevný riadiaci stav. Tieto premenné sú výhodné pre redukciu pravidiel pre postup signálu.



Obr. 3.7: Postup signálu a interakcia s pevným riadiacim stavom

Prázdny nosič

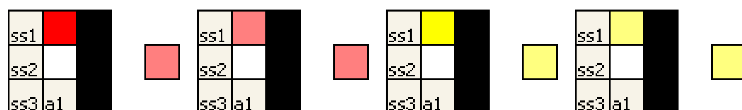
Prázdny nosič (tyrkysová farba) slúži ako náhrada za nulový stav. Pri strete s ním sa signály neprepisujú na cieľové pevné stavy, ale ostávajú signálmi.



Obr. 3.8: Prázdny nosič

Zmena signálu na nosič

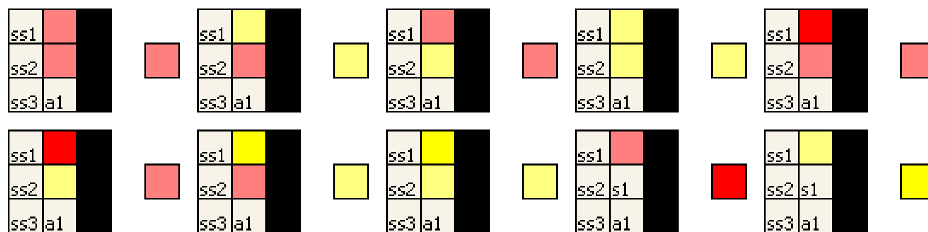
Pri strete signálu (čerevná a ružová farba) s pevným stavom sa tento mení na zodpovedajúci nosič (svetlé varianty farieb signálov). V príklade máme len dva typy nosičov. Základným stavom je pevný stav 1 a noseným signálom je jeden z dvoch signálov.



Obr. 3.9: Zmena signálu na nosič

Vstup do nosiča

Nosiče a signály vstupujúce do nosičov sa menia na zodpovedajúce nosiče a nosiče odovzdávajú svoj signál ďalej. Premenné `ss1-3` reprezentujúce pevné cieľové stavy sú nutné kvôli smeru výstavby konfigurácie zľava doprava, premenná `a1` označuje ľubovoľný stav vo vnútri slučky a významne redukuje počet potrebných pravidiel.

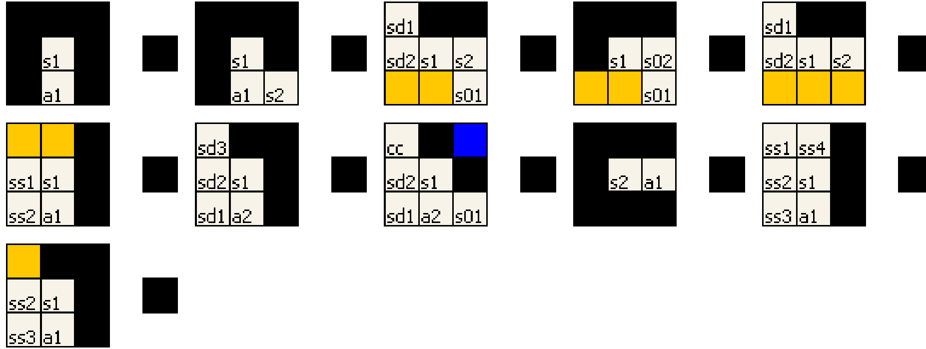


Obr. 3.10: Vstup do nosiča

Zánik signálu

Posledné signály zanikajú. Opäť využívame premennú `sd1-3` reprezentujúcu nulový stav a pevný riadiaci stav (oranžová farba), ktorý slúži

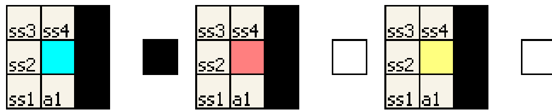
ako oporná stena pre pohyb signálov. Všetky pravidlá menia stavy na nulové stavy a využívajú sa v horných a ľavých koncoch signálového prúdu, ktoré takto zanikajú.



Obr. 3.11: Zánik signálu

Zánik nosiča

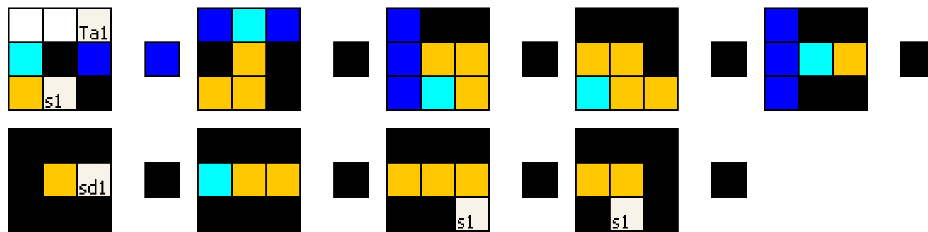
Posledné nosiče sa menia na cieľový pevný stav, ktorý reprezentujú. Máme len prázdny nosič (tyrkysová farba) a nosiče so základným pevným stavom 1. Nosiče sa menia na pevný stav vtedy, keď je vľavo a nad nimi nejaký pevný stav (premenné `ss1-3`).



Obr. 3.12: Zánik nosiča a zmena na cieľový pevný stav

Vyčistenie okolia

Pre správne splnenie zadania sa okolie očistí od nadbytočných stavov a ostáva len cieľová konfigurácia. Prvé pravidlo využíva premennú `Ta1` pre transportované stavy na zavretie brány a ucelenie obalu slučky (modrá farba). Všetky ostatné pravidlá riadia zánik nadbytočných stavov, hlavne pevného riadiaceho stavu (oranžová farba).



Obr. 3.13: Vyčistenie okolia

3.3 Komprimácia stavov

Pri skúmaní behu automatu s nosičovými stavmi vyvstáva otázka, či by nebolo možné navrhnúť automat tak, aby jeden stav držal informáciu o cieľovom stave viacerých buniek naraz. Ak by sme uvažovali rovnaký princíp, tak je možné, aby signál určoval stavy pre tri bunky naraz, keďže veľkosť okolia je 1. Každý signál by určoval $n^3_{\text{pevné}}$ trojíc. Celkovo by takto len počet nosičov bol aspoň $n_{\text{nosiče}} \geq n_{\text{signály}} n_{\text{pevné}} = (n^3_{\text{pevné}}) n_{\text{pevné}} = n^4_{\text{pevné}}$.

Pri našich implementačných možnostiach budeme uvažovať len prípad $n_{\text{pevné}} = 2$. Určíme pravdepodobnostnú funkciu $f_P(m, 3n, k)$ udávajúcu pravdepodobnosť, že pri konfigurácii veľkosti m x $3n$ bude využitých presne k rôznych trojíc stavov. Najprv definujeme rekurzívnu funkciu $v_1(n, k)$ ako počet variácií dĺžky n na k prvkoch takých, že sa tam každý prvok vyskytuje aspoň raz.

$$v_1(n,k) = k^n - \sum_{i=2}^{k} \binom{k}{i-1} v_1(n, k-1)$$
$$v_1(n,1) = 1^n = 1$$

Potom pre našu pravdepodobnostnú funkciu platí

$$f_P(m, 3n, k) = \frac{\binom{8}{k} v_1(n,k)}{8^{mn}}$$

Už pri konfiguráciách o veľkosti 10×10 je pravdepodobnosť, že využijeme všetkých 8 možných trojíc vyššia ako 90%, a preto za skutočný počet stavov potrebných k takémuto prístupu budeme považovať hore definovanú hornú hranicu.

Určíme teda výsledný počet stavov pre túto metódu. Budeme používať rovnaké typy stavov ako u predošlej metódy, ich počet sa však bude líšiť.

Základnými stavmi budeme opäť myslieť päť základných stavov Tempestiho slučky.

Pevné stavy sú stavmi cieľovej konfigurácie, uvažujeme len dva stavy.

Signály budú pri tejto metóde reprezentovať celú trojicu stavov a ich počet je premenlivý podľa konfigurácie a platí $n_{\text{signály}} \in [2, 9]$, pretože opäť využijeme aj *riadiaci signál*. Označme $n_{\text{komprimované}}$ ako počet rôznych trojíc stavov vyskytujúcich sa v cieľovej konfigurácii. Potom platí $n_{\text{signály}} = n_{\text{komprimované}} + 1$.

Transportované signály je kategória stavov, ktorú sme zaviedli, aby sme predišli interferencii, ktorá vzniká na rohoch slučky. Vzniká preto, lebo pri prepise jedného stavu na tri existujú samozrejme aj pravidlá, ktoré závisia len na jednej rohovej bunke okolia (obr. 3.14). Táto interferencia by sa dala ošetriť pridaním ďalších pravidiel, ale pretože počet stavov sa týmto omnoho nezväčší a úprava pôvodnych pravidiel Tempestiho slučky je nežiadúca, vyriešime tento problém pridaním nových $n_{\text{signály}}$ stavov. Tieto stavy budú z hľadiska budovania cieľovej konfigurácie neaktívne. Zaktivizujú sa pri vstupe do slučky prepisom na aktívne signály. Slúžia teda len ako informácia o signále, ktorý reprezentujú, ale žiadne pravidlá okrem transportačných sa na nich nevzťahujú.

