VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

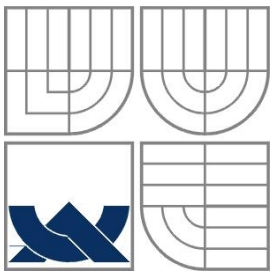
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SIMULÁTOR NANOPOČÍTAČE NA BÁZI CELULÁRNÍHO AUTOMATU

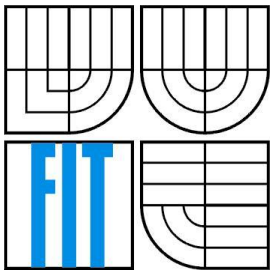
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. DUŠAN KMEŤ
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SIMULÁTOR NANOPOČÍTAČE NA BÁZI CELULÁRNÍHO AUTOMATU

A NANOCOMPUTER SIMULATOR USING CELLULAR AUTOMATON

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DUŠAN KMEŤ

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. LUKÁŠ SEKANINA, Ph.D.

BRNO 2012

Zde bude vloženo zadání

Abstrakt

Tato diplomová práce se zabývá realizací simulátoru na bázi asynchronního celulárního automatu simulujícího obvody odolné vůči zpoždění. Celulární automaty mají z pohledu nanotechnologií několik zajímavých vlastnosti využitelných při fyzické implementaci nanopočítačů na nich založených. Těmito jsou hlavně sebereplikace, regulérnost a výrazný paralelismus. V práci je uveden vztah mezi nanopočítači a celulárními automaty, přičemž důraz je kladen na možnosti využití asynchronního módu časování. Asynchronní celulární pole vycházející z modelu asynchronního celulárního automatu by mohly být vhodnou architekturou pro budoucí nanopočítače, což bylo důvodem implementace simulátoru uvedené technologie. Jeho funkčnost byla ověřena na základě experimentů.

Abstract

This master thesis deals with the realization of a simulator based on asynchronous cellular automata simulating delay insensitive circuits. In connection with nanotechnology, cellular automata have several interesting properties, such as self-replication, regular structure and high parallelism that make them very useful as models for some types of nanocomputers. This text describes the relationship between cellular automata and nanotechnology. Emphasis is given to the possibility of using asynchronous timing mode. Asynchronous cellular arrays based on asynchronous cellular automata could prove to be a suitable architecture for future nanocomputer, which was the reason for implementation of this simulator. The simulator's functionality was verified by experiments.

Klíčová slova

celulární automat, nanopočítače, asynchronní obvody, obvody odolné vůči opoždění, simulátor, celulární pole, nanotechnológie

Keywords

cellular automata, nanocomputer, asynchronous circuits, delay insensitive circuits, simulator, cellular arrays, nanotechnology

Citace

Dušan Kmeť: Simulátor nanopočítače na bázi celulárního automatu, diplomová práce, Brno, FIT VUT v Brně, 2012

Simulátor nanopočítače na bázi celulárního automatu

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Prof. Ing. Lukáša Sekaninu, Ph.D. a uviedol som všetky literárne zdroje a publikácie, z ktorých som čerpal.

……
Dušan Kmeť
23. 5 2012

Poďakovanie

Týmto by som chcel poďakovať pánovi Prof. Ing. Lukášovi Sekaninovi, Ph.D. za odbornú konzultáciu a pripomienky pri vypracovaní tejto diplomovej práce.

Obsah

1 Úvod

Už od 60-tých rokov platí v oblasti vývoja počítačov Moorov zákon, ktorý tvrdí, že počet tranzistorov integrovaných na čipe sa každé dva roky zdvojnásobí. Tento trend bol doteraz dobre splniteľný, pretože používané technológie umožňovali minimalizovať komponenty obvodov. Avšak technologický vývoj sa dostáva do situácie, kde ekonomické a hlavne fyzikálne limity budú brániť v pokračovaní miniaturizácie tranzistorov, navyše pri ďalšom zvyšovaní hustoty integrácie vznikajú problémy spojené so vznikom prebytočného tepla [11]. Východisko z tejto situácie sú nanotechnológie, ktoré predpokladajú, že je možné manipulovať s atómami alebo molekulami. Aj keď v dnešnej dobe je technologicky možné pracovať s hmotou na úrovni nano rozmerov, oproti konvenčnému prístupu existuje rada nevýhod, ako sú nepreskúmaná architektúra, nízka spoľahlivosť, vysoká cena, takmer žiadna výrobná infraštruktúra, a iné. Napriek rady problémov spojených s nanotechnológiami, je stále narastajúci záujem o túto technológiu, pretože výhody sú značné.

Existuje niekoľko druhov prístupov k nanotechnológiam. Jednou z nich je využitie znalostí synchrónnych obvodov založených na logických členoch a nahradenie technológie CMOS nejakou nanotechnológiou založenou na atómoch či molekulách. Takáto činnosť vyžaduje vysokú presnosť a znalosť technológie, čo sa určite odrazí na cene. Tento prístup k návrhu je možné charakterizovať ako zhora – nadol. Ďalšou možnosťou je opustiť konvenčné technológie a obvody realizovať na iných architektúrach. Tieto architektúry by mohli byť založené na modele celulárnych automatov zložených z buniek, ktorých hlavným rysom je pravidelná štruktúra. To môže viesť k masívnej výrobe za nízku cenu, napríklad chemickou reakciou. Aby bolo možné realizovať nanopočítače na modeloch založených na celulárnych automatoch, je nutné k tomu prispôsobiť aj návrh obvodov. Zatiaľ čo logické obvody dnes obvykle spoliehajú na synchronizáciu, u celulárnych automatov by fyzická implementácia synchronizácie nebola efektívna. Existujú možnosti, ako zabezpečiť synchronizáciu na vyššej úrovni, napríklad simulovaním celulárneho automatu, ale réžia je príliš veľká a výkon nepostačujúci. Vývoj sa preto začína zameriavať na asynchrónne obvody, známe predovšetkým absenciou globálneho synchronizačného signálu, modulárnosťou a predovšetkým odolnosťou voči oneskoreniu. Napriek týmto výhodám, boli rozšírené synchrónne obvody, čo mohlo byť spôsobené jednoduchším návrhom, lepšou testovateľnosťou a hlbšími znalosťami v porovnaní s asynchrónnymi. V poslednej dobe v spojitosti s nanotechnológiami stále narastá záujem práve o asynchrónne obvody, konkrétne o ich podtriedu a to obvody odolné voči oneskoreniu.

V poslednom desaťročí vznikali viaceré systémy vychádzajúce z modelu celulárnych automatov s cieľom mať možnosť realizácie asynchrónnych obvodov. Jeden z prvých systémov predstavila v roku 2004 skupina autorov Susumu Adachi, Ferdinand Peper, a Jia Lee [5], [12]. Jednalo sa o teoretický model, takzvaný CA s vlastným časovaním, ktorý bol schopný realizácie ľubovoľného asynchrónneho obvodu, inak povedané bol schopný výpočtu. Následne bola prezentovaná jeho praktickejšia forma pod názvom asynchrónne celulárne polia, prvýkrát uvedená v článku [8], z ktorého vychádza aj táto diplomová práca.

Cieľom diplomovej práce je realizovať simulátor založený na modele celulárnych automatov s vlastným časovaním využívajúci asynchrónne celulárne polia. V článku, z ktorého vychádzame, boli predstavené základné primitíva na tvorbu obvodov. V tejto práci sa pokúsime ukázať, že na základe týchto primitív sme schopný konštruovať zložitejšie asynchrónne obvody, dokonca i nanopočítače.

Čitateľ je v druhej kapitole zoznámený s celulárnymi automatmi, ktorých pochopenie bude v tejto práci kľúčové.

V tretej kapitole sú bližšie vysvetlené nanotechnológie, ich súvislosť s celulárnymi automatmi a rozdiel medzi synchrónnym a asynchrónnym prístupom. Dôsledkom asynchrónnych obvodov sú odvody odolné voči oneskoreniu, ktorým je tiež venovaná značná pozornosť. V závere kapitoly bude kladený dôraz na toleranciu chýb a to jednak pri výpočte, ale aj pri výrobe, ktorých prekonanie alebo minimalizácia je pre úspešný výpočet pomocou nanopočítačov rozhodujúcim faktorom.

Asynchrónne celulárne polia, pomocou ktorých je možné realizovať asynchrónne obvody, sú detailne rozobrané v kapitole štyri. Kapitola vysvetľuje princíp fungovania výpočtu na celulárnych poliach a spôsob predávania informácie medzi bunkami. Uvedené sú základné primitíva na distribúciu signálu v poli buniek, ale tiež je ukázaná možnosť, ako z jednoduchých častí zložiť základné komponenty nanopočítačov.

Kapitola päť sa venuje návrhu simulátora celulárnych polí. Proces návrhu bude rozdelený na dve časti. V prvej časti bude popísaná simulačná vrstva a navrhnutý algoritmus vykonávajúci aktualizáciu buniek. Na simulačnú vrstvu bude nadväzovať druhá časť, kde bude navrhnuté užívateľské rozhranie a architektúra umožňujúca efektívne napojenie na simulačnú vrstvu.

Na základe tohto návrhu bude v kapitole šesť popísaná implementácia simulátoru. Zameriame sa na konkrétne implementačné detaily spojené s realizovaním kvalitného užívateľského rozhrania a aplikácie optimalizovanej k návrhu modulárne založených obvodov.

Experimenty a testovanie funkčnosti simulátoru nájdeme v kapitole sedem. Na testovanie boli vybrané obecne známe komponenty, ako je AND hradlo, register, úplná jednobitová sčítačka a iné. V závere kapitoly bude zhrnutie vykonaných testov i celého procesu testovania.

2 Celulárne automaty

Inšpirácia k vytvoreniu celulárnych automatov pochádza zo živých bunečných systémov, ktoré majú podobnú štruktúru ako z nich odvodený výpočtový model. Keďže sa jedná o model, bola vykonaná určitá abstrakcia. Bunečná štruktúra je reprezentovaná diskrétnym n-rozmerným priestorom. Komplexný stav buniek je vyjadrený na základe stavu pamäti. Interakcia medzi bunkami na základe proteínov bola prevedená na matematické funkcie alebo pravidlá, ktoré určujú ako sa bude stav bunky meniť v čase v závislosti na stavu susedných buniek [10].

V úvode kapitoly budú popísané základné vlastnosti a definície celulárnych automatov ako výpočtového modelu prebrané z [5]. Dôležité bude pochopenie prechodových funkcií a spôsob aktualizácie automatu. Ako prvý bude formálne definovaný obecný celulárny automat. Nasledovať bude popis modifikovaného celulárneho automatu založeného na asynchrónnej aktualizácii buniek, z ktorého budú vychádzať nasledovné kapitoly, preto je dôležité jeho pochopenie.

2.1 Základné vlastnosti

Celulárne automaty (CA) pôvodne zaviedli Ulam a von Neumann v 40-tých rokoch minulého storočia ako model pre vyšetrovanie chovania zložitých systémov a sebareplikácie.

Celulárny automat je dynamický systém, ktorý je možné interpretovať ako kolekciu konečných automatov nazývaných bunky, kde každá z nich sa môže nachádzať v jednom stave z konečnej množiny stavov. Geometrické usporiadanie buniek CA je špecifikované jeho dimenziou, ktorá môže byť jednorozmerná, dvojrozmerná, prípadne i viacrozmerná. Stav bunky je aktualizovaný synchrónne v diskrétnych časových krokoch v závislosti na lokálnom prechodovom pravidle, ktoré určuje nasledujúci stav každej bunky v závislosti na aktuálnom stave tejto bunky a na stavoch buniek v jej susedstve. Konfiguráciou celulárneho automatu rozumieme súhrnný stav všetkých buniek automatu v kroku *i*. Výpočet celulárneho automatu je definovaný ako postupnosť konfigurácií, kde z konfigurácii v kroku *i* do konfigurácie v kroku $i + 1$ sa dostaneme aplikovaním lokálneho prechodového pravidla vo všetkých bunkách. Pre obecný typ celulárneho automatu sú charakteristické vlastnosti paralelizmu, homogenity a lokality [7].

Paralelizmom máme na mysli skutočnosť, že výpočet hodnôt stavov všetkých prvkov celulárneho automatu prebieha súčasne. Táto vlastnosť je veľmi výhodná pre využitie v paralelných systémoch a tým i dosiahnutie veľkej rýchlosti pri behu simulácie. Avšak pri behu na bežných strojoch sa musí paralelizmus simulovať.

Homogenita v pôvodnom zmysle vyjadrovala totožnosť prechodovej lokálnej funkcie pre všetky bunky automatu, ale to neplatí u neuniformných celulárnych automatoch.

Lokalita hovorí, že nový stav závisí iba na stave konkrétnej bunky a na stavu susedstva. Táto vlastnosť bude kľúčovou v oblasti nanopočítačov, pretože je postačujúce zabezpečiť komunikáciu medzi bunkami na lokálnej úrovni.

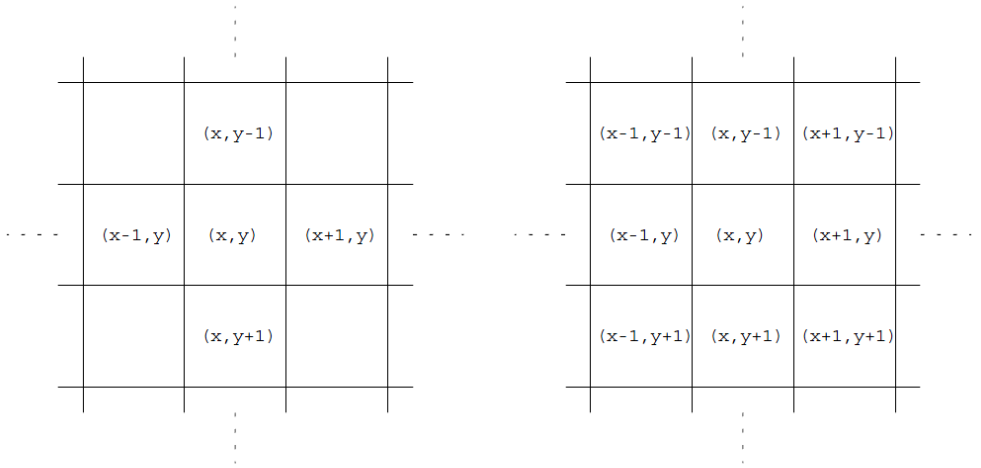
Štruktúra celulárneho automatu môže byť charakterizovaná *geometriou bunečnej mriežky*, *tvarom susedstva* a *počtom stavov bunky*. Bunky sú organizované do pravidelných mriežok. Najčastejšie sú mriežky tvorené bunkami štvorcového charakteru, ale exitujú CA, kde bunky majú hexagonálny alebo trojuholníkový tvar. Stav bunky reprezentuje stavová premenná. Stavová premenná môže nadobúdať jednu hodnotu z konečnej množiny prípustných hodnôt bunky Q. Často sa definuje špeciálny stav q_0, ktorý predstavuje neaktívny stav bunky.

2.2 Definícia celulárneho automatu

Definícia 2.1: *Deterministický celulárny automat (alebo jednoduchšie celulárny automat) je systém definovaný päticou* $A = (Z^d, N, Q, f, q_0)$, *kde Z je množina celých čísel takých, že* Z^d *reprezentuje d-rozmerné pole buniek, pričom* $d \geq 1$ *a* $d \in \mathcal{N}$. *N sa nazýva index okolia určený n-ticou* $(n_1, n_2, \ldots, n_n)$, *kde každé* $n_i \in Z^d$ *a* $n_1 = (0, 0, \ldots, 0)$. *Majme bunku* $a \in Z^d$, *potom každá bunka v množine* $S = \{(a+n_1), \ldots, (a+n_n)\}$ *je susednou bunkou bunky a. Množina S je indukovaná množinou N a nazýva sa okolie bunky* $a \in Z^d$. *Ďalej Q je konečná neprázdna množina prípustných stavov bunky, pričom bunka je vždy v nejakom stave* $q \in Q$. *Funkcia f definovaná ako* $f : Q^n \rightarrow Q$, *kde n je počet susedných buniek, je mapovanie volané lokálna prechodová funkcia.* $q_0 \in Q$ *je kľudový stav, kedy* $f(q_0, \ldots, q_0) = q_0$. *Konfigurácia v celulárnom automate A je mapovanie* $c : Z^d \rightarrow Q$, *čo priraďuje každej bunke* $a \in Z^d$ *stav* $q \in Q$ *[5].*

2.2.1 Okolie bunky

Formálne sme si definovali celulárny automat s dimenziou *d*. V praxi sa najčastejšie používajú jedno a dvojdimenzionálne celulárne automaty. V tejto práci bude pre nás dôležitý práve dvojdimenzionálny CA. Existujú dva známe modely rozlišujúce sa v počte buniek tvoriacich okolie bunky. Prvým je *von Neumannovo* okolie, v ktorom každá bunka interaguje so štyrmi susednými bunkami umiestnenými v horizontálnom (ľavá a pravá bunka) a vertikálnom (vrchná a spodná bunka) smere. Druhým je *Moorovo* okolie, v ktorom každá bunka interaguje s ôsmimi susednými bunkami. Moorovo okolie rozširuje *von Neumannovo* okolie o bunky umiestnené na diagonálach. Formálne definované:



Obrázok 2.1: Dvojdimenzionálne pole s a) von Neumannovým okolím a b) Moorovým okolím.[]

Definícia 2.2: Nech $A = (Z^2, N, Q, f, q_0)$ *je dvojdimenzionálny CA. Vravýme, že A má von Neumannove oklie (Obrázok 2.1a), ak:*

$$N = ((0,0), (0,-1), (1,0), (0,1), (-1,0)).$$

A má Moorove okolie, (Obrázok 2.1b) ak:

$$N = ((0,0), (0,-1), (1,-1), (1,0), (1,1), (0,1), (-1,1), (-1,0), (-1,-1)).$$

2.2.2 Počiatočný stav

Pre korektnú funkčnosť celulárneho systému v súlade s prechodovými funkciami je nutné definovať počiatočný stav všetkých buniek systému. Toto nastavenie môže byť označené ako *počiatočná konfigurácia*.

2.2.3 Prechodová funkcia

Prechodová funkcia f určuje nový stav bunky na základe jej stavu a stavu okolitých buniek. Realizácia prechodovej funkcie nie je presne daná a existujú viaceré variácie. Zvyčajne je prechodová funkcia definovaná zoznamom lokálnych prechodových pravidiel, kde ľavá strana pravidla (Left-Hand-Side LHS) je tvorená stavom bunky a stavmi jej susedov a pravá strana pravidla (Right-Hand-Side RHS) určuje nový stav bunky. Formálne zapísané $f: Q^n \rightarrow Q$, kde n je počet susedov vrátane bunky samej. Podľa charakteru množiny je celulárne automaty možné klasifikovať na *neuniformné*, kde rôzne bunky môžu mať rôzne pravidlá, a *uniformné,* kde prechodové pravidlá všetkých buniek sú totožné. Uniformnosť prechodových pravidiel je výhodná vlastnosť pri realizácii celulárneho automatu s veľkým množstvom buniek za pomerne nižšiu cenu ako pri neuniformných.

Aj keď lokálne prechodové pravidlá môžu byť samé o sebe veľmi jednoduché, celulárny automat môže vykazovať veľmi zložité globálne chovanie, ktoré je možné považovať za *emergentnú vlastnosť výpočtového systému.*

2.3 Asynchrónny celulárny automat

Synchronizácia celulárneho automatu úzko súvisí so spôsobom aplikácie prechodových pravidiel. Viaceré modely a aplikácie, ktoré boli dodnes predstavené spoliehali na *synchrónny režim*. To znamená, že CA sa nachádza v konfigurácii i a do konfigurácii $i+1$ prejde aplikovaním prechodových pravidiel na všetky bunky. Okrem toho synchrónny mód časovania vyžaduje dočasné uloženie stavu buniek, aby bolo možné pristupovať k starým stavom buniek, zatiaľ čo sú bunky aktualizované. Pretože všetky bunky synchrónneho celulárneho automatu sú aktualizované v každom časovom kroku, prechodové pravidlá by mali byť definované pre všetky možné kombinácie stavov, ktoré môžu v okolí nastať.

Opakom synchrónnej aktualizácie je asynchrónna aktualizácia. V *asynchrónnom režime* je náhodne vybraná jedna bunka v každom časovom okamihu ako kandidát na aktualizáciu. V prípade, že existuje prechodové pravidlo, ktorého ľavá strana odpovedá stavu vybranej bunky a okoliu, bude stav bunky prepísaný podľa pravej strany prechodového pravidla. Ak stav bunky nevyhovuje žiadnemu zo sady pravidiel, bunka nebude aktualizovaná. To prináša veľké zlepšenie v redukcii počtu pravidiel, pretože je nutné definovať len pravidlá kedy má nastať zmena. Aj keď sada pravidiel je redukovaná, celulárny automat je stále deterministický. Výhodou je, že asynchrónny mód nevyžaduje dočasné uloženie stavu buniek, keďže je aktualizovaná práve jedna bunka. Inak povedané, každá bunka je aktualizovaná okamžite na základe stavu okolia. Model využívajúci asynchrónny mód časovania je nazývaný *asynchrónny celulárny automat* (ACA).

Definícia 2.3: *Nech A = (Z^2, N, Q, f, q0) je dvojdimenzionálny CA. A je asynchrónne aktualizovaný, ak každý časový krok maximálne jedna náhodne vybraná bunka zo Z^2 podstúpi zmene stavu, pričom každá bunka má pravdepodonosť od 0 do 1 byť vybraná [5].*

Asynchrónne časovanie môže byť implementované niekoľkými spôsobmi. Jednou z možností je náhodný výber buniek, avšak tým môže vzniknúť vyhladovanie určitých buniek, keďže spôsob výberu je pravdepodobnostný. Ďalšou variantou je simulovaný náhodný výber, kde by sme zabezpečili, aby všetky bunky podstúpili zmene stavu.

V praxi je ale možné do procesu aktualizácie zaviesť určité heuristiky a vytvoriť kombináciu synchrónneho a asynchrónneho časovania. Tým je myslené, že aj keď budeme používať ACA aktualizácia bude riadená. Predstavme si stav, kde by boli vybrané všetky bunky z celulárneho priestoru. Ak bunka a jej okolie neboli zmenené, potom je zrejmé, že ak by bola vybraná procesom náhodného výberu k žiadnej zmene by nemohlo dôjsť. Takže je zbytočné aby bola vybraná. Riešením je náhodne vyberať bunky len z podmnožiny buniek tvorenou bunkami, kde buď bunka alebo nejaká bunka z jej okolia boli zmenené.

3 Celulárne automaty a nanopočítače

V súvislosti s platnosťou Moorovho zákona sa často hovorí o nanotechnológiách, teda o možnosti vyrábania čipov, ktoré budú zapracovávať a uchovávať informácie reprezentované objektami, ako sú jednotlivé elektróny, atómy alebo molekuly. Nanotechnológie a nanovedy pracujú s objektami o veľkosti do 100 nm (do 1 nm sa vojdú zhruba 3-4 atómy) [4].

V tejto kapitole bude čitateľ uvedený do problematiky nanotechnológií. Bude porovnaná metóda návrhu „zhora dole“ s metódou „zdola hore“, ktorá je kľúčovou v tejto oblasti. Ďalej sa zameriame na odvetvie nanotechnológií, konkrétne na nanopočítače. Uvedieme si stručnú históriu a pokusy o vytvorenie nanopočítačov. V súvislosti s nanopočítačmi sa často hovorí o asynchrónnych obvodoch a obvodoch odolných voči oneskoreniu, a preto sa tejto problematike budeme hlbšie venovať. Na konci kapitoly budú zhrnuté problémy spojené s nanotechnológiami, ktoré sú momentálne kritické z hľadiska fyzickej implementácie a spoľahlivosti nanopočítačov.

3.1 Nanotechnológie

Nanotechnológia je všeobecné označenie (častí) vedných odborov, ktoré sa zaoberajú tvorbou a využívaním technológií v meradle rádovo nanometrov. Hlavným cieľom nanotechnológie je schopnosť manipulovať s materiálom na atomárnej úrovni. Vo svete elektroniky a informatiky je to odvetvie, ktoré sa zaoberá návrhom a výrobou extrémne malých elektronických obvodov a vstavaných systémov na molekulárnej úrovni.

V oblasti počítačov by nanotechnológie mali umožniť, aby boli prekonané limity konvenčnej technológie CMOS a umožniť ďalšiu miniaturizáciu. V súčasnosti je najdôležitejšou úlohou zaistiť spoľahlivosť budúcich nanopočítačov. Ak by spoľahlivosť elementárnych komponent nanopočítačov bola rovnaká ako je spoľahlivosť dnešných logických členov, nebudú nanohradlá takmer vôbec použiteľné, pretože vzhľadom k ich výrazne väčšiemu počtu v systéme bude pravdepodobnosť poruchy celého systému neprijateľne vysoká.

V oblasti molekulárnej elektroniky existujú dva konštrukčné prístupy. Metóda „zhora dolu“ sa snaží presne vyrobiť a umiestniť komponenty v rámci obvodu. Príkladom môže byť dvojrozmerné pole molekulárnych vodičov, kde sú v priesečníkoch namontované molekulárne prepínače. Ich stav určuje logickú nulu alebo jednotku. Je tak možné vytvoriť logickú sieť alebo pamäť s vysokou hustotou integrácie. To však vyžaduje veľmi presné umiestnenie vodičov.

Druhou možnosťou je prístup „zdola nahor“, kedy by sa molekuly mohli vlastnou schopnosťou spontánne samoorganizovať do cieľových stabilných štruktúr. Základom tohto prístupu je vhodná chemická syntéza, ktorá dnes umožňuje pripraviť takmer ľubovoľné molekulárne štruktúry. Tento prístup by mohol byť výrazne lacnejší ako postup zhora dolu. Viac informácii v [4].

3.2 História nanopočítačov

Nasledovný prehľad výskumu a experimentov v oblasti nanopočítačov bol prevažne prevzatý z článku [1]. Uvedieme si najvýznamnejšie projekty za posledné desaťročia či aj realizované, alebo len teoreticky navrhnuté.

V 90tých rokoch Carter pracoval na dizajne *chemického nanopočítača* založeného na molekulárnych poliach. Carter bol chemik a zameriaval sa na niektoré operácie buniek, ktoré mohli byť realizované postupnou výmenou medzi jednoduchou a dvojitou väzbou v molekulách. Táto myšlienka nakoniec nebola v oblasti počítačov implementovaná, ale išlo o prvý systematický návrh nanopočítačov.

S CA čiastočne súvisí problematika *kvantových celulárnych automatov* (angl. Quantum Cellular Automaton QCA), ktoré boli navrhnuté za účelom realizácie nanopočítača. Každá bunka pozostávala zo 4 kvantových bodiek, z ktorých 2 obsahovali elektrón. Body sú umiestnené do rohov bunky. Podľa Columbovho zákona vo vnútri bunky sa 2 elektróny snažia udržať maximálnu vzdialenosť medzi sebou, resp. sa medzi sebou vzájomne odpudzujú, čo má za následok, že elektróny zaujmú pozíciu v protiľahlých rohoch. Tým vznikajú 2 možné stavy umiestnenia elektrónov, kde jeden stav je možné interpretovať ako logickú 1 a opačný ako logickú 0. V určitej mierke je Columbova interakcia aj medzi susednými bunkami práve kvôli susediacim elektrónom, čo má za následok, že pri preklopení stavu bunky je ovplyvnený stav susedných buniek. Táto interakcia môže byť použitá na vedenie signálu v danom modeli a tiež možné použiť na implementáciu AND a NOT hradiel. Tento model sľubuje nízku spotrebu energie, kde nie je používaný tok elektrónov, ale tunelovanie elektrónov medzi kvantovými bodkami vo vnútri bunky. Tento model funguje len teoreticky, pretože požaduje veľmi nízku operačnú teplotu a presné umiestnenie kvantových bodiek.

Dvoj dimenzionálny celulárny *automat využívajúci optické signály* k spusteniu prechodu stavu buniek bol predstavený v roku 1994 pánom Biafore. Tento model mal veľmi jednoduché bunky, pretože na realizáciu prechodovej funkcie uprednostňoval fyzické interakcie pred definovanou sadou prechodových pravidiel. Používaním optického hodinového signálu o rôznej vlnovej dĺžke dodávaný v špecifickom poradí, model dokázal kontrolovať (aspoň teoreticky), ktoré bunky budú interagovať s okolím a kedy táto interakcia nastane.

Heinrich, Lutz, Gupta, a Eigler (2002) predstavili celulárny automat, ktorý bol fyzicky realizovaný v nano rozmeroch. Je založený na fyzickej interakcii medzi CO molekulami, ktoré sú usporiadané na povrchu medi, tak aby tvorili *molekulárne kaskády*. Niektoré z konfigurácii CO molekúl sú nestabilné, ale je s veľkou pravdepodobnosťou možné predpovedanie do akej konfigurácie bude zmenená. Vhodným usporiadaním CO molekúl je možné realizovať veľké množstvo funkcií, ako kríženie signálov, AND hradlá a iné. Tieto typy systémov boli experimentálne overené, že sú schopné jednoduchého výpočtu, ale boli príliš pomalé.

