SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dokumentácia projektu pre zimný semester

Prostredie pre návrh digitálnych systémov

Tímový projekt

Akademický rok: 2010/2011

Študijný program: PKSS

Vedúci projektu: Ing. Peter Pištek

Tím č. 2: Bc. R. Chytil, Bc. M. Jánoš, Bc. T. Lőrincz, Bc. T. Takács, Bc. R. Virkler

Obsah

0 Úvod

Táto kapitola obsahuje informácie o zadaní projektu, účelu a rozsahu dokumentu, použitých skratkách a výrazoch a použitej notácie.

0.1 Zadanie projektu

Každý zo študentov odboru PKSS sa počas svojho štúdia stretol s viacerými prostrediami pre návrh digitálnych systémov. Jedná sa o rôzne úrovne návrhu od klasických kombinačných logických obvodov , cez použitie Petriho sietí na opis správania systému až po návrh procesorov alebo ASIC obvodov. Základným problémom je nízka podpora programových systémov pri testovaní niektorých spôsobov návrhu a následnej simulácie, poprípade syntézy. Z tohto dôvodu je cieľom vytvoriť prostredie, s ktorým by študenti mohli pracovať na čo najväčšom počte predmetov zameraných na návrh digitálnych systémov a zastrešovalo by všetky metodiky návrhu na rôznych úrovniach. Významné z pohľadu výskumu na našej fakulte je rovnako aj vytvorenie prostredia pre testovanie jednotlivých skúmaných metód (v rámci ústavu, fakulty, SR,..).

Niektoré z hlavných cieľov projektu:

- Základ pre modulárny aplikačný systém umožňujúci prácu s čo najväčším množstvom metodík návrhu.
- Umožniť dodatočné pridávanie nových metodík.
- Podporovať súborové štandardy - BLIF, KISS, SLIF, atď.
- Zahrnutý grafický editor pre klasické hradlové obvody, Petriho siete, Konečné stavové automaty, prípadne iné grafické modely používané pri návrhu.
- Podpora simulácie daných grafických modelov (príkladom je PIPE pre Petriho siete alebo LOG pre hradlové obvody)

Cieľom Tímového projektu je v prvom kroku vytvorenie základu pre daný modulárny systém. Je nutné podrobne zanalyzovať súvisiacu problematiku a implementovať prostredie spolu s grafickým editorom do ktorého by bolo možné pridávať jednotlivé metodiky návrhu a podporované modely.

0.2 Účel a rozsah dokumentu

V súčasnosti neexistuje postačujúci modulárny programový systém pre návrh digitálnych obvodov. Pritom prostriedkov je k dispozícii dosť. Tento dokument sa preto venuje najmä problematike návrhu základu takéhoto systému, ktorý obsahuje rôzne moduly podľa určitých oblastí návrhu, syntéze a simulácie digitálnych systémov. Vznikol ako dokumentácia k predmetu Tímový projekt 1 v 1. ročníku inžinierskeho štúdia. Je výsledkov práce piatich študentov, členov tímu. Cieľom a účelom dokumentu je poskytnúť prostredie na návrh digitálnych systémov, v ktorom bude možné spravovať, ovládať a pridávať moduly z rôznych oblastí tematiky. Analytická časť opisuje vybrané oblasti problematiky ako programy na syntézu logických obvodov, Petriho sietí, súborové formáty a binárne vyhľadávacie stromy, a prehľad existujúcich riešení, teda nami vybrané časti zadania, ktorými za budeme ďalej zaoberať. Nasleduje špecifikácia požiadaviek na systém a hrubý návrh riešenia požiadaviek špecifikovaných v predchádzajúcej kapitole, ktorý predstavuje prvý pohľad na podobu výsledného produktu a obsahuje výber implementačného prostredia, architektúru systému spolu s modulmi pre načítanie, grafický editor a simulátor obvodiv. Systém je navrhnutý tak, aby sa mohol rozšíriť o ďalšie moduly. Nakoniec uvádzame zdroje, z ktorých sme čerpali pri tvorbe dokumentu.

0.3 Použité skratky a výrazy

BDD (*Binary Decision Diagram*) - binárny rozhodovací diagram

BLIF-MV (*Berkeley Logic Interchange Format*-Multi Value) – natívny súborový formát vyvinutý univerzitou Berkeley na zápis (viachodnotových) logických obvodov

CST (*Co-Singleton Transform*) – transformácia z multihodnotovej logiky do binárnej

CTL (*Computation Tree Logic*) - výpočet stromu logiky

EQN (*Equation Format*) – rovnicový formát zápisu obvodov

FSM (*Final State Machine*) – konečný stavový automat

FPGA (*Field Programmable Gate Array*) – programovateľné pole logických hradiel

HTML (Hypertext Markup Language) – hypertextový značkovací jazyk, používa sa na tvorbu internetových stránok

KISS – (Keep It Sipmle Stupid) – súborový formát

L Language – L jazyk, vychádzajúci z programovacích jazykov C, Tcl a Perl, ktorých vlastnosti kombinuje

MDD (Multivalued Decision Diagram) – viachodnotový rozhodovací diagram

MUX (*Multiplexor*) – multiplexor, prepína viac vstupných kanálov na jeden výstupný

MV (*Mulivalued*) – viachodnotové, napr. premenné

MVSIS (*Multivalued Logig Synthesis*) - program na syntézu a overovanie logických obvodov s viachodnotovými premennými

PLA *(*Programmable Logic Array) – programovateľne logické pole

PN (Petri Net) – petriho sieť

PNML (*Petri Net Markup Language*) – jazyk na zápis Petriho sietí

SDC, ODC (*Satisfiability, Observability Don‘t-care*) - splniteľnosť a pozorovateľnosť don’t care stavov

SOP (*Sum Of Product*) - súčet súčinov

STG (*Signal Transition Graph*) – graf zmeny signálu, popisuje správanie asynchrónnych obvodov

VHDL (VHSIC Hardware Description Language) – jazyk na popis hardvéru

VIS (*Verification Interacting with Synthesis*) – program na syntézu a overovanie log. obvodov

XML (*eXtensible Markup Language*) – jazyk na značkovanie, rozšírenie HTML

0.4 Použitá notácia

V tomto dokumente je použitá notácia UML. Použité typy diagramov a im prislúchajúce techniky opisu problémovej oblasti sú uvedené a vysvetlené nižšie.

0.4.1 Diagram činností

Obr. 0.1: Počiatočný stav

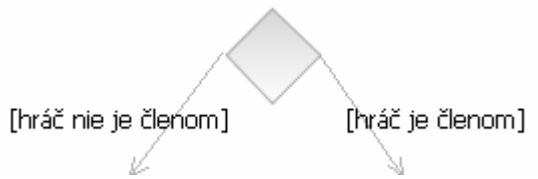
Počiatočný stav predstavuje začiatok vykonávania sa sledu činnosti.

Obr. 0.2: Konečný stav

Konečný stav prestavuje koniec vykonávania sa sledu(postupnosti) činnosti(operácie).

Obr. 0.3: Časový sled činností

Šípka na obr. 0.3 popisuje sled pripojených činností v čase, pričom činnosť nachádzajúca sa napravo(na ktorú šípka ukazuje) sa začne vykonávať keď činnosť naľavo skončí.



Obr. 0.4: Rozhodovací blok

Rozhodovací blok znamená, že na základe hodnoty uvedeného slovného výroku sa zvolí buď ľavý alebo pravý sled činnosti.

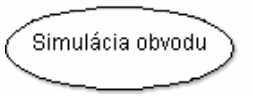
Obr. 0.5: Stav

Stav je pomenovaná situácia objektu počas jeho existencie, v ktorej objekt spĺňa nejakú podmienku, čaká alebo vykonáva nejakú činnosť. Inak povedané *stav* je množina okolností alebo atribútov, ktoré charakterizujú popisovaný objekt.

0.4.2 Diagram prípadov použitia

Obr. 0.6: Používateľ

Používateľ predstavuje pomenovanú úlohu, ktorú tento hrá vo vytváranom systéme.



Obr. 0.7: Prípad použitia

Prípad použitia pomáha porozumieť, štruktúrovať a porovnať základné požiadavky na systém. Dá sa charakterizovať ako pomenovaná a štruktúrovaným textom opísaná typická interakcia medzi používateľom a systémom.

Obr. 0.8: Asociácia

Šípka na obr. 0.8 predstavuje priradenie *prípadu použitia používateľovi*, s ktorým ho spája.

1 Analýza problému

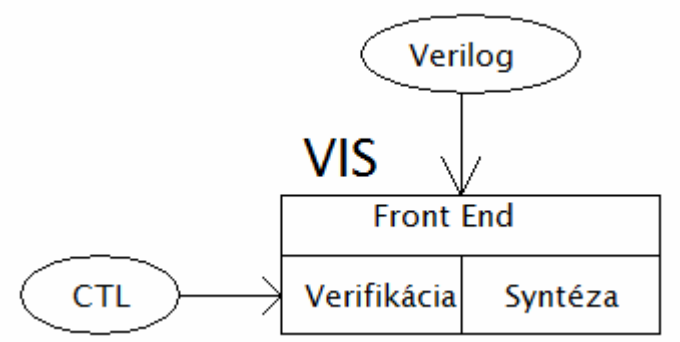
Táto kapitola obsahuje popis existujúcich riešení a najzákladnejších súborových formátov.