Obr. 3.14: Dva typy pravidiel spôsobujúce interferenciu mimo slučky

Nosiče sú v tomto prípade len stavy reprezentujúce cieľový pevný stav *1* a nosený signál. Ich počet je teda spolu s prázdnym nosičom $n_{\text{nosiče}} = n_{\text{komprimované}} + 1$.

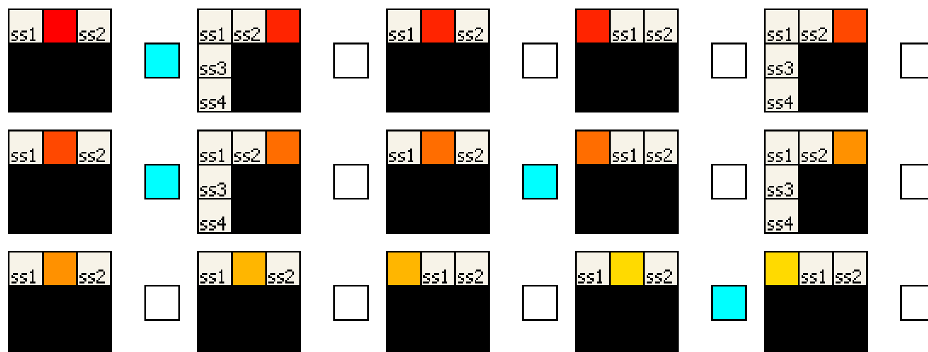
Pevný riadiaci stav opäť sluźí na riadenie pohybu signálov.

Celkový počet stavov je teda:

$$\begin{aligned}
n_{\text{spolu}} &= n_{\text{základné}} - 1 + n_{\text{pevné}} + n_{\text{transportované}} + n_{\text{signály}} + n_{\text{nosiče}} + 1 \\
&= 5 + 2 + n_{\text{signály}} + n_{\text{komprimované}} + 1 + n_{\text{komprimované}} + 1 \\
&= n_{\text{komprimované}} + 1 + 2n_{\text{komprimované}} + 9 \\
&= 3n_{\text{komprimované}} + 10
\end{aligned}$$

Pri využití všetkých možných trojíc stavov je výsledný počet stavov $n_{spolu} = 24 + 10 = 34$.

Pri tejto metóde sa používajú takmer rovnaké pravidlá automatu ako pri základnej nosičovej metóde. Výnimku tvoria prepis transportovaných signálov na signály pri vstupe do vnútra slučky a samotný prepis signálov na pevné stavy. Každý signál určuje stavy trom bunkám, a preto pre každý stav máme tri prepisovacie pravidlá (bez dosadenia premenných). Na obrázku 3.15 máme pravidlá pre 7 rôznych signálov. Pravidlá, ktoré prepisujú na nulový stav, nepridávame, lebo bunky sú pôvodne v nulovom stave. Výnimkou je stredný stĺpec, ktorý nesmie obsahovať po vstupe signálu nulový stav, aby boli ďalšie signály transportované, a preto sa prepisuje na prázdny nosič.



Obr. 3.15: Pri strete signálu s pevným stavom sa všetky tri bunky v rade prepisujú na cieľový pevný stav. Premenná `ss1-4` predstavuje pevné stavy a prázdny nosič je tyrkysový.

Samozrejmou výhodou komprimácie stavov je kratšia dátová sekvencia obiehajúca okolo slučky a tým aj menší potrebný priestor pre automat. Vypočítame veľkosť kompresie pre všeobecnú konfiguráciu veľkosti $m \times 3n$.

Najprv si pre základnú, nekompresnú metódu musíme uvedomiť, že stĺpec dĺžky n sa zapíše kódom dĺžky $n+1$ v prípade, že všetky stavy stĺpca sú nenulové. Ak máme od konca stĺpca m nulových stavov, tak kód bude mať dĺžku $n+1-m$. Definujeme si náhodnú premennú X ako počet ušetrených nulových stavov od konca stĺpca. Ušetriť môžeme maximálne $n-2$ stavov z princípu pravidiel. Preto máme

$$\Pr[X=i] = p_i = \frac{1}{2^{i+1}} \text{ pre } i = 1, 2, \ldots, n-2$$

Potom pre strednú hodnotu platí

$$E[X] = \sum_{i=0}^{n-2} \frac{i}{2^{i+1}}$$

Úpravou tejto konvergentnej rady získame strednú hodnotu ako

$$E[X] = 1 - n(\frac{1}{2})^{n-1}$$

Už pre malé n sa stredná hodnota blíži rýchlo k 1. To znamená, že stredná dĺžka kódu bude pre stĺpce dĺžky n rovná n.

Tento postup zopakujeme pre kompresnú metódu, kde je jediný rozdiel v tom, že sa berú trojice ako celky a u tých sa za koniec stĺpca považuje trojica obsahujúca nuly, teda jedna z ôsmich trojíc. Pre náhodnú veličinu X_{kompr} nám rovnakým postupom vyjde stredná hodnota

$$E[X_{kompr}] = \sum_{i=0}^{n-2} \frac{7i}{8^{i+1}} \to \frac{1}{7} \text{ pre } n \to \infty$$

Teda stredná hodnota sa blíži k $\frac{1}{7}$ a môžeme určiť kompresiu v priemernom prípade. Stredná dĺžka kódu pre konfiguráciu rozmerov $m \times 3n$ je pri základnej metóde

$$E[K] = ((m+1) - E[X])3n = 3mn$$

Pri kompresnej metóde je to

$$E[K_{kompr}] = ((m+1) - E[X_{kompr}])n = (m+1-\frac{1}{7})n = (m+\frac{6}{7})n$$

Stredná hodnota kompresie, teda ušetrenej dĺžky kódu je

$$E\left[(1-\frac{K_{kompr}}{K})100\%\right] = (1-\frac{(m+\frac{6}{7})n}{3mn})100\% = (1-\frac{m+\frac{6}{7}}{3m})100\%$$
$$= (\frac{14m-6}{21m})100\%,$$

čo je napríklad pre konfiguráciu výšky 10 približne 63.8%.

Tento prístup je teda možné využívať z implementačných dôvodov len pre konfigurácie o dvoch stavoch. Priemerná komprimácia potrebného signálového kódu je $(\frac{14m-6}{21m})100\%$ pre m rovné výške konfigurácie a počet potrebných stavov je $n_{\text{spolu}} = 3n_{\text{komprimované}} + 10$.

3.4 Implementácia

Hlavná trieda aplikácie pre generovanie automatov *TempestiLoopBuilder* sa nachádza v balíku *thesis.tempesti*. Táto abstraktná trieda má dve implementácie a to *CarrierTLBuilder* a *CompressedCarrierTLBuilder*. Tieto triedy reprezentujú dve rôzne metódy ako generovať automaty. Postup generovania je z implementačného hľadiska rovnaký pre obe metódy:

- Načítanie pôvodných pravidiel Tempestiho slučky z externých súborov.
- Statická trieda *StateNumberComputer* priradzujúca význam konkrétnym stavom, resp. ich číselnej reprezentácii.
- Vnorená trieda *TransitionBuilder*, ktorá pridáva už pravidlá automatu a premenné špecifické pre generovanie konfigurácií.
- Vnorená trieda *SignalCode*, ktorá zo zadanej konfigurácie vytvorí medzikód, ktorý sa použije na vytvorenie signálového kódu cirkulujúceho okolo slučky.

Generovanie automatu sa spúšťa v `Menu`→`Thesis`→`Create CarrierTL`. Po otvorení dialógového okna je možné nastaviť metódu (`Method`) a vstupnú konfiguráciu (`Side pattern`). Po výbere konfigurácie aplikácia zobrazuje informáciu o konfigurácii a možnosti použita oboch metód. Je možné zvoliť

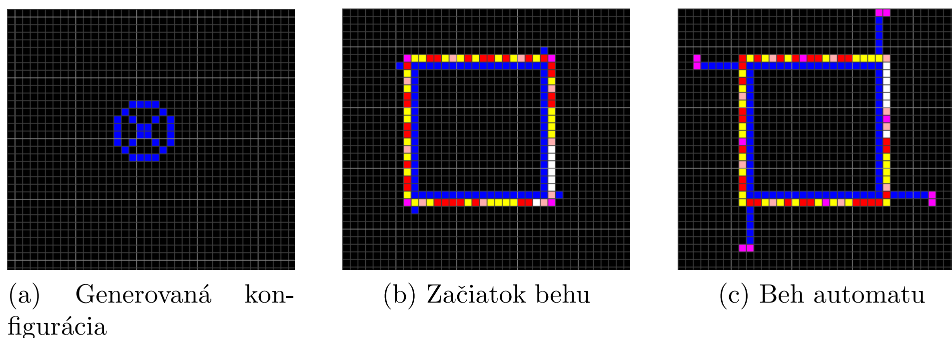
aj výstupné súbory pre prechodovú funkciu (`Output rule file`) a konfiguráciu (`Output pattern`) a takisto veľkosť slučky (`TL size`). Táto veľkosť je minimálnou veľkosťou a v prípade potreby bude vygenerovaná väčšia slučka. Poslednou možnosťou je okamžité zobrazenie vytvoreného automatu do prostredia (`Set to workspace`).