Dizajny uvedených celulárnych automatov využívali fyzické interakcie na vykonávanie prechodov. To viedlo k extrémne jednoduchým bunkám, kde ich funkcionalita bola limitovaná množinou povolených prechodov. Prídavná funkcionalita, ako oprava chýb alebo konfigurácia buniek na určitý stav je ťažko implementovateľná. Z toho dôvodu boli vymyslené modely založené na komplexných bunkách, ktoré by boli vyrobiteľné technikou "zdola nahor" a organizované v poliach. Príkladom tohto prístupu je *Cell Matrix* (Durbeck & Macias, 2001). Každá bunka pozostáva z približne 100 bytov pamäte a desiatok hradiel. Bunky je možné nakonfigurovať tak, aby vykonávali jednoduché operácie, ako logické hradlá, binárne sčítačky a iné.

3.3 Celulárny Automat s vlastným časovaním

Asynchrónny celulárny automat, v ktorom prechod bunky do nového stavu je spustený prechodom v nejakej susednej bunke sa nazýva c*elulárny automat s vlastným časovaním*, v literatúre označovaný STCA (angl. Self-Timed Cellular Automaton). STCA je modifikáciou asynchrónnych CA zameraný na lokálnu interakciu buniek. Tento automat bol predstavený v článku [5] a potom

ďalšie pokusy o návrh a realizáciu obvodov boli v [3]. Model vykonáva aktualizácie buniek na rovnakom princípe ako je tomu u synchrónnych automatov. V ten samý čas ponúka flexibilitu asynchrónnych celulárnych automatov, ktorou je myslené aktualizovanie maximálne jednej bunky v čase, a tým odpadajú nevýhody synchrónnych obvodov spomenuté v kapitole 2.3. STCA model sľubuje vhodné vlastnosti pri realizácii výpočtu založeného na celulárnych automatoch v molekulárnej mierke. V tomto zmysle je vlastné časovanie zvlášť užitočné kvôli jeho kompatibilite s chovaním molekúl a vyhnutiu sa problémom spojenými so šírením synchronizačného signálu.

3.4 Asynchrónne obvody

Synchrónny mód časovania v porovnaní s asynchrónnym prináša radu problémov, medzi ktorými sú hlavne spotreba energie a odvádzanie tepla. Tieto problémy majú tendenciu sa zhoršovať pri vyššej úrovni integrácie, a preto boli motivujúcim faktorom pre výskum asynchrónneho režimu, kde časovanie funguje na lokálnej úrovni výmenou signálom, napríklad pomocou techniky *handshaking* (zaslanie požiadavky a čakanie na odpoveď). Absencia globálneho synchronizačného signálu má veľké výhody pri fyzickej implementácii, čo je jeden z hlavných dôvodov, prečo sú dnes študované asynchrónne obvody vo vzťahu s nanopočítačmi. Okrem toho asynchrónne obvody disponujú viacerými vlastnosťami vhodnými pre implementáciu nanopočítačov. O jednotlivých výhodách si povieme podrobnejšie v nasledovných podkapitolách.

Pojem asynchrónne obvody reprezentuje viaceré typy obvodov, ktoré je možné rozdeliť do piatich tried [3]. Najvšeobecnejšia trieda je tvorená asynchrónnymi obvodmi, ktoré využívajú časových predpokladov v rámci obvodu a v interakcii medzi obvodom a prostredím, napríklad obmedzenosť veľkosti oneskorenia. Táto trieda je najmenej odolná voči neočakávanému chovaniu.

Druhú triedu predstavujú *obvody s vlastným časovaním* (angl. self-timed circuits). Elementy obvodov s vlastným časovaním nevykonávajú žiadne odhady pri komunikácii medzi nimi. Vlastnosť odolnosti voči oneskoreniu vzniká kvôli spôsobu komunikácie vo vnútri elementov, ktorá je prísne kontrolovaná. Napríklad vodiče vo vnútri prvku môžu požadovať obmedzené alebo zanedbateľné oneskorenie.

Treťou triedou sú *obvody nezávislé na rýchlosti* (angl. speed-independent circuits), kde operácie prvkov obvodu nekladú dôraz na obmedzenosť meškania, ale vodiče teda spojenie medzi prvkami vyžaduje nulové alebo zanedbateľné oneskorenie.

Tieto obvody sú v skutočnosti veľmi podobné obvodom v štvrtej skupine, *obvody kvázi odolné voči oneskoreniu* (angl. quasi-delay-insensitive circuits), ktoré sú odolné voči oneskoreniu, až nato, že požadujú takzvaný *isochronic fork*, čo je prvok, ktorého všetky výstupné vetvy majú rovnaké oneskorenie.

Piata skupina predstavuje *obvody odolné voči oneskoreniu* (angl. delay-insensitive circuits), ktoré budú podrobnejšie vysvetlené v nasledovnej podkapitole. Tieto obvody sú najrobustnejšie zo všetkých asynchrónnych obvodov kvôli ich tolerancii k oneskoreniu vo vnútri prvku, a tiež pri komunikácii medzi nimi.

3.4.1 Obvody odolné voči oneskoreniu

Obvody odolné voči oneskoreniu sú obvody, kde správne fungovanie výpočtu nie je ovplyvnené oneskorením v elementoch obvodu a prepojovacích signáloch. Operácie v obvodoch odolných voči oneskoreniu sú riadené signálmi. Element obvodu je prevažne neaktívny, dokým neprijme určitú sadu vstupných signálov, na ktoré je schopný reagovať. Potom vykoná výpočet (môže sa jednať aj

o primitívnu činnosť, akou je preposlanie signálu), pošle na výstup signály a opäť sa stáva neaktívnym. Z princípu nie je nutné časovanie. Obvody odolné voči oneskoreniu patria medzi *asynchrónne obvody*, ktoré majú oproti *synchrónnym obvodom* niekoľko výhod. V rámci implementácie nanopočítačov a spojitosti s nanotechnológiami sú najvýraznejšie tieto [3]:

- **Nie je potrebné zavedenie synchronizačného (clock) signálu**, čo má za výhodu ušetrenie miesta v obvodoch ale aj prispieva k homogenite (dôležitá vlastnosť pri výrobe založenej na molekulárnej syntéze alebo inej výrobe v nano mierke) systému.

- **Zníženie spotreby energie a úniku tepla**, čo je spôsobené tým, že nie je nutné, aby bol aktívny celý obvod naraz, ale len časti, kde sa menia signály. Zatiaľ čo u asynchrónnych obvodoch je táto vlastnosť automatická, u synchrónnych obvodov všetky prvky musia prejsť časovaním, alebo je nutné implementovať ďalšiu logiku, ktorá to zabezpečí.

- **Absencia problémov s časovaním signálov**. Viacmenej sú odstránené všetky problémy spojené s časovaním, ako je napríklad nutnosť signál dostať do určitého miesta v rámci jedného hodinového cyklu (sú techniky, ktoré tento problém riešia, ale vyžadujú navyše logiku).

- **Nezávislosť na fyzickej implementácii a podmienkach**. Zmeny v časovaní signálov dané fyzikálnymi vlastnosťami alebo implementáciou neovplyvňujú správnosť operácií asynchrónnych obvodov**,** hlavne ak sú to obvody odolné voči oneskoreniu (variant asynchrónnych obvodov).

- **Lepší priemerný než najhorší výkon**. Lepšie je, keď asynchrónny obvod operuje tak rýchlo, ako to obvod dovoľuje, než aby bol limitovaný najpomalejšími časťami obvodu, ktoré v synchrónnych systémoch limituje frekvencia globálneho hodinového signálu.

- **Modularita.** Asynchrónne obvody môžu byť rozdelené do modulov, ktoré môžu byť navrhnuté bez závislosti na iných moduloch**.** Modulárnosť zjednodušuje konfigurovateľnosť obvodu a je možné uvažovať aj o čiastočnej rekonfigurácii. Modul môže byť zložený z jednoduchších modulov a spojením všetkých potrebných modulov by bolo možné zostrojiť nanopočítač odolný voči oneskoreniu (ak je modul odolný voči oneskoreniu, teda aj výsledok zložený z modulov bude tiež odolný voči oneskoreniu).

Kým v synchrónnych obvodoch je binárna informácia prenášaná jedným signálom (signálom sa rozumie stav, ktorý je odlišný od kľudového stavu, napríklad signál môže byt interpretovaný ako vysoká impedancia, zatiaľ čo nízka impedancia predstavuje absenciu signálu), informácia v obvodoch odolných voči oneskoreniu (DI obvody) býva reprezentovaná výskytom signálu v určitej časti z vodičov. Na preloženie do binárnej hodnoty je možné použiť dva signály. Jeden signál pre log.0 a druhý pre log.1. Tento spôsob interpretácie logických hodnôt sa volá dvojdrôtové kódovanie (angl. dual-rail encoding). Absencia signálu v oboch vodičoch indikuje, že nie je prenášaná žiadna informácia.

DI obvody bývajú zostavené z modulov. Modul je prvok s konečným počtom vstupných a výstupných vodičov a konečným počtom stavov. Keď príjme určité signály na vstupných vodičoch, modul vykoná operáciu a jej výsledok môže zmeniť (nastaviť) signál na nejakom

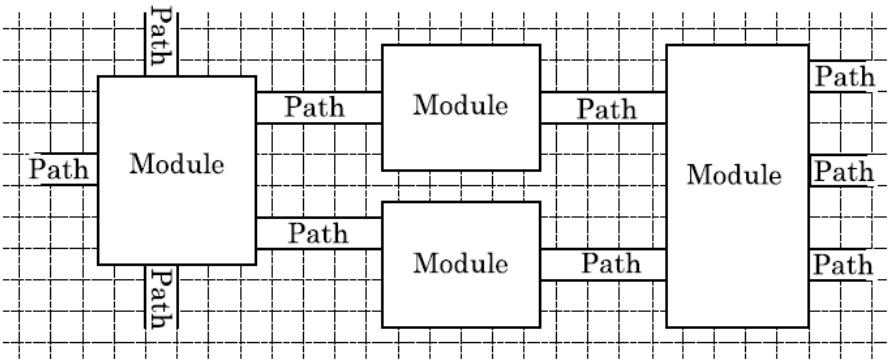
výstupnom vodiči. Moduly môžu byť organizované v hierarchických štruktúrach a vytvorené z modulárnych sietí a v celku môže tvoriť počítač odolný voči oneskoreniu.

Obvody odolné voči oneskoreniu majú vhodné vlastnosti k implementácii založenej na asynchrónnych celulárnych automatoch s vlastným časovaním (STCA). Príklad takejto implementácie sú *asynchrónne celulárne polia* podrobnejšie popísané v kapitole 4.

Aj keď majú asynchrónne obvody radu nevýhod, je o ne stále rastúci záujem, pretože vyššie uvedené výhody sa ešte viac prejavujú pri vyšších hustotách integrácie. Hlavnými nevýhodami sú menej dostupné návrhy, testovania, a výrobné infraštruktúry a tiež odborné znalosti v porovnaní so synchrónnymi systémami (viac je možne sa dočítať v [3]). Ďalším problémom je existencia porúch a fakt, že ich tolerancia (nemali by ovplyvňovať výpočet) vyžaduje určitú réžiu pri fyzickej implementácii.

3.4.2 Celulárne polia

Obvody odolné voči oneskoreniu môžu byť realizované v *celulárnych poliach* konfiguráciou buniek tvoriace *moduly* a prepojené medzi sebou pomocou *ciest* (obrázok 3.1).



Obrázok 3.1: Moduly prepojené cestami v celulárnom poli [3].

Cestou je myslená neprerušená postupnosť buniek, po ktorých je signál prenášaný zo zdrojového do cieľového modulu. Cesty v celulárnych poliach majú rovnakú úlohu, ako vodiče v elektronických obvodoch. V rámci celulárnych polí je *signál* definovaný ako zmena stavu bunky na ceste v smere od jedného konca (*zdroj*) k druhému koncu (*cieľ*). Na jednej ceste môže byť viacero rôznych signálov, ale tieto signály sa nemôžu vzájomne ovplyvňovať, to znamená, že rôzne signály nemôžu byť zlúčené do jedného, a tiež jeden signál nemôže sa samovoľne rozdeliť na viacero signálov. Ak bol signál vyslaný zo zdroja, nie je žiadna možnosť jeho zrušenia, alebo vrátenia. Signál bude putovať smerom k cieľu a doba, za ktorú sa tam dostane, je daná dĺžkou cesty. Oneskorenie signálu je určené konečným počtom krokov (prechodov).

Ak signál dosiahne cieľu, modul vykoná operáciu, čo zvyčajne spôsobí generovanie signálu na jednej alebo viacerých výstupných cestách. Niekedy aby modul mohol vykonať operáciu, musí čakať na príchod signálu z inej cesty. V tomto prípade je vstupný signál nazvaný *čakajúci* (angl. pending). Prenos signálov po cestách a ich spracovanie modulmi môže nadobúdať ľubovoľné konečné oneskorenie bez toho, aby to malo nejaké dôsledky na správnu funkciu obvodu. Inými slovami, nie sú žiadne časové obmedzenia na signály. Viaceré nasledujúce vstupné signály na ceste do modulu sú vždy striedané v čase s výstupným signálom produkovaný modulom ako odpoveď na vstupný signál. Tento spôsob výpočtu efektívne bráni modulu používať viaceré nasledujúce vstupné signály z jednej cesty v rámci vykonania operácie. Odlišná situácia nastáva, keď modul má dva

vstupné signály z rozdielnych ciest, a môže spracovať v čase maximálne jednu. V tomto prípade môže modul náhodne zvoliť, ktorý zo vstupných signálov bude spracovaný ako prvý.

Tak ako ľubovoľný logický obvod v synchrónnych systémoch môže byť zostrojený z určitej sady primitívnych prvkov, napríklad AND hradlá a NOT hradlá, podobne aj DI odvod môže byť realizovaný pomocou sady primitív, nazývanej *univerzum*. Pre praktické účely univerzálnosťou môže byť považovaná schopnosť, na základe sady primitív skonštruovať obvody, ktoré dovoľujú vykonávať ten istý typ výpočtu, ako konvenčné počítače [13]. Logické hradlá nie sú primitívnou sadou pre obvody odolné voči oneskoreniu, pretože im chýba funkcionalita na vysporiadanie sa s každým možným usporiadaním udalostí na vstupných vodičoch. Pre obvody odolné voči oneskoreniu existuje iná sada primitív, pomocou ktorej je možná realizácia základných logických obvodov s tým rozdielom, že takto vytvorené logické hradlá budú rovnako ako ich primitíva odolné voči oneskoreniu. Pre celulárne polia boli vymyslené rôzne sady primitív, dokonca niektorých univerzálnosť bola aj formálne overená.

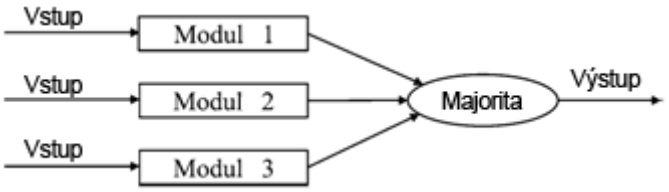
3.5 Tolerancia porúch

Tolerancia porúch bude dôležitá u nanopočítačov, pretože spoľahlivosť ich komponent je limitovaná určitými faktormi. Prvým faktorom je individuálne chovanie elektrických prvkov, častíc, molekúl alebo iného materiálu vhodného pre nanopočítače, pretože toto chovanie neprebieha podľa zákonov klasickej fyziky, ale musí byť popisované podľa kvantovej teórie. Druhým faktorom sú chyby spôsobené pri výrobe, ktorých pravdepodobnosť sa zvyšuje vzhľadom k narastajúcej hustote integrácie obvodov.

3.5.1 Tolerancia porúch pri výpočte

Pri výpočte môžu nastať poruchy, s ktorými by sa systém mal vedieť vysporiadať. Takáto porucha môže byť spôsobená napríklad zmenou prostredia vplývajúceho na systém. Inak povedané, cieľom *výpočtu s toleranciou porúch* (fault tolerance computing) je zvýšenie *spoľahlivosti* systému, kde spoľahlivosť môže byť definovaná ako schopnosť systému splniť určité požiadavky alebo úlohy na akceptovateľnej úrovni dôveryhodnosti, a to buď v prítomnosti alebo neprítomnosti porúch. Spoľahlivosť (dependability) systému je posudzovaná viacerými ukazovateľmi, a to bezporuchovosť (reliability), dostupnosť (availability), odolnosť voči chybám (fault coverage) a bezpečnosť (safety). Bezporuchovosť systému je obvykle definovaná ako pravdepodobnosť, že systém bude v čase *t* fungovať bez poruchy. Dostupnosť systému je pravdepodobnosť, že systém je k dispozícii na vykonanie úlohy v čase *t*. Odolnosť voči poruchám je schopnosť systému sa spamätať z poruchy a pokračovať vo výpočte. Bezpečnosť je pravdepodobnosť, že systém bude schopný i napriek poruchám pracovať správne [6].

Najčastejším riešením na zvýšenie *spoľahlivosti systému* býva pridanie určitej redundancie. Na hardwarovej úrovni je možné implementovať TMR (angl. triple modular redundancy, obrázok 3.1), kde výpočet je realizovaný v troch moduloch a potom sa určí výstup podľa majority. Aj keď tento prístup sa dokáže vysporiadať s maximálne jednou chybou, je možné hierarchicky kombinovať TMR a tak dosiahnuť ďalšie zvýšenie spoľahlivosti.



Obrázok 3.1: TMR - Triple modular redundancy [6].

3.5.2 Tolerancia poruchy pri výrobe

Ďalej je často požadovaná schopnosť systému správne fungovať, ak jedna alebo viacej jeho častí sú permanentne poškodené, kedy poškodenie vzniklo pri výrobe. Vysporiadanie sa s týmto druhom porúch bude tiež dôležitou vlastnosťou nanopočítačov, pretože poruchy ľahko vznikajú hlavne pri výrobe „zdola nahor“. Chyby sú permanentné, a preto je nutné ich detekovať a izolovať od nepoškodenej časti, aby nebola ovplyvnená správna funkčnosť systému. Jednou z možností je rekonfigurácia alebo čiastočná rekonfigurácia celulárneho priestoru, ale fyzická implementácia nie je jednoduchá, ak je vôbec efektívne realizovateľná [1].

3.5.3 Tolerancia porúch u nanopočítačov

Cieľom výskumu v oblasti odolnosti voči poruchám u nanopočítačov je viac uprednostňovať zlepšenie spoľahlivosti systému zlepšením kvality výrobnej fázy pred zlepšením mechanizmov odolnosti proti poruchám, ktoré sa uplatnia pri výpočte. Každá technika vyžaduje určitú redundanciu, čo zvyšuje ako cenu, tak aj čas vývoja. Taktiež redundancia má vplyv na výkonnosť, únik tepla, váhu a veľkosť systému. Dobre navrhnutý mechanizmus odolnosti voči poruchám volí rozumný kompromis medzi mierou spoľahlivosti a množstvom použitej redundancie [6].

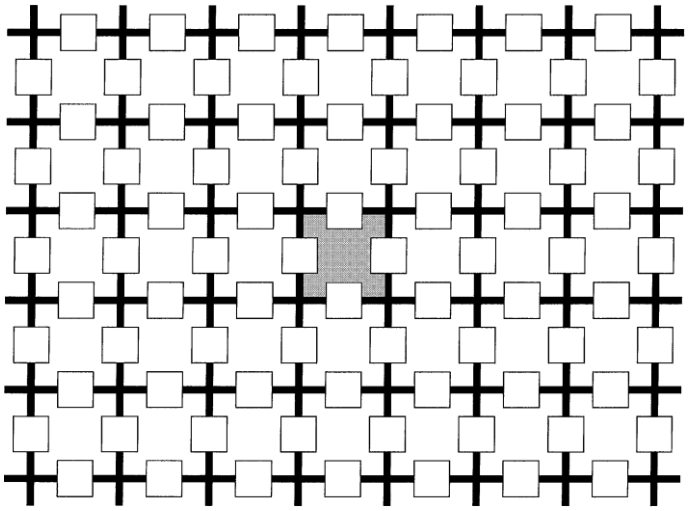
4 Asynchrónne celulárne polia

V tejto kapitole sa budeme podrobne venovať celulárnym poliam, ktoré sa ukazujú byť vhodnou technológiou na realizovanie nanopočítačov. Budeme rozoberať 2-dimenzionálne polia. Na základe tejto techniky by bolo možné skonštruovať aj 3-dimenzionálne štruktúry, avšak tie nebudú predmetom tejto práce a preto sa im nebudeme ani bližšie venovať. Po definovaní základných princípov bude na základe nich ukázaná možnosť implementácie asynchrónnych obvodov.

Asynchrónne Celulárne polia boli zavedené v [3] a ďalej skúmané v [12] a [8]. Z publikácie [8] boli prebrané takmer všetky obrázky.

4.1 Princíp celulárnych polí

Celulárne pole je 2D štruktúra identických buniek, kde každá má štyroch susedov (Obr. 4.1). Bunka má štyri pamäte o veľkosti dvoch bitov, pričom jedna pamäť je zdieľaná medzi susednými bunkami. Zdieľanie je možné z toho dôvodu, že nikdy nebudú naraz aktualizované dve susedné bunky. Hodnoty uložené v pamätiach určujú *stav bunky*. Stav všetkých buniek daný stavom ich pamätí sa v celulárnych automatoch volá *konfigurácia*. Na obrázku 4.3b je možné vidieť tri rôzne konfigurácie po aplikovaní prechodových pravidiel niekoľkokrát po sebe.

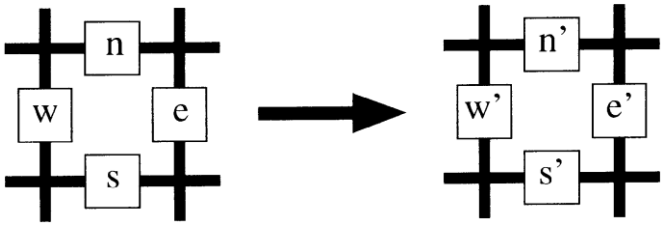


Obr. 4.1: Celulárne pole buniek (šedá plocha označuje jednu bunku), kde každá má prístup do štyroch pamätí (malý biely štvorček). Pamäť je zdieľaná dvoma susednými bunkami, a stav pamäte je menený aplikovaním prechodových pravidiel [8].

4.1.1 Aktualizácia stavu bunky

Bunka môže zmeniť stav štyroch pamätí, s ktorými je prepojená, operáciou nazývanou *prechod*, a tým prejsť do nového stavu. Prechod je popísaný *prechodovým pravidlom*, ktoré musí mať špecifikovaný aktuálny a nový stav pamätí bunky. Prechodové pravidlo sa aplikuje v prípade, ak ľavá strana pravidla sa zhoduje s kombináciou stavov pamätí bunky. Pravá strana pravidla popisuje stav pamätí po aplikovaní pravidla (Obr. 4.2). Na popísanie funkčnosti celulárneho automatu

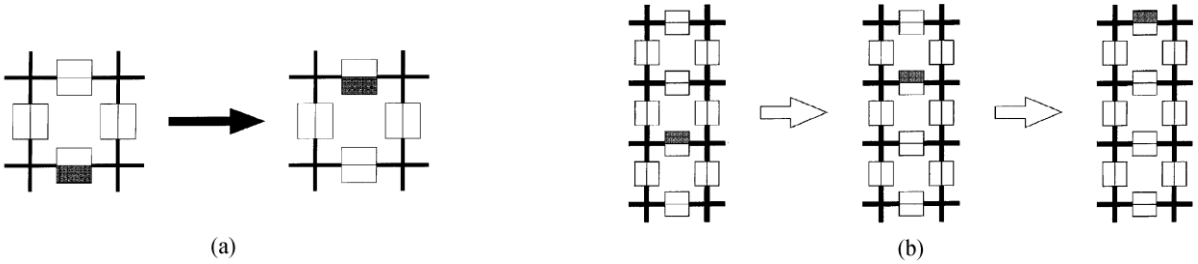
spravidla nestačí jedno pravidlo, ale podľa náročnosti ich treba viac. Takáto skupina pravidiel sa nazýva *tabuľka prechodových pravidiel*.



Obr. 4.2: Prechodové pravidlo popisujúce zmenu stavu pamätí spojených s bunkou. Ak stavy pamätí w, n, e a s bunky sa zhodujú s ľavou stranou pravidla, pravidlo môže byť aplikované na bunku, a pamäte bunky nadobudnú stavy w´, n´, e´ a s´ dané pravou stranou pravidla [8].

4.1.2 Aplikácia prechodového pravidla

Na obrázku 4.3a je ukázaný príklad prechodového pravidla, ktoré posunie informáciu o jeden bit nahor. Po aplikovaní tohto pravidla dvakrát je na obrázku 4.3b ukázaná postupnosť konfigurácii, kde je hodnota propagovaná *cestou buniek* zdola nahor. Po každej aplikácii sa informácia presunula o jednu pozíciu vyššie.



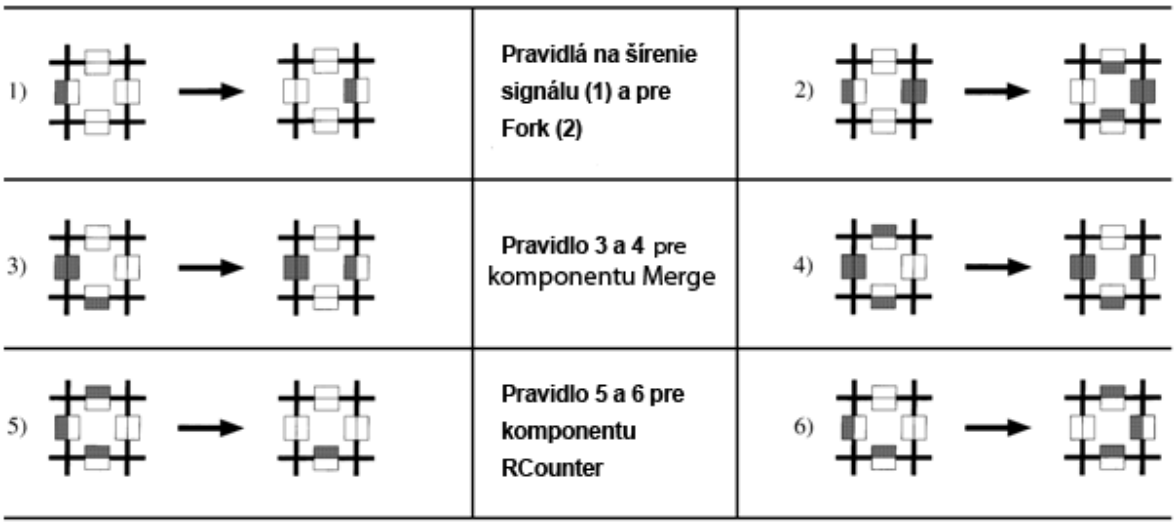
Obr. 4.3: (a) Prechodové pravidlo ukazujúce propagáciu signálu a (b) aplikovanie tohto pravidla dvakrát po sebe. Pamäť obsahuje dva bity, pruhovaný blok označuje hodnotu 1 a biely hodnotu 0 (nie logickú hodnotu, ale existenciu signálu) [8].

4.1.3 Šírenie signálov v celulárnych poliach

Kým v konvenčných obvodoch je šírenie signálu medzi logickými hradlami možné zabezpečiť prepojením elektrickými vodičmi, celulárne automaty sú tvorené homogénnou štruktúrou buniek, ktoré môžu komunikovať iba s okolitými bunkami. Nepripadá v úvahu žiadna možnosť, kde by sa komponenty obvodov mohli prepájať vodičmi, čo by tiež porušovalo podmienku *lokality*. Preto je nutné na vedenie signálov použiť bunky celulárneho automatu a prechodové pravidlá musia byť navrhnuté tak, aby bolo myslené aj na túto požiadavku. V kapitole 4.4 uvidíme, že nám bude stačiť jedno, maximálne dve pravidlá a sme schopný šíriť signál kdekoľvek v celulárnom poli.

4.2 Implementácia asynchrónnych obvodov na celulárnych poliach

Aby bolo možné zostrojiť zložitejšie obvody, musíme si najskôr definovať základné primitíva. Chovanie týchto primitív je závislé na tabuľke prechodových pravidiel. Pre naše účely by mali stačiť pravidlá na obrázku 4.4 aby sme s nimi definovali chovanie našich primitív, ktoré budú podrobne popísane v kapitole 4.5. Návrh primitív a pravidiel s nimi spojených výrazne ovplyvňuje zložitosť buniek a tiež proces výroby nanopočítačov. Musíme myslieť na to, že nanopočítače založené na modele celulárnych automatov kladú dôraz na lokálnosť, a preto prechodové pravidlá musia byť implementované pre každú bunku osobitne.



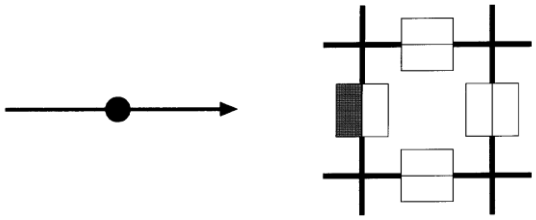
Obrázok 4.4: Prechodové pravidlá nutné pre simuláciu obvodov odolných voči oneskoreniu na celulárnych automatoch [8].

4.3 Základné primitíva

Základné primitíva majú úlohu viac-menej zabezpečiť šírenie signálu v celulárnom poli. V porovnaní s logickými prvkami ako je AND, OR alebo NOT v synchrónnych obvodoch sú na primitívnejšej úrovni, avšak ich prepojením je možné vytvoriť napríklad ľubovoľné logické hradlá a obvody.