1.1 VIS

VIS (Verification Interacting with Synthesis) je systém určený na formálnu verifikáciu, syntézu a simuláciu konečných stavových systémov [1]. Bol vyvíjaný spoločne na univerzitách University of California at Berkeley, University of Colorado at Boulder a tiež University of Texas, Austin. Hlavným cieľom jeho vývoja bolo zlepšiť prvú generáciu takýchto systémov ako HSIS a SMV. Celkovo je systém VIS rozdelený na tri časti a to:

- Front End
- Verifikácia
- Syntéza

Na obr. 1.1 je zobrazené toto principiálne rozdelenie ako blokový diagram.



Obr. 1.1: Blokový diagram systému VIS

Časť Front End slúži na načítanie vstupného súboru a jeho prerobenie do hierarchického systému opísaného vo formáte BLIF-MV. Ako vstupný súbor sa VIS používa súbor vo formáte Verilog alebo súbor vo formáte BLIF-MV. Súčasťou systému VIS je aj utilita

VL2MV, ktorý slúži na prerobenie Verilog súboru do formátu BLIF-MV. Po načítaní vstupného súboru nasleduje verifikácia. Verifikačné jadro slúži na kontrolu modelu a na verifikáciu sa používa CTL [2]. Na syntézu sa využíva systém SIS a je ju možné vykonať len pre dvojhodnotové obvody.

Ako už bolo spomenuté VIS pracuje so vstupným súborom vo formáte Verilog, pričom špecifikácia systému v takomto formáte pozostáva z jedného alebo viacerých modulov. Pomocou VL2MV vie VIS prerobiť vstupný Verilog súbor do formátu BLIF-MV. Tento súborový formát je modifikáciou súborové formátu BLIF. Pomocou BLIF-MV vieme vytvoriť hierarchické viac vrstvové (multi-level) modely. Taktiež použitím formátu BLIF-MV vieme pracovať s viachodnotovými premennými a môžeme opísať nedeterministické správanie systému, lebo umožňuje používať nedeterministické vstupy.

Formálna verifikácia v systéme VIS pozostáva z nasledujúcich častí ako vytvorenie vnútornej reprezentácie konečného stavového systému, kontrola modelu, kontrola ekvivalencie, simulácia [2]. Po načítaní súboru BLIF-MV príkazom *read_blif_mv* a zadaní príkazu *init_verify* je BLIF-MV opis systému uložený ako hierarchický strom. Tento hierarchický opis pozostáva z modulov prípadne podmodulov. Príkazom *print_hierarchy_stats* zobrazíme informácie o hierarchii, *print_models* zobrazí štatistiku všetkých modelom v hierarchii a *print_io* zobrazí informácie o vstupoch a výstupoch. Súčasťou verifikácie je prerobenie sieťovej reprezentácie do funkcionálneho opisu. Ten opisuje výstup a nasledujúci stav premenných pomocou vstupu a aktuálneho stavu premenných. VIS na toto využíva BDD a tiež ich rozšírenie MDD. Toto nastáva po zadaní príkazov *flatten_hierarchy*, *static_order* a *build_partition_mdds*. Všetky tieto tri príkazy nahrádza spomínaný príkaz *init_verify*.

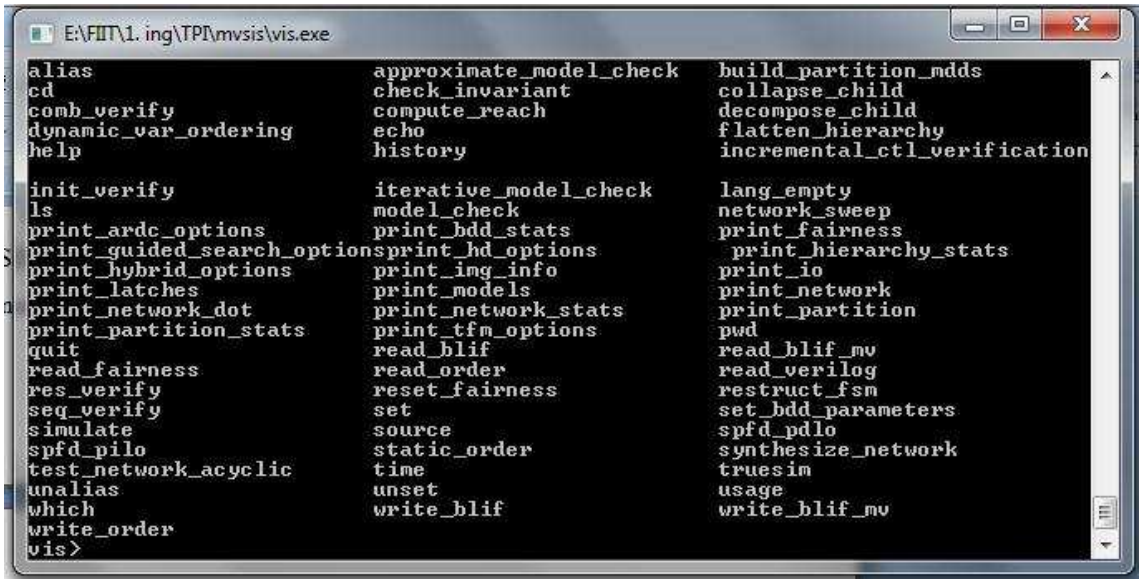
VIS umožňuje tiež overiť dosiahnuteľnosť jednotlivých stavov v konečnom stavovom automate (FSM) a definovať obmedzenia spravodlivosti, ktoré odstraňujú nechcené správanie systému. Ďalšou možnosťou je kontrola modelu pomocou príkazu *model_check*. Kontrola sa vykonáva pomocou kontroly vzorcov uvedených v jazyku CTL. Ďalšou schopnosťou je kontrola kombinačnej alebo sekvenčnej ekvivalencie dvoch sietí. Vlastnosti uvedené v tomto

však neboli prakticky overené, keďže sa jedná o komplexnú problematiku a vzorové súbory na ich otestovanie neboli k dispozícií. Vytvorenie vlastných súborov by vyžadovalo hlbšiu analýzu systému VIS, formátu BLIF-MV a jazyka CTL.

Zaujímavou možnosťou, ktorú systém VIS podporuje je simulácia pomocou príkazu *simulate*. Tento príkaz vie vygenerovať zadané množstvo náhodných vstupov alebo pracovať so vstupným súborom. Následne sú zobrazené hodnoty výstupov pre vstupné hodnoty.

Ako bolo spomenuté, VIS umožňuje vykonať syntézu kombinačných dvojhodnotových obvodov pomocou systému SIS. Táto funkcia však tiež nebola prakticky overená, keďže sme nemali k dispozícií korektne nainštalovaný systém SIS.

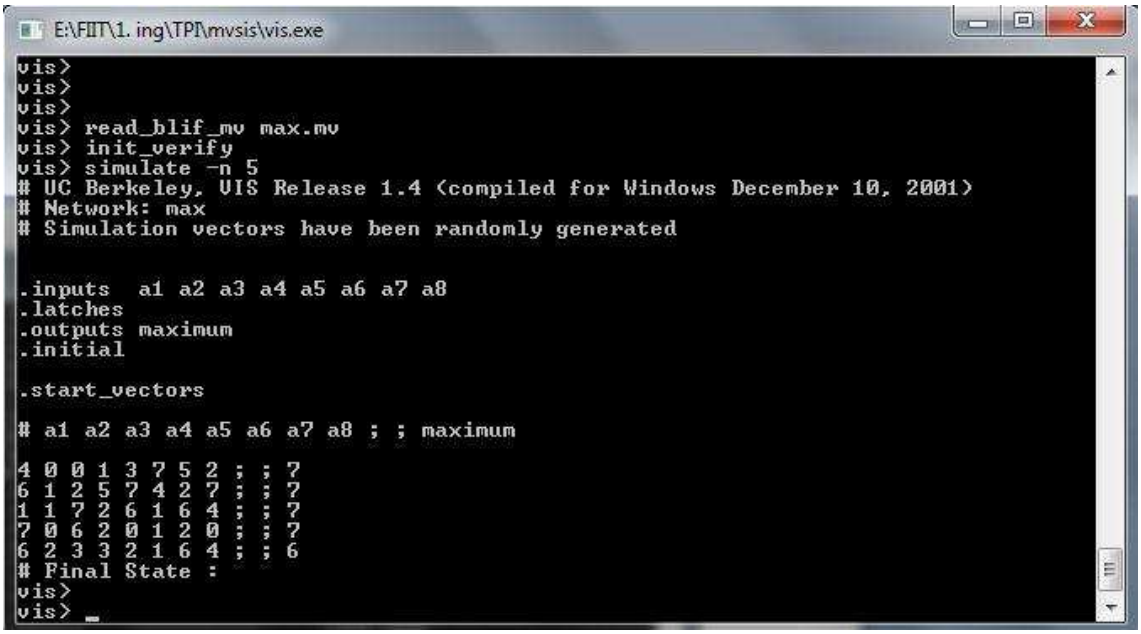
Prostredie systému VIS sa ovláda prostredníctvom príkazového riadka. Testovaná bola verzia pre Windows stiahnuteľná z [3], pretože linuxovú distribúciu sa nepodarilo korektne nainštalovať. Na obr. 1.2 je zobrazené prostredie programu VIS.



Obr. 1.2: Prostredie systému VIS s dostupnými príkazmi

Na obr. 1.3 je zobrazené načítanie súboru *max.mv*, ktorý opisuje správanie systému s 8 vstupmi a jedným výstupom. Na vstupoch môžu byť hodnoty od 0 po 7. Na výstup maximálna zo vstupných hodnôt. Následne je systém verifikovaný príkazom *init_verify*

a odsimulovaný pre 5 náhodných vstupov príkazom *simulate –n 5*. Súbor *max.mv* sa nachádza v prílohe A1.