Kapitola 4

Príklady generovaných automatov

4.1 Nosičová metóda

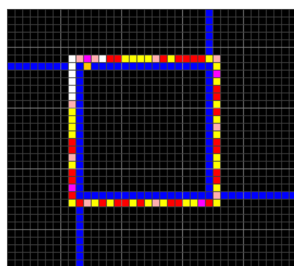
Generujeme automat s vedľajšou konfiguráciou o dvoch stavoch veľkosti 8×8.



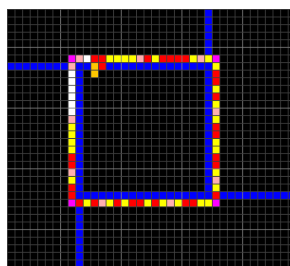
(a) Generovaná konfigurácia (b) Začiatok behu (c) Beh automatu

Obr. 4.1: Nosičová metóda

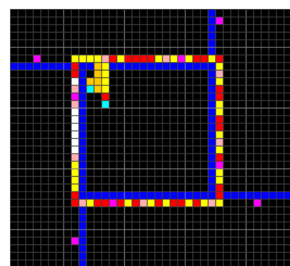
Vygenerovaný automat má 790 pravidiel s premennými, po dosadení premenných má 40,293 pravidiel. Na vybudovanie konfigurácie od začiatku behu je potrebných 119 diskrétnych prechodov. Tento príklad ukazuje beh automatu vytvoreného nosičovou metódou pre jednoduchú, relatívne malú konfiguráciu o dvoch stavoch. Konfigurácia sa veľkosťou podobá na konfiguráciu v Tempestiho slučke, avšak pri metóde z Tempestiho slučky by ju nebolo možné vytvoriť kvôli zložitej štruktúre.



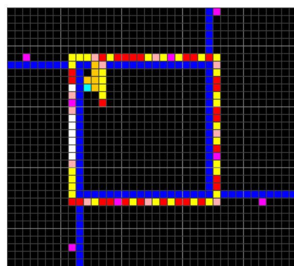
(a) Kontrolná sekvencia spúšťa tvorbu konfigurácie



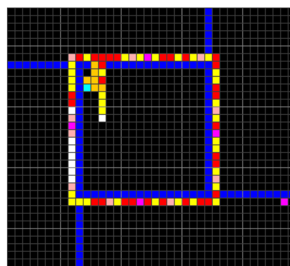
(b) Signály postupujú dovnútra slučky



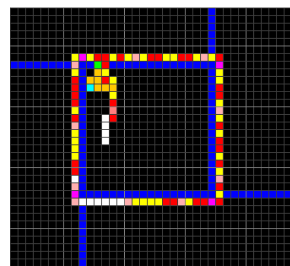
(c) Červený signál vstupuje do nulového stavu a vytvára prázdny nosič



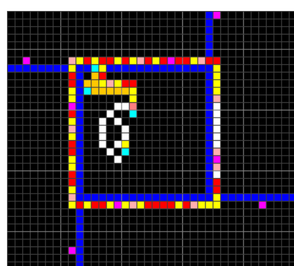
(d) Červený signál vstupuje do prázdneho nosiča



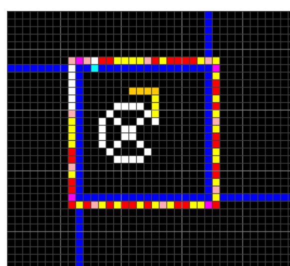
(e) Žltý signál vstupuje do nulového stavu a vytvára pevný stav 1



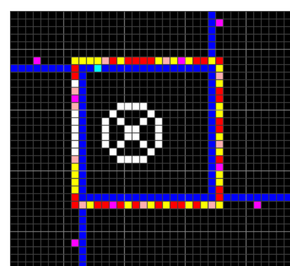
(f) Prúd je odklonený do druhého stĺpca



(g) Polovica konfigurácie je hotová



(h) Brána je zavretá, vyčisťuje sa okolie



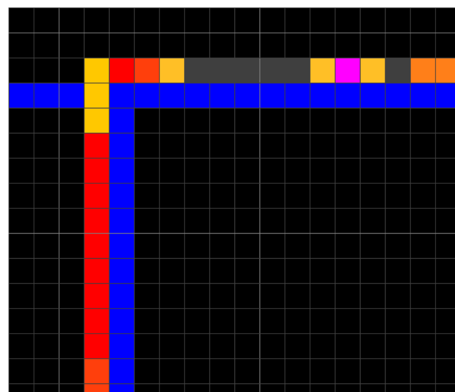
(i) Konfigurácia je hotová, kód sa začína kopírovať na vedľajšie slučky

Obr. 4.2: Nosičová metóda

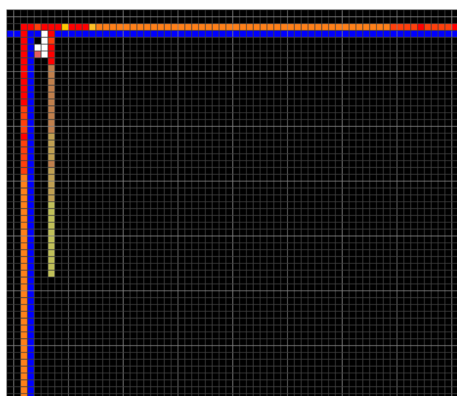
4.2 Nosičová metóda pre viacero cieľových stavov

Generujeme automat s vedľajšou konfiguráciou o štyroch stavoch veľkosti 100×75.

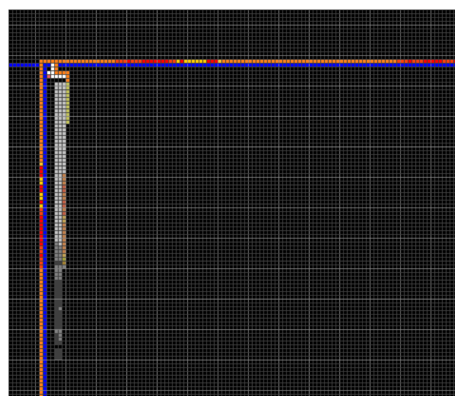
(a) Generovaná konfigurácia



(b) Kontrolná sekvencia



(c) Vytváranie konfigurácie

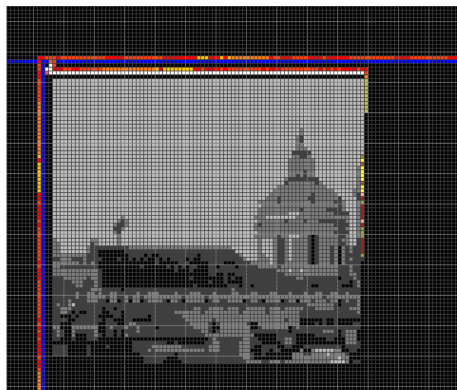


(d) Vytváranie konfigurácie

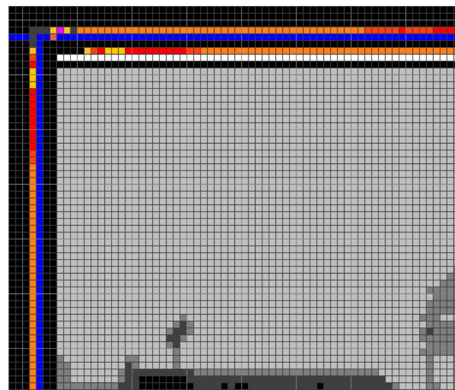
Obr. 4.3: Nosičová metóda pre 4 stavy

Vygenerovaný automat má 1060 pravidiel s premennými, po dosadení premenných má 991,041 pravidiel. Touto metódou je možné vytvoriť ľubovoľne veľkú konfiguráciu. Táto konfigurácia je obsahom takmer 60-krát väčšia ako konfigurácia v tvare písmen "LSL" v pôvodnej Tempestiho slučke. Dĺžka hrany slučky narastá lineárne s obsahom konfigurácie. Rozmery tejto slučky sú 1866x1866 a dĺžka jej hrany je takmer 85-krát väčšia ako u Tempestiho

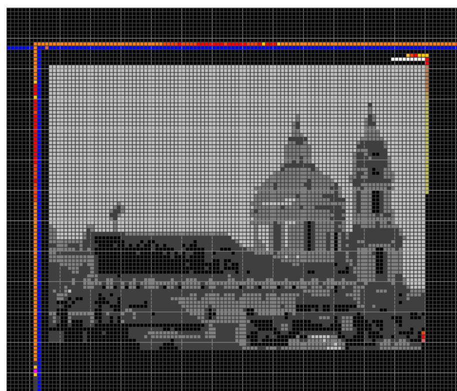
slučky (22x22). Počet krokov automatu na vytvorenie tejto konfigurácie od východzej polohy je 11,345.