4.3.1 Signál

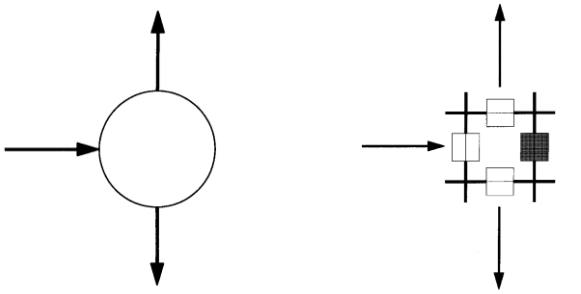
Signál slúži na propagáciu signálu v jednom smere buniek, teda buď vertikálne alebo horizontálne. V prípade viacerých signálov na jednej ceste, budú ostatné signály čakať (tento režim sa nazýva „*pending*“), dokým sa nespracujú signály pred ním. Signál sa nikdy nemôže prepísať iným signálom.



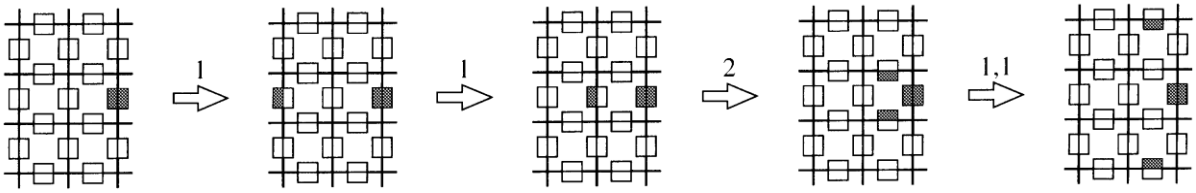
Obrázok 4.5 Signál – schematická značka, konfigurácia [8].

4.3.2 Fork

Prvok *Fork* je jediným prvkom schopný z duplikácie signálu, teda z jedného signálu vytvoriť dva signály. *Fork* má jeden vstup a dva výstupy. V prípade, že na vstup príde signál, na každom z výstupov sa generuje nový signál. Proces šírenia signálu je znázornený na obrázku 4.7, kde je vidieť, že najskôr sa dvakrát aplikovalo pravidlo 1. Keď sa signál dostal k čiernej bunke bolo aplikované pravidlo 2 a potom aplikovaním pravidla 1 sa signály šíria na oba výstupy.



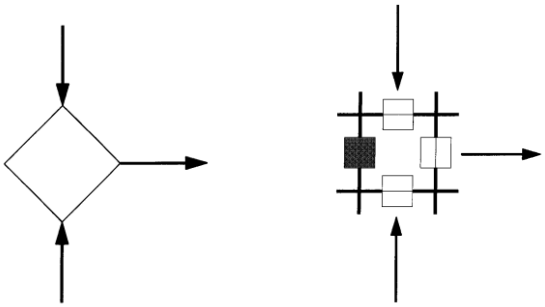
Obrázok 4.6 Fork – schematická značka, konfigurácia [8].



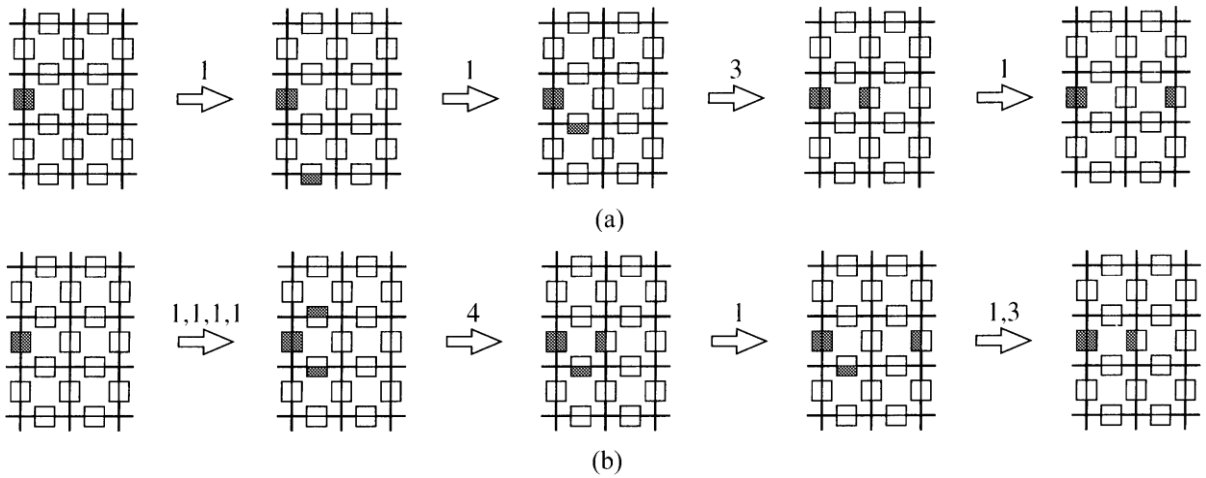
Obrázok 4.7 Fork – priebeh signálov,
číslo nad šípkou značí číslo použitého pravidla [8].

4.3.3 Merge

Merge má opačnú funkciu ako *Fork*, ale nie je schopný spojiť dva signály dokopy. Má dva vstupy a jeden výstup. V prípade, že na jeden vstup príde signál, je poslaný na výstup, čo je možné vidieť na obr. 4.9a. Ak by na vstup prišli naraz dva signály (obr. 4.9b), jeden z nich bude náhodne vybraný a poslaný na výstup, zatiaľ čo druhý bude čakať na vstupe. Po spracovaní prvého bude na výstup poslaný aj druhý signál. Signály sú spracované sekvenčne a aj v prípade príchodu oboch vstupov v rovnaký časový okamih. Pretože *Merge* posiela signál na cestu kolmú na vstupnú cestu, býva často používaný na zmenu smeru toku signálu.



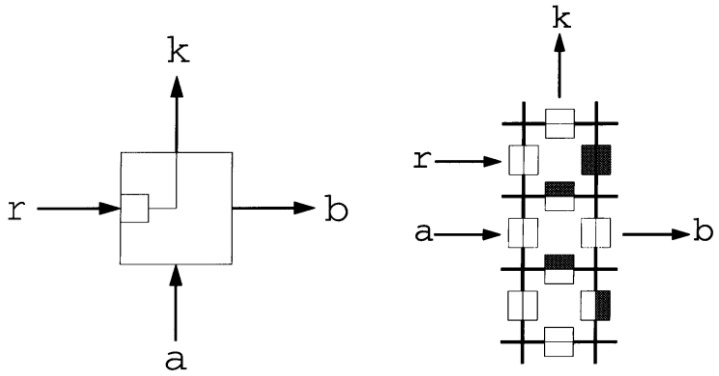
Obrázok 4.8 Merge – schematická značka, konfigurácia [8].



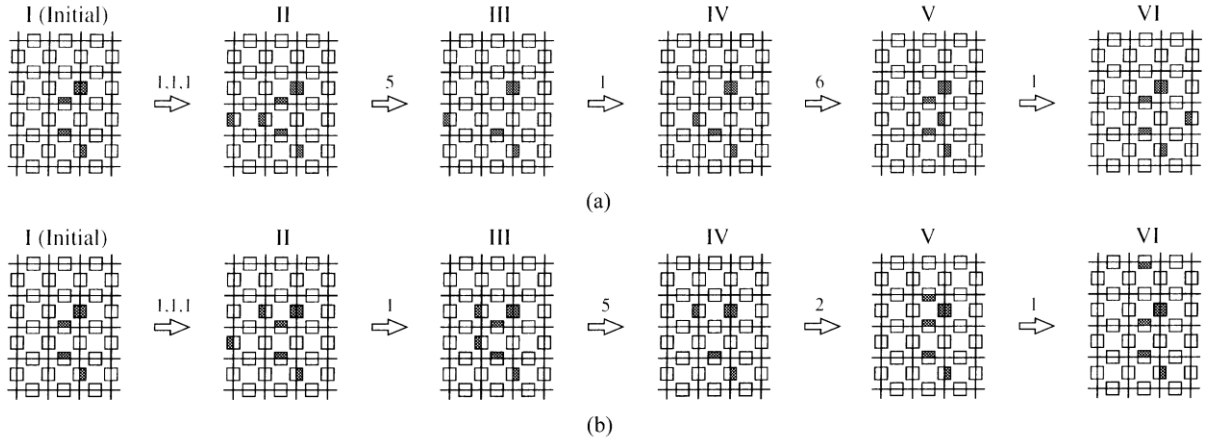
Obrázok 4.9 Merge, a) spracovanie jedného vstupného signálu (pravidlo 1 po prvom obrázku znamená príchod signálu zo spodnej strany), b) spracovanie dvoch vstupných signálov (štyri razy aplikované pravidlo 1 po prvom obrázku znamená prísun signálu z vrchu a zo spodku, dvakrát aplikované zhora, dvakrát zospodu) [8].

4.3.4 R-Counter

R-Counter (*Resettable Modulo 2 Counter*) prijíma signály na vstupe „*a*“ a každý druhý signál posiela na výstup „*b*“. Po odoslaní výstupného signálu je prvok vo východzom stave. Prvok môže byť resetovaný (prejde do východzieho stavu) príchodom signálu na vstup „*r*“, čo produkuje signál na výstup „*k*“. R-Counter je v podstate modifikovaný *Resettable Join modul* predstavený v [8], kde dva vstupy boli zlúčené do jedného vstupu „*a*“. R-Counter je reprezentovaný konfiguráciou troch buniek, z ktorých prvé dve podstupujú prechodom. Tretia (spodná) slúži aby udržiavala jednotnú konfiguráciu. Inak povedané, ak by tretia bunka nebola nakonfigurovaná, na druhú bunku by bolo možné aplikovať pravidlo číslo 1, čo by spôsobilo nechcený tok signálu.



Obrázok 4.10 R-Counter – Schematická značka, konfigurácia [8].



Obrázok 4.11 R-Counter a) spracovanie dvoch signálov na vstupe „a“, b) spracovanie signálu na vstupe „a“ a vstupe „r“ [8].

Ak vstupný signál prijde cestou „a“, aplikovaním pravidla 5 (všetky pravidlá na obr. 4.4) prejde konfigurácia do špeciálneho stavu, ktorý je možné vidieť na tretej konfigurácii v obrázku 4.11a. Ak cestou „a“ prijde ďalší signál (4. konfigurácia v obrázku 4.11a) aplikovaním 6. pravidla bude konfigurácia primitíva v pôvodnom stave a na výstupnej ceste „b“ sa objaví signál.

Ak by ako prvý prišiel signál na cestu „r“, ako je to v obr. 4.11b, ostane sa čakať, dokým na cestu „a“ neprijde nejaký signál. Potom sa aplikuje pravidlo 5 a R-Counter sa dostane opäť do špeciálneho stavu. Následne sa spracuje signál na ceste „r“, vysiela sa výstup na cestu „k“ a obvod ostáva v pôvodnom stave. Keby signály „a“ a „r“ prišli v opačnom poradí výsledok by bol rovnaký.

Poslednou možnosťou by bolo, keby na vstup „a“ prišli 2 signály a na „r“ jeden, potom by sa vykonal jeden z popísaných prípadov a jeden signál by ostal čakať.

4.4 Moduly zložené z primitív

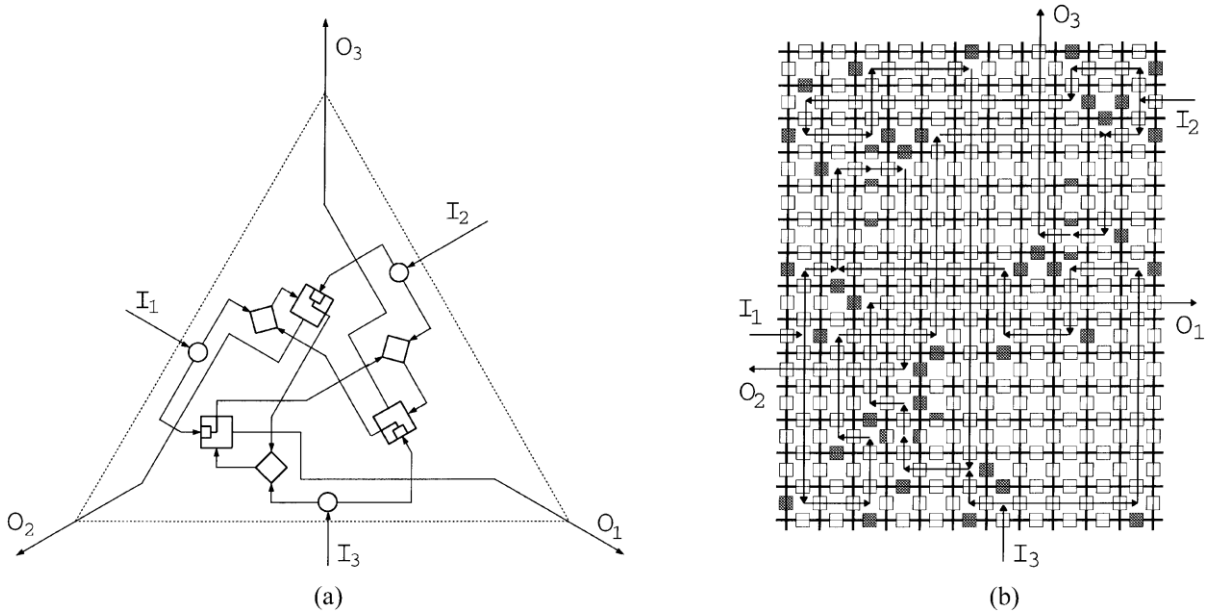
Teraz sme si popísali základné primitíva, z ktorých by sa dali skladať zložitejšie obvody. Avšak musíme myslieť aj nato, že pracujeme s bunkami v 2D mriežke a stanú sa prípady keby budeme musieť krížiť dve cesty. Tento problém by bol v 3D štruktúre buniek podobne ako v viacplošných dosiek plošných spojom lepšie riešiteľný, ale keďže tento návrh je momentálne pre 2D štruktúru, je

nutné spraviť elementy schopné kríženia signálov. V článku [1] je uvedený element *crossing sequencer*, ale pri návrhu obvodov sa skoro vždy dá spraviť dizajn bez nutnosti ho používať.

4.4.1 Tria

Tria je základný synchronizačný modul v asynchrónnych obvodoch, pretože umožňuje zadržať signál, pokým na jeho jeden zo vstupov nepríde nejaký ďalší signál.

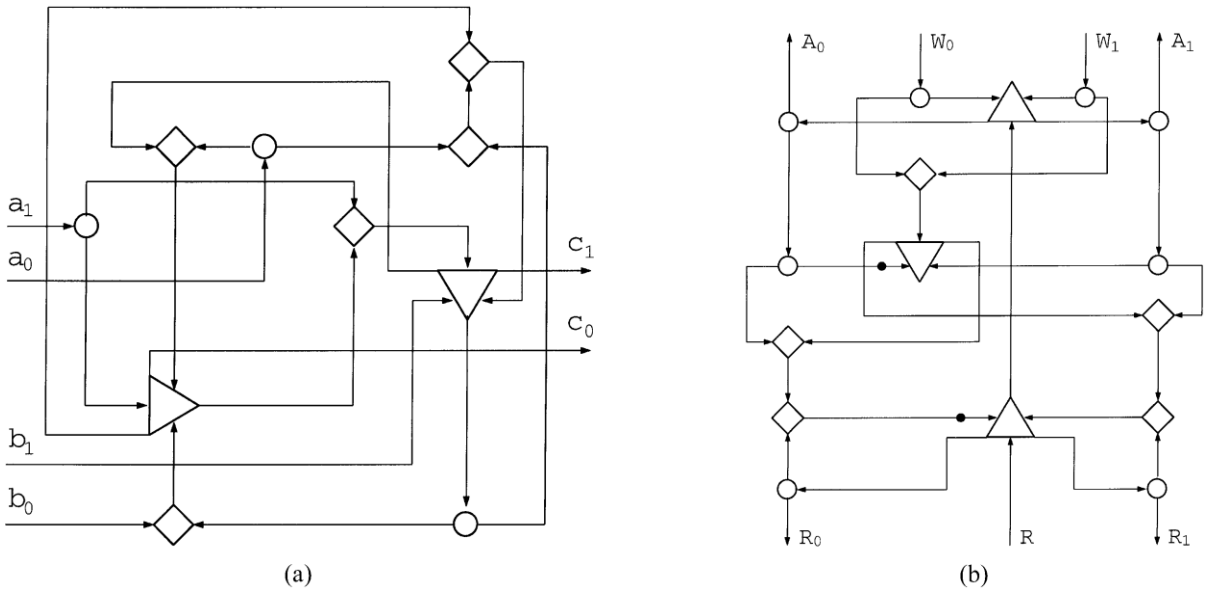
Tria má tri vstupy a tri výstupy, kde každý výstup sa vzťahuje k dvom vstupom. Ak Tria príjme jeden vstupný signál, ostane čakať na príchod druhého signálu. Po prijatí dvoch signálov je na výstup ležiaci medzi vstupmi poslaný signál.



Obrázok 4.12 TRIA – Schéma zapojenia, implementácia v celulárnych poliach [8].

4.5 Základné komponenty nanopočítačov

Doteraz uvedené primitíva a moduly slúžili na prenos signálov. Rozdielom oproti synchrónnym obvodom je, že u synchrónnych obvodoch prítomnosť signálu alebo napätia znamenalo log. 1 a respektíve log. 0. U asynchrónnych obvodoch, keď v danom momente nie je na vstupe žiadny signál, nemôžeme to považovať za log. 0, pretože signál sa mohol oneskoriť, alebo sa niekde čaká na iný signál. Preto sa pri reprezentovaní logických hodnôt používa *dvojdrôtové* (*dual-rail) kódovanie*, ktoré kóduje logické hodnoty pomocou dvoch bitov. Log. 0 je kódovaná kombináciou „01“ (1 je najmenej významový bit) a log. 1 je reprezentovaná „10“. Hodnoty „11“ a „00“ sú neplatné kódové kombinácie. Ďalej budú ešte ukázaný princíp komponent NAND (Obr. 4.13a) a 1-bitovej pamäte (Obr. 4.13b). Z jednobitovej pamäte je možné navrhnúť 1-bitový čítač. Spojením *n* čítačov je možné postaviť n-bitový čítač. Postupne by sa dali zostrojovať zložitejšie moduly až k obvodom, ktoré by boli prakticky použiteľné. Príklad schémy je možné nájsť v literatúre [3], ale z dôvodu že v danej literatúre používajú inú sadu primitív, je nutné prispôsobiť schému prechodovým pravidlám definovaných v tejto práci.



Obrázok 4.13 Ďalšie komponenty, a) dvojdrôtovo kódované NAND hradlo, b) dvojdrôtovo kódovaná 1-bitová pamäť [8].

5 Návrh implementácie simulátoru celulárnych polí

V tejto kapitole je popísaná aplikácia na simuláciu asynchrónnych obvodov realizovaných technológiou celulárnych polí. Kapitola sa nebude zaoberať implementačnými detailami, ale skôr bude zameraná na analýzu problému. Načrtne vhodné riešenie a dátové štruktúry pre potreby simulácie. Implementácia bude podrobnejšie popísaná až v kapitole 6.

Na začiatku tejto kapitoly budú definované požiadavky na aplikáciu. Potom bude nasledovať podkapitola zameraná na výber programovacieho jazyka. Ďalšie podkapitoly sa venujú už konkrétnemu návrhu aplikácie, ktorý je rozdelený na dve časti. V prvej časti sa budeme zaoberať návrhom simulačnej vrstvy. V druhej časti sa zameriame na návrh prezentačnej vrstvy, zahrňujúcej hlavne zobrazovanie a editáciu celulárneho poľa s možnosťou sledovania simulácie. Vrstvy sú navrhnuté tak, aby boli použiteľné samostatne, čo dovoľuje spúšťať aplikáciu bez, alebo s užívateľským rozhraním.

5.1 Neformálna špecifikácia

Cieľom projektu je aplikácia umožňujúca simulovať nanopočítače a jeho komponenty založené na báze celulárnych automatov. Aplikácia by mala byť schopná simulovať primerane veľké komponenty v dostatočne krátkom čase.

Dôraz je kladený tiež na intuitívnosť užívateľského rozhrania, ktoré by samozrejme malo užívateľovi poskytovať dostatočné možnosti ako pri simulácii, taktiež aj pri návrhu obvodov za účelom simulácie. Rozhranie preto musí z hľadiska simulovateľnosti disponovať možnosťou zobrazenia priebehu simulácie a nastavenia rýchlosti simulácie. Z hľadiska dobrej navrhovateľnosti musí podporovať editáciu a presúvanie buniek a približovanie a odďaľovanie plochy.

Aby bolo možné simulovanie nanopočítača, je nutné najskôr navrhnutie jednotlivých komponent z ktorých budú zložené komplexnejšie komponenty. Preto užívateľské rozhranie by malo mať možnosť okrem simulovania načítaných konfigurácii aj vytváranie jednotlivých komponent a spájanie ich do väčších. Aplikácia by preto mala užívateľovi poskytovať vhodné pomôcky pri práci s komponentami, ako je napríklad možnosť výberu, vloženie na určité miesto a presúvanie komponent.

Aplikácia musí byť schopná fungovať bez grafického užívateľského rozhrania. Na základe parametrov predaných pri spustení by mala vykonať simuláciu a výstup vhodne vrátiť alebo uložiť do súboru. Tým bude možné jej prepojenie s ďalšími aplikáciami.

5.2 Java

Na základe analýzy požiadaviek na systém, bol ako implementačný jazyk zvolená Java. Je to objektovo orientovaný programovací jazyk. Syntax bola odvodená od jazyka C++. Objektový model je jednoduchší a nemá toľko nízkoúrovňových konštrukcií. Aplikácie napísané v Jave sú kompilované do byte kódu, ktorý je interpretovaný pomocou Java Virtual Machine (JVM). Vďaka tomu sú aplikácie nezávisle na architektúre.

Java používa ku správe pamäti automatický garbage collector, ktorý sa stará o pridelenú pamäť po celú dobu života objektov. Programátor určuje kedy bude objekt vytvorený a garbage collector sa stará o uvoľňovanie pamäti. Ak objekt už naďalej nie je používaný (neukazuje naň žiadna referencia) bude týmto nástrojom odstránený z pamäte. K vykonaniu údržby pamäte môže dôjsť kedykoľvek, v ideálnom prípade ak je program nečinný. Explicitná správa pamäte nie je možná.

Java nepodporuje ukazovatele známe z jazyka C,C++. V Jave sú objekty predávané jedine cez referenciu. Táto vlastnosť umožňuje garbage collectoru realokáciu pamäte. Primitívne dátové typy nie sú v jave realizované ako objekty, a preto ich používanie nemá negatívne dôsledky na výkon, ako by to mu bolo v čisto objektovom jazyku.

Dôvodom používania Javy bol fakt, že výhody plynúce z jej používania značne prevyšovali nad nevýhodami. Medzi nevýhody by som zaradil iba pomalší výpočet napríklad v porovnaní s jazykom C++. Je to dané tým, že Java nie je prekladaná do strojového kódu, ako je tomu u jazyku C++. Program teda nepracuje priamo s pamäťou, ale pristupuje k nej prostredníctvom JVM. Tento problém je dnes možné minimalizovať použitím *Just-in-time* (JIT) prekladača. Podľa typu použitých konštrukcií dokážu JIT prekladače veľkú časť a niekedy aj celý byte kód preložiť do natívneho strojového kódu, ktorý je následne spustený. Preklad je samozrejme optimalizovaný podľa architektúry, na ktorej je program kompilovaný.

Avšak spomínal som, že existuje rada výhod. To že program je spúšťaný cez JVM umožňuje prenositeľnosť medzi rôznymi platformami. Nespornou výhodou je bohatá sada dostupných knižníc. Knižnice sú prevažne dobre zdokumentované a existuje množstvo tutoriálov a príkladov. Bohatá podpora je v rámci komunity a fór, a preto je vývoj v Jave porovnaní s ostatnými jazykmi oveľa menej náročnejší na čas. Taktiež nároky na programátorské schopnosti nie sú príliš veľké v porovnaní s napríklad C++. Pre vývoj GUI existujú viaceré programy, medzi ktorými je najznámejší NetBeans. Taktiež disponuje podpornými nástrojmi pri tvorbe užívateľského rozhrania.

5.3 Simulátor

Navrhovaný simulátor je určený na simulovanie asynchrónnych obvodov. Hlavnými požiadavkami pri návrhu simulátoru bola *dobrá konfigurovateľnosť* s možnosťou jednoduchého *napojenia na užívateľské* rozhranie. Pod pojmom dobrá konfigurovateľnosť je myslená možnosť používania ľubovoľnej sady prechodových pravidiel, nastavovanie rýchlosti simulácii, schopnosť spracovať každý formát vstupu a podobne aj ukladanie do ľubovoľne zvoleného výstupu. Medzi hlavné ciele patrí samozrejme aj škálovateľnosť, teda dôraz je kladený na rýchlosť a schopnosť simulovania obvodov a modulov s veľkými rozmermi. Veľkosť konfigurácie, ktorú je simulátor schopný simulovať, je ovplyvnená jedine veľkosťou pamäte.

Väčšina aplikácií na simulovanie úloh založených na modelu celulárnych automatov má práve problém s rýchlosťou. To vyplýva z dôvodu nutnosti aplikovať prechodové pravidlá na každú bunku. Algoritmus používaný pri nastavovaní nového stavu buniek bol navrhnutý s cieľom minimalizovať priechod mriežkou a minimalizovať počet buniek, ktoré budú podliehať testovaniu aplikovateľnosti nejakého prechodového pravidla.

5.3.1 Bunka a celulárne pole

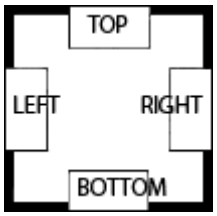
Celulárne pole môže byť reprezentované viacerými možnosťami. Najjednoduchším spôsobom je uloženie buniek do dvojrozmernej matice. Takýto spôsob poskytuje rýchle vyhľadávanie buniek na

základe súradníc x a y, ktoré určujú pozíciu bunky v ploche. Existujú aj iné techniky implementácie, ako je napríklad hashovacie pole, ktoré majú menšie pamäťové nároky v prípade riedko rozmiestnených buniek, ale nájdenie bunky má väčšiu časovú náročnosť. V celulárnych automatoch môžu byť teoreticky využívané všetky bunky. Pretože pri aplikácii prechodových pravidiel je nutné poznať stav bunky a stav okolitých buniek, je dôležitý rýchly prístup k susedným bunkám. Z toho dôvodu je dvojrozmerná matica optimálnou dátovou štruktúrou na reprezentáciu celulárneho poľa.

V aplikácii preto budú bunky uložené v dvojrozmernom poli `cells[][]` typu `Cell`. Toto pole bude umiestnené v triede `CellGrid`, ktorá bude umožňovať pristupovať k bunkám na požadovaných pozíciách a tiež bude mať informáciu o počte riadkov a stĺpcov matice.

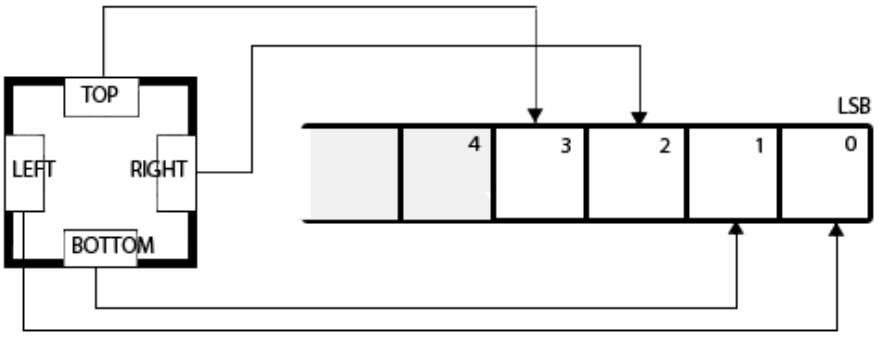
Keďže sa jedná o celulárne automaty, ako už bolo spomenuté, je vhodné rýchlo a pohodlne pristupovať k susedným bunkám. Preto je každej bunke predaná inštancia triedy `CellGrid`. Bunka musí minimálne obsahovať atribúty `row`, `col` určujúce polohu bunky a `cellGrid` ako referenciu na objekt s uloženými bunkami. Tieto atribúty nie je možné za behu aplikácie meniť, aby sa zachovala integrita modelu (riadok a stĺpec bunky musí byť v súlade s jej skutočnou pozíciou v poli `cells`). Pomocou metódy `getNestedCell(int relativeRow, int relativeCol)` je možné získať referencie na susedné bunky. Napríklad TOP bunku je možné získať volaním `getNestedCell(-1, 0)`, a pretože každá bunka má nastavenú pozíciu v poli, jednoduchým výpočtom `getCell(row+relativeRow, col+relativeCol)` je susedná bunka okamžite sprístupnená.

Trieda `Cell` musí mať atribút `state`, v ktorom je vhodne zakódovaný stav bunky. Bunka je zobrazená na obrázku 5.1. Vidíme, že jej stav je daný nastavením štyroch pamätí na hodnotu 1 alebo 0. Na zakódovanie celého stavu nám budú stačiť 4 bity. Stav bude i napriek tomu uchovaný v dátovom type integer, pretože Java s týmto typom vie optimálne pracovať. Pri použití iných dátových typov by dokonca mohlo dôjsť k spomaleniu aplikácie spôsobené konverziou iného typu na integer.