Obr. 1.3: Práca v systéme VIS

Systém VIS môžeme zhodnotiť ako komplexný. Obsahuje pomerne veľké množstvo funkcií pričom sa nám nepodarilo všetky otestovať. Ich otestovanie by si vyžadovalo pomerne široké štúdium danej problematiky. Napríklad pre overovanie modelu by bolo potrebné naštudovanie si problematiky zápisu vzorcov v jazyku CTL. Systém VIS by bolo možné využiť volaním z nášho programu napríklad pri simulácii správania obvodu zapísaného v súborovom formáte BLIF-MV.

1.2 SIS a MVSIS

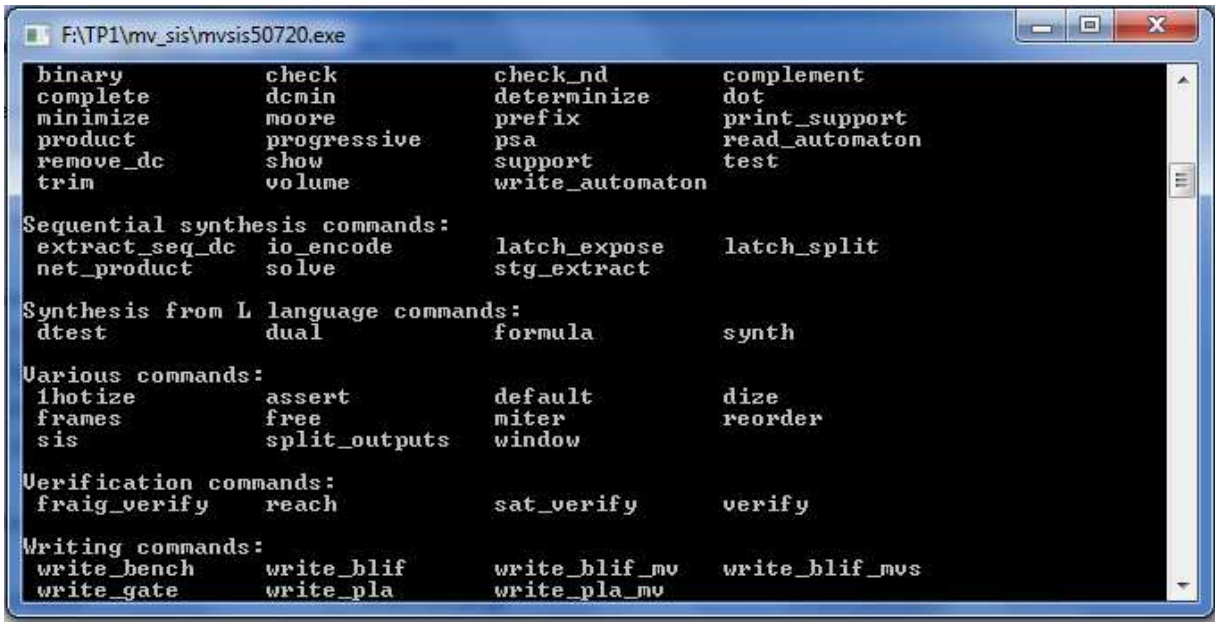
MVSIS je rozšírenie systému SIS o možnosť premenných nadobúdať viac hodnôt, každá s vlastným rozsahom. Ako SIS aj MVSIS je interaktívny nástroj. Vyvíjané na univerzite Berkeley v Kalifornii, USA. Zameriava sa na optimalizačné algoritmy, ktoré zdokonaľujú kvalitu obvodov generovaných automatickými nástrojmi na syntézu a súčasne prispôsobenie pre praktické využitie. Keď sa používa len v rámci binárnej logiky, správa sa presne ako SIS s tým rozdielom, že je rýchlejší, využíva menej pamäti a môže byť aplikovaný na rozsiahlejšie návrhy. MVSIS sa stane aj jeho úplnou náhradou v širšom ponímaní. Podobnosť medzi nimi je taká markantná, že na popis obidvoch postačuje MVSIS, navyše SIS je už zastaraná technológia a v súčasnosti sa nepoužíva. Podporuje údajové štruktúry a procedúry potrebné na technologicky nezávislú a MV (*Multi-valued MV,* viachodnotové) logickú syntézu. Obsahuje nové zdrojové kódy spolu s niektorými balíkmi prepožičanými od SIS a VIS. Najnovšia verzia je MVSIS 3.0, tá je však vo vývoji [6]. Preto sa nasledujúce riadky budú týkať najmä verzie 2.0.

Viacúrovňová viachodnotová logická syntéza má mnohostranné využitie:

- ✓ Logická syntéza pre MV hardvérové zariadenia (obvody CMOS s prúdovým módom, optické logické obvody)
- ✓ Počiatočná manipulácia s popisom hardvéru pred jeho zakódovaním do binárnej podoby a spracovaním štandardným programom na logickú syntézu. MV je prirodzený spôsob na popis procedúr na vyššej úrovni
- ✓ Popredná časť pre softvérový kompilátor od okamihu keď softvér prirodzene podporuje vyhodnocovanie MV premenných v jednom cykle. Silné transformácie logickej syntézy môžu byť aplikované na kompilátory zamerané na vnorené aplikácie.
- ✓ Poskytuje optimalizácie: zjednodušenie uzla, výber jadier a kociek, párovanie a kódovanie a manipulácie so sieťou

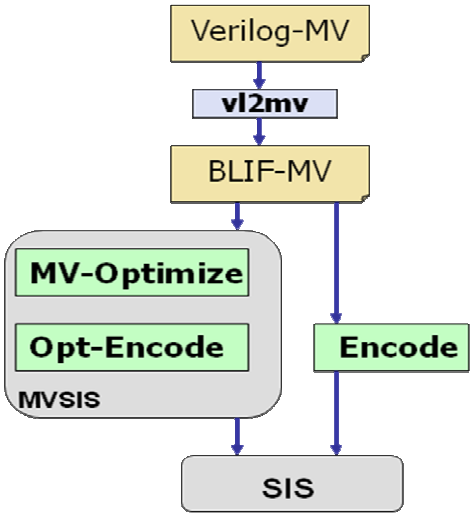
Mnoho používaných algoritmov obsahujú verzie založené na SOP a súčasne na BDD. To umožňuje dynamicky zvoliť verziu, ktorá je rýchlejšia za daných logických okolností. Minimalizácia uzla siete je rozšírená pomocou BDD, aby sa dalo poradiť s veľmi veľkými

dvojúrovňovými funkciami keď program Espresso zlyhá. Algoritmy *pair_decode*, *merge* a *encode* slúžia na konverziu do viachodnotovej oblasti a z nej. Umožňujú vykonať prieskum optimalizačného potenciálu v MV oblasti predtým, než sú všetky signály skonvertované do binárnej podoby.



Obr. 1.4: Prostredie programu MVSIS

Program MVSIS bolo možné vyskúšať ako konzolovú aplikáciu ovládanú príkazovým riadkom pod OS Windows. Jeho hlavné okno je na obr. 1.4



Obr. 1.5 Blokový diagram systému MVSIS

MV obvod sa do programu MVSIS vkladá vo formáte sieťového zoznamu MV uzlov príkazom *read_blif_mv*. Tak ako súbory BLIF-MV môžu byť generované programom Verilog pre neskoršie využitie programom VIS (vl2mv) (Obr. 1.5), môže sa tiež ním vypisovať pomocou VIS. Binárne siete sú špecifikované v BLIF formáte môžu byť čítané obdobným spôsobom. Po načítaní špecifikácie návrhu, je táto konvertovaná do MV siete, čo predstavuje reprezentáciu využívanú v rámci MVSIS. MV sieť je sieť uzlov, každý uzol predstavuje MV reláciu s jediným MV výstupom. Množina vstupných uzlov sa označuje ako CI (Combination Input) a výstupných uzlov CO (Combination Output). Funkcie združené do počiatku hodnôt (i-sety, príklad nižšie) sú uložené vo forme SOP alebo BDD.