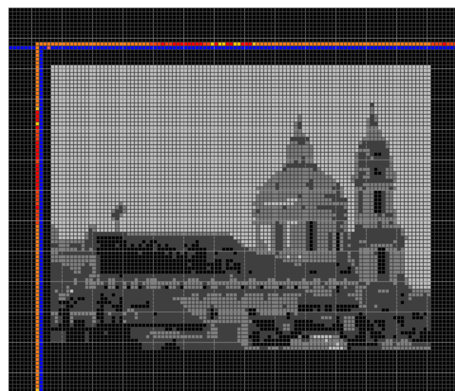
(a) Vytváranie konfigurácie



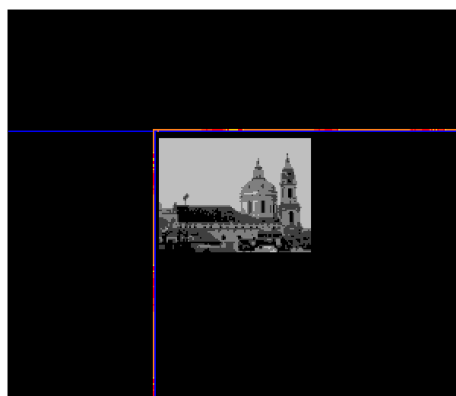
(b) Čistenie priestoru



(c) Čistenie priestoru



(d) Konfigurácia vytvorená

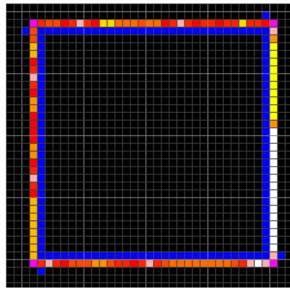


(a) Konfigurácia vytvorená

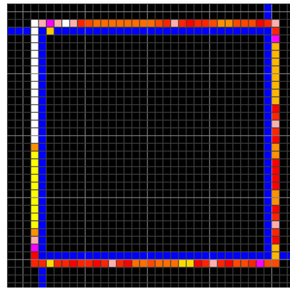
Obr. 4.4: Nosičová metóda pre 4 stavy

4.3 Komprimačná metóda

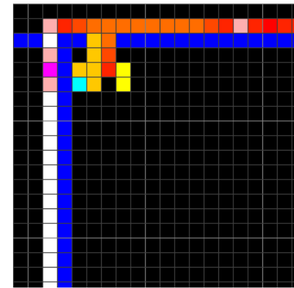
Generujeme automat s vedľajšou konfiguráciou o dvoch stavoch veľkosti 23×12.



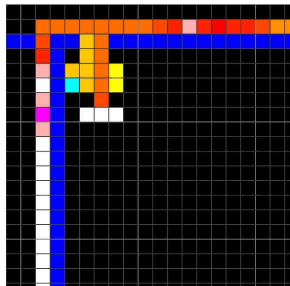
(a) Generovaná konfigurácia



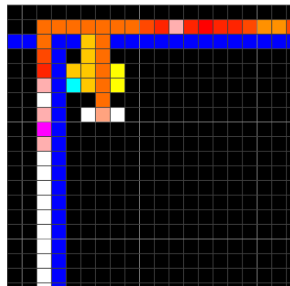
(b) Kontrolná sekvencia



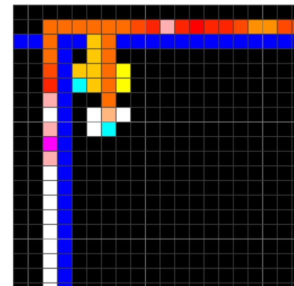
(c) Dva žlté stavy napravo sú nutné pre riadenie vykresľovania



(d) Červený signál vstupuje do nulového a vytvára tri pevné stavy naraz

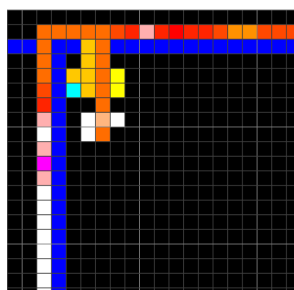


(e) Nasledujúci signál vstupuje do pevného stavu a vytvára nosič

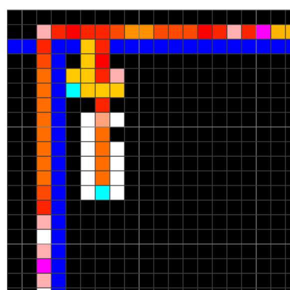


(f) Signál vychádza z nosiča a pri strete s nulovým stavom vytvára ďalšiu trojicu stavov `(1,0,0)`

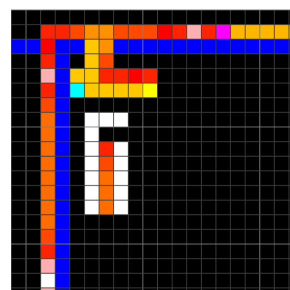
Obr. 4.5: Komprimačná metóda



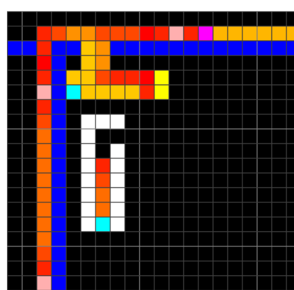
(a) Ďalší signál vstupuje do prázdneho nosiča



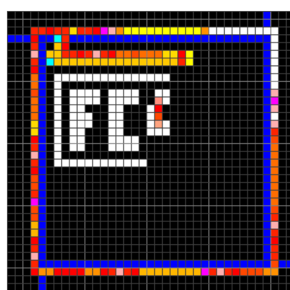
(b) Začína sa odklon do druhého stĺpca pomocou riadiaceho stavu



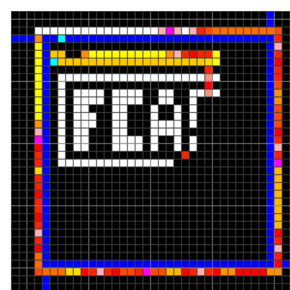
(c) Odklon do druhého stĺpca



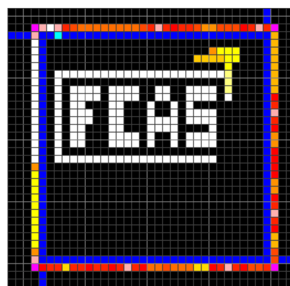
(d) Opäť potrebujeme dva žlté stavy napravo pre riadenie prúdu



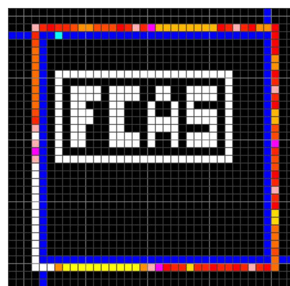
(e) Konfigurácia takmer hotová



(f) Brána sa zavrela a začína sa čistenie okolia



(g) Tesne pred ukončením tvorby konfigurácie

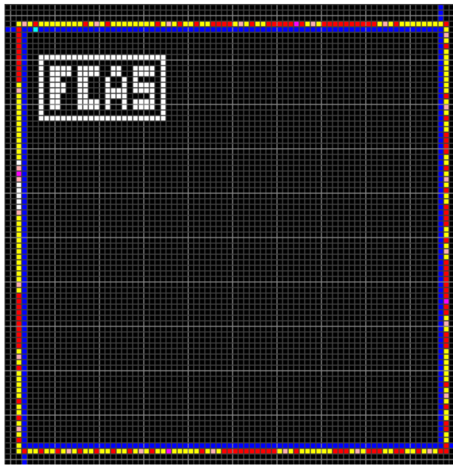


(h) Konfigurácia hotová

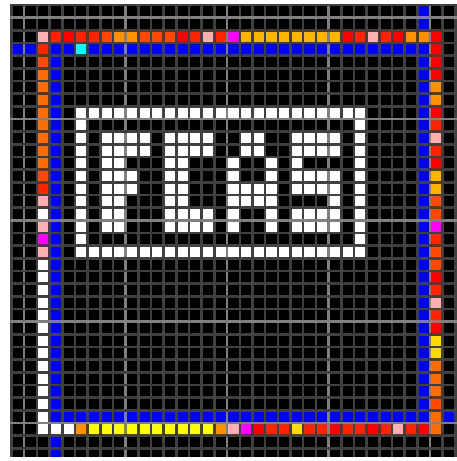
Obr. 4.6: Komprimačná metóda

Vygenerovaný automat má 883 pravidiel s premennými, po dosadení premenných má 1,809,474 pravidiel. Počet krokov automatu na vytvorenie tejto konfigurácie od východzej polohy je 202.

Na nasledujúcom obrázku sú pre porovnanie výstupné automaty z oboch metód pre rovnakú cieľovú konfiguráciu. Pri nosičovej metóde má automat rozmery 80x80 a pri komprimačnej metóde je to 34x34 Došlo teda ku kompresii o takmer 82%. Počet používaných stavov podľa týchto metód je 13 a 34.



(a) Nosičová metóda



(b) Komprimačná metóda

Obr. 4.7: Porovnanie veľkosti slučky pre obe metódy

4.4 Záver

V práci sme ukázali, že je možné vytvoriť seba-replikujúci automat, ktorý bude vyrábať počas svojho behu ľubovoľnú konfiguráciu. Ukázali sme, že je takéto automaty možné pomocou vhodného princípu vytvárať automaticky. Seba-replikujúce automaty nemusia byť založené na slučke, nosičový princíp je možné uplatniť u ľubovoľného automatu, ktorý dokáže uchovávať informácie v podobe datových stavov a takisto je schopný tieto stavy podľa potreby buď interpretovať alebo len prepisovať resp. zachovávať.

Ako je zrejmé z uvedených príkladov, je možné takto vytvárať ľubovoľne veľké konfigurácie, avšak rozmery slučky rastú kvadraticky s veľkosťou konfigurácie. Automat, ktorý by nemal tvar slučky, ale bol by schopný datové stavy uchovávať, by nemusel rásť kvadraticky a dokonca by mohol byť menší ako samotná konfigurácia. Je to preto, lebo posledná postupnosť nulových stavov stĺpca nemusí byť reprezentovaná signálmi.