Obrázok 5.1: Bunka s označenými pamäťami.

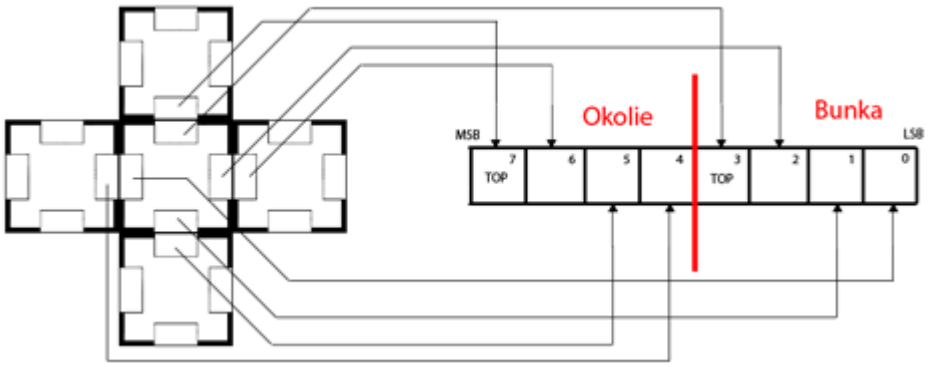
Nasledovný obrázok ukazuje zakódovanie stavu bunky v atribúte `state`. Využívané sú prvé 4 bity. Napríklad pre nakonfigurovanie bunky na stav, v ktorom TOP a RIGHT sú nastavené na 1, ostatné na 0, by `state` musel mať hodnotu dekadicky 12 (binárne 1100).



Obrázok 5.2: Mapovanie stavu bunky do typu Integer.

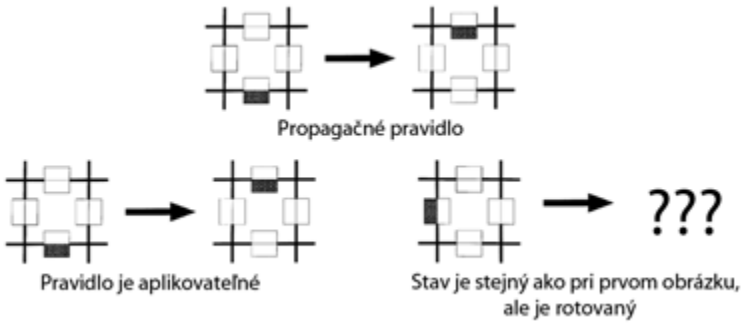
5.3.2 Prechodové pravidlá

Prechodové pravidlo bude v aplikácii modelované triedou `Rule`. Tá bude mať atribúty `currentState` reprezentujúci aktuálny stav, a `nextState` pre nasledujúci stav. Aby pravidlo vedelo efektívne testovať, či je možná zmena stavu, musí byť stav okolia rozumne zakódovaný. Podobne ako je stav bunky mapovaný na integer, je tento spôsob použitý pri mapovaní pravidla. Toto kódovanie dovoľuje zistiť, či je pravidlo aplikovateľné na základe jednoduchého porovnania dvoch hodnôt. Mapovanie je zobrazené na obrázku 5.4, kde vidíme, že mapovanie bunky je zachované a v čísle sa používajú ďalšie 4 bity symbolizujúce stav okolia.



Obrázok 5.4: Mapovanie stavu bunky a okolia do typu Integer.

Pravidlo musí byť schopné určiť, či je aplikovateľné na bunku a jej okolie. To zabezpečí metóda `boolean match(int state)`, kam bude predaný zakódovaný stav bunky s okolím, a vrátením `true` alebo `false` rozhodne, či sa zhoduje alebo nie. Ak pravidlo uspeje, nasledovný stav je získaný metódou `getMatchedState()`. Triviálny spôsob implementácie metódy `match` by mohol byť realizovaný porovnaním `currentState` a `state`. V prípade rovnosti by sa stav buniek zmenil na `nextState`. Čo by samozrejme fungovalo, ak by konfigurácia bola rovnaká ako pri definovaní pravidla. Avšak konfigurácia môže byť pootočená o 90° a porovnanie by skončilo nezhodou. Preto je nutné, aby každé pravidlo bolo schopné správne porovnávať všetky 4 varianty pootočenia stavu. Problém je zobrazený na obrázku 5.3.



Obrázok 5.3: Problém pri aplikovaní pravidla, keď je stav rotovaný.

Riešením je vytvoriť triedu `RotatedRule`, ktorá bude rozširovať vlastnosti triedy `Rule`. `RotatedRule` si pri inicializácii vytvorí všetky štyri varianty pravidla. Metóda `match` bude prechádzať vytvorenými variantmi, a ak nájde zhodu, vráti `true`. Nasledovný stav bude dostupný volaním metódy `getMatchedState`, ktorý sa zapamätá v metóde `match`.

5.3.3 Algoritmus na aktualizáciu buniek

Triviálnym riešením je prechádzať všetky bunky a testovať, či je možné aplikovať pravidlo a prípadne zmeniť stav bunky. Takéto riešenie nie je optimálne. Navrhnutý algoritmus prechádza všetkými bunkami jedine v prvej iterácii. V ďalších iteráciách sa pracuje s bunkami, ktoré boli ovplyvnené. Ovplyvnené bunky sú vždy bunky okolia vrátane konkrétnej bunky. To je dané tým, že pravidlo pri zmene stavu nemení len stav konkrétnej bunky, ale aj stav okolitých buniek. Inak povedané, bunky, ktoré v ďalšej iterácii nebudú môcť byť zmenené, sa ignorujú. Poloformálny zápis algoritmu vyzerá takto:

1. Vytvor *hlavnú frontu* buniek.
2. Do hlavnej fronty pridaj všetky bunky.
3. Vytvor *pomocnú frontu*, premiestni do nej obsah *hlavnej fronty* (*hlavná fronta* ostane prázdna) a všetky bunky v *pomocnej fronte* označ ako INVALID.
4. Prechádzaj *pomocnú frontu* buniek dokým nie je prázdna.
   4.1. Z *dočasnej fronty* vyber prvú bunku.
   4.2. Ak je bunka označená ako VALID, pokračuj krokom 4.1.
   4.3. Označ bunku ako VALID.
   4.4. Ak je možné aplikovať prechodové pravidlo, zmeň stav bunky a jej okolie.
5. Ak nie je *hlavná fronta* prázdna potom pokračuj krokom 3.
6. Vráť finálnu konfiguráciu (ďalšia aktualizácia je zbytočná).

Vstupom algoritmu je *pole prechodových pravidiel* a *počiatočná konfigurácia buniek*. Výstupom je *konfigurácia buniek* zmenená podľa prechodových pravidiel.

Algoritmus pracuje s dvoma frontami, *hlavnou* a *pomocnou*. Hlavná fronta sa napĺňa počas iterácie, zatiaľ čo pomocná sa vyprázdňuje. Algoritmus by samozrejme mohol pracovať s jednou frontou, kde by sa bunky vyberali od začiatku a nové pridávali na koniec. Aj keď riešenie by to bolo funkčné, používanie 2 front má svoje výhody. Napríklad pri spustení simulácie by sme potrebovali sledovať šírenie signálu a zaujímal by nás počet iterácií, za ktoré sa dostane signál z bodu A do B. To by sa dalo spočítať pomerne jednoducho. Je to počet iterácii algoritmu. Inak povedané, aktualizácia všetkých buniek fronty predstavuje jeden paralelný krok. Pri variante s jednou frontou je ale počet iterácií počtom aktualizovaných buniek a nie sme schopný jednoduchým spôsobom spočítať počet krokov.

Aj keď to možno nie je na prvý pohľad vidno, v algoritme sú 2 cykly. Vonkajší cyklus je hodne závislý na konfigurácii plochy. Počet iterácii môže byť teoreticky nekonečný, čo by malo byť určite ošetrené aj pri implementácii nejakou vhodnou maximálnou hodnotou. Tento cyklus je daný bodmi 3 a 5.

V bode 3 sa obsah hlavnej fronty premiestni do pomocnej. Hlavná fronta sa vyčistí a bude pripravená na uloženie buniek pre ďalšiu iteráciu. Tieto bunky budú postupne pridávané vo vnútornom cykle. V bode 4 sa začne prechádzať pomocná fronta. Je možné buď vyberať vždy prvý prvok alebo len prejsť jej obsahom. Druhý spôsob je efektívnejší, pretože prechod frontou je určite jednoduchší ako operácia výberu, čo je spojené s mazaním prvého prvku a realokáciou pamäte.

V každej iterácii vnútorného cyklu je spracovaná jedna bunka. Pretože algoritmus pri pridávaní nových buniek do hlavnej fronty nekontroluje, či táto bunka vo fronte existuje, je nutné

vedieť, že bunka už bola aktualizovaná. Bunka nemôže byť aktualizovaná dvakrát počas jednej iterácie. Takže riešením je pridať bunke informáciu o validite. VALID znamená, že bunka už v danej iterácii bola spracovaná, INVALID čaká na spracovanie. Tento stav je kontrolovaný hneď na začiatku vnútornej iterácie. Ak je v stave VALID, pokračuje sa ďalšou bunkou. Ak je INVALID, vidíme, že okamžite sa mení na VALID a nemôže nastať situácia, že by bunka bola spracovaná dvakrát.

Ďalej sa zisťuje, či je možné na bunku aplikovať nejaké pravidlo. V prípade, že sa také pravidlo nájde, ďalšie pravidlá sa už ignorujú. Podľa nájdeného pravidla je zaktualizovaný stav bunky a stav okolitých buniek. Všetky zmenené bunky sú pridané do hlavnej fronty. A tu vzniká problém opakovania buniek. Aktualizované môžu byť napríklad 2 susedné bunky, čo spôsobí dvojité pridanie. Riešením by bola tiež kontrola, či bunka už vo fronte existuje, no vzhľadom k tomu, že sa snažíme vymyslieť najrýchlejšie riešenie, je test booleovskej premennej omnoho rýchlejší ako vyhľadávanie v poli či fronte.

Časová zložitosť algoritmu je daná veľkosťou plochy, počtu prechodových pravidiel a tiež závisí od počtu ovplyvnených buniek v rámci iterácii. Povedzme, že vonkajší cyklus bude mať *a* iterácií. Počet riadkov a stĺpcov bude *n*, teda veľkosť plochy je n^2. Počet pravidiel je *m*. Zložitosť triviálneho algoritmu je teda $O(an^2m)$. Navrhnutý algoritmus pracuje lepšie. Časová zložitosť prvej iterácie je $O(n^2m)$. V ďalších iteráciách sa aktualizujú len zmenené bunky a ich počet je hodne závislý od konfigurácie, ale zo skúseností sa ukazuje, že tento počet sa pohybuje okolo čísla *kn*, kde *k* je nezáporné číslo blízke 1, čo môžeme považovať za konštantu. Teda zložitosť ostatných iterácií je $O((a-1)nm)$. Výsledná zložitosť je:

$$O(n^2m) + O((a-1)nm) = O(n^2m + (a-1)nm)$$

5.3.4 Plánovanie aktualizácií

Navrhnutý algoritmus funguje výborne, dokým je spúšťaný bez grafického rozhrania. V prípade že by sme chceli zobraziť priebeh simulácie, jeho činnosť by sa musela spomaliť natoľko, aby užívateľ bol schopný sledovať zmeny. Aktívne čakanie nie je moc vhodné, pretože zbytočne vyťažuje procesor. Ako jedna z možností bola rozdeliť algoritmus medzi viaceré triedy. Trieda `CellsUpdater` sa bude starať o aktualizáciu buniek, teda bude implementovať vnorený cyklus algoritmu, čo je v podstate celý krok 4. Aktualizácia plochy sa vykoná volaním metódy `update`, pričom si pamätá hlavnú frontu medzi jednotlivými volaniami. `Update` vracia informáciu o tom, či nastala nejaká zmena, čo bude dôležitá informácia pre plánovač. `CellsUpdater` sám nikdy nevolá `update`. Metóda `update` môže byť volaná nejakým iným objektom, nazývaným plánovač, ktorý bude mať na starosti plánovanie aktualizácii. V simulátore budú implementované dva typy plánovača.

Východzím plánovačom je objekt triedy `IterableScheduler`, ktorý funguje presne podľa uvedeného algoritmu a v cykle volá metódu `update`. Tento plánovač je extrémne rýchly, ale nie je možné nastaviť rýchlosť simulácie, respektíve oneskorenie medzi jednotlivými volaniami `update`. To znamená, že pracuje v cykle, dokým sa vykonávajú nejaké zmeny. Taktiež ho nie je možné zastaviť v priebehu simulácie.

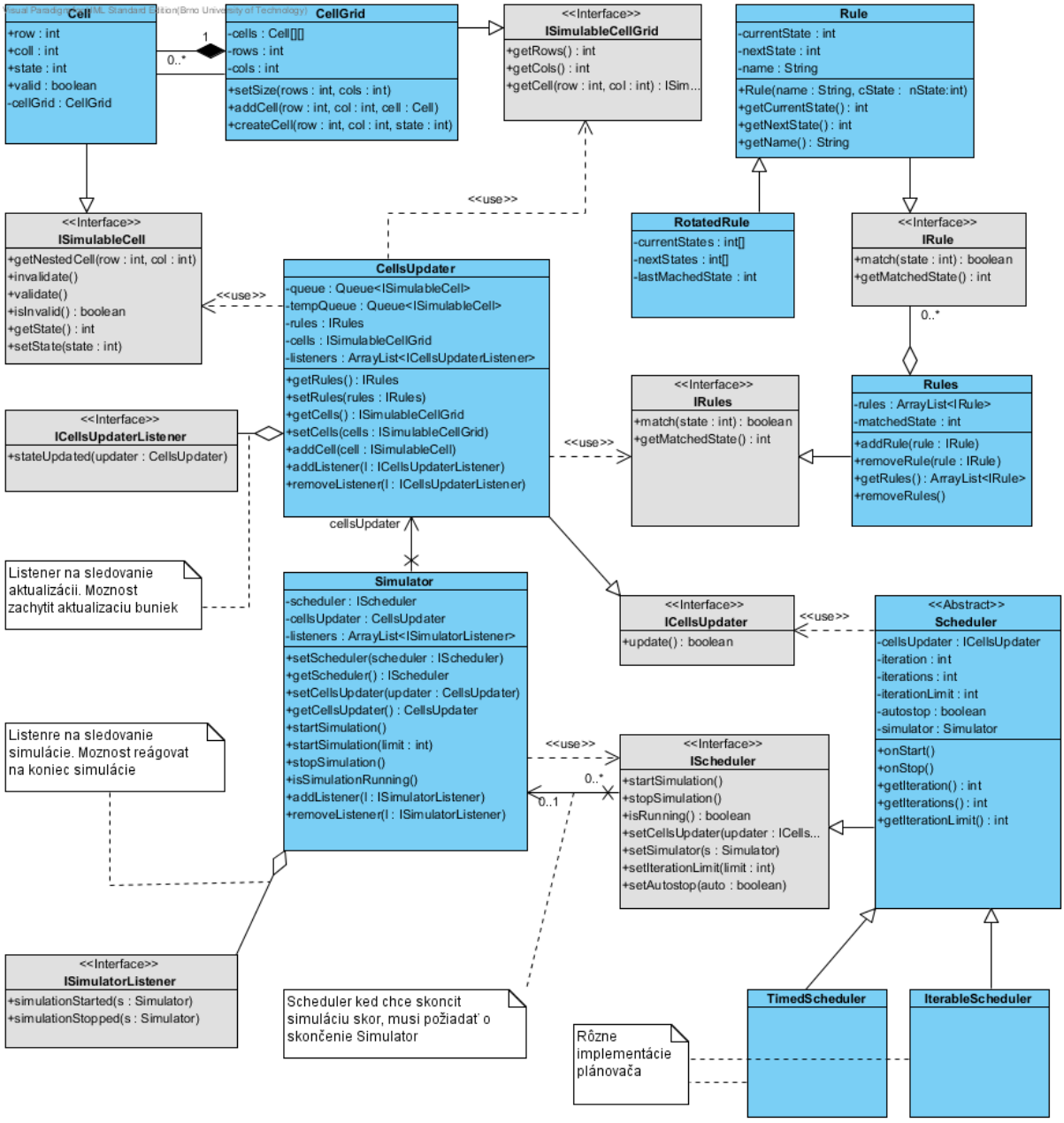
Druhým plánovačom je objekt triedy `TimedScheduler`, ktorý je vhodný pri napojení na grafické užívateľské rozhranie. Tento objekt využíva pre plánovanie triedu `java.util.Timer`. Tá umožňuje registrovať úlohu a spustiť ju v stanovený čas. Úloha bude veľmi triviálna a jej náplňou bude volanie metódy `update`. `TimedScheduler` sa postará po vykonaní úlohy o registráciu ďalšej úlohy, ktorá sa spusti s definovaným oneskorením. Oneskorenie je možné meniť i v priebehu

simulácie, čo dovoľuje riadiť rýchlosť priamo z užívateľského rozhrania. `TimedScheduler` je tiež možné kedykoľvek zastaviť a následne spustiť. To umožňuje užívateľovi simulovanie po krokoch.

Oba plánovače budú podporovať nastavenie maximálneho počtu iterácií. V prípade, že nebude nastavené, plánovač bude plánovať vždy ďalšie volanie metódy `update`, dokým sa vykoná nejaká zmena.

5.3.5 Diagram návrhových tried

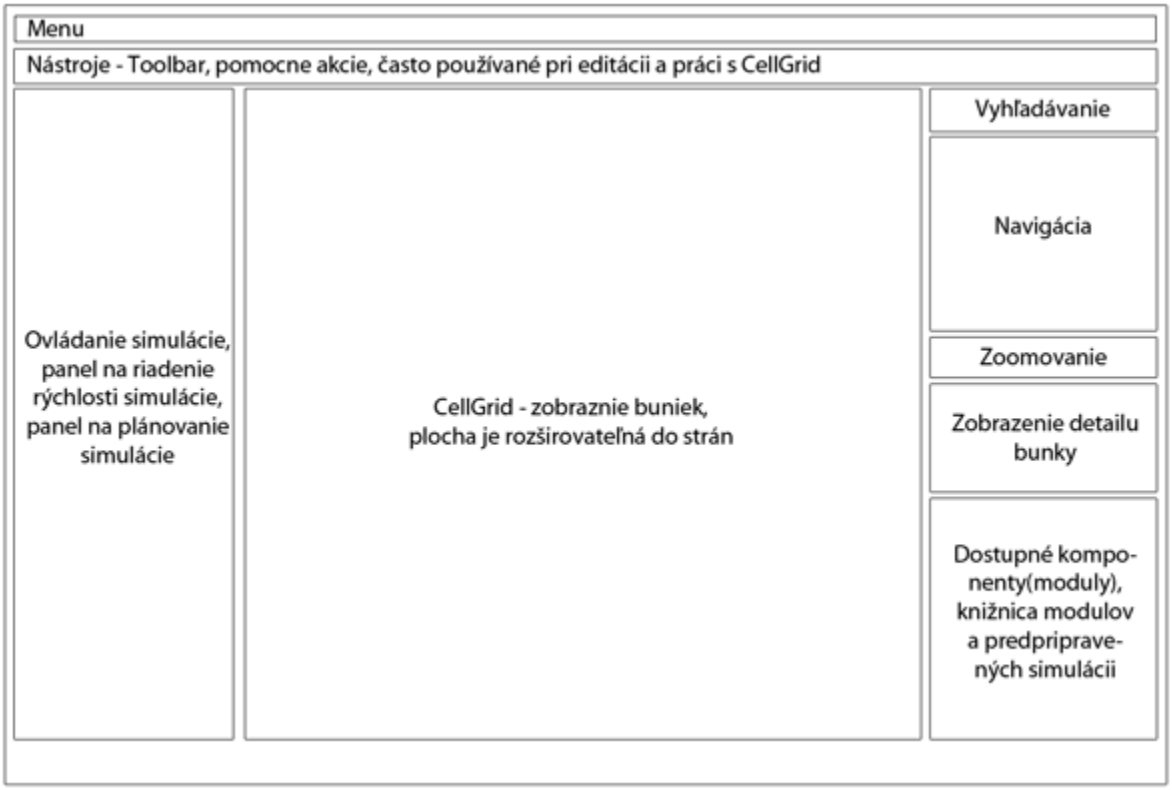
Nasledovný diagram (obrázok 5.4) ukazuje závislosti medzi triedami. Kvôli prehľadnosti sú zakreslené len triedy priamo súvisiace s návrhom. Ďalšie pomocné triedy je možné nájsť v programovej dokumentácii. Taktiež kvôli úspore miesta nie sú zobrazené všetky metódy. Hlavne boli vynechané privátne a pomocné funkcie. Všetky triedy simulátoru sú zabalené v balíčku Simulator, na ktorý bude v nasledujúcej podkapitole zobrazené napojenie užívateľské rozhranie.



Obrázok 5.4: Diagram návrhových tried simulátora. Zobrazené sú triedy týkajúce sa simulácie.

5.4 Grafické užívateľské rozhranie

Návrh grafického rozhrania definuje umiestnenie jednotlivých funkčných blokov a spôsob komunikácie s užívateľom. Taktiež zahrňuje ovládanie komponent a prepojenie komponent medzi sebou. Cieľom návrhu bolo, aby aplikácia poskytovala užívateľovi dostatočné možnosti pri editovaní plochy a práci s veľkým počtom buniek.



Obrázok 5.5. Diagram ukazujúci rozloženie prvkov v GUI.

Hlavnou časťou rozhrania je práve plocha na upravovanie buniek, pomenovaná `GridPanel`. Pretože väčšina práce s programom bude spojená práve s `GridPanel` musí fungovať intuitívne. Základom je práca s bunkami. `GridPanel` musí vedieť detekovať kliknutie na bunku. V prípade, že užívateľ klikne na nejakú z pamätí, `GridPanel` vyvolá udalosť s informáciou o bunke a detekovanou pamäťou. Poznamenajme, že panel patrí do prezentačnej vrstvy a preto jeho úlohou je zobrazovať dáta, nie však ich meniť. Zmena dát sa musí vykonať v inej časti aplikácie. Komponenta zaznamená operácie užívateľa a vyvolá udalosti, na ktoré bude možné reagovať. Viac o tejto technike bude popísané v kapitole návrh architektúry (5.6). Editovanie buniek nie je ale postačujúce. Návrh obvodov si vyžaduje presnosť a dosť často sa môže stať, že užívateľ bude potrebovať presunúť konfiguráciu alebo zmazať, prípadne zistí že nejaká konfigurácia bola nastavená tam, kde nechcel. V takýchto prípadoch rekonfigurácia štýlom prenastaviť chybné bunky by bola v rozpore so snahou spraviť jednoduché a intuitívne rozhranie. Riešením je mať možnosť označovať bunky a tým tvoriť selekcie. Nad selekciou bude možné vykonávať operácie presúvanie, mazanie, kopírovanie a vkladanie vybraných buniek. Tieto operácie by mali byť postačujúce pre základnú editáciu.

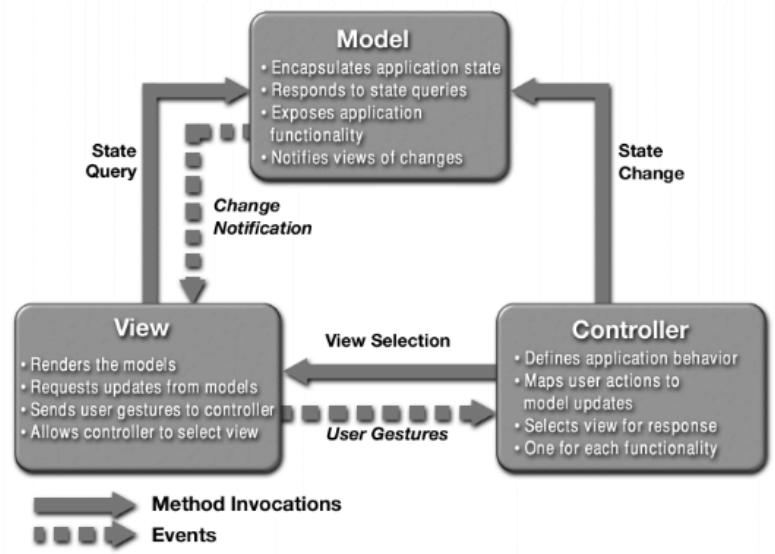
Obecne pri návrhu obvodov sa vždy zložitejšie časti navrhujú z nejakých predvytvorených jednoduchších komponent, a k ním je pridaná logika navyše. To isté platí aj pri návrhu asynchrónnych obvodov. Preto je nutné, aby si užívateľ mohol vytvárať moduly a tie komponovať do väčších celkov. Modulom je myslená konfigurácia buniek zložená z vlastnej konfigurácie a prípadne z vnorených modulov. Modulárnosť je základom tvorby každej väčšej aplikácie. Ak vyžadujeme, aby program mohol simulovať zložité obvody, dokonca nanopočítače, je nutné, rozložiť konfiguráciu do modulov. Podpora modulárnosti bude dôležitou funkcionalitou `GridPanelu` a musí zahŕňať základné operácie nad modulmi, a tými sú *vkladanie*, *mazanie* a *posúvanie*. To, že som neuviedol *editáciu* modulu, nie je chyba, ale zámer. Môže sa zdať táto podmienka zvláštna, dokonca je možné tvrdiť, že modul má určite nejaké nevyužité miesto, ktoré by sa po jeho vložení hodilo, ale ďalej je uvedené vysvetlenie, prečo je tento prístup špatný. Zásahom do vloženého modulu by jednak mohli nastať chyby úpravou či nepozornosťou, ale väčší dopad by to malo v praxi pri realizácii obvodov. Modul môže byť nejakou technológiou, napríklad chemickou syntézou prefabrikovaný, a potom ďalej používaný bez znalosti obsahu. Je možné si ho predstaviť ako čiernu skrinku, ktorá niečo robí, ale nevieme ako to robí. Povolením editácie vloženého modulu by takto syntetizované moduly boli nepoužiteľné, lebo náš modul by bol čiastočne modifikovaný. Modul sa môže upravovať len keď je otvorený (nastavovanie stavu buniek modulu), čo je požadovaná funkcionalita. Poslednou vlastnosťou panelu je podpora odďaľovania plochy. Tým by sme mali navrhnutú komponentu na prácu s celulárnou mriežkou a rozsiahlymi konfiguráciami.

Medzi ďalšie špeciálne komponenty patrí `NavigationPanel`. Panel bude zobrazovať náhľad celej plochy, kde užívateľ musí vidieť organizáciu modulov a tiež viditeľnú oblasť buniek na obrazovke. Navigátor by mal pracovať intuitívne, čo zahŕňa presunutie viditeľnej plochy v `GridPanele` po kliknutí myšou v navigačnej oblasti. Poslednou komponentou, ktorú treba pripraviť, je `CellPanel` na zobrazenie informácii o bunke. Tento panel by mal umožňovať editáciu všetkých atribútov bunky, ktoré užívateľ môže meniť. Tými sú hodnoty pamätí (spomenuté v kapitole 5.3) a potom názov bunky. Riadok a pozícia sú dostupné len na čítanie. Na implementovanie ostatných funkcií stačí použiť komponenty z knižnice Swing. Finálne rozloženie funkčných blokov je možné vidieť na obrázku 5.5. Tento návrh je len konceptuálny a nezobrazuje priamo užívateľské rozhranie, slúži na predstavu ako bude aplikácia rozložená.

5.4.1 Návrh architektúry

Štruktúra programu je navrhnutá podľa architektonického vzoru Model-View-Controller (MVC). Ten sa začal používať hlavne v desktopových aplikáciách a dnes patrí medzi najpoužívanejšie architektonické vzory. Jeho znalosť je nutná pre pochopenie funkčnosti simulátoru a spôsobu prepojenia komponent, preto si popíšeme jeho princíp. Návrhový vzor MVC [15] rozdeľuje štruktúru aplikácie do troch oblastí zodpovednosti:

- *Model* (model) je doménovo špecifická reprezentácia dát a informácii.
- *View* (pohľad) interpretuje dáta reprezentované modelom do podoby vhodnej k interaktívnej prezentácii užívateľovi.
- *Controller* (radič) reaguje na udalosti (typicky pochádzajúce od užívateľa) a zaisťuje zmeny v modele alebo v pohľade.



Obrázok 5.6: Architektonický vzor Model-View-Controller [14].

Obrázok 5.6 ukazuje vzťah medzi jednotlivými časťami vzoru MVC. Šípky znázorňujú smer komunikácie. Hoci tento koncept môže byť realizovaný rôznym spôsobom, obecne platí:

Užívateľ vykoná nejakú akciu v užívateľskom rozhraní (napr. stlačí tlačidlo). Radič dostane oznámenie o tejto akcii z objektu užívateľského rozhrania.

Radič pristúpi k modelu a v prípade potreby ho aktualizuje na základe vykonanej užívateľskej akcie (napr. zaktualizuje prvky formuláru).