MV funkcia, každá výstupná hodnota sa nazýva i-set, napr. pre funkciu F:

$$F(u,v,w):\{0,1\}\times\{0,1,2\}\times\{0,1,2\}\rightarrow\{0,1,2\}$$

0-set: $F^{\{0\}} = u^{\{0\}}v^{\{0\}} + u^{\{0\}}v^{\{1\}}w^{\{0,1\}}$

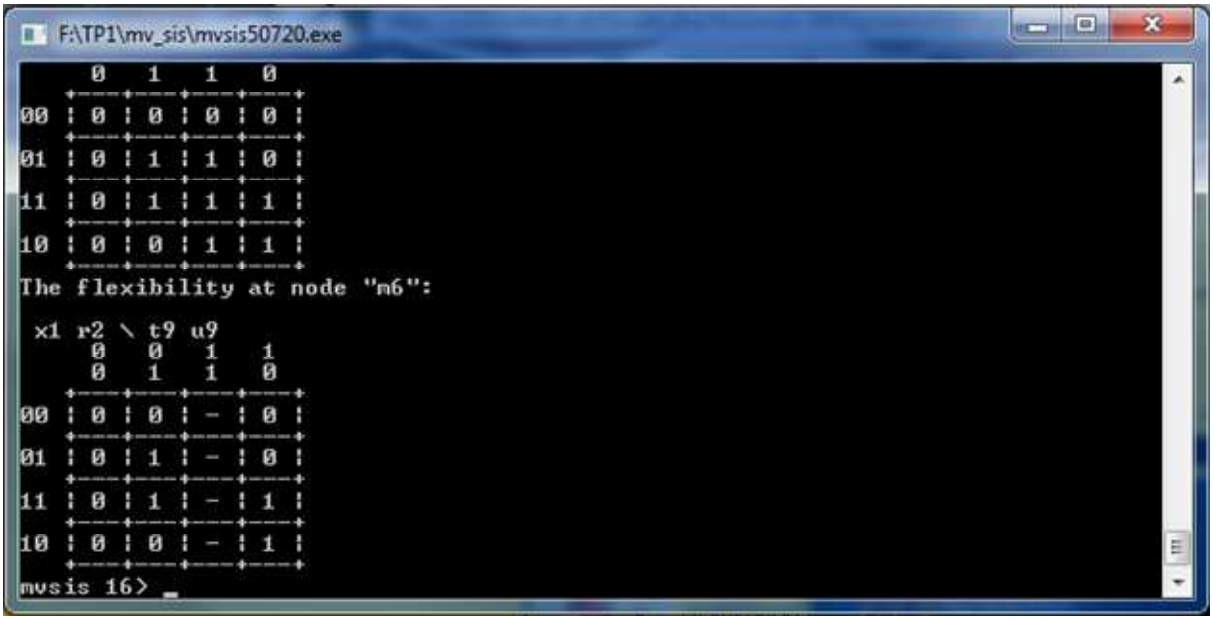
1-set: $F^{\{1\}} =$ <východisková hodnota>

2-set: $F^{\{2\}} = u^{\{1\}}v^{\{1,2\}} + u^{\{1\}}v^{\{0\}}w^{\{1,2\}}$

Len jedna MV premenná sa tu spája s výstupom každého uzla. Hrana spája uzly *i* a *j* ak nejaký z i-setov v *j* závisí explicitne od premennej spájanej s uzlom *i*. Sieť má množinu primárnych vstupov a množinu uzlov určených ako výstupy zo siete. Významný rozdiel

oproti iným MV metódam je, že tu každá premenná x_n má svoj vlastný rozsah hodnôt $\{0,1,\cdots,|P_n|-1\}$. S hodnotami sa zaobchádza jednotne. Podporuje tiež sekvenčné MV siete s MV hradlami (latch), ale nepodporuje sekvenčnú optimalizáciu. Počiatočná špecifikácia, tiež aj aktuálna sieť, môže obsahovať nedeterministické reláciu medzi uzlami. To môže zapríčiniť, že jeden alebo viac uzlov z primárnych výstupov sa stane nedeterministickým. Výsledok syntézy môže byť podmnožinou špecifikovaných počiatočných relácií. Toto je dané do platnosti ak je sieť dobre definovaná.

Jednou z možností ako zjednodušiť i-sety v MV uzle je použiť príkaz *simplify*, ktorý využíva binárny logický minimalizátor ESPRESSO-MV [20]. Cieľom takéhoto minimalizátora je nájsť logickú reprezentáciu s minimálnym množstvom implikantov (kociek, napr. napr. $X^{\{0,2\}}Z^{\{0,1\}}$) a literálov (napr. $X^{\{0,2\}}$) so zachovaním funkčnosti. Po vykonaní operácie zjednodušenia je každý i-set nahradený zjednodušenou verziou, ak nové funkcie boli upravené vzhľadom na hodnotu ich použitia. Pre každý uzol, po načítaní iniciálneho súboru, je jeden z i-setov vybraný ako východisková hodnota. Koncept východiskového i-setu je užitočný, pretože sa neprejaví, kým si ho príkaz nevyžiada. Hodnoty uzlov a štatistík siete sú založené len na nevýchodiskových i-setoch. Avšak príkazom *reset_default* docielime vyhľadanie a nastavenie východiskovej hodnoty uzla na základe jeho hodnoty. Jednou z najsilnejších metód zjednodušenia uzla v sieti je príkaz *fullsimp*. Pri vykonávaní tejto funkcie pre viacúrovňové MV siete je automaticky generovaná množina *don't-care* („nestarám sa“, nedefinovaných) stavov. Využívajú sa podmnožiny SDC a ODC množín. MV výpočtové techniky sú využívané na ich mapovanie do lokálnej množiny vstupov každého uzla. Obsahuje možnosť boolovskej substitúcie. Ak trvá alokácia uzla príliš dlho, zjednodušovanie daného uzla sa ukončí. A len miestne SDC je použité na minimalizáciu uzla. Ďalšia, omnoho silnejšia metóda sa nazýva *mfs*, vykonáva tie isté kroky ako *fullsimp*, sú tu avšak isté rozdiely v prispôsobivosti, heuristike a reprezentácii. Ak sa príkaz použije s prepínačom „-k“ nasledovaným menom uzla, ten sa zjednoduší a zobrazí sa Karnaughova mapa obidvoch iniciálnych relácií v uzle a odvodená úplná relácia flexibility, ktorá je obyčajne nedeterministická. Opätovné vykonanie príkazu zobrazí relácie medzi uzlami po minimalizácii úplnej relácie flexibility. Príklad je na obr 1.6.



Obr. 1.6: Príklad práce s príkazom *fullsimp* (bez prepínača *-k*)

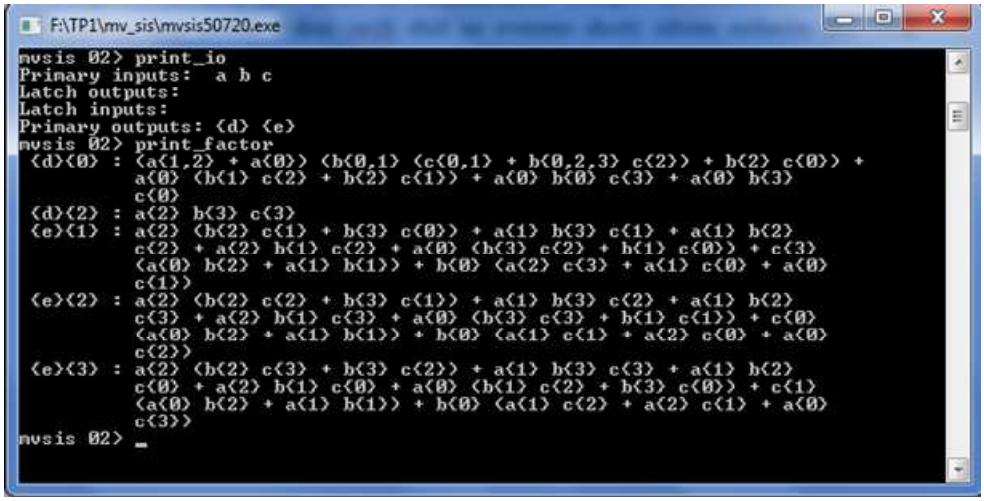
Algebrické metódy zahŕňajú metódy na nájdenie spoločných pod – výrazov, polo – algebrické delenie, dekompozícia MV siete a faktorovanie SOP formy. Technológia nazývaná CST (Co-Singleton Transform) využíva kódovanie MV do binárnej formy vytvorením bijekcie medzi MV a binárnymi výrazmi [4]. Na to využíva algebrické binárne operácie. Relevantné príkazy patriace do tejto oblasti sú *fxu*, ktorý zo všetkých uzlov v sieti vyberie vyhovujúci spoločný faktor a vytvorí nové, staré pritom znovu vyobrazí v súlade s novými, *decomp* uskutočňuje faktoring každého uzla (uzol popisuje algebrický výraz, funkcia, zložený zo súčinu súčtov nazývaných faktory), týmto spôsobom sa vytvoria pomocné uzly. Príkaz *strash* je podobný ako *decomp*, len nahradzuje pôvodnú sieť funkcionálne ekvivalentnou sieťou zloženou len z členov AND a invertorov.

MVSIS obsahuje tiež príkazy pre manipuláciu so sieťou. Príkaz *collapse* konvertuje celú MV sieť aby i-sety pre každý výstup boli len v medziach primárnych vstupov. Potom počet uzlov v sieti sa bude rovnať počtu kombinačných výstupov. *Elimination* uzla pozostáva v jeho odstránení zo siete pri nahradení uzlovej relácie so vstupom obvodu (fanout). Uzol nebude eliminovaný ak jeho hodnota nepresiahne určitú prahovú hodnotu. *Merging* je operácia určená špeciálne pre MV oblasť. Zoznam uzlov zlúči do jedného MV uzla pomocou vytvorenia jedného i-setu pre každú kombináciu hodnôt uzla, ktorý sa má zlúčiť. Na rozdiel

od tohto, príkaz *pair_decode* je automatický, ktorý hľadá premenné na zlúčenie. Tie obsahujú kombinačné vstupy. Automaticky hľadá sľubných kandidátov a ohodnotí ich pre možné šetrenie odhadnutím i-setov, ktoré by mohli byť vytvorené. Táto operácia je podobná operácií *input pairing* v PLA [20] uskutočňovanej v minulosti. Je to jeden z možných spôsobov na vytvorenie prostredných MV uzlov. V opačnom smere pracuje *encode*. Snaží sa nájsť vyhovujúce zakódovanie pre každú MV premennú v sieti s výnimkou primárnych vstupov a výstupov. Vykonáva sa v smere od kombinačných výstupov ku kombinačným vstupom v topologickom poradí. Príkaz *verify* slúži na overenie zhody súboru zadaným ako vstup príkazu s aktuálnou sieťou.