V práci uvádzame metódu komprimácie viacerých stavov do jedného. Posledný príklad dobre ilustruje, aké zmenšenie priestoru je možné touto metódou dosiahnúť. Z implementačných dôvodov je používaná len pre cieľové konfigurácie o dvoch stavoch.

Možným vylepšením alebo pokračovaním práce môže byť skúmanie automatov, ktoré nemajú tvar slučky a teda vytvorenie automatu, ktorý dokáže vytvoriť konfiguráciu väčšiu ako on sám. Ďalším vylepšením môže byť skúmanie možnosti implementácie prechodových funkcií *Rule Table* pre vyšší počet stavov, keďže teraz program umožňuje generovať automaty pre konfigurácie len do piatich stavov a pri komprimačnej metóde len pre dvojstavové konfigurácie.

Dodatok A

FCAS - užívateľská dokumentácia

A.1 Úvod

Aplikácia je simulátorom vývoja celulárnych automatov. V prípade potreby využíva algoritmus Hash-Life[3], ktorý umožňuje extrémne zvýšiť rýchlosť výpočtu. Niektoré prechodové funkcie môžu potrebovať aj milióny krokov, aby sa naplno prejavili ich skúmané vlastnosti a vďaka tomuto algoritmu je možné takéto automaty simulovať. Aplikácia podporuje niekoľko rôznych formátov zápisu automatu, čo zaručuje plnú kompatibilitu s už používanými prostrediami. Podporuje multi-automatové prostredie, teda beh viacerých automatov naraz. Prostredie je plne prispôsobiteľné, čo sa dosahuje oknovým charakterom programu. Program podporuje päť rôznych prechodových funkcií:

- Life - prepokladá automat s dvoma stavmi a veľkosťou okolia 1. Definícia funkcie obsahuje počet živých (v stave 1) buniek v okolí potrebných pre oživenie bunky alebo jej urdžanie pri živote.

- Generations - umožňuje automaty s ľubovoľným počtom stavov. Bunky v tomto prípade nezomierajú, ale starnú (zvyšuje sa číslo ich stavu namiesto nastavenia na nulu). Funkcia je takisto závislá od počtu živých buniek v okolí.

- Larger Than Life - ako Generations, ale umožnuje navyše meniť veľkosť okolia automatu.

- Cyclic CA - číslo stavu bunky stúpa cyklicky (modulo počet stavov), ale len vtedy, ak je v okolí buniek zadefinované množstvo buniek v cieľovom stave.

- Rule Tables - všeobecný zápis prechodovej funkcie, pomocou ktorého je možné zadefinovať ľubovoľnú funkciu, špeciálne predošlé štyri. Tento spôsob definície je však oveľa menej názorny a preto sa používa len u komplikovaných automatov.

Hlavným účelom aplikácie je vytvoriť prívetivé prostredie pre prácu s viacerými automatmi naraz a umožniť pracovať s prechodovými funkciami typu Rule Tables pokiaľ možno najintuitívnejšie, keďže návrh konkrétneho automatu s funkciou tohoto typu môže byť veľmi neprehľadný. Rovnako dôležitá je možnosť ukladania už vytvorených automatov, prechodových funkcií a konfigurácií.

A.2 Rýchly návod

Po spustení aplikácie sa zobrazí štandartné prostredie, ktoré si užívateľ môže ľubovoľne upraviť. V strede sa nachádza automat s nulovým stavom v každej bunke. Pracovný názov automatu je na jeho záložke a typicky má tvar `A#1 inf-hlife`, kde číslo definuje poradie automatu, v ktorom bol vytvorený a text za identifikátorom je krátky popis automatu, napr. `inf-hlife` je priestorovo neobmedzený automat implementujúci algoritmus Hash-life. Priestorovo obmedzené automaty majú v identifikátore ich rozmery.

A.2.1 Rules - zadávanie prechodových funkcií

Vľavo dole je okno so záložkami, ktoré obsahuje definíciu prechodovej funkcie. Každá definícia má na záložke identifikátor automatu, ku ktorému patrí. Po nastavení prechodovej funkcie tlačidlo *Apply rule* nastaví prechodovú funkciu automatu. Prechodové funkcie sa nastavujú priamočiaro, podľa ich definície, jedinú výnimku tvorí typ funkcie Rule Tables, ktorej prechody sa nastavujú stlačením tlačidlo *Edit* a ich priamou editáciou ako opíšeme v nasledujúcom odstavci. Prechodová funkcia sa dá uložiť stlačením tlačidla *Save rule* buď ako .mcl alebo .table súbor.

V prípade potreby je možné ku každému automatu získať okno s jeho prechodovou funkciou `Menu`→`Automaton`→`Rule Definition`.

A.2.2 Editácia prechodovej funkcie Rule Tables

Toto okno má dva spôsoby zobrazovania prechodov funkcie a to textový a grafický. Tieto spôsoby sa prepínajú tlačidlom *Graphical/Textual*. Grafický mód slúži na lepšie zorientovanie sa v pravidlách, ale definíciu pravidiel nie je možné editovať priamo, len pridaním pravidla na koniec a to pomocou tlačidla *Add transition* na spodnej časti okna. Pridanie pravidla je priamočiare, výzor pravidla sa nastavuje graficky, stlačením ľavého tlačidla myši sa nastaví konkrétny stav, pravým tlačidlom sa pridáva premenná. Tlačidlo *Filter* aj s textovým poľom sa používa v grafickom móde na filtráciu pravidiel pomocou "AND" filtra. Pre sprehľadnenie je grafický mód okrem pravidiel a premenných schopný rozoznávať komentáre, nadpisy a veľké nadpisy, ktoré sa v texte definujú predponami `#`, `##` a `###` v takomto poradí. Pri chybe v syntaxi hlási okno chybu a stav automatu po stlačení *Apply rule* je nedefinovaný.

A.2.3 Nový automat

Nový prázdny automat je možné vytvoriť v `Menu`→`Automaton`→`New`. Tri typy automatov sú:

- Priestorovo obmedzený automat, kde sa zadávajú aj jeho rozmery
- Neobmedzený automat bez algoritmu Hash-life
- Neobmedzený automat používajúci algoritmus Hash-life

Takisto je možné už existujúci automat tzv. vyčistiť, teda nastaviť všetkým bunkám stav 0 a to v `Menu`→`Automaton`→`Clear`→`(identifikátor automatu)`, alebo je možné existujúci automat kopírovať v
`Menu`→`Automaton`→`Copy`→`(identifikátor automatu)`.

A.2.4 Načítanie a uloženie automatu

Vľavo hore je umiestnené okno *Rules*, ktoré slúži na rýchle načítavanie už vytvorených a uložených prechodových funkcií a konfigurácií v závislosti od načítaného súboru. Koreňovým adresárom je "rules" v adresári aplikácie. Formáty súborov sú bližšie špecifikované v sekcii A.4. Po kliknutí na vybraný súbor sa vytvorí nový automat s danou prechodovou funkciou, poprípade konfiguráciou. Po zavretí tohoto okna je možné ho opakovane otvoriť v

Menu→Window→Rule Tree, alebo ak došlo k zmene v adresári "rules", je možné ho načítať v Menu→Window→Refresh Rule Tree.

Automat je možné uložiť v Menu→File→Save Automaton→(identifikátor automatu). Ukladá sa do súboru s príponou .mcl, výnimkou je automat s prechodovou funkciou Rule Tables, ktorý sa ukladá do súboru s príponou .table. V tomto prípade sa neukladá konfigurácia automatu a musí sa uložiť manuálne pomocou *Pattern Library* do súboru .rle ako je popísané neskôr.

A.2.5 Základné ovládacie prvky

Na hornej lište pod menu sa nachádza základne ovládanie automatu a to:

- Zastavenie bežiaceho automatu.
- Jeden krok automatu.
- Spustenie behu automatu.
- Nastavenie rýchlosti, resp. prestávky medzi jednotlivými krokmi.
- Označenie - interakcia myšou bude vyvolávat označenie regiónu automatu.
- Kreslenie - priame nastavenie stavov bunkám automatu.
- Posúvanie - posúvanie zobrazenej časti automatu.
- Premiestňovanie - premiestňovanie označených konfigurácií.

A.2.6 Knižnica konfigurácií - Pattern Library

Pri práci s automatmi je niekedy vhodné mať možnosť uložiť si niektoré konfigurácie a na to slúži tzv. *Pattern Library*. Ovládanie knižnice je intuitívne, konfigurácia sa vyberá priamo kliknutím no súborový strom, ktorého koreňovým adresárom je "patterns" v adresári aplikácie. V prípade potreby je možné knižnicu vyvolať v Menu→Window→Pattern Library a znovunačítať adresárovú štruktúru vo vlastnom menu File→Refresh tree. Konfigurácie sa ukladajú do súborov spríponou .rle.

Knižnica umožňuje vytvoriť konfiguráciu z ľubovoľného obrázku vo File→Convert from image. V konvertovacom okne je možné načítať obrázok, nastaviť veľkosť konfigurácie a počet stavov a takisto určiť tzv. *Threshold*, čo je

konštanta, ktorá sa používa pri priraďovaní farieb pixelom a je užitočné ju meniť ak má obrázok netypický jas, resp. silné farby. Vytvorený pixelizovaný obrázok sa uloží ako konfigurácia do súboru.