Model je len iný názov pre doménovú vrstvu. Doménová logika spracuje zmenené dáta (napr. automaticky doplní nejaké hodnoty do formuláru). Niektoré aplikácie používajú mechanizmus pre perzistentné uloženie dát (napr. databázu). To je však otázka vzťahu medzi doménovú a dátovú vrstvu, ktorá nie je architektúrou MVC pokrytá. Náš model je uložený na disku vo forme konfiguračného súboru.

Pohľad použije model pre zobrazenie zaktualizovaných dát užívateľom (napr. vykreslí graf na základe získaných dát z formuláru). Pohľad získava dáta priamo z modelu, zatiaľ čo model nepotrebuje žiadne informácie o pohľade (je na nej nezávislý). Avšak je možné použiť návrhový vzor pozorovateľ (angl. Observer), umožňujúci modelu informovať akúkoľvek komponentu o prípadných zmenách dát. V tom prípade sa view zaregistruje u modelu ako príjemca týchto informácií. Je dôležité podotknúť, že radič neodovzdáva doménové objekty (model) komponente pohľadu, napriek tomu jej môže poslať príkaz, aby svoj obsah podľa modelu aktualizovala. Táto technika bude používaná aj v našom návrhu. Užívateľské rozhranie čaká na ďalšiu akciu užívateľa, ktorá celý cyklus začne znovu.

5.4.2 Návrh aplikácie

V kapitole 5.4 sme si navrhli užívateľské rozhranie. V tejto podkapitole si navrhneme štruktúru tried na zabezpečenie požadovaného chovania. Poznamenajme, že balíček Simulátor už obsahuje triedy `CellGrid` a `Cell` definujúce štruktúru buniek. Ich vlastnosti sa výborne hodia pri zobrazovaní buniek, avšak ich bude nutné rozšíriť pre potreby modularizácie.

Pred návrhom aplikácie si musíme analyzovať dáta, s ktorými ideme pracovať. Predmetom simulácie sú asynchrónne obvody, vyznačujúce sa modulárnosťou. To znamená, že navrhovaný modul môže byť zložený z *konfigurácie buniek* a jednoduchších *predprípravných modulov*, ktoré sa

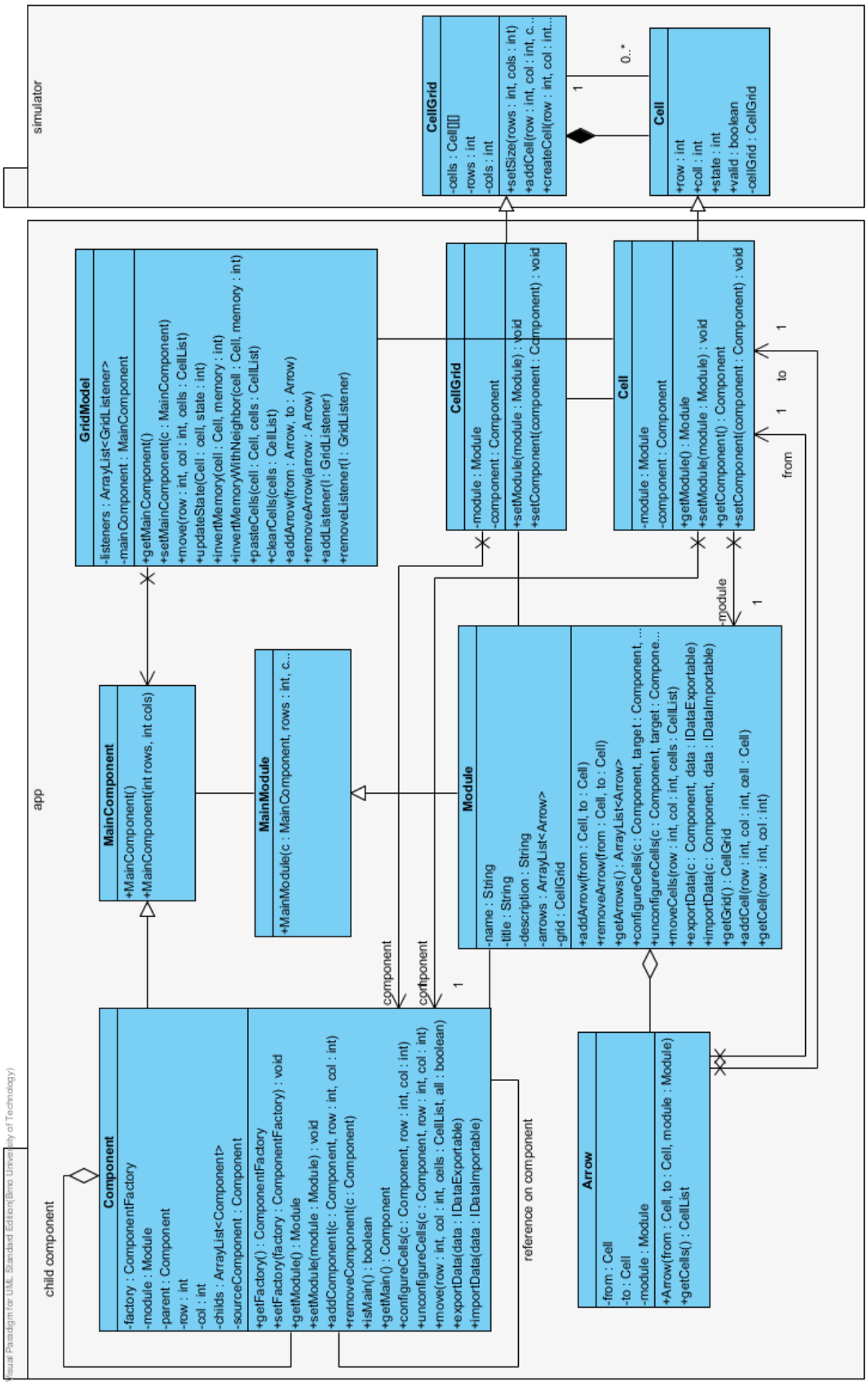
môžu a pravdepodobne aj budú opakovať. Moduly môžu byť rozmiestnené ľubovoľne bez toho aby sa prekrývali a samozrejme môžu obsahovať i ďalšie vnorené moduly.

Povaha týchto dát je tvoriť stromovú štruktúru modulov. Jedným z možných riešení by bolo vytvoriť triedu implementujúcu skladanie modulov s tým, že modul bude niesť informáciu o svojej konfigurácii. Avšak pri mnohonásobnom použití modulu, réžia spojená s organizovaním stromu je minimálna, ale udržiavanie konfigurácie v každom objekte (module) je neefektívne, pretože vyťažuje veľa pamäte. Poznamenajme, že konfigurácia určitého modulu je vždy rovnaká, jediné čo sa mení je pozícia, na ktorú je vložený. Z toho dôvodu budú vytvorené dve triedy, a to trieda `Component` na udržiavanie informácii o štruktúre stromu. Jednotlivé položky stromu budeme nazývať komponentami. Každá komponenta bude mať referenciu práve na jeden objekt triedy `Module`, v ktorom bude uložená informácia o jeho konfigurácii. Na modul môže byť odkazované z viacerých komponent, čo je hlavným cieľom nášho návrhu. Návrh tried je možné vidieť v diagrame návrhových tried zobrazenom na obrázku 5.7.

Trieda `Component` umožňuje skladanie komponent a tak vytvoriť komponentový strom. Vnorenú komponentu je možné pridať na konkrétne miesto metódou `addComponent(component, row, col)`, alebo odstrániť volaním `removeComponent(component)`. Pri pridávaní komponenty je nutné otestovať vložiteľnosť na požadovanú pozíciu, teda či je pre neho dostatok miesta, alebo či sa nebude prekrývať s inou komponentou. Tým by sme mali v aplikácii podporovanú hierarchickú štruktúru a znovupoužiteľnosť vytvorených obvodov.

Tu ale vzniká problém, ako tento komponentový strom zobraziť a ako ho simulovať. Každý modul odkazovaný z komponenty má v sebe uchovanú nejakú konfiguráciu buniek. Ale `GridPanel` je schopný zobraziť práve jednu takúto konfiguráciu. Teda keby sme mu predali koreňový modul, zobrazil by jeho bunky, ale nie bunky vnorených modulov. Taktiež simulátor požaduje plne nakonfigurovanú plochu a s modulmi pracovať nevie. Riešením je exportovanie konfigurácie z vnorených modulov do koreňového modulu. K tomu bude slúžiť metóda `configureCells(module, row, col)`, ktorá nakonfiguruje modul predaný v parametri od zadaného riadku a stĺpcu. Metódu je možné rozšíriť na rekurzívnu verziu, kedy okrem predania vlastnej konfigurácie bude toto volanie propagovať svojim synom. Nakoniec stačí mať prístup ku koreňovému modulu a vždy po zmene syna, alebo pri potrebe získať celú konfiguráciu, vykoná volanie metódy `configureCells(this, 0, 0)`. Komplementárnou funkciou je `unconfigureCells(module, row, col)`, ktorá vymaže svoju konfiguráciu z požadovaného modulu.

`GridPanel` bude vždy zobrazovať *koreňový* modul s prevzatou konfiguráciou synovských modulov. Aby užívateľ mohol pracovať s modulmi, musí ich `GridPanel` odlíšiť od ostatných buniek. Ideálne by bolo zistiť, či bunka patrí nejakej komponente a na základe tejto informácie ju vykresliť inou farbou alebo iným orámovaním. Problémom je, že každá bunka je priradená k nejakej komponente. Riešením môže byť vytvoriť triedu `RootComponent` a objekt tejto triedy bude *koreňovou* komponentou v strome komponent. Metóda `isRoot()` by potom vracala informáciu o tom, či je alebo nie je komponenta koreňová.



Obrázok 5.7 Diagram návrhových tried aplikácie.

6 Implementácia

Táto kapitola sa zaoberá realizáciou simulátoru asynchrónnych obvodov na báze celulárneho automatu. Najskôr je popísaná implementácia simulačnej časti nasledovaná popisom implementácie užívateľského rozhrania so zameraním na detaily spojené s navigáciou, zoomovaním plochy, riadením simulácie, práce s modulmi a podporou histórie zmien. V závere kapitoly sa budeme venovať možnostiam konfigurácie, predovšetkým ako nastaviť prechodové pravidlá a stav buniek.

6.1 Implementácia simulátoru

Celý proces realizácie vychádzal z analýzy uvedenej v kapitole 5.3. Pri implementácii som kládol dôraz na znovupoužiteľnosť kódu a aby prípadné integrovanie do iného systému vyžadovalo minimum úsilia. Preto je simulátor navrhnutý ako jeden balíček bez závislosti na užívateľskom rozhraní a doménovej logiky aplikácie. Jedinou závislosťou je knižnica *jdom-2.0* pre prácu s XML súbormi. Balíček *simulátor* som rozdelil z dôvodu väčšej prehľadnosti na ďalšie balíčky zapuzdrujúce logicky súvisiace triedy. Štruktúra balíčkov je nasledovná:

- *simulator* – pomocné triedy a výnimky
- *simulator.cells* – bunky, sieť buniek a načítavanie a ukladanie stavu buniek
- *simulator.cells.adapter* – rôzne implementácie ukladania buniek
- *simulator.config* – konfigurácia a kontajner služieb
- *simulator.rules* – prechodové pravidla a načítavanie týchto pravidiel
- *simulator.rules.adapter* – rôzne implementácie ukladania pravidiel
- *simulator.scheduling* – plánovanie, `IterableScheduler` a `TimedScheduler`
- *simulator.simulation* – triedy vykonávajúce simuláciu
- *simulator.utils* – pomocné triedy

6.1.1 Vkladanie závislostí

Triedy boli implementované na princípe techniky známej pod názvom *vkladanie závislostí* (angl. Dependency Injection [16]). Vkladanie závislostí (ďalej DI) sa často používa v objektovo orientovanom programovaní na vkladanie závislostí medzi časťami programu tak, aby jedna komponenta mohla používať druhú, bez znalosti o tom, ako sa má požadovaná komponenta vyrobiť. Bez použitia DI, ak náš objekt potrebuje používať služby iného objektu, zodpovedá za celý jeho životný cyklus v čom je zahrnutá aj inicializácia. Pri používaní DI je táto zodpovednosť objektu odobraná, a má ju na starosti Dependency Provider. Náš objekt potom potrebuje len referenciu na dependency providera, slovensky ho môžeme nazvať poskytovateľ závislostí, a provider mu je schopný dodať viacero komponent (záleží aké vie vyrobiť, alebo aké má zaregistrované), ktoré spĺňajú požiadavky nášho objektu. Každá z týchto komponent môže nášmu objektu poskytovať rozdielne služby.

DI zahrňuje minimálne 3 elementy, ktoré medzi sebou musia spolupracovať. *Konzument*, teda objekt požadujúci služby. *Poskytovateľa* týchto služieb mu dodá DI provider, ktorý zodpovedá za celý životný cyklus poskytovateľa služieb. Pod pojmom poskytovateľ služieb si môžeme predstaviť objekt implementujúci rozhranie. Konzument prakticky požaduje objekt určitého

rozhrania. A poslednou časťou DI je práve *DI kontajner*, ktorý môže byť implementovaný viacerými spôsobmi, napr. lokátor služieb, abstraktná továreň, továrenské metódy alebo pomocou nejakého frameworku, napríklad Spring.

DI kontajner v simulátore je implementovaný technikou továrenských metód - vytváranie služieb je definované v triede `Container`. Pred používaním služieb je vhodné inicializovať kontajner volaním metódy `setConfig(Config config)`. Podľa nastavení budú poskladané a pripravené poskytované služby, respektíve objekty. Náš kontajner podporuje nasledovné služby:

```
public Simulator          getSimulator();
public CellsUpdater       getCellsUpdater();
public IScheduler         getScheduler();
public TimedScheduler     getTimedScheduler();
public IterableScheduler  getIterableScheduler();
public RulesLoader        getRulesLoader();
public Rules              getRules();
public CellsLoader        getCellsLoader();
```

V triede `Config` je možné nastaviť:

```
public String rulesPath;                 – cesta k definovaným pravidlám
public String cellsPath;                 – cesta ku konfigurácii buniek
public boolean timedSimulation = true;   – použiť TimedScheduler?
public int period = 100;                 – perióda aktualiázicií
public int iterationsLimit = LIMIT_INFINITY; – maximálny počet iterácií algoritmu
```

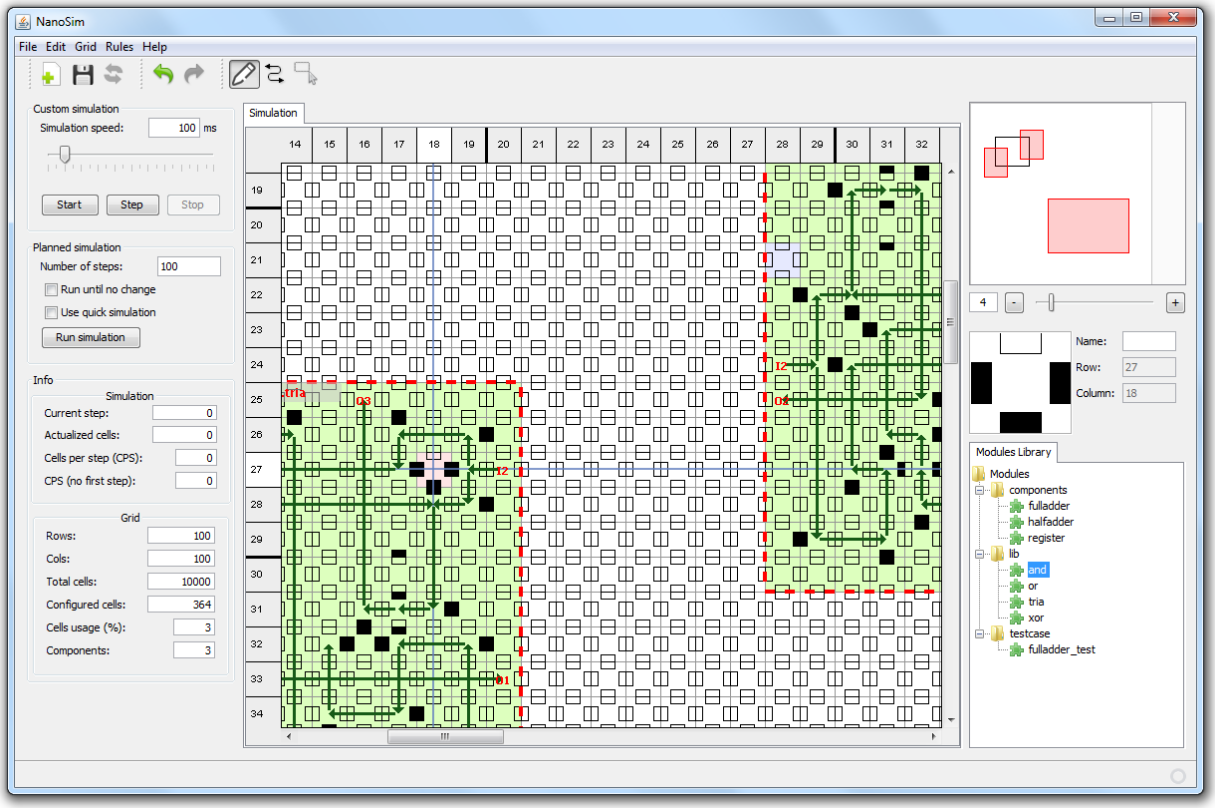
6.2 Implementácia GUI

V rámci implementácie GUI sú zahrnuté tri zložky. Tvorba komponent prezentačnej vrstvy, implementácia aplikačnej domény a riadenie komunikácie medzi nimi. Jednotlivé časti odpovedajú práve architektúre model-view-controller z kapitoly 5.4.1.

Podobne ako simulátor, tak aj GUI bol rozdelený do balíčkov. Triedy sú zabalené podľa logickej súvislosti. Hlavné balíčky sú nasledovné:

- *app* – hlavné triedy aplikácie, `JFrame`, `Application` a iné
- *app.config* – konfigurácia prostredia, uchovanie užívateľských nastavení
- *app.history* – správa histórie zmien pri editácii plochy
- *app.model* – modely
- *app.modules* – implementácia modulárnosti
- *app.modules.loader* – perzistencia modulov, načítavanie a ukladanie do súboru
- *app.cells* – rozšírenie buniek simulátoru
- *app.cells.loader* – načítavanie rozšíreného nastavenia buniek
- *app.view* – vlastné komponenty UI
- *app.view.grid* – `GridPanel` komponenta na prácu s bunkami a pomocné triedy
- *app.view.grid.plugins.** – rozšírenia pre `GridPanel`
- *app.view.simpanel* – triedy na ovládanie simulácie
- *app.view.modulestree* – triedy pre načítanie stromu komponent

Obrázok 6.1 ukazuje finálnu implementáciu užívateľského rozhrania. Funkciu a spôsob realizácie si popíšeme v nasledovných podkapitolách.



Obrázok 6.1: Hlavné okno simulátoru.

6.2.1 Podpora modulov

Štruktúra modulov tvorí veľkú časť doménovej vrstvy (tiež označovanú ako model). Preto si prácu s nimi podrobnejšie popíšeme. V čase spustenia programu sú všetky moduly uložené na disku v XML súboroch. Modul je načítaný zo súboru až v čase keď je potrebný a nie je nikde k dispozícii. Požadovanú činnosť zabezpečuje kooperácia tried `ComponentFactory`, `IComponentLoader` a `ComponentStorage`. Životný cyklus týchto tried je nasledovný:

V prípade, že potrebujeme inštanciu modulu určitého typu (momentálne typom sa nemyslí trieda ale typ modulu, napr.: *lib.tria*, *lib.and*, *lib.xor* a iné), požiadame o neho `ComponentStorage`. Ten zistí, či tento modul, resp. komponenta je už registrovaná. Ďalej budeme hovoriť o komponentách, pretože každá komponenta odkazuje na práve jeden modul a `ComponentStorage` vracia objekty triedy `Component`. Ak komponenta nebola nájdená, `ComponentStorage` požiada `ComponentFactory` volaním `createComponent(String name)` o jej vytvorenie. Avšak samotná fabrika (objekt triedy `ComponentFactory` zabezpečujúci vyrobenie komponent) pracovať priamo so súbormi nevie. Na to slúžia triedy implementujúce rozhranie `IComponentLoader`. V našej aplikácii je implementovaný `XmlComponentLoader` pracujúci s XML súbormi, avšak je možné implementovať aj inú triedu, napríklad pracujúcu s databázou. `XmlComponentLoader` má pomocou metódy `setComponentsDir()` nastavený adresár, kde sú uložené XML konfigurácie. Takže napríklad fabrika predá žiadosť o vytvorenie komponenty *lib.and* objektu triedy `XmlComponentLoader`. Ten pomocou kódu:

```
componentsDir + "/" + name.replace(".", "/") + ".xml"
```

prevedie názov na cestu ku komponente (*[componentsDir]/lib/and.xml*), preto je nutné aby požadovaný názov bol synchrónny s fyzickým umiestnením na disku. V tomto riešení nie je možné presúvanie komponent vo fyzickom umiestnení. `XmlComponentLoader` na základe konfigurácie XML vytvorí objekt triedy `Component`.

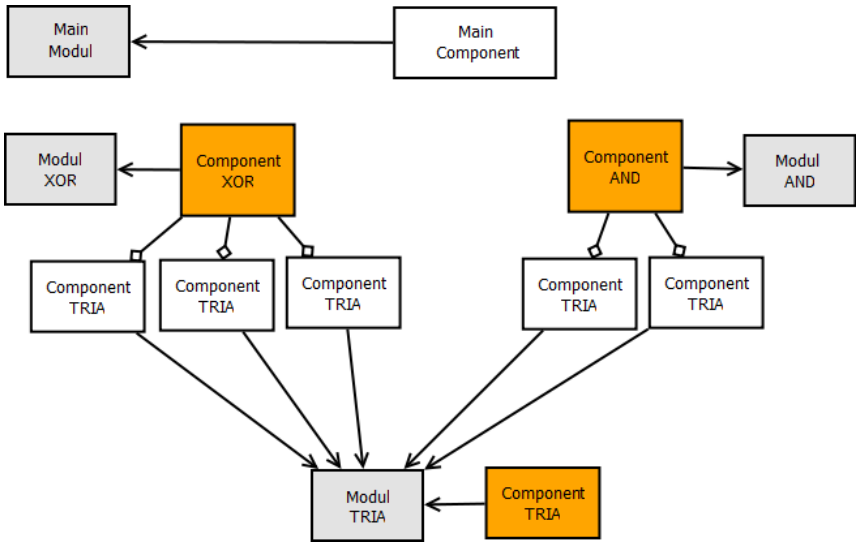
Na obrázku 6.2 vidíme, že konfiguračný súbor môže obsahovať aj vnorené komponenty, definované názvom a pozíciou bunky, od ktorej bude modul vložený. `XmlComponentLoader` je zodpovedný za zostavenie celej komponenty vrátane načítania vnorených komponent. Aby nedochádzalo k opätovnému načítaniu už existujúcich komponent, `XmlComponentLoader` nemôže rekurzívne tvoriť synovské komponenty. Tým by sme nedosiahli takmer žiadnu optimalizáciu. Preto mu je predaná inštancia triedy `ComponentStorage`, ktorú požiada v prípade potreby na vytvorenie ďalšej komponenty. `ComponentStorage` požadovanú komponentu už môže mať pripravenú, v tom prípade vracia jej referenciu, alebo znova požiada `IComponentLoader` prostredníctvom `ComponentFactory` o jej vytvorenie. Ak je komponenta kompletne vytvorená, je vrátená do triedy `ComponentStorage`. Tým je životný cyklus načítania komponenty skončený.

```
<?xml version="1.0" encoding="UTF-8"?>
<component>
    <components>
        <component>
            <name>lib.tria</name>
            <position row="11" col="6" />
        </component>
        <component>
            <name>lib.tria</name>
            <position row="9" col="27" />
        </component>
    </components>
    <module>
        <title>AND</title>
        <description>AND</description>
        <cells rows="30" cols="45">
            <cell row="0" col="32">0010</cell>
```

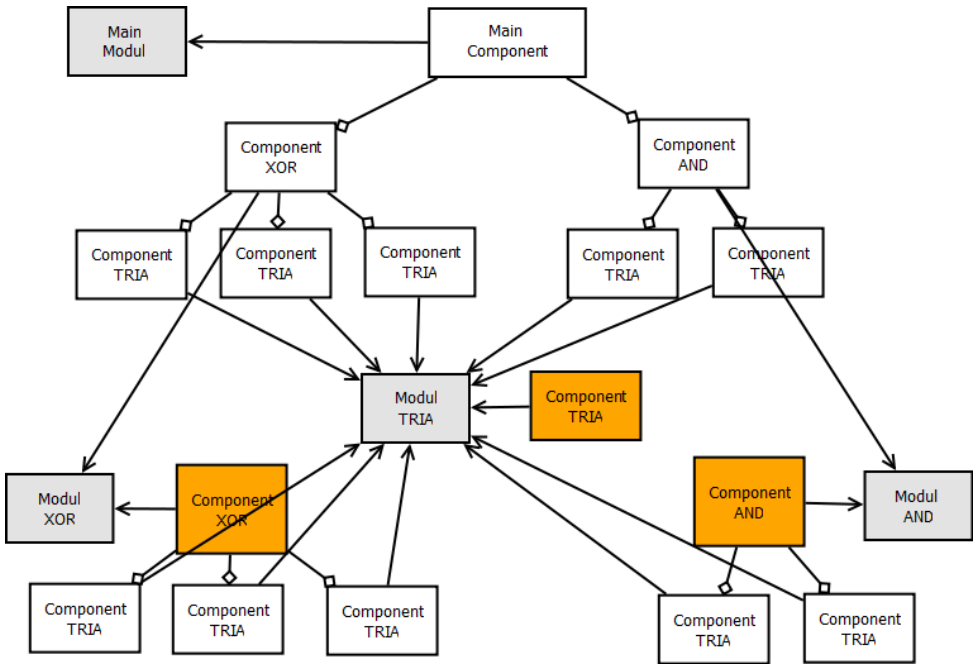
Obrázok 6.2: Príklad súboru XML s definovaním vnorených komponent.

Zoznámili sme sa s procesom načítavania komponent. Teraz nasleduje technika tvorby stromu komponent. Najskôr si ale musíme definovať spôsob prepojenia uzlov v strome. Každá komponenta, teda uzol môže mať ľubovoľný počet synov. Kvôli prechodu stromom je synovi predaná referencia na rodiča. Koreňový uzol nemá žiadneho rodiča. Každá komponenta je prepojená s nejakým modulom.

Predstavme si situáciu, že chceme pridať komponentu XOR a AND k hlavnej (Main) komponente (tento stav ukazuje obrázok 6.3). Triviálnym riešením by bolo priame pridanie týchto komponent do Main s tým, že by sa každej nastavil rodič. Problém by nastal, ak by sme chceli XOR pridať do iného stromu. Jeho rodič by sa zmenil a tým by sa porušila konzistencia stromu. Z toho dôvodu bolo pridávanie komponent realizované kopírovaním stromu komponenty a teda celý podstrom je vybudovaný odznovu. Aby nebola porušená konzistencia stromu, pri kopírovaní sa nastavujú referencie na moduly a nových rodičov. Pridanú komponentu a kopírovaný podstrom je možné vidieť na obrázku 6.4.



Obrázok 6.3: Komponenty AND, XOR a TRIA bez pripojenia do koreňovej komponenty. Šedou farbou sú označené moduly, oranžová farba predstavuje moduly uložené v `ComponentStorage`.



Obrázok 6.4: Strom komponent po pripojení komponent AND a XOR.

6.2.2 Grid panel

Grid panel bol navrhnutý tak, aby práca s bunkami a komponentami bola čo najjednoduchšia a intuitívna. Pretože požiadaviek na neho stále pribúdalo, rozhodol som sa ho implementovať s možnosťou rozširovania pomocou rozšírení (angl. Plugins). Panel sám o sebe je celkom jednoduchý. Hlavnými funkciami sú renderovanie buniek, vykreslenie postranného pravítka na zobrazenie pozície buniek v riadku a stĺpci a zobrazenie "terču" ukazujúci pozíciu myši.

Ostatné funkcie sú implementované na strane pluginov, resp. rozšírení. Panel umožňuje pridávať a odoberať rozšírenia pomocou metód `installPlugin(IGridPlugin plugin)` a `uninstallPlugin(IGridPlugin plugin)`. Rozšírení môže byť inštalovaných viacero, niektoré sa ale môžu medzi sebou biť, preto existuje metóda `setActivePlugin(String name)`, ktorá nastavuje práve jedno aktívne rozšírenie, ostatné sú deaktivované. Rozšírenia využívajú návrhový vzor pozorovateľ (angl. Observer Pattern). Pri inštalovaní alebo aktivácii sú volané metódy `install(GridPanel panel, CellController controller)` a `uninstall(GridPanel panel, CellController controller)`, v ktorých si môže registrovať poslucháča (angl. Listener). Všetky zmeny nad dátami sú realizované prostredníctvom radiča. Grid panel informuje registrované rozšírenia iba o udalostiach spojené s renderovaním nasledovnými metódami:

- `beforePaintComponent` - volané na začiatku vykresľovania
- `afterPaintComponent` - volané na konci vykresľovania
- `beforePaintCells` - volané pred vykreslením buniek
- `afterPaintCells` - volané po vykreslení buniek

Grid panel môže registrovať ľubovoľné rozšírenie implementujúce rozhranie `IGridPlugin`. Ako základné rozšírenia pre prácu so simuláciou a tvorby vlastnej simulácie boli vytvorené nasledovné rozšírenia:

`ZoomGridPlugin` umožňuje po stlačení medzerníka a skrolovaním kolieska na myši generovať udalosť oznamujúcu zmenu priblíženia. Udalosť je v aplikácii pomocou objektu triedy `ZoomListener` zachytená a patrične spracovaná.