MVSIS prirodzene obsahuje príkazy na zobrazovanie, čítanie a zápis do/zo súboru. Príkazy na čítanie automaticky rozpoznajú rozdiel medzi súborovými formátmi BLIF a BLIF-MV a budú sa správať vhodne. Príkaz *print* zobrazí všetky uzly, SOP formu pre každý i-set. Príkazom *print_factor* sa zobrazí faktorový tvar všetkých uzlov každého nevýchodiskového i-setu. Buď sa zadá zoznam uzlov ako argument alebo „*“ pre všetky uzly ako u SIS. Veľkosť rozsahu každej premennej zobrazíme cez *print_range*, hodnotu uzlov získame zadaním *print_value*, *print_stats* zobrazí štatistiky obsahujúce informácie o názve siete, počte kombinačných vstupov/výstupov, počet uzlov, kociek, a literálov, vstupy a výstupy siete zobrazí príkaz *print_io*.

Príkazy na manipuláciu s názvami premenných sú *chng_name* a *reset_names*, s uzlami *rename* a *namemode* (mód krátkych alebo dlhých mien). Východiskovú hodnotu pre uzol docielime príkazom *default*. Na prácu s nedeterministickými sieťami slúži *dize*, s oknom, čo je vlastne špeciálny zoznam uzlov, *window* s funkčnou reprezentáciou *free*, s klastrovaním *club*, s nastavením globálnych parametrov *set* a *alias* (ktorý vytvára príkazom prezývky). Zoznam všetkých príkazov uvidíme po vykonaní *help*.



Obr. 1.7: Príklad práce s príkazmi print_io a print_factor

Na obr. 1.7 je znázornený príklad výstupu zobrazovacích príkazov zo súboru *full_adder_4.mv* nachádzajúci sa v prílohe A2.

Prehľad skupín príkazov:

Základné (napr. *alias, history, set, undo, echo, ls, source, unset, quit, help, unalias*)

Čítacie (napr. *read_kiss, read_blif, read_pla, read_blif_mv, read_pla_mv*)

Zapisujúce (napr. *write_blif, write_pla, write_blif_mv, write_pla_mv*)

Tlačové (napr. *print_map_stats, print_spec, print_value, show_network, write_gml*)

Časovacie (napr. *fraig_retime, retime*)

Mapujúce (napr. *attach, fpga, map, super, resm*)

Menujúce (napr. *chng_name, rename, reset_name*)

Overujúce (napr. *fraig_verify, reach, sat_verify, verify*)

Vykonávajúce kombinačnú syntézu (napr. *collapse, eliminate, encode, decomp, simplify*)

Vykonávajúce sekvenčnú syntézu (napr. *io_encode, solve, net_product, latch_split*)

Vykonávajúce sekvenčnú syntézu na základe STG (napr. *complete, minimize, test, trim*)

Vykonávajúce syntézu z L jazyka (napr. *dtest, dual, formula, synth*)

Rozličné (napr. *frames, sis, assert, window, dize, reorder, free, split_outputs*)

V časti o MVSIS boli spomenuté základné pojmy, vzťahy a definície. Ako vidíme z vyššie popísaných riadkov je problematika MVSIS naozaj široká. Poskytuje prehľad najzákladnejších príkazov na prácu v programovom prostredí. Na úplné pochopenie by bolo treba sa oboznámiť bližšie s pojmami ako kocka, literál, i-set, fanin, fanout, atď. Čo by značne presahovalo problematiku aplikácií pre vnorené systémy. V našom programe by sa dala použiť časť venujúca sa prácou so súborovým systémom BLIF-MV, ktorý je najrozšírenejši.

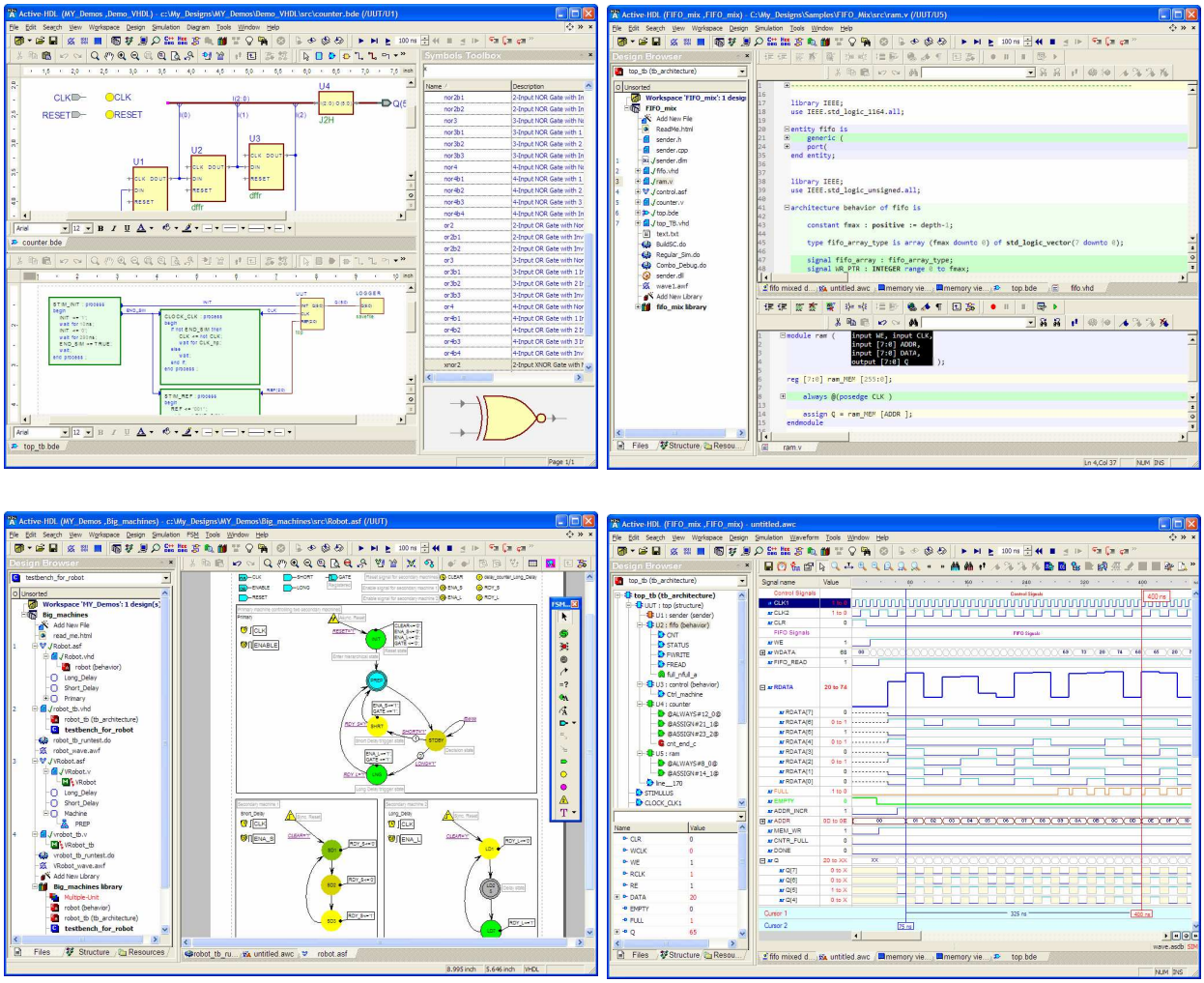
1.3 Active HDL

Je to nástroj založený na FPGA[7]. Umožňuje grafický návrh digitálnych obvodov, nastavenie jednotlivých parametrov jednotlivých častí. Rovnako podporuje aj voľné programovanie pri procese analýzy alebo syntézy v plain texte. Nástroj umožňuje aj návrh a generovanie stavových diagramov ako aj diagramov výstupných priebehov. Je to výborný nástroj pre návrh a vyhodnotenie digitálnych systémov.

Možnosti:

- ✓ grafický návrh a editácia návrhu
- ✓ code2grafic <-> grafic2code
- ✓ import/export do štandardných formátov
- ✓ podpora viacerých jazykov ako VHDL, Verilog, System Verilog, System C
- ✓ automatická testbench generácia

- ✓ pokročilejšie debagovanie
- ✓ možnosť simulácie pomocou matlabu/simulinku
- ✓ PCB design interface
- ✓ HTML a pdf generátor dokumentácie



Obr. 1.8: Okná programu Acvite HDL [8]

1.4 Log

Log[21] je jeden z nástrojom umožňujúcich návrh digitálnych systémov a ich simuláciu v reálnom čase. Grafické prostredie však nie je priateľské pre používateľa, treba si na neho dlhšie zvykať a ťažšie sa hľadajú samotné logické členy. Rovnako aj výstup nie je do štandardných formátov, ale je zrozumiteľný len pre log. Neumožňuje programovanie v plain texte. Neumožňuje zobrazenie výsledkov simulácie v časovej osi vo forme grafov. Nástroj nie je vhodný pre hlbšiu prácu.