A.2.7 Zmena stavu - State picker

S automatmi je možná interakcia priamym nastavovaním stavu jeho bunkám. Na výber konkrétneho stavu slúži *State Picker*, ktorý okrem výberu stavu umožňuje nastaviť nasledujúce schémy farieb:

- Linear priraďuje farby stavom ako lineárny prechod medzi dvoma zadanými farbami.
- Random priraďuje farby náhodne.
- Contrast 10 používa 10 základných kontrastných farieb. Zvyšné stavy dotvorí náhodne, preto sa odporúča používať len pri 10-stavových automatoch.
- Carrier TL sa používa pri automatoch vygenerovaných metódou "Carrier/Nosič" aplikáciou pre daný počet stavov a takisto využíva lineárne prechody medzi danými farbami.
- Compressed TL je takmer identický s predošlou triedou až na spôsob prechodu medzi farbami, ktorý je špecifický pre metódu "Compressed Carrier/Komprimovaný Nosič".

Posledné dve metódy sa používajú pri generovaní konkrétnych automatov a pre ich netriviálnosť v tomto texte nebudú popísané. Po výbere stavu je možná nastavovať stavy všetkým automatom a takisto konfiguráciam v *Pattern Library*.

A.2.8 Náhodné rozloženie stavov

V `Menu`→`Automaton`→`Fill` je možné pre priestorovo obmedzený automat nechať náhodne nastaviť jeho stavy, čo môže byť užitočné napríklad pri simulovaní automatu s prechodovou funkciou Cyclic CA. Po výbere priestorovo obmedzeného automatu sa otvorí nové okno s nastavením nasledovných parametrov:

- Area - plocha pokrytia nenulovými stavmi vyjadrená percentuálne.

- Max state - horná hranica pre čísla stavov, ktoré budú automatu nastavované. V prípade hornej hranice väčšej ako umožňuje prechodová funkcia automatu, budú vyššie stavy ignorované.
- Radius - polomer pokrytia so stredom v strede automatu. Polomer rovný nule znamená pokrytie celého automatu.

A.3 Používanie algoritmu Hash-life

Algoritmus Hash-life je možné plynule vypínať a zapínať, prípadne meniť veľkosť kroku, teda počet prechodov automatu medzi zobrazeniami jeho stavu. Veľkosť kroku je z povahy algoritmu mocnina dvojky a nastavuje sa v `Menu`→`Automaton`→`Step size` zadaním veľkosti kroku a stlačením klávesy `Enter`. Aplikácia neumožní zadanie veľkosti kroku inej ako mocnina dvojky a zaokrúhli vstup na najbližšiu takúto mocninu. Je možné takisto zapnúť aj režim, v ktorom program sám plynule zvyšuje veľkosť kroku spolu so zväčšujúcou sa plochou aktívnej časti automatu. Tento režim sa zapína v `Menu`→`Automaton`→`Fast Hashlife`.

A.4 Podporované formáty

Program bude používať syntax zápisu konfigurácie založenú na formáte RLE, MCL[10] a TABLE[11]. Formát RLE, navrhnutý D. Buckinghamom, je spôsob ako úsporne zapisovať konfigurácie automatov. Formát MCL je jeho rozšírením a okrem konfigurácie obsahuje aj informáciu o automate a jeho prechodovej funkcii. Formát TABLE je určený pre definovanie prechodových funkcií vo všeobecnom zápise Rule Table. Tento formát neobsahuje informáciu o konfigurácii automatu ani o automate samotnom.

A.4.1 RLE

Formát RLE využíva kombinácie písmen a čísel na lineárne definovanie konfigurácie:

- Stav 0 sa označuje znakom `.`.
- Ostatné stavy su označované znakmi `[A,B,...,X]` pre stavy 1,2,...,24. Pre vyššie stavy sa používajú prefixy `[a,b,...,x]`, ktoré reprezentujú

násobky čísla 24, teda napríklad [aA,aB,...,aX] zodpovedá stavom 25,26,...48.

- Koniec riadku sa označuje $.
- Číslo n pred stavom znamená n buniek s nasledovným stavom.
- *Príklad*: 5A.2B.$$2aA

A.4.2 MCL

Formát MCL využíva formát RLE, ale pred jeho použitím opisuje automat a jeho prechodovú funkciu. Takisto povoľuje komentáre.

- Riadok začínajúci #D je komentár. Táto aplikácia považuje každý riadok začínajúci # za komentár.
- Riadok #GAME (názov rodiny CA), kde názvom rodiny môžu byť nasledovné: Life, Larger than Life, Generations, Cyclic CA, určuje typ prechodovej funkcie.
- Riadok #BOARD (šírka)x(výška) určuje rozmery automatu.
- Riadok #RULE (...) určuje prechodovú funkciu, ktorá sa pre rôzne automaty líši a je definovaná nasledovne:

  Life S/B

  - S Počet živých buniek, pri ktorých bunka prežíva.
  - B Počet živých buniek, pri ktorých bunka ožíva.

  Generations S/B/C

  - S,B Definované ako pri Life s rozdielom, že živé bunky nezomierajú, ale starnú, teda ich stav sa zvyšuje o 1 a bunky staršie ako 1 neprispievajú k novým bunkám.
  - C Počet stavov automatu.

  Larger than Life R,C,M,S,B,N

  - R Polomer okruhu buniek, ktorých stavy sú parametrami prechodovej funkcie.
  - C Počet stavov automatu.
  - M Centrálna bunka prispieva k výpočtu (1) alebo nie (0).

S Intervaly počtov živých buniek, pri ktorých bunka prežíva, napríklad S2..7.

B Intervaly počtov živých buniek, pri ktorých bunka ožíva, napríklad B1..3.

N Typ okolia bunky, NN pre von Neumanove, NM pre Moorovo.

Cyclic CA R,T,C,N

R Polomer okruhu susedských buniek.

T Počet buniek okolia v nasledujúcom stave potrebných na prechod do ďalšieho stavu.

C Počet stavov automatu.

N Typ okolia bunky, NN pre von Neumanove, NM pre Moorovo.

- Riadky #L (...) na konci definície využívajú syntax RLE na zápis konfigurácie automatu.

Príklad definície automatu v MCL formáte

```
#GAME Larger than Life
#RULE R10,C255,M1,S2..3,B3..3,NM
#BOARD 120x120
#D Most simple patterns expand into lines and blocks
#D of color, except for the really simple ones, which
#D form stable power blocks (literally).
#D
#D Discovered by Charles A. Rockafellor, March 2000
#L 4A$4.AA15.6A$6.AA11.AA$8.3A5.3A$11.AA..A$13.AA$12.A
#L ..AA$11.A5.AA$11.A7.AA$10.A10.3A$9.A14.AA$8.A17.AA$
#L 7.A20.AA$6.A23.AA$5.A26.3A$4.A30.A$36.A$37.A$38.A$
#L 39.AA$41.AA$42.A
```

A.4.3 TABLE

Formát TABLE je určený pre definíciu všeobecného zápisu prechodovej funkcie, kde sú definované pravidlá pre konkrétne konfigurácie okolia. Ak pre nejaké okolie pravidlo chýba, bunka ostáva po prechode v rovnakom stave.

- Riadok začínajúci # je komentár.

- Povinný riadok `n_states:N`, kde `N` je počet stavov automatu.
- Povinný riadok `neighborhood:(typ okolia)` definuje okolie a možnosti sú `vonNeumann` a `Moore`.
- Povinný riadok `symmetries:(typ symetrie)` sa používa pri pravidlách, ktoré sú priestorovo symetrické a povoľuje hodnoty `none`, `rotate4`, `rotate8`, `reflect`, `rotate4reflect` a `rotate8reflect`.

Ďalej sa syntax líši podľa druhu okolia:

von Neumann
: Nasledujú riadky typu `CNESWC`, kde písmená `C` predstavujú stav bunky, pre ktorú pravidlo definujeme a výsledný stav po prechode. Písmená `NESW` predstavujú stavy buniek v okolí podľa názvov hlavných svetových strán v anglickom jazyku.

Moore
: Nasledujú riadky typu `CNNEESESSWWC`, kde písmená `C` predstavujú stav bunky, pre ktorú pravidlo definujeme a výsledný stav po prechode. Písmená a dvojice písmen `NNEESESSWW` predstavujú stavy buniek v okolí podľa názvov ôsmich svetových strán v anglickom jazyku.

Príklad definície automatu v TABLE formáte

```
# Langton's Loops
#
# C.G.Langton. "Self-reproduction in cellular automata."
# Physica D, Vol. 10, pages 135-144, 1984.
#
# transition rules from: http://necsi.org/postdocs/sayama/
# /sdsr/java/loops.java
# credits: "Self-Replicating Loops & Ant, Programmed by
# Eli Bachmutsky, Copyleft Feb.1999"
#
# states: 8
# rules: 219
# format: CNESWC
#
n_states:8
neighborhood:vonNeumann
```

```
symmetries:rotate4
000000
000012
000020
000030
000050
000063
000071
000112
000122
000132
000212
000220
000230
000262
...
```

Dodatok B

FCAS - programová dokumentácia

Program je napísaný v programovacom jazyku Java, na spustenie je potrebné prostredie Java JRE 6.0. Program využíva externú knižnicu InfoNode Docking Windows pre prostredie s oknami. Program sa delí do štyroch hlavných balíkov: *fcas*, *fcas.automatons*, *fcas.ruleComputers*, *fcas.gui*.