`CellnameGridPlugin` zobrazuje názvy buniek. Názvy sú zobrazené až na konci procesu vykresľovania, aby neboli ničím prekryté.

`GragGridPlugin` je užitočné rozšírenie pri pohybe po mriežke. Stlačením medzerníka a pohybom premiestňuje viditeľnú časť buniek. Tento efekt je tiež možné docieliť stlačením pravého tlačidla na myši.

`ComponentsViewGridPlugin` slúži na zobrazenie komponent. Komponenty sú podfarbené a orámované aby ich bolo lepšie vidieť. V triede je možné nastavovať atribúty ovplyvňujúce spôsob vykresľovania. Atribút `colorizeComponent` umožňuje vypnúť podfarbovanie komponent a atribút `deep` nastavuje úroveň zanorenia pokiaľ sa majú komponenty zobrazovať.

Uvedené rozšírenia pracujú počas celého behu programu. Nie je tomu tak ale u ostatných rozšírení. Tie sú totiž na ovládanie pomerne náročné a ich paralelný beh by mohol zmiasť užívateľa. Hovoríme o týchto rozšíreniach:

`SelectGridPlugin` umožňuje označovať ťahom myši skupinu buniek, teda vytvárať *selekcie*. So selekciou je možné pracovať aj v ostatných rozšíreniach. `SelectGridPlugin` obsahuje niekoľko metód na programové ovládanie a prácu so selekciou, takže vytvorenie selekcie a označenie buniek môže byť realizované programátorom. Viac o tom ako sa pracuje so selekciami bude v užívateľskej príručke.

Jednoduchým rozšírením je `EditGridPlugin`, ktorý zachytí udalosť kliknutia na bunku a predá ju radičovi, ktorý zmení stav bunky. Táto činnosť bola realizovaná formou rozšírenia práve preto, aby sa nebila s možnosťami selekcie a šípok.

Posledným implementovaným rozšírením je `ArrowsGridPlugin`. Umožňuje kreslenie šípok. Šípka nemá žiadnu špeciálnu úlohu, slúži pre užívateľa ako informácia o toku signálu.

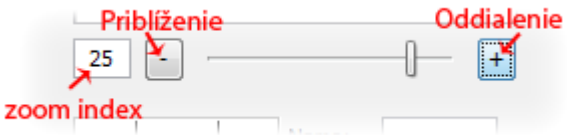
6.2.3 Navigácia

Navigácia dáva užívateľovi prehľad o rozmiestnení modulov v rámci celej konfigurácie. Ako vidíme na obrázku 6.1, veľkosť navigátora je fixná a preto je treba prepočítať rozmery plochy tak, aby čo najlepšie vyplnila tento priestor. To sa deje v metóde `setGridPanelArea`, ktorá sa musí volať vždy po zmene veľkosti mriežky. Navigátor je prepojený s komponentou `GridPanel` a po kliknutí alebo ťahaním myšou v navigačnom panely navigátor nastavuje zobrazovanú časť buniek, čo umožňuje rýchle presúvanie zobrazenej oblasti.

Náhľad sa automaticky aktualizuje vždy po pridaní alebo odobratí modulu. Ak nastane nejaká zmena v stromovej štruktúre modulov, je táto udalosť propagovaná až do koreňového modulu. Aby navigátor vedel o týchto udalostiach, registruje si `ComponentListener` do koreňového modulu získaného z objektu `gridModel`. Podobným spôsobom sú propagované aj zmeny polohy modulu.

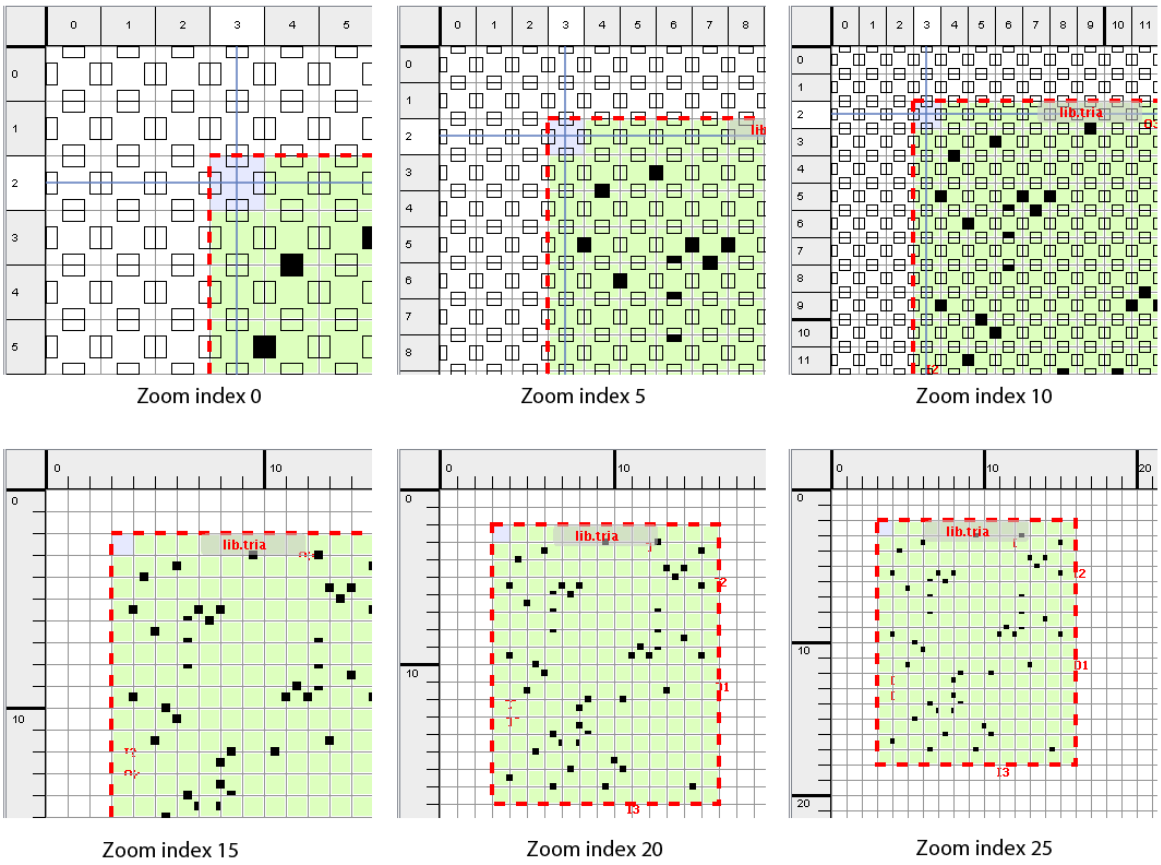
6.2.4 Zoomovanie

Navrhovanie obvodov bez možnosti zmeny úrovne priblíženia je nepraktické. Možností na realizáciu boli viaceré. Jednou z nich bolo ukladanie vyrenderovanej plochy do buffrovaného obrázku a ten aplikovaním transformácie zobraziť zmenšený. Táto technika sa ukázala pri väčšom počte buniek pomalá, bolo nutné vymyslieť iný spôsob. Riešením je vytvoriť triedu `CellDimension`, kde sú definované rozmery bunky a jej pamäti. Volaním metódy `setScale(double scale)` sa prepočítajú rozmery bunky. `CellDimension` si uchováva kópiu pôvodných rozmerov, čiže napríklad volaním `setScale(2.0)` bude bunka dvakrát zmenšená, následným volaním `setScale(1.0)` bude mať pôvodné rozmery. Z obrázku 6.5 vidíme, že posuvník nastavuje celé čísla z intervalu *<0, 30>*, preto je nutné túto hodnotu previesť do vhodnejšieho intervalu (napr.: *1.0 + zoomIndex/10*).



Obrázok 6.5: Zoom panel na zmenu priblíženia.

Na obrázku 6.6 je zobrazený priebeh odďaľovania. V prípade, ak *zoom index* dosiahne hodnotu 10 sa plocha stáva mierne neprehľadnou. Preto `GridPanel` podľa úrovne priblíženia prispôsobuje renderovanie buniek. Taktiež sa prispôsobuje postranné pravítko.



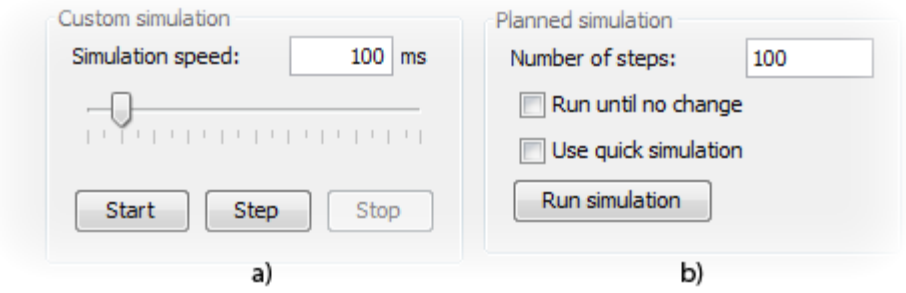
Obrázok 6.6: Rôzne úrovne oddialenia plochy.

Maximálna hĺbka priblíženia je momentálne nastavená na hodnotu 30. Túto hodnotu je aplikácia schopná vhodne priblížiť. Obecne `GridPanel` je schopný zobraziť ľubovoľne veľký *zoom index*, ale výsledok by nemusel byť pre užívateľa prehľadný.

6.2.5 Riadenie simulácie

Simuláciu je možné spúšťať dvomi spôsobmi, s *nedefinovaným* alebo *preddefinovaným* ukončením. Nedefinovaným ukončením je myslená simulácia bežiaca donekonečna. Ukončená môže byť jedine stlačením tlačidla Stop (Obrázok 6.7a). Simulácia sa bude vykonávať podľa nastavenej rýchlosti. Pri zmene hodnoty rýchlosti je táto zmena zachytená a poslaná plánovačovi simulácie. Taktiež je možná editácia buniek počas prebiehajúcej simulácie.

Preddefinované ukončenie sa nastavuje v panely zobrazenom na obrázku 6.7b. Tu je možné nastaviť rýchlu simuláciu, kedy priebeh simulácie nebude vizualizovaný. Na plánovanie sa použije `IterableScheduler` uvedený v kapitole 5.3.4. Ďalším nastavením je "Run until no change". Ak je zapnuté, potom sa ignoruje zvolený počet krokov a simulácia bude prebiehať dokým sú bunky zmenené.

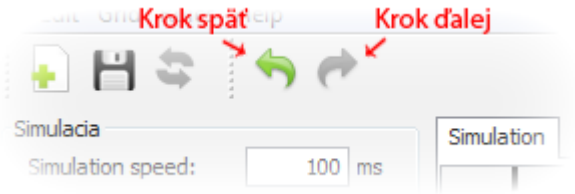


Obrázok 6.7: Panely na riadenie simulácie, a) panel na ovládanie rýchlosti simulácie, b) panel na odsimulovanie prednastavených počet krokov.

6.2.6 História zmien

Po každej zmene konfigurácie plochy, buď pridaním komponenty alebo zmenou nejakej bunky sa stav hlavného modulu ukladá do histórie zmien. Počet zapamätaných krokov je nastavený na 50. Užívateľ sa môže počas vytvárania simulácie v prípade omylu vrátiť späť. Zmeny sa zaznamenávajú len ak je simulácia vypnutá. V priebehu simulácie sú všetky zásahy užívateľa do celulárnej plochy ignorované. Stav modulu sa zapamätá i po skončení spustenej simulácie. Vrátením späť je možné obnoviť stav modulu pred zahájením simulácie. Obrázok 6.8 ukazuje tlačidlá na zmenu v histórii.

O ukladanie a načítavanie sa stará trieda `HistoryManager`. Manager nepracuje priamo s hlavným modulom, ale s objektmi triedy `HistoryRecord` implementujúci rozhrania `IDataExportable` a `IDataImportable`. Pretože ukladanie stavu celej hlavnej komponenty by bolo pamäťovo náročné, boli v triedach `Component` a `Module` implementované metódy `exportData(IDataExportable data)` a `importData (IDataImportable data)`, ktoré exportujú minimálne množstvo dát. Manager si sleduje maximálnu kapacitu záznamov a po prekročení limitu je prvý záznam odstránený.



Obrázok 6.8: Ovládanie histórie zmien pri úpravách.

6.3 Konfigurácia

Aplikácia bola navrhnutá tak, aby vedela spracovať rôzne konfiguračné formáty. Pri spúšťaní simulátoru bez grafického rozhrania, kde sa nevyžadujú pokročilé nastavenia, sú preferované konfigurácie v súbore *TXT*. Tieto nastavenia je samozrejme možné spracovať aj v grafickom režime, ale nebude v nich dostatok informácii pre užívateľa. Pri spustení v grafickom prostredí sa používajú *XML* konfigurácie. Nastavovať je možné prechodové pravidlá, stavy buniek a moduly.

6.3.1 Pravidlá

Pravidlá sa definujú v XML dokumente. Všetky pravidlá sú zabalené v značke `rules`. Pravidlo je definované v značke `rule`. Pravidlo obsahuje tri povinné elementy:

- `name` - pomenovanie pravidla
- `crnt` - zakódovaná ľavá strana pravidla
- `next` - zakódovaná pravá strana pravidla

Na obrázku 6.9 vidíme príklad definovanie propagačného pravidla.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rules>
    <rule>
        <name>Propagation rule (1)</name>
        <crnt>00010000</crnt>
        <next>00000100</next>
    </rule>
</rules>
```

Obrázok 6.9: XML Konfigurácia prechodových pravidiel.

6.3.2 Bunky

Bunky je možné definovať *xml* ale aj v *textovom* formáte. Textový formát môže byť vhodnejší pri prepojení s iným programom, je jednoduchší na zápis pri manuálnom nastavovaní. Príklad definície buniek je možné vidieť na obrázku 6.10 vľavo. Na prvých dvoch riadkoch je definovaný rozmer mriežky. potom nasleduje výpis buniek. Bunky sa zapisujú nasledovne: *RIADOK,STLPEC:STAV_BUNKY*. Jednotlivé bunky sú oddelené bodkočiarkou. Nie je nutné uvádzať bunky so stavom 0000, je to východzí stav.

Xml konfigurácia je zobrazená na obrázku 6.10 vpravo. Hlavným elementom je `cells` s povinnými atribútmi `rows` a `cols` udávajúce rozmer plochy. `Cells` môže obsahovať ľubovoľný počet elementov `cell` s povinnými atribútmi `row` a `col` určujúce polohu bunky. Hodnotou elementu je stav bunky zadávaný podľa uloženia v dátovom type *integer*. Technika, akou sa mapuje integer na stav bunky bola vysvetlená v kapitole 5.3.1.

```
rows:10
cols:10
cells:
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;0,0:0010;
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<cells rows="10" cols="10">
    <cell row="1" col="0">0100</cell>
    <cell row="1" col="1">0001</cell>
    <cell row="2" col="9">0010</cell>
    <cell row="3" col="9">1000</cell>
    <cell row="4" col="10">0010</cell>
    <cell row="4" col="13">0010</cell>
    <cell row="4" col="15">0100</cell>
    <cell row="4" col="16">0001</cell>
    <cell row="5" col="8">0100</cell>
</cells>
```

Obrázok 6.10: Konfigurácia buniek, textový a XML formát.

6.3.3 Komponenty

Nastavenie komponenty zahŕňa okrem definície buniek aj ďalšie dáta, ktoré je možné vizualizovať v GUI. Hlavným elementom je `component`, ktorý obsahuje povinný element `module` a voliteľný

`components`. `Components` sa používa pre vloženie vnorených komponent, ktorých nastavenie sa skladá z 2 značiek. `Name` je názov požadovanej komponenty, podľa ktorého bude nájdená a vytvorená komponenta. Ďalšou značkou je `position`, ktorá udáva pozíciu vloženia komponenty.

Modul je zložený z povinných elementov `title`, `description` a `cells`. `Cells` sme si už definovali v predchádzajúcej podkapitole a jeho význam je rovnaký. Jedinou zmenou je podpora atribútu `name`, ktorý nastavuje názov bunky. `Title` a `description` sú informácie o komponente, ktoré môžu byť zobrazené užívateľovi po otvorení modulu. Voliteľnou položkou je `arrows`. V `arrows` môžu byť umiestnené elementy `arrow` definujúce šípku. Pre zobrazenie šípky je nutné vedieť zdrojovú a cieľovú bunku. Tie sú definované práve povinnými atribútmi `fromRow`, `fromCol`, `toRow` a `toCol`. Príklad konfigurácie modulu vidíme na obrázku 6.11.