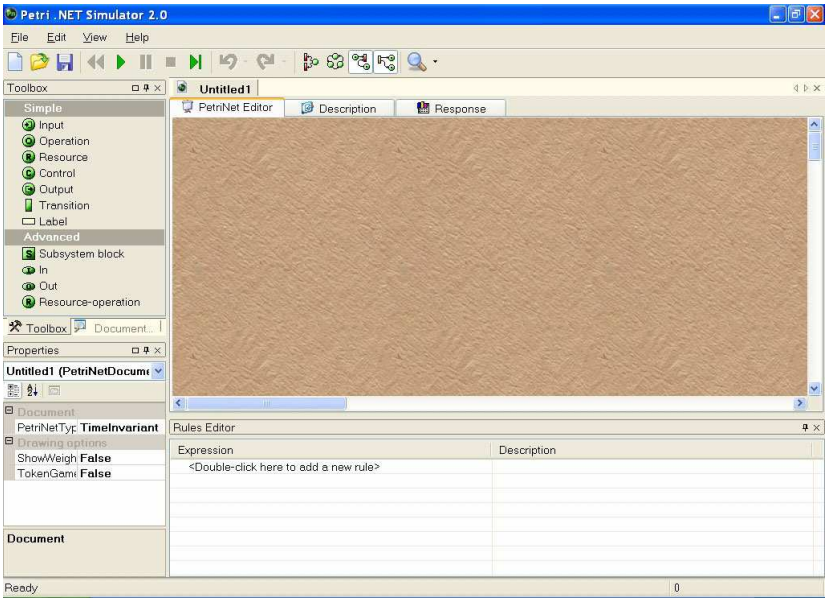
1.5 Petri .NET simulátor

Je to aplikácia pre navrhnutie, nakreslenie a simuláciu Petriho sietí[9]. Je navrhnutý pre modelovanie, analýzu a simuláciu flexibilných výrobných systémov, ale môže byť taktiež použitý pre iné systémy.

Základné vlastnosti:

- jednoduché kreslenie Petriho sieti. Je možné nakresliť mnoho tvarov PN.
- Objekty PN (miesta a prechody) môžu byť spájané do podsystémov. PN je tak ľahšie porozumieť a udržovať
- Simulácia časových PN
- Simulácia môže zobraziť animácie značiek a čas spracovania
- Simulácia môže byť spustená v reálnom čase, alebo je ju možné zobraziť vo forme tabuľky alebo ako oscilogram. Oscilogram sa dá použiť pri výbere časového intervalu, ktorý bude použitý pre vykonanie analýzy siete
- Použitie kontrolného algoritmu cez editor pravidiel na PN model aby sa vytvoril stabilný systém

- Export modelu PN alebo oscilogramu do zdokonaleného metafile formátu, alebo jednoducho preniesť model do aplikácie tretej strany cez clipboard. Exportovať ako tabuľku, výhodne pri písaní výsledkov simulácie.
-



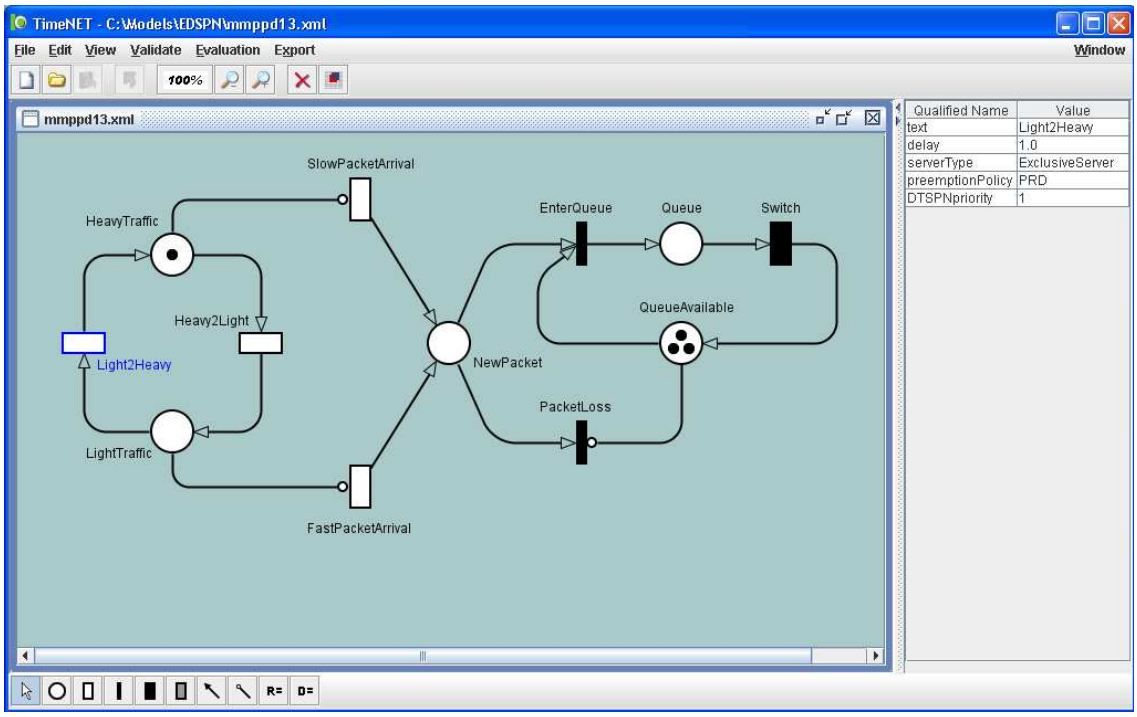
Obr. 1.9: Okno programu Petri Net Simulator [9]

1.6 TimeNET

Softwarový balíček TimeNET (vo verzii 4), grafický a interaktívny nástroj premodelovanie stochastických PN a stochastických farebných PN. TimeNET je vyvinutý Real-Time Systems a Robotics Technickej Univerzity Berlin. Projekt bol motivovaný potrebou silného softvéru pre efektívne hodnotenie časovaných Petriho sieti s ľubovoľným časom spúšťania prechodov. TimeNET a jeho predchodca DSPNexpress boli ovplyvnené skúsenosťami s inými dobre známymi nástrojmi pre PN ako GreatSPN a SPNP [10].

Základné vylepšenia vo verzii 4.0:

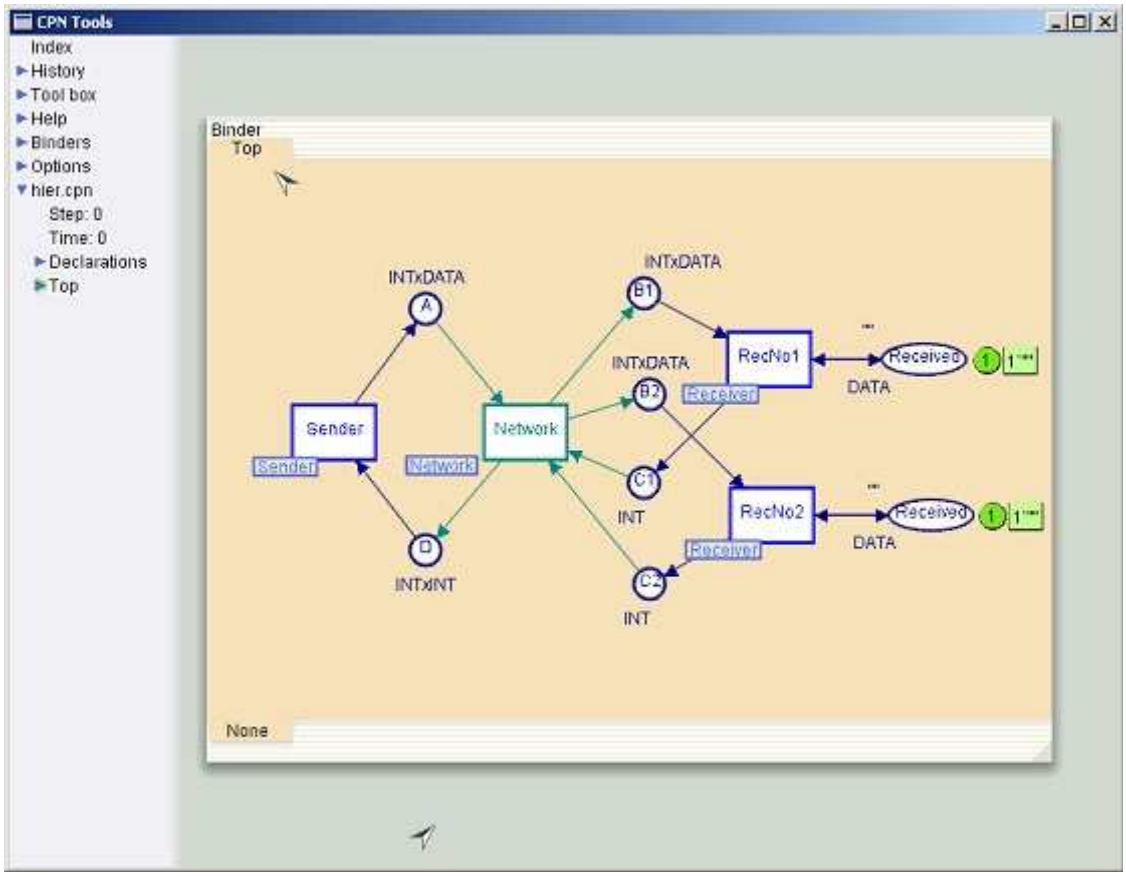
- všeobecné grafické rozhranie v prostredí JAVA založené na reprezentácií XML sieťovej triedy, ľahko rozšíriteľné pre väčšinu aplikácií založených na princípe grafu
- užívateľské prostredie a algoritmus zhodnotenia bežia pod operačným systémom Windows® a Linux
- modelovania a simulácia stochastických farebných PN
- grafické zobrazenie výsledku simulácie stochastických farebných PN
- nezávislé komponenty pred modelovanie, simuláciu, analýzu a výsledkov čo umožňuje GUI byť spustené na inom počítači ako modul analýzy.