B.1 Návrh - UML

Do UML diagramu boli pre prehľadnosť vybraté len najdôležitejšie triedy a je logický rozdelený na *Automaty a prechodové funkcie* a *GUI*.

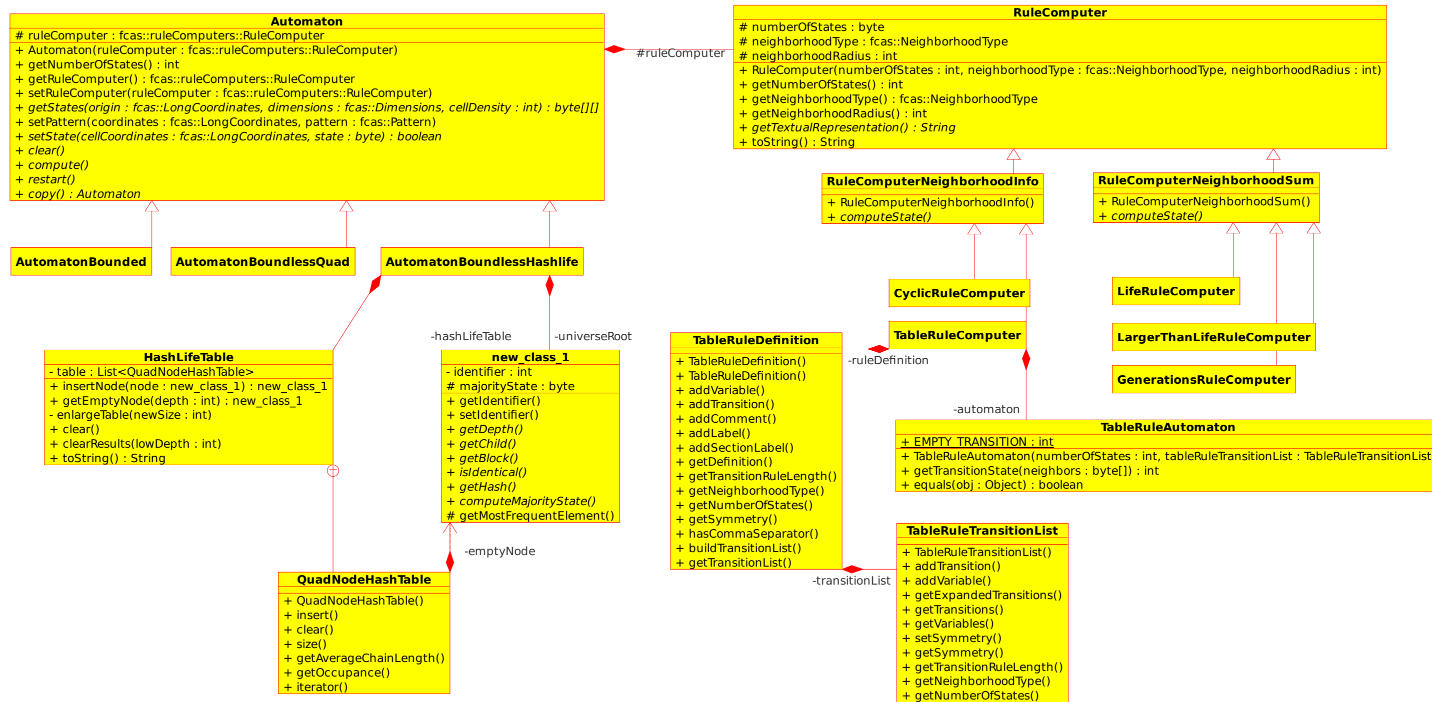


Diagram: Automaton & Rule Computer Page 1

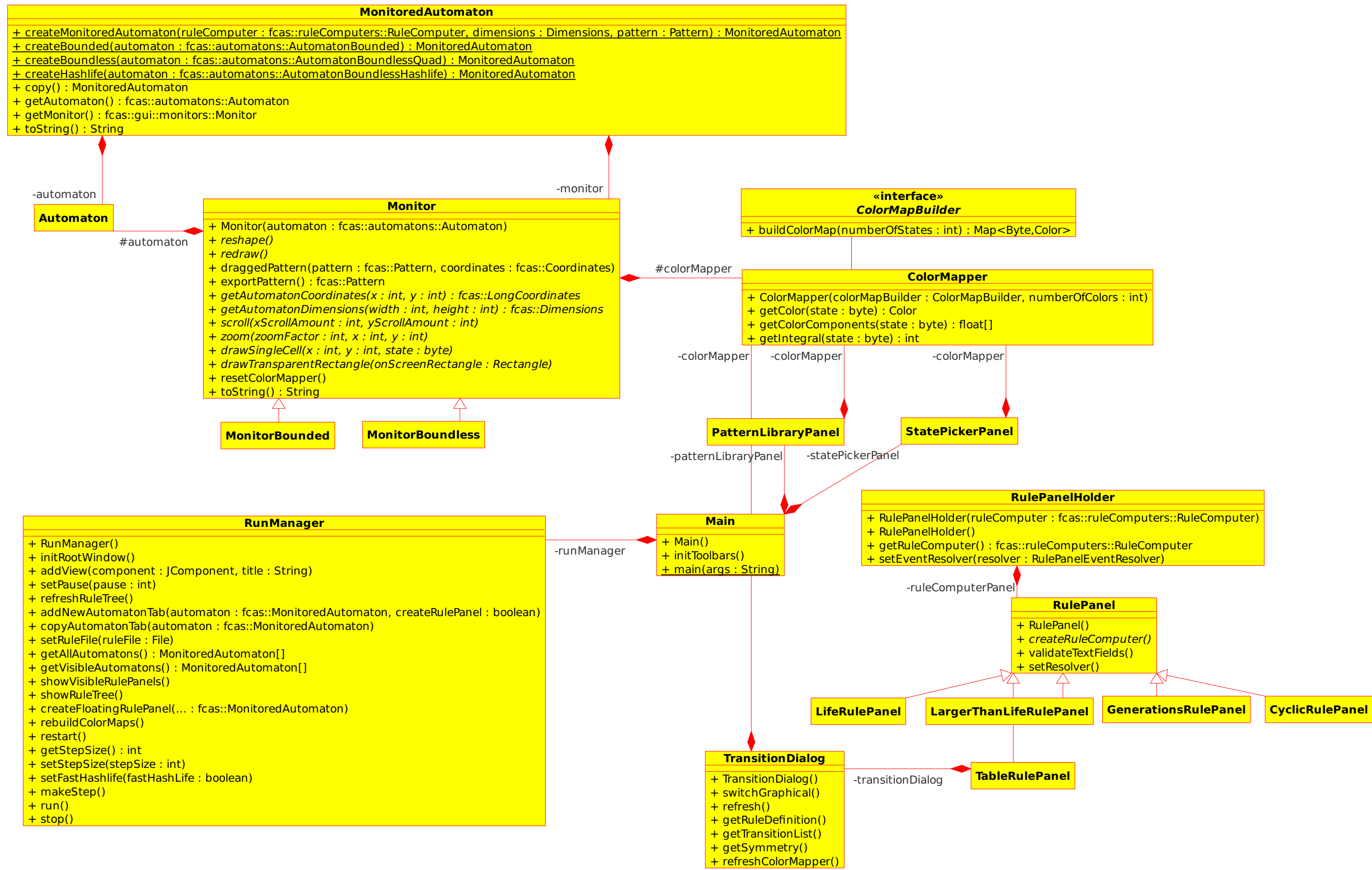


Diagram: GUI Page 1

B.2 Automaton - automat

Automaton je základnou triedou balíka *fcas.automatons* a aj celého programu. Táto abstraktná trieda reprezentuje samotný celulárny automat a jej najdôležitejšími metódami sú *Automaton.getStates()* a *Automaton.compute()*, teda rozhranie medzi automatmi a monitormi a operácia jedného diskrétneho kroku. Trieda *Automaton* obsahuje chránený atribút *Automaton.ruleComputer*, ktorý obsahuje inštanciu druhej najdôležitejšej triedy: *RuleComputer*.

AutomatonBounded je jednoduchý automat reprezentovaný ako dvojrozmerné pole bytov (každý automat používa pre uloženie stavov byte kvôli úspore z čoho však plynie obmedzenie maximálneho počtu stavov na 256). Pomyslený povrch tohoto automatu je torus a jeho protiľahlé strany sú totožné, takže bunky na pravom majú v okolí bunky z ľavého okraja a pod.

AutomatonBoundlessQuad je automat s neobmedzeným priestorom implementovaný ako strom, kde každý vrchol má štyroch nasledovníkov, teda tzv. quad-tree. Pokrýva len priestor, ktorý obsahuje aktívne bunky. Najmenší prvok tohoto automatu je dvojrozmerné pole bytov ľubovoľnej veľkosti (program momentálne využíva veľkosť bloku 12x12). Na každej vrstve sú uzly prepojené so svojimi priamymi susedmi pre jednoduchosť vyhľadávania susedov pre každú bunku. Trieda umožňuje rôzne operácie pre prácu so stromom, hlavne expanziu vo vrcholoch, kde vzniká možnosť, že automat nebude pokrývať priestor s aktívnymi bunkami alebo napr. zabezpečovanie prepájania listových uzlov.