```
<?xml version="1.0" encoding="UTF-8"?>
<component>
    <components>
        <component>
            <name>lib.tria</name>
            <position row="11" col="6" />
        </component>
    </components>
    <module>
        <title>AND</title>
        <description>Logicka komponenta</description>
        <cells rows="30" cols="45">
            <cell row="0" col="32">0010</cell>
            <cell row="1" col="0">0100</cell>
            <cell row="25" col="44" name="c1">0000</cell>
        </cells>
        <arrows>
            <arrow fromRow="8" fromCol="0" toRow="8" toCol="4" />
            <arrow fromRow="8" fromCol="4" toRow="7" toCol="4" />
        </arrows>
    </module>
</component>
```

Obrázok 6.11: XML Konfigurácia komponenty.

7 Testovanie a experimenty

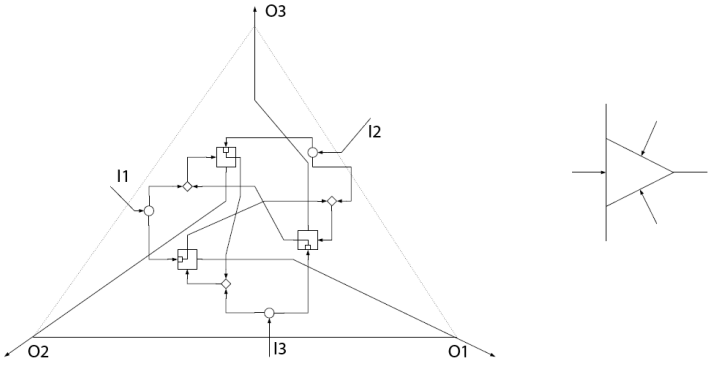
Pretože nikde nebol dostupný program simulujúci obvody odolne voči oneskoreniu na celulárnych poliach, proces testovania a experimentovania so simulátorom bol dôležitý. Pri testovaní sa zameriame na overenie funkčnosti simulátora. Na začiatku kapitoly budú podrobnejšie popísané experimenty s komponentou *Tria*. Potom budú nasledovať základne logické obvody, z ktorých je možné realizovať počítače. Posledným simulovaným obvodom bude úplná jednobitová sčítačka demonštrujúca schopnosť simulovať i netriviálne obvody z pohľadu implementácie v celulárnych poliach. V závere kapitoly bude uvedené zhrnutie celého procesu simulácie, ale aj návrh obvodov odolných voči oneskoreniu.

7.1 Testovanie funkčnosti simulátoru

Funkčnosť simulátoru bola testovaná na simuláciou logických obvodov. Cieľom testovania bolo ukázať, že asynchrónne obvody implementované pomocou technológie celulárnych polí sú schopné logicky rovnakého výpočtu ako ekvivalentné synchrónne logické obvody založené na tranzistorovej logike. Experimenty boli prevažne zamerané na testovanie správnej funkčnosti obvodov, teda či obvod na požadovaný vstup reaguje očakávaným výstupom. Simulované obvody (okrem *Tria* modulu) využívali *dvojdrôtové* (angl. Dual Rail) kódovanie vysvetlené v kapitole 4.5. Preto každý vstup a výstup sa skladá z dvoch ciest označené indexmi 1 a 0, logickú hodnotu je možné odvodiť podľa dvojdrôtového kódovania.

7.1.1 Tria

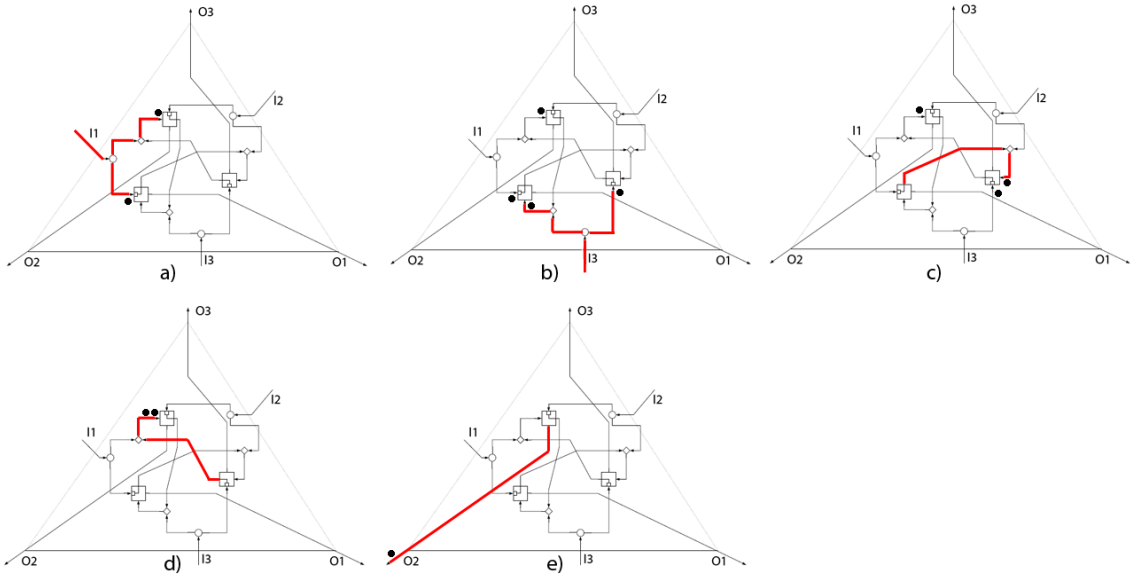
Tria je základným obvodom používaný vo všetkých asynchrónnych obvodoch práve kvôli jej synchronizačným vlastnostiam, bez ktorých by nebolo možné počítať na celulárnych poliach. Inak povedané, modul *Tria* vykonáva lokálnu synchronizáciu v asynchrónnych obvodoch. Schéma na obrázku 7.1 ukazuje príchod signálov na jednotlivé obvody a následne generovanie signálu na príslušnú výstupnú cestu. Realizácia na celulárnych poliach je na obrázku 7.4.



Obrázok 7.1: Schéma modulu Tria [2].

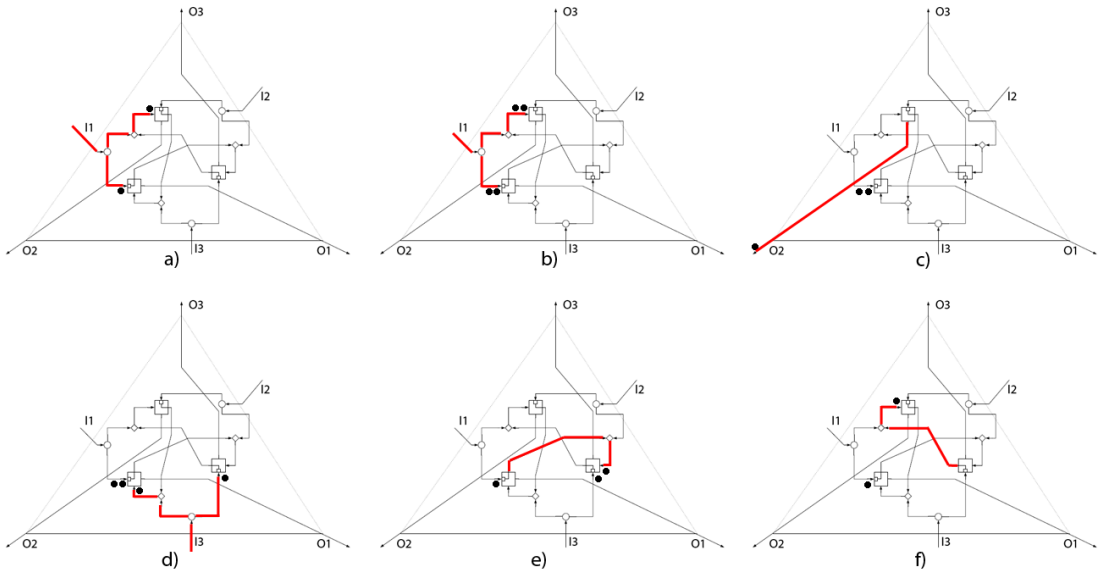
Princíp činnosti modulu ukazuje obrázok 7.2. Signál prichádza na vstup I_1 (obr. 7.2a), kde je rozdelený a zastavený prvkom *R-Counter*. V tomto stave sa nachádza až dokým na jeden z prvkov *R-Counter* nepríde ďalší signál. Uvažujme najskôr správne chovanie, kedy ďalší vstupný signál môže prísť na vstup I_3 alebo I_2. Dajme tomu príde signál na I_3 (7.2b). Prvok *R-Counter* zareaguje na

vstup a generuje signál zaslaný k druhému *R-Counteru* (obr. 7.2.c), čo spúšťa podobnú reakciu a signál je zaslaný na tretí *R-Counter* (obr. 7.2d). Príchod druhého signálu na vstup *a* prvku *R-Counter* spôsobí generovanie signálu na výstup *b*, teda signál sa dostáva na výstup *O2* a modul *Tria* sa nachádza v pôvodnom stave. Celý proces sa podobným spôsobom zopakuje pre ľubovoľnú kombináciu dvoch rôznych vstupných signálov.

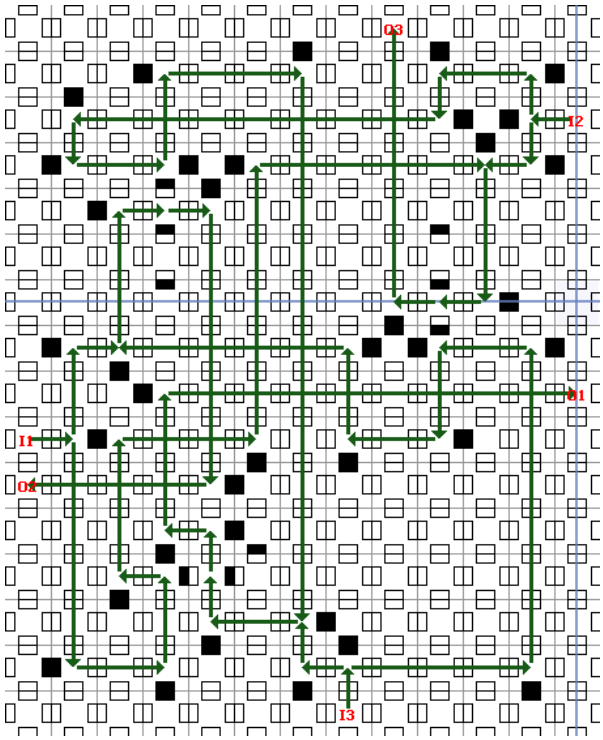


Obrázok 7.2 Správne chovanie modulu Tria,
čierna bodka ukazuje miesto pokiaľ bolo možné propagovať signál,
červená čiara ukazuje cestu šírenia signálu (táto konvencia platí aj pre ostatné obrázky).

Problém nastane, keď signál príde opakovane viackrát na jednu vstupnú cestu (obr. 7.3). Dochádza k okamžitému generovaniu výstupu bez čakania na druhý vstup. Stav do ktorého sa obvod dostane nazývame *nekonzistentný*, pretože by za správneho fungovania nemal nikdy nastať. Modul má šancu príchodom signálu na vstup *I3* sa dostať opäť do *konzistentného* stavu (obr. 7.3d-f), avšak ak by medzi tým prišli na ďalšie vstupy (*I1*, *I2*) nové signály, modul by pravdepodobne fungoval nesprávne. Uvedená situácia by pri správnom návrhu obvodu a za predpokladu nevzniknutia poruchy nemala nikdy nastať.



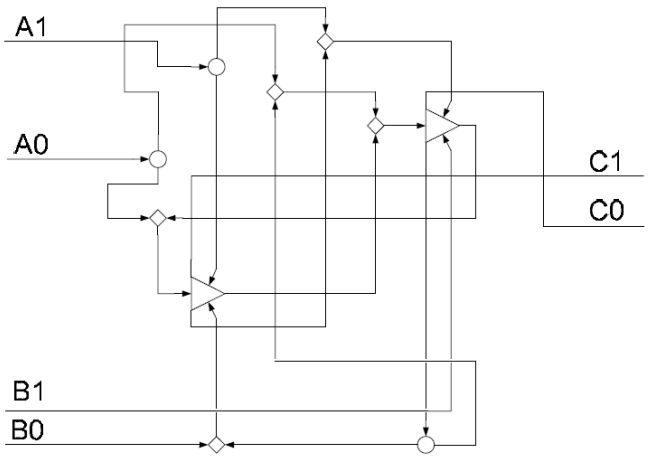
Obrázok 7.3: Chybné chovanie modulu Tria.



Obrázok 7.4: Rozloženie modulu Tria v celulárnom poli.

7.1.2 AND, OR, NAND, NOR

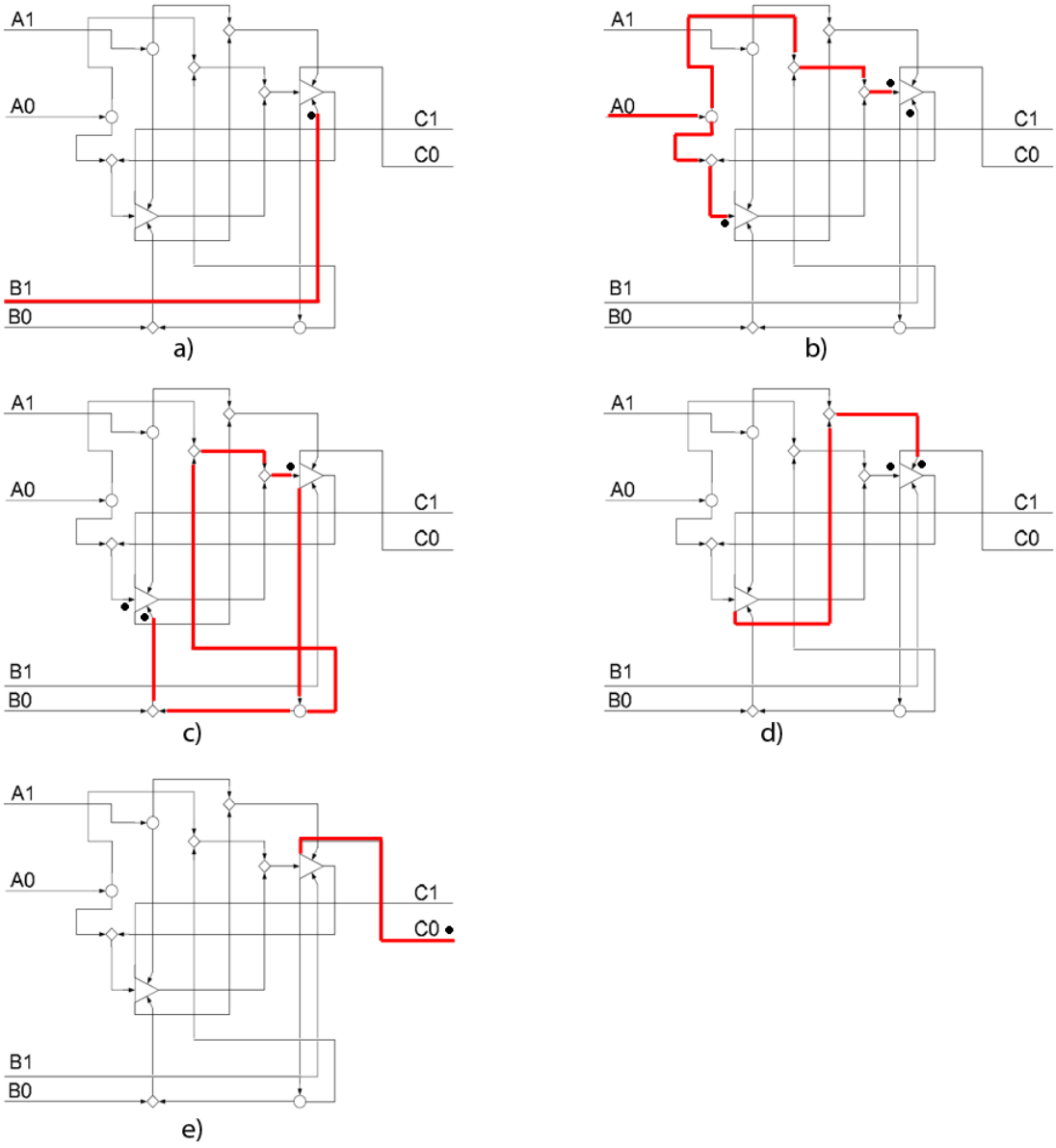
Prvým testovaným logickým obvodom bolo hradlo AND zobrazené na obrázku 7.5. Rozloženie v celulárnom poli je zobrazené na obrázku 7.7. Písmená *A* a *B* označujú vstupy, *C* je výstup. Po aplikovaní všetkých vstupov sa ukázalo, že obvod pracuje podľa očakávaní. Testované kombinácie s informáciou o kódovanej logickej hodnote sú zobrazené v tabuľke 7.1.



Obrázok 7.5: Asynchrónne hradlo AND [2].

Aby sa signál objavil na výstupe, je nutné, aby na vstup prišli obe kódové informácie. Táto vlastnosť sa v asynchrónnych obvodoch nazýva lokálna synchronizácia. V prípade oneskorenia jedného zo vstupov, obvod čaká na jeho príchod. Po príchode oneskoreného signálu sa preklopia

stavy synchronizačných prvkov *Tria*, signál sa dostane na výstup a obvod zostane vo východzom stave. Pre lepšiu predstavu na obrázku 7.6 je ukázané ako k takejto situácii môže dôjsť.



Obrázok 7.6: Šírenie signálu v obvode AND.

Čierna bodka ukazuje miesto pokiaľ bolo možné propagovať signál. Vidíme, že čierna bodka je vždy umiestnená pri prvku *Tria*, čo je dané jej synchronizačnými vlastnosťami. Červená čiara ukazuje cestu šírenia signálu. Okrem prvku *Tria* sú použité aj *Merge* a *Split*, ktoré ako vidíme buď spájajú alebo rozdeľujú signál. Podobnou analýzou je možné overiť všetky ďalšie možnosti.

Zaujímavým prípadom je však situácia, kedy pred príchodom oneskoreného signálu, povedzme na vstup *A*, by prišiel znova signál na vstup *B*. Takéto chovanie by v praxi nemalo nikdy nastať, avšak nejaká chyba by to mohla spôsobiť. Je zrejmé, že reakcia obvodu pri takejto udalosti je silne závislá od prvku *Tria*. Pretože tento projekt je zameraný na realizáciu simulátoru a otestovanie jeho funkčnosti, *nebudeme* sa podobným analýzam podrobnejšie venovať. Hlbšie analýzy prvkov *Tria* a *AND* v spojitosti s testovateľnosťou obvodov je možné nájsť v [16].

A1	A0	**A**	B1	B0	**B**	C1	C0	**C**
0	1	**0**	0	1	**0**	0	1	**0**
0	1	**0**	1	0	**1**	0	1	**0**
1	0	**1**	0	1	**0**	0	1	**0**
1	0	**1**	1	0	**1**	1	0	**1**

Tabuľka 7.1: Testované hodnoty pre hradlo AND, zvýraznené sú logické hodnoty.

Vďaka používaniu dvojdrôtového kódovania môžeme zmenou interpretácie významu indexovania bitov na vstupných alebo výstupných signáloch zmeniť chovanie obvodu *AND*. Prehodením indexovania výstupného signálu *C*, teda *C1* by sme používali ako *C0* a opačne *C0* ako *C1* by sme v skutočnosti dosiahli negáciu, teda obvod by fungoval ako *NAND*.

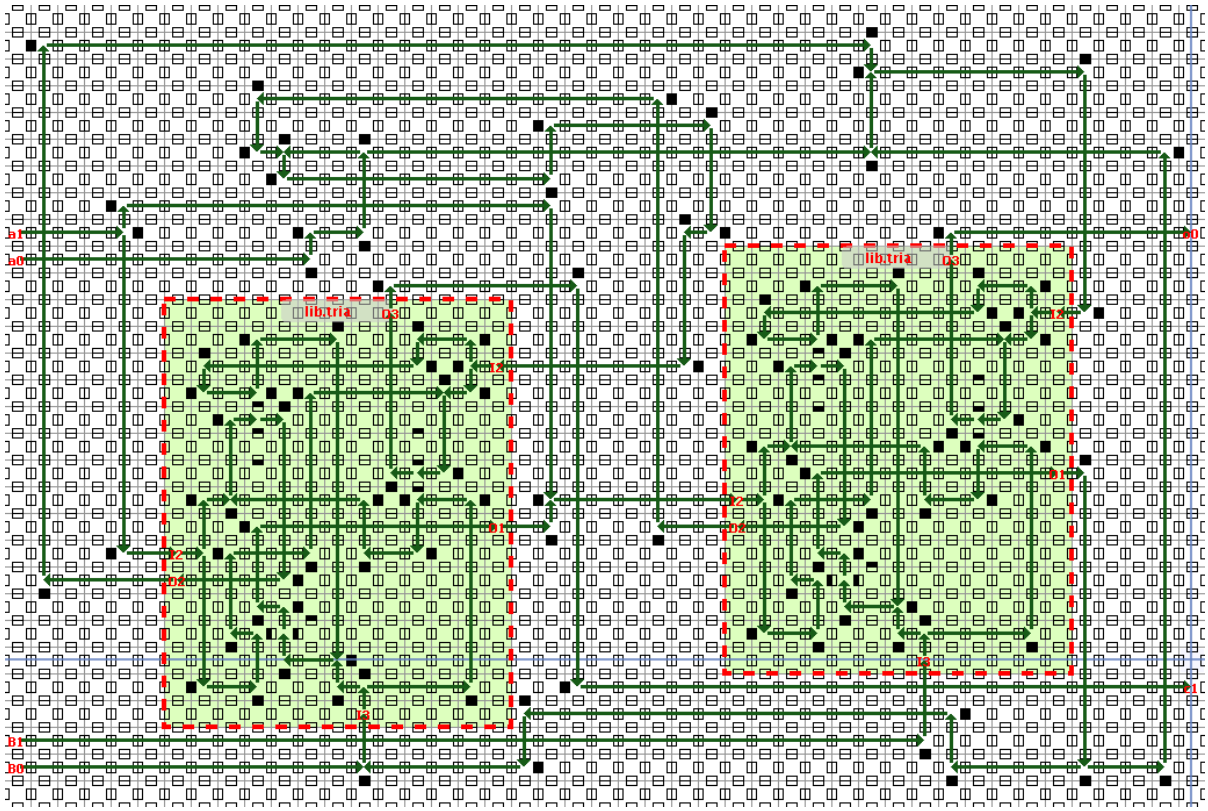
Prehodením indexov na vstupných a výstupných signáloch bude navrhnutý obvod fungovať ako *OR*. Tento prípad je vidno v tabuľke 7.2, kde hodnoty v stĺpcoch *Ax, Bx* a *Cx* ostali rovnaké, jediné čo sa zmenilo sú indexy signálov. Vidíme, že logické hodnoty odpovedajú logickému prvku *OR*. Prehodením indexácie vstupných signálov a ponechaním *C* nezmeneného v hradle *AND* realizujeme hradlo *NOR*. Tento prípad zobrazuje tabuľka 7.3.

A0	A1	**A**	B0	B1	**B**	C0	C1	**C**
0	1	**1**	0	1	**1**	0	1	**1**
0	1	**1**	1	0	**0**	0	1	**1**
1	0	**0**	0	1	**1**	0	1	**1**
1	0	**0**	1	0	**0**	1	0	**0**

Tabuľka 7.2: Testované hodnoty pre hradlo OR voči hradlu AND, prehodená indexácia vstupov A, B a výstupu C.

A0	A1	**A**	B0	B1	**B**	C1	C0	**C**
0	1	**1**	0	1	**1**	0	1	**0**
0	1	**1**	1	0	**0**	0	1	**0**
1	0	**0**	0	1	**1**	0	1	**0**
1	0	**0**	1	0	**0**	1	0	**1**

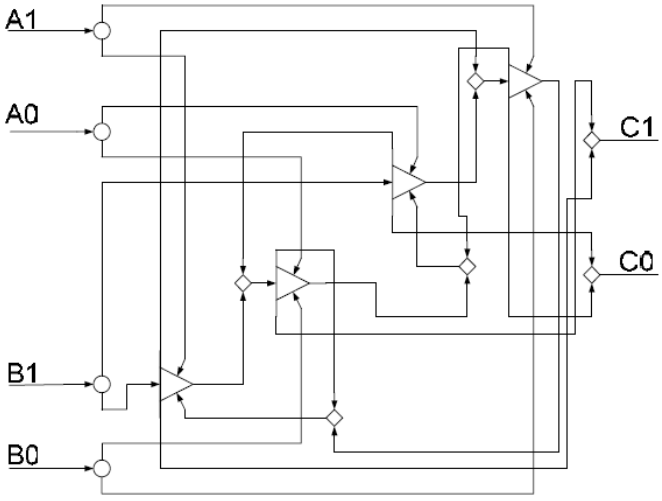
Tabuľka 7.2: Testované hodnoty pre hradlo NOR, prehodená indexácia vstupov A a B.



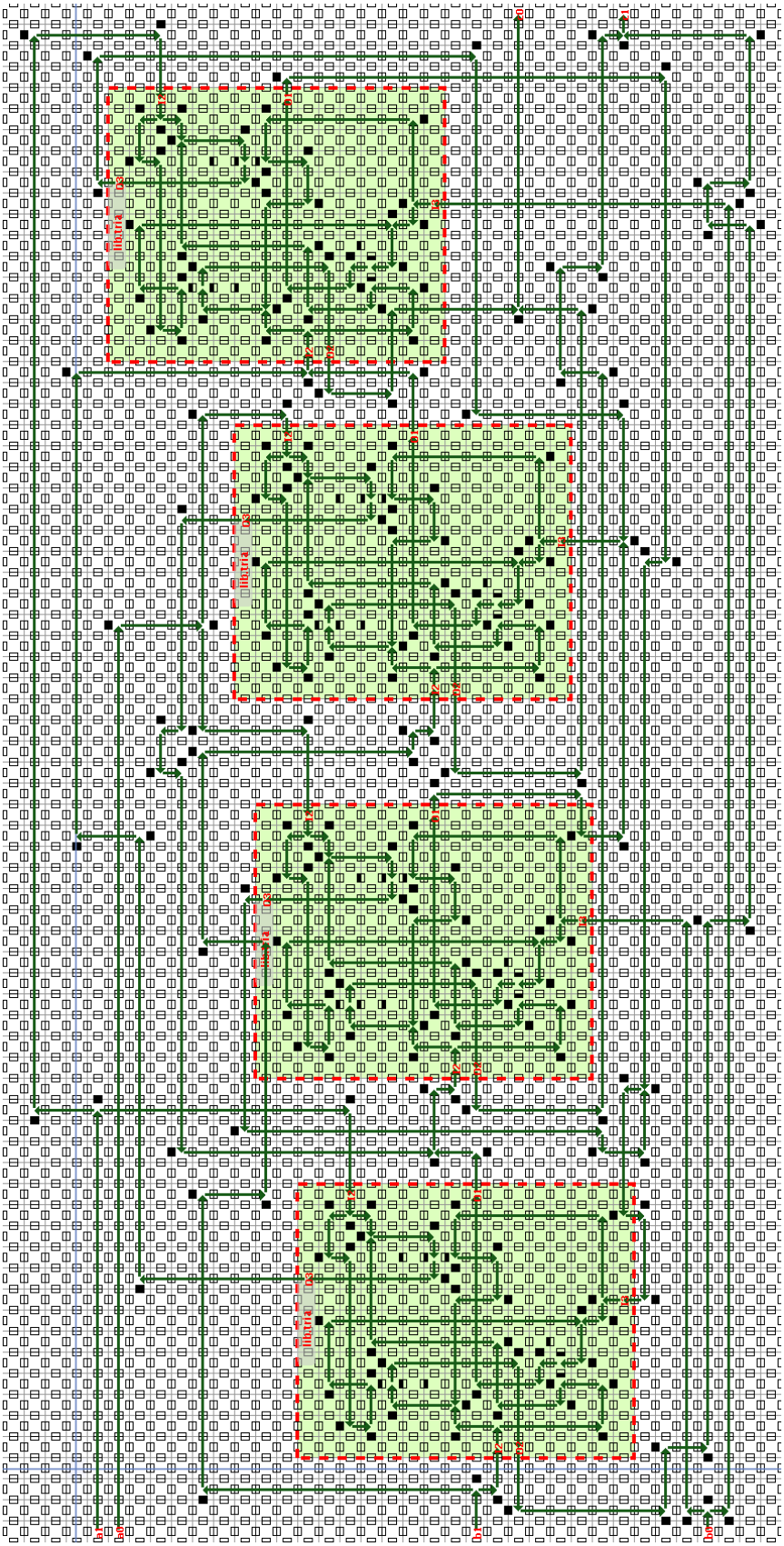
Obrázok 7.7: Rozloženie asynchrónneho hradla AND v celulárnom poli.

7.1.3 XOR

Ďalším testovaným logickým obvodom bol XOR. Zo schémy (obrázok 7.8) vidíme že je zhruba dva krát zložitejší ako AND. Má dva vstupy A,B a jeden výstup C kódované dvojdrôtovo. Implementáciu v celulárnych poliach vidíme na obrázku 7.9. Obvod fungoval správne na všetky možné kombinácie logických hodnôt.



Obrázok 7.8: Asynchrónne hradlo XOR [2].

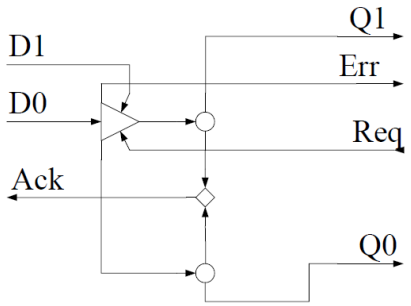


Obrázok 7.9: Rozloženie asynchrónneho hradla XOR v celulárnom poli.

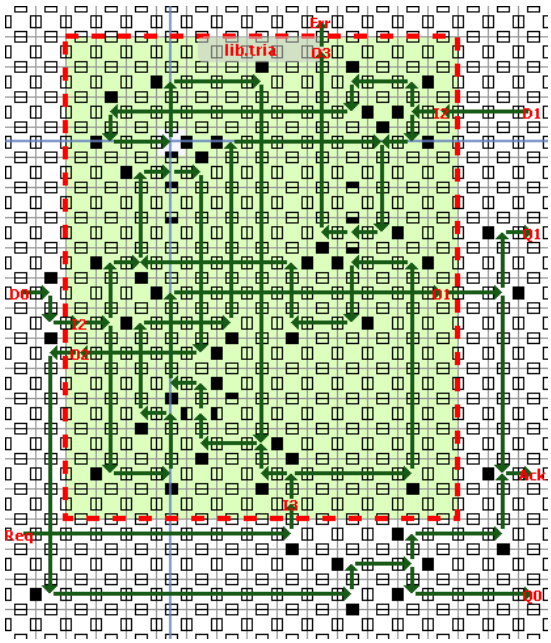
7.1.4 Register

Register je obvod, ktorý slúži na dočasné zapamätanie stavu (na rozdiel od pamäte, ktorá si pamätá informáciu trvale). Stav, teda logická *1* alebo *0* je zapamätaná až do doby, kým nebude obvod požiadaný o túto hodnotu. Stav sa udržiava zapamätaný vďaka synchronizačnej vlastnosti prvku *TRIA*. Ako už vieme, *TRIA* generuje výstup po príchode dvoch vstupov. Schéma zapojenia je na obrázku 7.10 a rozloženie na celulárnych poliach ukazuje obrázok 7.11.

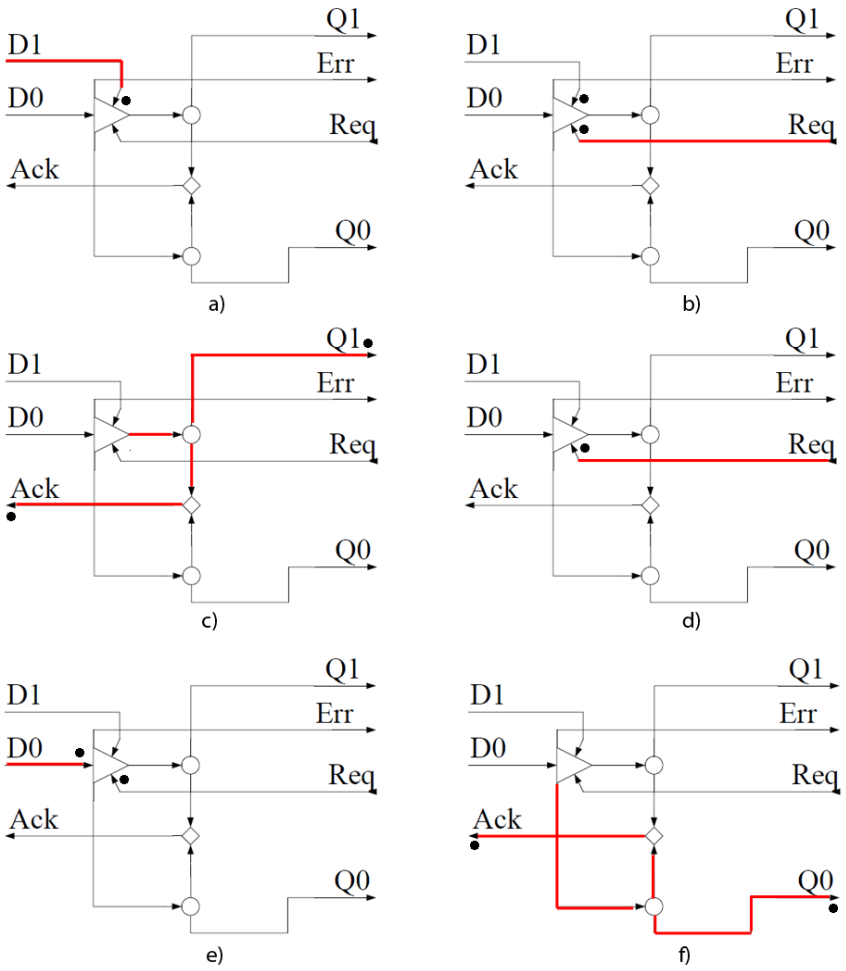
Na obrázku 7.12 si ukážeme ako asynchrónny register funguje. Používaním dvojdrôtového kódovania je zabezpečené, že k prvku *TRIA* bude privedený signál buď cestou *D1* alebo *D0* avšak nikdy zároveň. *TRIA* bude čakať na príchod ďalšieho signálu, pretože potrebuje dva vstupné signály aby došlo ku generovaniu výstupu. Tým je *Req* signál. Prvý signál je na *D1*, druhý prichádza na *Req*, takže sa bude generovať signál putujúci smerom na výstup *Q1*. Zároveň sa v prvku *Split* signál rozdelí a príde na výstup *Ack*, čím môžeme detekovať prečítanie z registra. Register je po operácii čítania vo východzom stave a čaká na príchod ďalšieho signálu na *D1* alebo *D0*. V prípade, že príde požiadavok na získanie stavu (signál na *Req*) ale *D1* alebo *D0* má oneskorenie, obvod bude v stave čakania. Príchodom signálu *D1* alebo *D0* *TRIA* odošle na patričný výstup signál. Signál *Err* je generovaný ak na vstup príde *D1* a *D0* skôr, ako bol jeden z nich prečítaný, čo je zakázaný stav v dvojdrôtovom kódovaní.



Obrázok 7.10: Asynchrónny Register [2].



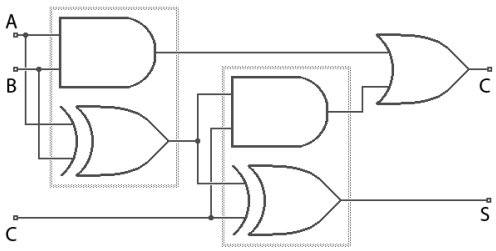
Obrázok 7.11: Rozloženie asynchrónneho Registra v celulárnom poli.



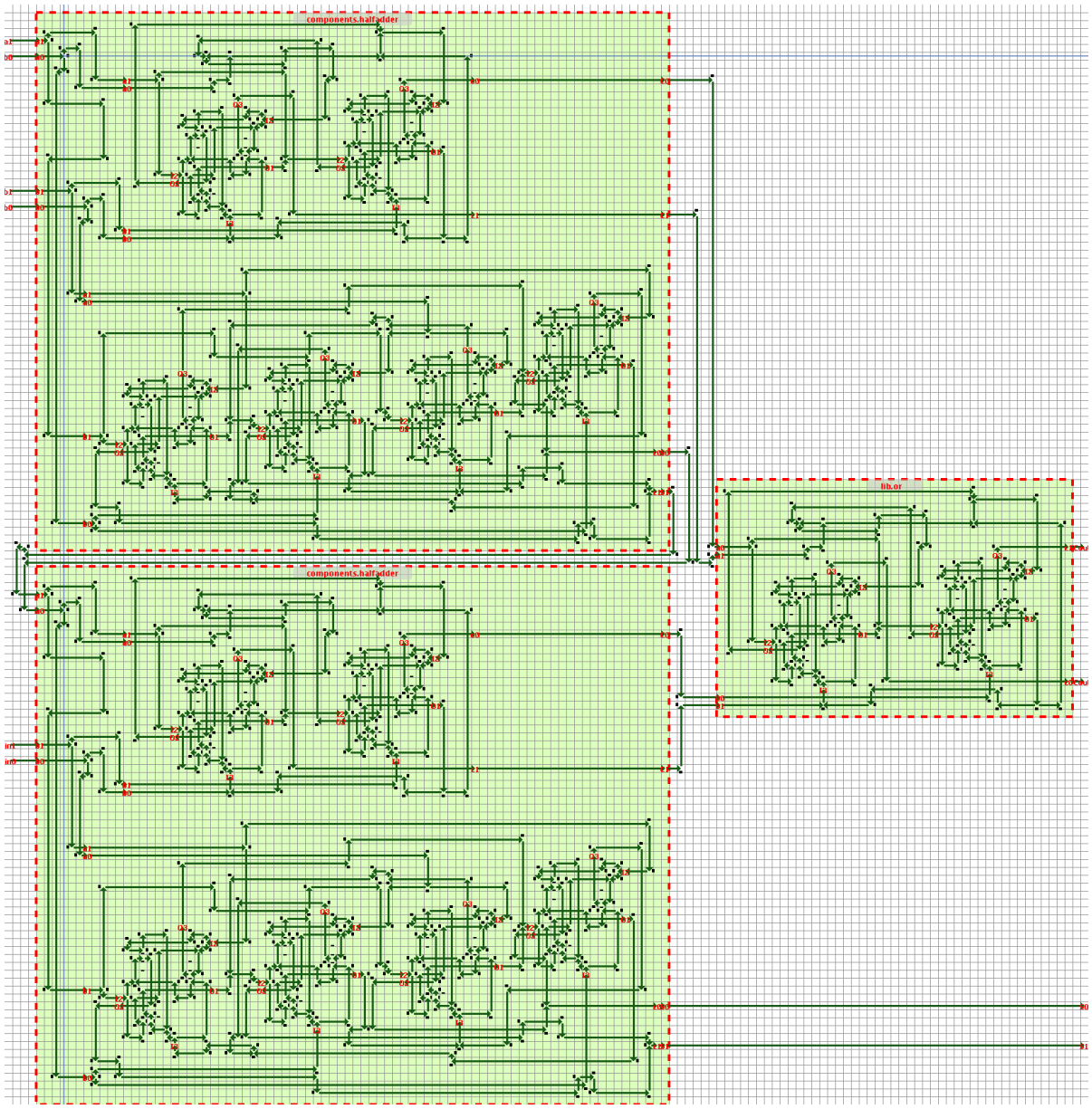
Obrázok 7.12: Šírenie signálu v Registri. Obrázky a,b,c ukazujú prípad, kedy ako prvý prichádza signál na D1, na obrázkoch d,e,f ako prvý príde Req signál.

7.1.5 1-bitová úplná sčítačka

Na základe navrhnutých obvodov AND a XOR bola zostrojená polovičná sčítačka. Prepojením dvoch polovičných sčítačiek s pripojeným hradlom OR bola realizovaná úplná sčítačka. Schéma zapojenia je zobrazená na obrázku 7.13. Obrázok 7.14 ukazuje implementáciu sčítačky v celulárnom poli. Kvôli veľkosti konfigurácie bola sčítačka výrazne zmenšená. Po overení funkčnosti vykazovala správne chovanie, ale kvôli jej zložitosti nebudeme ukazovať ako pracuje. V podstate je zložená prepojením modulov resp. komponent, o ktorých vieme, že fungovali správne, preto je zrejmé že sčítačka bude taktiež fungovať správne.



Obrázok 7.13: Schéma úplnej 1-bitovej sčítačky [17].



Obrázok 7.14: Rozloženie asynchrónnej úplnej 1-bitovej sčítačky v celulárnom poli.

7.2 Zhodnotenie testovania a práce so simulátorom

Pri zhodnotení sa zameriame na dva faktory. Prvým faktorom je navrhovateľnosť. Návrh asynchrónnych obvodov bol výrazne zložitejší v porovnaní so synchrónnymi obvodmi. Kým u synchrónnych obvodoch je globálne časovanie, ktoré mnohonásobne uľahčuje návrh, asynchrónne obvody musia byť navrhnuté tak, aby fungovali korektne bez synchronizácie. Dôležité je tiež aby obvod reagoval správne na všetky platné vstupné kombinácie signálov. Náročnosť návrhu sa prejavila hlavne pri práci s elementárnymi komponentami, zatiaľ čo návrh komponent zložených z logických prvkov bol relatívne triviálny. Návrh by sa komplikoval ak by sme potrebovali zaviesť synchronizáciu medzi komponentmi na princípe požiadavka a odpoveď.

Druhým faktorom, ktorý je nutné zhodnotiť je práca so simulátorom. Implementovať základné logické obvody v celulárnom poli nebolo zložité. Výbornou pomôckou boli šípky ukazujúce šírenie signálu. Na základe nich bolo ľahké si pri návrhu spraviť predstavu o rozmiestnení prvkov a toku signálu. Užívateľ preto môže efektívne využiť plochu buniek a nemal by sa ľahko dopustiť chyby. Pri simulovaní väčších komponent program pracoval efektívne a rýchlo reagoval na akcie užívateľa. Avšak umiestnenie komponenty a prepojenie s ďalšími komponentami už vyžadovalo veľa času. Komponenty musia byť umiestnené presne na bunku, tak isto aj prepojenia si vyžadujú maximálnu presnosť. To núti užívateľa k neustálemu približovaniu a odďaľovaniu plochy. Naše simulované obvody boli o veľkosti 300 x 300 buniek, čo nie je málo ale problém so stálym zoomovaním bol primeraný. Ďalšie funkcie ako pohyb po mriežke, nastavovanie stavu bunky, kreslenie šípok a práca s komponentami fungovali presne podľa návrhu a užívateľsky boli výborne zvládnuté.

Simulácie prebiehali podľa očakávaní a nedochádzalo ani ku žiadnym kolíziám. Rýchlosť simulátora bola postupne počas vývoja optimalizovaná. Aktuálna verzia pracuje dostatočne rýchlo a určite by si poradila aj s väčšími obvodmi.

8 Záver