Obr. 1.10: Okno programu TimeNet [11]

1.7 CPN Tools

CPN Tools je nástroj pre editáciu a analýzy farebných PN. GUI je založené na pokročilých interaktívnych technikách [12]. Aplikácia používa pre informovanie používateľa kontextové chybové hlásenia a indikuje závislosti medzi jednotlivými elementmi PN. Nástroj poskytuje kontrolu syntaxe a generácie kódu, čo sa vykonáva počas konštruovania siete. Rýchly simulátor zvládne časované aj nečasovaná PN. Môžu byť vygenerované a analyzované úplné alebo čiastočné vyhodnotenia PN. Štandardné vyhodnotenie obsahuje informácie ako ohraničenie, živosť.



Obr. 1.11: Okno programu CPN Tools [12]

1.8 BLIF

Cieľom BLIF-u [17] je opísať hierarchický logický obvod v textovej podobe. Obvod je ľubovoľná kombinačná alebo sekvenčná sieť logických funkcií. Obvod môže byť braný ako orientovaný graf kombinačných logických uzlov a sekvenčných logických elementov.

Každý uzol opisuje dvojúrovňová, jednovýstupová logická funkcia. Každá spätná väzba musí byť ošetrená. Každá sieť (alebo signál) má iba jeden parameter, a signál, alebo brána, ktorá riadi signál môže byť pomenovaná len jednoznačne.

V nasledujúcom opise < > označujú povinné a [ ] označujú nepovinné parametre.

1.8.1 Modely

Model je zjednodušený hierarchický obvod. BLIF súbor môže obsahovať veľa modelov a referencie na modely opísané v iných BLIF súboroch. Model sa deklaruje nasledujúco:

.model <dekl-model-názov>
.inputs <dekl-vstup-zoznam>
.outputs <dekl-výstup-zoznam>
.clock <dekl-hodiny-zoznam>
<príkaz>
.
.
.
<príkaz>
.end

dekl-model-názov – je názov modelu

dekl-vstup-zoznam – je to zoznam reťazcov oddelených „bielymi znakmi“ (a ukončený znakom „konca riadku“), ktorý obsahuje formálny opis vstupných uzlov pre daný model. Ak je to prvý alebo jediný model, potom tieto uzly sú brané ako hlavné vstupy pre daný obvod. Je povolené zapísať viacero *.input* riadkov, a potom všetky vstupy sú spojené do jedného zoznamu.

dekl-výstup-zoznam - je to zoznam reťazcov oddelených „bielymi znakmi“ (a ukončený znakom „konca riadku“), ktorý obsahuje formálny opis výstupných uzlov pre daný model. Ak je to prvý alebo jediný model, potom tieto uzly sú brané ako hlavné výstupy pre daný obvod.

Je povolené zapísať viacero *.output* riadkov, a potom všetky výstupy sú spojené do jedného zoznamu.

dekl-hodiny-zoznam - je to zoznam reťazcov oddelených „bielymi znakmi“ (a ukončený znakom „konca riadku“), ktorý obsahuje hodinové signály pre daný model. Je povolené zapísať viacero .clock riadkov, a potom všetky hodinové signály sú spojené do jedného zoznamu.

Príkaz môže byť jeden z nasledujúcich:

<logické-hradlo> *<generický-zámok>* *<knižničné-hradlo>*

<referencia-na-model> *<referencia-na -podsúbor>* *<opis-fsm>*

<obmedzenie-hodin> *<obmedzenie-oneskorenia>*

BLIF dovoľuje, aby parametre .model, *.inputs*, *.oputputs*, *.clock* a *.end* boli nepovinné. Ak nie je špecifikovaný .model, potom *dekl-model-názov* bude mať hodnotu názvu BLIF súboru. Nie je povolené použiť rovnaký *dekl-model-názov* reťazec pre rôzne modely. Ak nie sú zadefinované vstupy príkazom *.inputs*, tak sa dajú odvodiť zo signálov, ktoré nie sú výstupmi žiadnych iných logických blokov. Podobne signály *.outputs* môžu byť odvodené zo signálov, ktoré nie sú vstupmi žiadnych iných logických blokov. Ak nie je uvedený žiadny *.clock* signál, tak potom ide čisto o kombinačný obvod, a .end je vsunutý na koniec, alebo pred začiatok ďalšieho modelu.

Prvý model , ktorý je uvedený v hlavnom BLIF súbore je ten, ktorý je vrátený používateľovi. Signál *.clock* a príkazy *<obmedzenie-hodin>* a *<obmedzenie-oneskorenia>* sú brané do úvahy len v prvom modeli. Všetky ostatné modely môžu byť vnorené v prvom modeli pomocou príkazu *<referencia-na-model>*.

Znak „#“ (mriežka) označuje začiatok komentára, ktorý začína týmto znakom a končí sa na konci riadku. Tento znak sa nemôže použiť v názvoch signálov. Ak je znak „\“ lomítko posledným znakom na riadku, tak potom príkaz pokračuje na ďalšom riadku. Za týmto znakom nemôže byť už žiadny iný znak.

Príklad:

```
.model jednoduchy
.inputs a b
.outputs c
.names a b c    # .names opisany neskor
11 1
.end

# nepomenovany model
.names a b \
c                       # '\' je tu len pre ukazku
11 1
```

Obidva modely opisujú ten istý obvod.

1.8.2 Logické hradlá

<*logické-hradlo*> priraďuje k signálu logickú funkciu v modeli. Táto funkcia môže byť použitá ako vstup pre ďalšie hradlo. <*logické-hradlo*> je definované nasledovne:

.names <vstup-1> <vstup-2> ... <vstup-n> <výstup>
<pravdivostná-tabuľka>

<*výstup*> - reťazec udávajúci názov logického hradla

<*vstup-1*> <*vstup-2*> ... <*vstup-n*> - reťazce definujúce vstupy hradla

<*pravdivostná-tabuľka*> - formálne udáva n-vstupový, 1-výstupový PLA opis logickej funkcie daného hradla. {0,1,-} je použité v n-bit širokej „vstupnej rovine“ a {0,1} je použité v 1-bit širokej „výstupnej rovine“. Stav „on“ je reprezentovaný 1-kou a stav „off“ je reprezentovaný 0-ou na výstupe.

Jednoduché logické hradlo s jednoduchou pravdivostnou tabuľkou:

```
.names v3 v6 j u78 v13.15
1--0 1
-1-1 1
0-11 1
```

Stav „-“ označuje nepoužitý signál. Všetky prvky na vstupe sú najprv AND-ované a potom všetky riadky sú OR-ované. Tým dostaneme posledný stĺpec, ktorý označuje výstup. Medzera medzi vstupom a výstupom je potrebná.

Preklad tohto logického hradla do sum-of-products opisu by bol nasledovný:

v13.15 = (v3 u78') + (v6 u78) + (v3' j u78)

Priradenie konštanty 0 k logickému hradlu „j“ vyzerá nasledovne:

.names j

Priradenie konštanty 1 k logickému hradlu „j“ vyzerá nasledovne:

.names j
1

Reťazec <*výstup*> jedného hradla môže byť použitý ako vstup pre iné logické hradlo ešte pred samotnou definíciou druhého hradla.

1.8.3 Vonkajšie Don´t Cares

Vonkajšie Don´t Cares („Nestarám sa“) označuje samostatnú sieť v modeli, a je špecifikovaná na konci modelu. Každá Don´t Cares funkcia, ktorá je špecifikovaná príkazom *.names* musí byť priradená k primárnemu výstupu hlavného modelu a špecifikovaná ako funkcia primárnych vstupov hlavného modelu (hierarchická špecifikácia Don´t Cares nie je podporovaná).

Don´t Cares sú špecifikované nasledovne:

.exdc
.names <vstup-1> <vstup-2> ... <vstup-n> <výstup>
<pravdivostná-tabuľka>

.exdc – označuje, že nasledujúci príkaz *.names* bude patriť k sieti vonkajších Don´t Cares

<*výstup*> - označuje hlavný výstup pre ktoré platia podmienky Don´t Cares

<*vstup-1*> <*vstup-2*> ... <*vstup-n*> - sú mená hlavných vstupov pre ktoré sú vyjadrené podmienky Don´t Cares.

<*pravdivostná-tabuľka*> - označuje n-vstupový a 1 výstupový PLA opis logických funkcií vyhovujúcich podmienkam Don´t Cares na výstupe.

Príklad na obvod s Don´t Cares:

.model a
.inputs x y
.outputs j
.subckt b x=x y=y j=j
.exdc
.names x j
1 1
.end

.model b
.inputs x y
.outputs j
.names x y j
11 1
.end

1.8.4 Preklápacie obvody a zámky

<*generický-zámok*> slúži na vytvorenie oneskorenia v obvode. Reprezentuje 1 bit pamäte alebo stavovú informáciu. <*generický-zámok*> môže byť využitý na vytvorenie ľubovoľného zámku alebo preklápacieho obvodu.