AutomatonBoundlessHashlife je priestorovo neobmedzený automat implementujúci algoritmus Hash-life. Je takisto reprezentovaný ako quad-tree, ale uzly nie sú mimo stromovej štruktúry vzájomne prepojené. Beh automatu je opísaný v sekcii B.4.

B.3 RuleComputer - prechodová funkcie

RuleComputer je reprezentáciou prechodovej funkcie a za atribúty má základné informácie o automate ako počet stavov, typ susedstva alebo veľkosť susedstva. Táto trieda má dve hlavné podtriedy: *RuleComputerNeighborhoodInfo* a *RuleComputerNeighborhoodSum*.

RuleComputerNeighborhoodInfo reprezentuje prechodové funkcie, ktoré pre svoj výpočet potrebujú počet živých buniek v okolí bunky: *LifeRuleComputer* (implementuje známu Conway's Game of Life), *GenerationsRuleComputer* a *LargerThanLifeRuleComputer*.

RuleComputerNeighborhoodInfo reprezentuje prechodové funkcie, ktoré pre svoj výpočet potrebujú kompletnú informáciu o susedoch bunky: *CyclicRuleComputer* a *TableRuleComputer*.

TableRuleComputer je úplným zovšeobecnením prechodovej funkcie, kde je možné nadefinovať ľubovoľné pravidlá tak ako je uvedené v sekcii A.4.3. Táto implementácia využíva triedy *TableRuleDefinition* a *TableRuleTransitionList*, kde prvá trieda reprezentuje celú kolekciu pravidiel včetne premenných, komentárov a nadpisov dvoch úrovní. Druhá trieda reprezentuje len premenné a pravidlá a tieto zapuzdruje, takže zatiaľ čo v definícii pravidiel môžu byť nezrovnalosti, napr. nedeklarované premenné alebo zlá dĺžka pravidla, v tejto triede sú už všetky pravidlá v korektnom tvare. Tieto pravidlá ďalej využíva *TableRuleAutomaton*, čo je reprezentácia konečného automatu, ktorý počíta prechody pre jednotlivé bunky pre *TableRuleComputer*.

TableRuleAutomaton táto reprezentácia je v skutočnosti najbližšia predstave celulárneho automatu, ktorého každá jednotlivá bunka má byť konečným automatom. V praxi je táto implementácia vhodná pre konštantý výpočet a pre priestorovú úspornosť. Konečný automat je implementovaný ako strom, kde každý uzol okrem listov má svojich nasledovníkov uložených v poli.

B.4 Hash-life

Algoritmus bol implementovaný podľa abstraktnej predlohy W. Gospera[3]. Automat je reprezentovaný ako quad-tree, jeho listové uzly sú 2x2 polia bytov a jeho najdôležitejšou vlastnosťou je, že obsahuje každý unikátny uzol len raz. Listové uzly sú rovnaké, ak sa ich polia 2x2 zhodujú a rekurzívne sú vyššie uzly rovnaké, ak sa zhodujú ich nasledovníci. Toto sa zabezpečuje hašovacou tabuľkou, kde je uložený každý existujúci vrchol. Tento program využíva obalovú triedu *HashLifeTable*, kde každá úroveň stromu má svoju vlastnú hašovaciu tabuľku *QuadNodeHashTable*.

QuadNodeHashTable je už konkrétna implementácia hašovacej tabuľky. Ideálne by sa mohlo hašovať podľa fyzickej adresy vrcholu v pamäti, kvôli obmedzeniu jazyka Java však hašujeme podľa identifikátora, ktorý dostane každý uzol pri svojom vzniku. Hašovanie je jednoduché, po prekročení limitu sa prehašováva naraz celá tabuľka do tabuľky dvojnásobnej veľkosti. Pri maximálnej zaplnenosti 85% je priemerná dĺžka reťazca okolo 1.5, čo je úplne dostačujúce. Pri tvorbe programu boli uvažované aj dynamické alebo rozšíriteľné formy hašovania, ale v tomto prípade by boli nevhodné, kedže operácia find musí byť extrémne rýchla a tieto spôsoby hašovania buď zvyšujú koštantný čas prístupu (Larson & Kajla), alebo potrebujú na rozšírenie rovnaký čas ako obyčajná tabuľka (Cormack). To samozrejme vyplýva z toho, že ich hlavný účel je hašovanie stránok pevného disku. Pri profilovaní programu nástrojmi NetBeans bolo zistené, že prehašovanie tabuľky naivným spôsobom neprináša takmer žiadne spomalenie a preto bolo vylúčené aj postupné prehašovávanie. Identifikátory uzlov sú uniformne rozdelené (ide o jednoduchú postupnosť prirodzených čísel) a preto používame jednoduchú hašovaciu funkciu $f(x_1, x_2, x_3, x_4) = x_1 + 3x_2 + 9x_3 + 27x_4$.

B.5 Infonode DockingWindows

Program využíva externú knižnicu *DockingWindows* od InfoNode (`http://www.infonode.net/index.html?idw`), ktorá umožnuje ľubovoľne prispôsobiteľné prostredie. Uživateľ može používať okná, okná so záložkami (*TabbedWindow*), delené okná (*SplitWindow*) a ich kombinácie. Táto trieda sa používa pomocou vnorenej triedy *Main.RunManager* implementujúcej návrhový vzor fasáda. Táto trieda umožňuje vytvárať nové okná, drží si informácie o existujúcich oknách a pri ich zatváraní vykonáva potrebné operácie (napr. zahodenie referencie na automat akonáhle bolo jeho okno uživateľom zatvorené). Pri vytváraní nových okien s automatmi alebo definíciami prechodových funkcií napríklad zabezpečuje, aby sa otvárali v tom istom záložkovom okne. Pri iniciovaní prechodu automatu spustí výpočet na všetkých viditeľných automatoch a pri spustení behu automatu vytvára vlákna, ktoré po danom intervale spúšťajú výpočty na všetkých viditeľných automatoch.

B.6 Monitor

Monitor je ústrednou triedou balíka fcas.monitors. Zabezpečuje grafické zobrazovanie automatov a interakciu užívateľa s automatmi, napr. jednotlivé nastavovanie stavov bunkám alebo prenášanie celých konfigurácií medzi automatmi alebo medzi automatom a knižnicou konfigurácií (*PatternLibraryPanel*). Monitor je abstraktná trieda, jej podtriedy musia implementovať v prvom rade metódy *redraw()*, *reshape()* a iné, napríklad *zoom()* a *scroll()*. Monitor má dve implementácie: *MonitorBounded* pre priestorovo obmedzené automaty a *MonitorBoundless* pre neobmedzené.

Pri vykresľovaní jednotlivých stavov potrebuje monitor presnú informáciu o zobrazení stavov do farieb a tú mu poskytuje *per*.

ColorMapper zobrazuje stavy na farby a vracia farby v rôznych reprezentáciách. V aplikácii sú inštancie *ColorMapper* vytvárané výhradne implementáciami rozhrania *ColorMapBuilder*.

ColorMapBuilder toto rozhranie má metódu *buildColorMap()*. Aplikácia obsahuje niekoľko implementácií, ktoré si užívateľ môže zvoliť v komponente *StatePickerPanel* tak, ako je uvedené v sekcii A.2.7.

B.7 Ošetrenie chýb

Na ošetrenie chýb používa aplikácia viacero mechanizmov, najmä ich logovanie do externého súboru. Ak si to typ chyby vyžaduje, napríklad chyba počas parsovania súboru s definíciou prechodovej funkcie, program zobrazí okno s informáciou o chybe. Podrobný opis chyby loguje trieda *Logger* do súboru *fcas.log* v koreňovom adresári programu.

Literatúra

[1] Byl John (1989): Self-Reproduction in small cellular automata, Physica D, vol. 34, 295-299.

[2] Codd Edgar F. (1968): Cellular Automata, Academic Press, New York.

[3] Gosper William (1984): Exploiting regularities in large cellular spaces, Physica D, vol. 10, no. 1-2, 75-80.

[4] Langton Christopher, G. (1984): Self-Reproduction in cellular automata, Physica D, vol. 10, 135-144

[5] Neumann, John v. (1966): Theory of Self-reproducing Automata, University of Illinois Press.

[6] Nobili Renato, Pesavento Umberto (1994): John von Neumann's Automata Revisited.

[7] Pesavento Umberto (1995): An implementation of von Neumann's self-reproducing machine. Artif. Life 2, 337-354.

[8] Reggia James A., Armentrout Steven L., Chou Hui-Hsien, Peng Yun (1993): Simple systems that exhibit self-directed replication, Science 259, 1282–1287.

[9] Tempesti Gianluca (1997): A robust multiplexer-based FPGA inspired by biological systems, Journal of Systems Architecture: the EUROMICRO Journal archive, vol. 43, issue 10, 719 - 733.

[10] Wôjtowicz Mirek: Formát MCL, http://www.mirekw.com/

[11] Formát TABLE,
http://code.google.com/p/ruletablerepository/wiki/TheFormat