Cieľom tejto diplomovej práce bolo implementovať a na praktických úlohách overiť funkčnosť simulátora schopného simulovať obvody budúcich nanopočítačov. Simulátor bol navrhnutý na báze celulárnych automatov. V histórii celulárnych automatov vznikali viaceré modely demonštrujúce chovanie celulárnych automatov [1], ktoré prevažne skúmali vplyv vstupných parametrov na výpočet automatu. Podľa typu úlohy boli modifikované napríklad spôsob synchronizácie, veľkosť okolia, prístup k pravidlám (globálny alebo lokálny) a iné. Z týchto modelov sa podarilo získať množstvo zaujímavých informácií. Problémom skúmaných automatov bol hlavne predpokladaný synchrónny režim, ktorého realizácia v oblasti nanopočítačov je neefektívna. Napriek tomu model celulárneho automatu bol stále považovaný za vhodný na realizáciu nanopočítačov. To viedlo k ďalšiemu výskumu a v [5] bola predstavená asynchrónna varianta pod názvom *Self-Timed CA (STCA)*. Model STCA nevyžaduje aktualizáciu všetkých buniek, naopak predpokladá neaktivitu buniek. Bunka sa zaktivuje, prípadne môže dôjsť k prechodu za predpokladu, že bola aktualizovaná nejaká susedná bunka. To viac odpovedalo fyzickej realizácii a preto predmetom ďalšieho výskumu bolo nájsť technológiu založenú na STCA, ktorá by bola schopná výpočtu. V [8] boli predstavené asynchrónne celulárne polia využívajúce model celulárneho automatu s vlastným časovaním.

Skúmanie ich vlastností a možnosti realizácie obvodov na nich postavených, bolo podnetom pre vytvorenie tejto diplomovej práce. Prísna lokálna interakcia medzi bunkami, ktorá je kľúčovou v celulárnych poliach nedovoľuje realizáciu synchrónnych obvodov. Avšak na druhú stranu asynchrónne obvody, by mohli byť veľmi efektívne implementované.

Asynchrónne obvody sú charakteristické odolnosťou voči oneskoreniu, ktorá nevyžaduje prídavnú synchronizačnú logiku, ale vychádza implicitne z princípu návrhu. Definovaním stavov a prechodových pravidiel celulárneho automatu je možné popísať chovanie buniek. Toto chovanie v rámci určitej skupiny buniek môže byť triviálne, ale vhodným prepojením buniek je možné skladať moduly, ktoré budú mať určité výpočtové schopnosti. Dôležitým poznatkom bolo tvrdenie, že v oblasti asynchrónnych obvodov existuje univerzálna sada elementárnych modulov, nazývaná univerzum, pomocou ktorej je možné zostrojiť obvody odolné voči oneskoreniu, ktoré sú schopné spočítať rovnakú triedu problémov ako konvenčné počítače realizované pomocou synchrónnych obvodov [8], [9].

V prvej polovici tejto práce boli podrobnejšie predstavené asynchrónne celulárne polia, ktoré vychádzajú z modelu asynchrónnych celulárnych automatov. Boli predstavené základné prvky, ich blokové schémy a možná implementácia. V druhej časti bol navrhnutý a implementovaný simulátor obvodov odolných voči oneskoreniu založených na celulárnych poliach. Na zvolených simuláciách sme overili funkčnosť simulátoru a ukázali, že je možná realizácia základných logických blokov, alebo aj zložitejších obvodov ako sčítačka.

Aj keď simulátor podporuje viaceré nástroje na zjednodušenie implementácie obvodov, je vyžadovaná veľká znalosť problému ako pri návrhu, tak pri vytváraní obvodov. Jedným z možných riešení je tvorbu konfigurácie modulu prenechať buď nejakému sofistikovanému algoritmu, alebo využiť evolučné algoritmy v spolupráci s navrhnutým simulátorom k tvorbe modulov na základe požadovaného chovania, čím by odpadli oba problémy, avšak je otázne aké spoľahlivé by takéto obvody boli.

Literatúra

[1] PEPER, F., J. LEE, S. ADACHI a T. ISOKAWA. *Cellular Nanocomputers*: *A Focused Review*. International Journal of Nanotechnology and Molecular Computation, 2009, s. 33-49. 1, 1.

[2] DI, J. a P. LALA. *Cellular Array-based Delay-insensitive Asynchronous Circuits Design and Test for Nanocomputing Systems*. Journal of Electronic Testing, 2007, s. 175-192. 23, 2-3.

[3] PEPER, F., J. LEE, S. ADACHI a S. MASHIKO. *Laying out circuits on asynchronous cellular arrays*: *a step towards feasible nanocomputers?*. Nanotechnology, 2003, 469–485. 14, 4.

[4] SEKANINA, L. *Evoluční hardware*: *od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Vyd. 1. Praha: Academia, 2009, 321 s. ISBN 978-80-200-1729-1.

[5] ADACHI, S., F. PEPER a J. LEE. *Computation by Asynchronously Updating Cellular Automata*. 2004, roč. 114, 1/2, s. 261-289.

[6] NIKOLIC, K, A SADEK a M FORSHAW. *Fault-tolerant techniques for nanocomputers*. Nanotechnology, 2002, 357–362. 13, 3.

[7] KOMENDA, T. *Sebereplikace v celulárnich systémech*. Brno, 2009. 85 s. Diplomová práce. FIT VUT v Brně.

[8] PEPER, F., J. LEE, F. ABO, T. ISOKAWA, S. ADACHI, N. MATSUI a S. MASHIKO. *Fault-Tolerance in Nanocomputers*: *A Cellular Array Approach*. IEEE Transactions on Nanotechnology, 2004, s. 187-201. 3, 1. ISSN 1536-125X.

[9] JIA LEE, F. PEPER, S. ADACHI a S. MASHIKO. *Universal delay-insensitive systems with buffering lines*. IEEE Trans. on Circuits and Systems, 2005, s. 742-754. 52, 4.

[10] ŠVANTNER, M. *Globálně řízené celulární automaty*. Brno, 2009. 70 s. Diplomová práce. FIT VUT v Brně.

[11] MICHELI, G. *Readings in hardware/software co design*. Vyd. 1. San Francisco: Morgan Kaufmann, 2002, 697 s. ISBN 15-586-0702-1.

[12] LEE, J., S. ADACHI, F. PEPER a S. MASHIKO. *Delay-insensitive computation in asynchronous cellular automata*. Journal of Computer and System Sciences. Inc. Orlando FL USA: Academic Press, 2005, s. 201-220. 70, 2.

[13] KELLER, R.M. *Towards a Theory of Universal Speed-Independent Modules*. IEEE Transactions on Computers, 1974, 21 - 33. C-23, 1.

[14] Java BluePrints: J2EE Patterns. *Model-View-Controller* [online]. Inc. Sun Microsystems. 2002 [cit. 2012-05-21]. Dostupné z: http://java.sun.com/blueprints/patterns/MVC-detailed.html

[15] LARMAN, C. *Applying UML and patterns*: *introduction to object-oriented analysis and design and iteractive development*. 3rd ed. New Jersey: Prentice-Hall, 2005, 703 s. ISBN 01-314-8906-2.

[16] Dependency injection. WIKIPEDIA. [online]. [cit. 2012-05-21]. Dostupné z: http://en.wikipedia.org/wiki/Dependency_injection

[17] Binárna sčítačka. WIKIPÉDIA. [online]. [cit. 2012-05-21]. Dostupné z: http://sk.wikipedia.org/wiki/Binárna_sčítačka

Príloha A

Užívateľská príručka

Príloha je doporučená pre koncového užívateľa, nie programátora. Slúži na rýchle sa zoznámenie so simulátorom. Užívateľovi poskytne nasledovné informácie:

- rýchle spustenie programu
- tvorba vlastných simulácií
- kompilácia a inštalácia

Minimálne požiadavky na simulátor

Rozlíšenie:	1000 x 700 px (doporučené 1600 x 1000 px)
Diskový priestor:	2 MB
Pamäť:	200 MB
CPU:	Intel 2 GHz (slabší bude tiež fungovať, ale môže byť pomalšia simulácia).
Software:	JAVA jdk1.7.*
Operačný systém:	Všetky OS s možnosťou spustenia Java aplikácií.

Ako si rýchlo vyskúšať simulátor?

1. Otvoríme CD a na disk prekopírujeme adresár *NanoSimSandBox*, v ktorom sa nachádza kompilovaná aplikácia. Na jej spustenie je nutné mať nainštalovanú *Javu(jdk 1.7)* (ak nie inštalovaná, tak si ju nainštalujte).
2. V adresári *NanoSimSandBox* na disku (*ďalej označované DIR*) kliknutím na *nanoSim.jar* spustíme simulátor.

Demo simulácia číslo 1.:

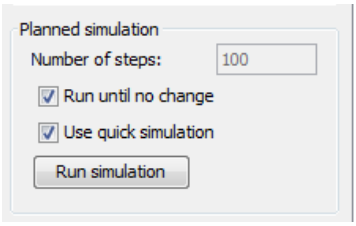
Príklad ukazuje spustenie simulácie s možnosťou sledovania zmien a toku signálu.

1. Otvoríme si testovaciu simuláciu (File > Load grid). Testovať je možné ľubovoľnú komponentu alebo už pripravené simulácie. Pripravené simulácie nájdeme v adresári *DIR/data/testcase.*
2. Vyberieme si súbor *tria_test.*
3. Simulácia je už pripravená, stačí stlačiť tlačidlo "Start" a simulátor vykoná simuláciu. Signál je nastavený na vstupoch *I1* a *I3*, po spracovaním komponentou *Tria* sa signál objavy na výstupe *O2*.

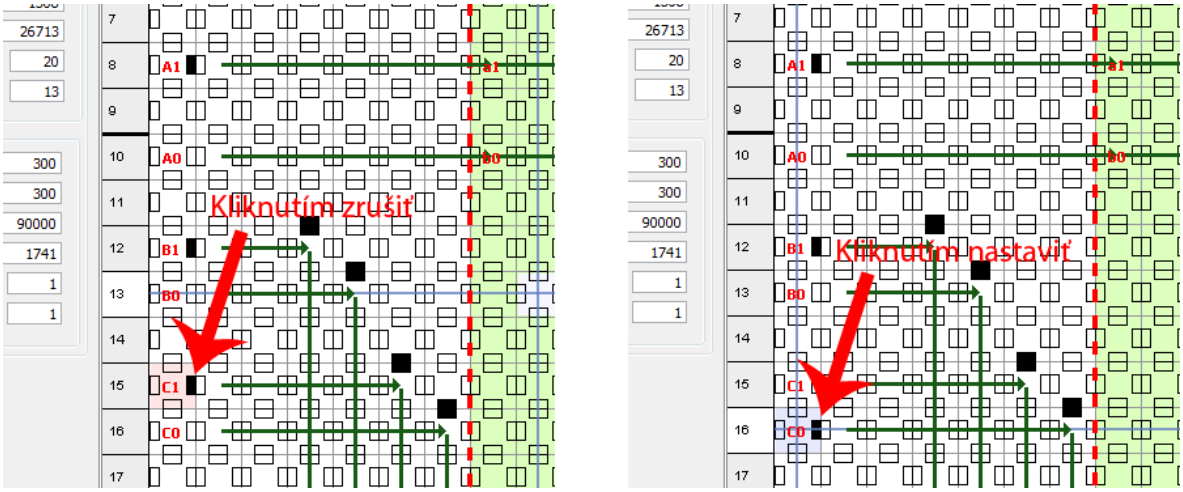
Demo simulácia číslo 2.:

Príklad ukazuje zložitejšiu simuláciu, ktorá bude nastavená pre rýchle vykonanie simulácie (bez grafickej vizualizácie) zameraná na testovanie funkčnosti komponenty.

1. Otvoríme si testovaciu simuláciu (File > Load grid)
2. Vyberieme si súbor *fulladder_test*. Sčítačka ma pripravené vstupné signály na vstupoch A1, B1 a C1. (A,B sú hodnoty, C je carry). Výsledok sa objavy na výstupe S(suma) a C(carry).
3. Simulátor okrem vizualizácie umožňuje takzvanú rýchlu simuláciu. Tú nastavíme zaškrtnutím "*Run until no change*" a "*Use quick simulation*" (podľa obrázka A.1).
   Run until no change - nastaví Simulácia bude bežať dokým dochádza k nejakej zmene.
   Use quick simulation - nastaví aby simulácia prebehla bez grafickej vizualizácie.
4. Stlačíme *Run simulation.*
5. Signál sa objavy na výstupe S1 a C1.
6. Simuláciu reštartneme (tretia ikona v Toolbare)
7. Zrušíme signál na C1 a nastavíme signál na C0 (kliknutím na štvorček v bunke sa zmení jeho stav na opačný, nastavenie je ukázané na obrázku A.2).
8. Stlačíme *Run simulation.*
9. Signál sa objavy na výstupe S0 a C1, čo značí, že carry je v log 1 a suma je v log 0.

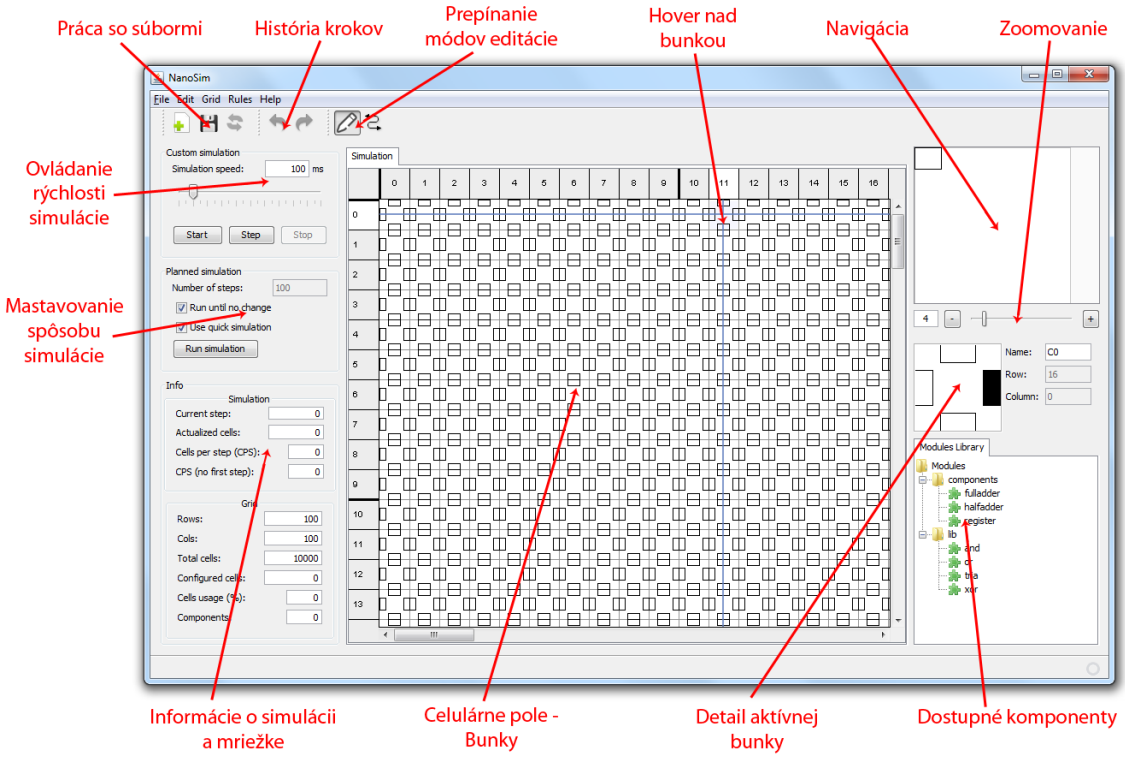


Obrázok A.1: Nastavenie rýchlej simulácie.



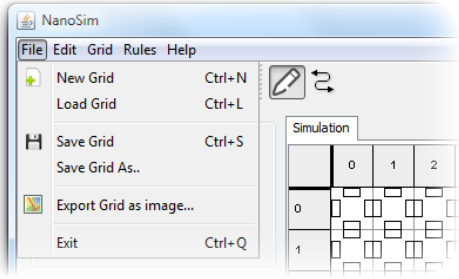
Obrázok A.2: Zmena konfigurácie prednastavených vstupov.

Ako pracovať s programom?



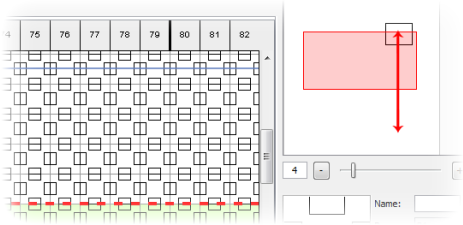
Obrázok A.3: Popis funkcií v simulátore

Program umožňuje pracovať s uloženými simuláciami alebo vytvárať nové simulácie. Tvorba nových simulácii bude vysvetlená v nasledovnej kapitole, tu sa naučíme ako program používať, aké funkcie poskytuje. Obrázok A.4 ukazuje základné možnosti pri práci so súbormi. Pomocou *New Grid* je možné vytvoriť novú nenakonfigurovanú mriežku. *Load Grid* umožňuje načítať už vytvorenú komponentu alebo simuláciu (čo je v podstate tiež komponentou). *Save Grid* a *Save Grid as* slúži na ukladanie simulácii (Pozor pri spustení z CD nie je možný zápis na CD). Funckia *Export Grid as image* ukladá celú plochu, aj časť, ktorá nie je viditeľná na monitore v aktuálne nastavenom priblížení do PNG obrázka. Niektoré možnosti je možné používať pomocou klávesových skratiek (vidíme z obrázka A.4).



Obrázok A.4: Menu.

Pri návrhu komponent je možné prechádzať v histórii zmien. Nato slúžia šípky v hornom toolbare. (obrázok A.3 - História krokov). Program si pamätá manipuláciu s komponentami, šípkami a bunkami. Pri načítaní novej komponenty sa história zmien vyčistí.

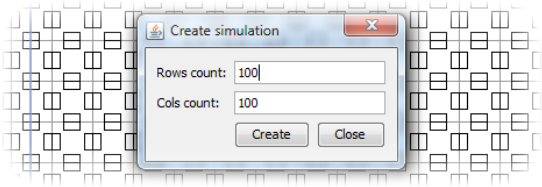


Obrázok A.5: Navigácia.

Navigácia ukazuje pozíciu obrazovky, kliknutím v navigácii je možné presúvanie zobrazenej plochy, *ťahom* stlačenej *myši v navigačnej oblasti* je možné plynulú posúvanie mriežky. Mriežku je možné posúvať *stlačením medzerníka a pravým tlačidlom myši v oblasti mriežky*.

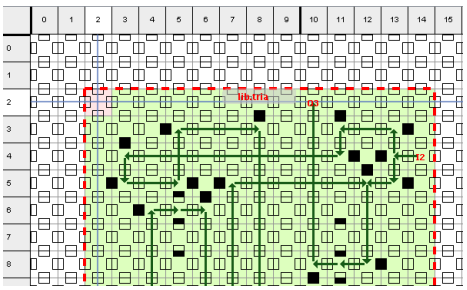
Tvorba vlastnej simulácie

Vlastnú simuláciu si môžeme vytvoriť nasledovne. Najskôr si otvoríme novú mriežku (Ctrl+N). Necháme veľkosť 100 na 100 a dáme *Create* (obr. A.6). Klikneme na bunku (3,3) a vložíme komponentu dvojklikom v zozname dostupných komponent (použijeme lib/tria). Komponenta sa nakonfiguruje od vybranej bunky (obr. A.7).



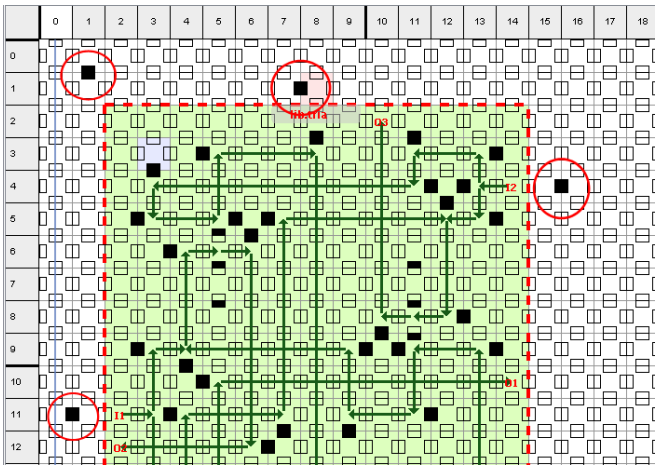
Obrázok A.6: Vytvorenie novej mriežky.

S komponentou je možné ďalej pracovať. Napr. posúvanie komponenty je možné kliknutím na nejakú bunku komponenty a potom stlačením SHIFT+ALT+ šípka v požadovanom smere.



Obrázok A.7: Pridanie komponenty.

Bunky sa konfigurujú buď stlačením ľavého tlačidla na myši, kedy sa zmení jedna pamäťová bunka, alebo stlačením prvého tlačidla myši, kedy sa zmení aj susedná pamäť (vytvorí sa štvorček). Nastavíme si konfiguráciu podľa obrázka A.8.

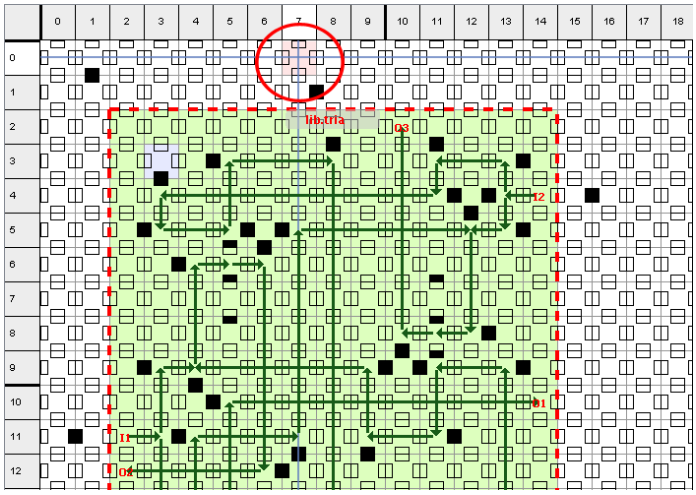


Obrázok A.8: Konfigurácia buniek.

Teraz by bolo možné spustiť simuláciu, ale ešte predtým si ukážeme *ako sa vytvárajú šípky*. Z navrhnutej konfigurácie je momentálne ťažko vidno ako obvod funguje. Preto si naznačíme tok signálu pomocou šípiek. V toolbare si prepneme mód editácie na kreslenie šípiek (obrázok A.9).

Obrázok A.9: Zmena módu kreslenia.

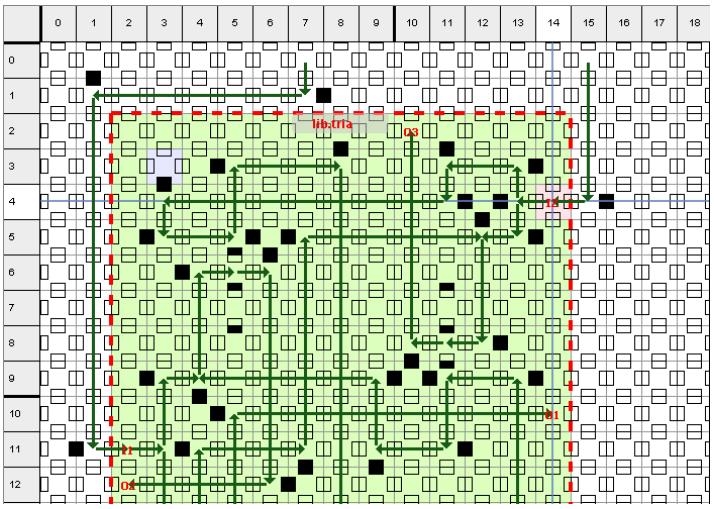
Šípky sa kreslia vždy od aktívnej bunky (podfarbená na červeno, posledná na ktorú bolo kliknuté). Klikneme na bunku (0,7) aby bola označená ako aktívna.



Obrázok A.10: Nastavenie aktívnej bunky.

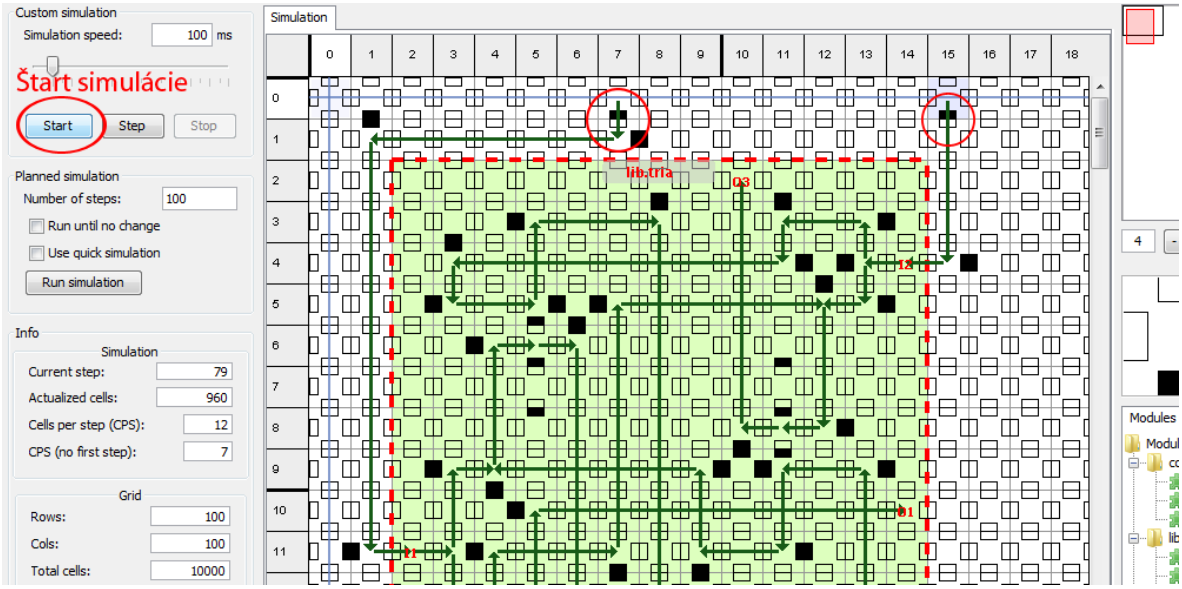
Stlačíme SHIFT a klikáme na bunky kade chceme viesť šípku. Správne by sme mali klikať na bunky (1,7), (1,1), (11,1) a (11,2). Potom pustiť SHIFT, kliknúť na bunku (0,15) a opäť stlačiť SHIFT a kliknúť na bunky (4,15) a (4,14). Tým by sme mali dostať konfiguráciu zobrazenú na obrázku A.11. Šípky je možne označiť ťahom myši a prípadne i vymazať stlačením DELETE.

Prípadne v editačnom móde označením buniek a stlačením SHIFT+DELETE, ktoré okrem zmazania buniek zmaže i označené šípky.



Obrázok A.11: Vytvorenie šípiek.

Teraz máme pripravenú prehľadnú simuláciu modulu *Tria*. Aby sme mohli začať simulovať, musíme nastaviť vstupné signály. To spravíme prepnutím módu mriežky späť na editáciu a nastavíme vstupné signály podľa obrázka A.12. Potom už môžeme odštartovať simuláciu kliknutím na *Start* a vidíme, že sa začala meniť konfigurácia. Simuláciu zastavíme stlačením *Stop*. Keď chceme proces simulácie zopakovať môžeme v nástrojovej lište pomocou šípiek vrátiť konfiguráciu pred začatím simulácie. Môžeme skúsiť použiť ďalšie možnosti na spustenie simulácie, napr. *Run until no change* a stlačiť *Run Simulation*, kedy sa simulácia automaticky zastaví, keď nebude čo simulovať.



Obrázok A.12: Nastavenie vstupného signálu a zahájenie simulácie.

Príloha B

Obsah CD

- Elektronická verzia diplomovej práce
- Zdrojové kódy implementovaných tried v jazyku Java
- SandBox - kompilovaná aplikácia s vytvorenými komponentami a simuláciami k rýchlemu vyskúšaniu simulátora
- XML Konfigurácie komponent vytvorených počas experimentovania so simulátorom