<*generický-zámok*> je deklarovaný nasledovne:

.latch <vstup> <výstup> [<typ> <ovládanie>] [<inic-hod>]

vstup – dátový vstup do zámku

výstup – výstup zámku

typ – má jednu hodnotu z množiny {fe, re, ah, al, as}, kde fe znamená „falling edge“ (dobežná hrana), re znamená „rising edge“ (nábežná hrana), ah znamená „active high“ (aktívne vysoko), al znamená „active low“ (aktívne nízko) a as znamená „asynchronous“ (asynchrónny).

ovládanie – je hodinový signál pre zámok. Môže to byť signál .clock modelu, výstup ľubovoľnej funkcie v modeli alebo slovo „NIL“ pre žiadny hodinový signál.

inic-hod – je začiatočný stav zámku, jeho hodnota môže byť {0,1,2,3}. 2 znamená „Don´t care“ (nestaráme sa) a 3 znamená „neznámy“, „nešpecifikovaný“.

Ak zámok nemá definovaný hodinový signál, je predpoklad, že zámok je kontrolovaný jedným globálnym hodinovým signálom. Správanie tohto signálu môže byť interpretované rôzne, záleží na algoritme programu ktorý daný model načítal. Preto by si mal používateľ dávať pozor na rôznu interpretáciu pri nešpecifikovanom hodinovom signáli.

Všetky spätné väzby musia ísť cez generický zámok. Čisto kombinačno-logické cykly nie sú povolené.

Príklady:

```
.inputs d                    # časovaný preklápací obvod
.output q
.clock c
.latch d q re c 0
.end
```

```
.inputs in                   # veľmi jednoduchý sekvenčný obvod
.outputs out
.latch out in 0
.names in out
0 1
.end
```

1.8.5 Knižničné hradlá

Príkaz *<knižničné-hradlo>* vytvorí inštanciu technologicky závislého logického hradla a priradí ho k uzlu, ktorý reprezentuje výstup logického hradla. Logická funkcia hradla a jeho známe technologicky závislé oneskorenie, vedenie, atď. sú zadefinované v s príkazom *<knižničné-hradlo>*.

<knižničné-hradlo> je jedno z nasledovných:

.gate <názov> <formálny-aktuálny-zoznam>
.mlatch <názov> <formálny-aktuálny-zoznam> <ovládanie> [<inic-hod>]

názov – je názov .gate alebo *.mlatch* inštancie. Hradlo alebo zámok s daným názvom musí byť prítomný v danej pracovnej knižnici.

formálny-aktuálny-zoznam – je namapovanie formálnych parametrov knižničného hradla na aktuálne signály v danom modeli. Formát zápisu je nasledovný:

formal1=aktual1 formal2=aktual2 ...

Všetky formálne parametre hradla musia byť špecifikované v *formálny-aktuálny-zoznam*-e a výstup musí byť na poslednom mieste.

ovládanie – je hodinový signál pre *.mlatch*, môže byť buď signál *.clock* daného modelu, výstup nejakej funkcie alebo slovo „NIL“ pre žiadny hodinový signál.

inic-hod – je začiatočný stav zámku, jeho hodnota môže byť {0,1,2,3}. 2 znamená „Don´t care“ (nestaráme sa) a 3 znamená „neznámy“, „nešpecifikovaný“.

.gate sa odkazuje na dvojúrovňovú reprezentáciu ľubovoľného jednovýstupového hradla z knižnice. .gate sa javí ako technologicky závislá interpretácia jediného logického hradla.

.mlatch sa odkazuje na zámok v knižnici. *.mlatch* sa javí ako technologicky závislá interpretácia jediného generického zámku alebo ako logické hradlo slúžiace na dátový vstup generického zámku.

.gate a *.mlatch* sú použité na opis obvodov ktoré používajú špecifickú knižnicu štandardných logických funkcií a ich technologicky závislé vlastnosti. Knižničné hradlo musí byť načítané skôr, než sa zavolá príkaz *.mlatch* alebo *.gate*.

Názov zodpovedá danému hradlu alebo zámku v knižnici. Názvy „nand2“, „inv“ a „jk_rising_edge“ v nasledujúcich príkladoch sú opisné názvy hradiel v knižnici. [18 - 19]

```
.inputs v1 v2
.outputs j
.gate nand2 A=v1 B=v2 O=x          # zadané: formálne parametre: A, B, O
.gate inv A=x O=j                  # zadané: formálne parametre: A & O
.end
```

Ten istý príklad môže byť opísaný aj v technologicky nezávislom tvare:

```
.inputs v1 v2
.outputs j
.names v1 v2 x
0- 1
-0 1
.names x j
0 1
.end
```

Podobne:

```
.inputs j kbar
.outputs out
.clock clk
.mlatch jk_rising_edge J=j K=k Q=q clk 1  # formálne parametre:J, K, Q
.names q out
0 1
.names kbar k
0 1
.end
```

A v technologicky nezávislom tvare:

```
.inputs j kbar
.outputs out
.clock clk
.latch temp q re clk 1  # .latch - zámok
.names j k q temp       # .names vstup pre .latch
-01 1
1-0 1
.names q out
0 1
.names kbar k
0 1
.end
```

1.9 PNML

Ľudia zaoberajúci sa problematikou Petriho sietí dlho hľadali prostriedky vhodné pre výmenu modelov v presne stanovenom a dohodnutom formáte. Rozmýšľali nad tým, aby tento formát alebo prostriedky boli definované v uznávanom štandarde. Nakoniec definovali a zaviedli štandard ISO/IEC 15909 [13].

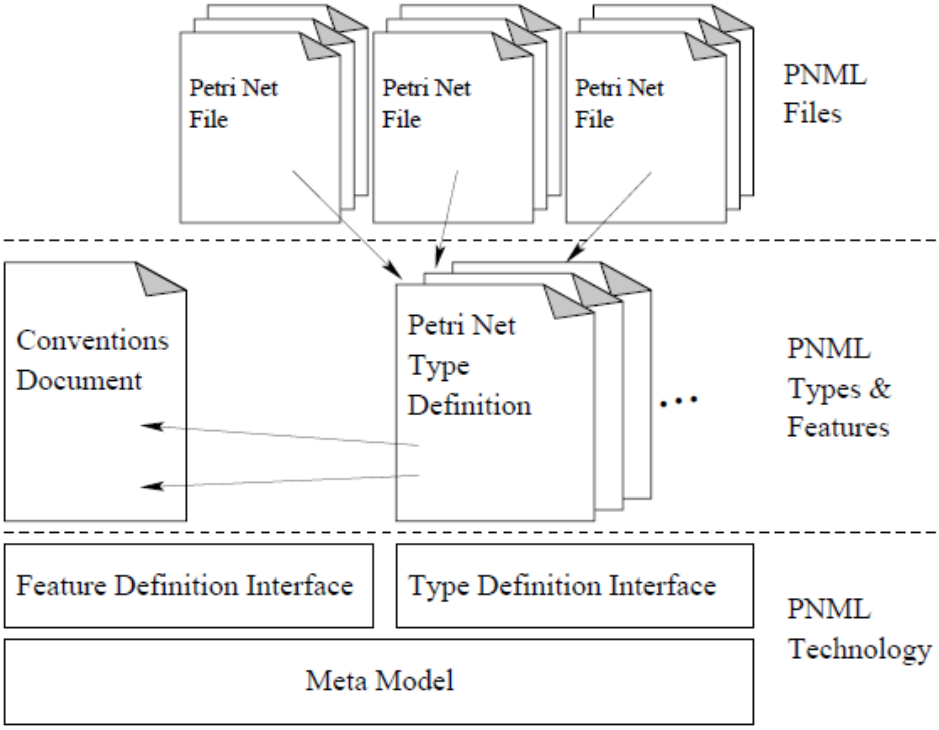
PNML je založený na štandardnom formáte XML. Pôvodne malo PNML slúžiť len pre JAVU, ale dopyt bol taký výrazný, že bola snaha o štandardizáciu Petriho sietí práve vo formáte XML. Veľmi dôležitým aspektom PNML je otvorenosť. Rozlišuje medzi všeobecnými črtami Petriho sietí ako aj medzi presne špecifikovanými črtami Petriho sietí.

Okrem toho PNML poskytuje zdieľanie špecifických vlastností, kedy môže určitú vlastnosť zdieľať viacero objektov súčasne.

Každá Petriho sieť sa skladá z kolekcie predefinovaných prvkov, avšak je možnosť vytvoriť si nové typy, ktoré budú využívané pre pokročilejšiu stavbu Petriho sietí. To sa vytvára pomocou DTD alebo XML schém[14].

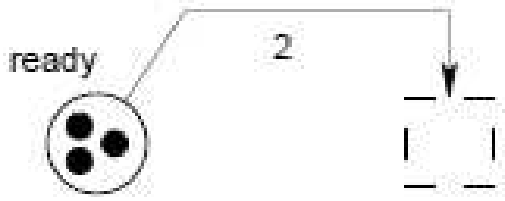
Celkové usporiadanie PNML súborov sa skladá z priechodov, miest a oblúkov, pričom každý jeden je jednoznačne definovaný pomocou špecifického atribútu. Všetky značky môžu byť vybavené ďalšími značkami v prípade potreby. Môže to byť napríklad Initial Marking alebo meno.

Základná syntax rovnako ponúka aj tagy pre prácu s grafickým režimom, kedy chceme definovať prvok ako uzol, jeho umiestnenie a podobne. Pre ďalšie špecifické vlastnosti si môžeme definovať aj ďalšie tagy, ktoré budú vykonávať požadovanú funkcionalitu. Avšak ak vytvárame nové funkcie, musia byť definované v globálnom dokumente, pre zachovanie konzistencie.



Obr. 1.12: Štruktúra PNML[14]

Príklady:



Obr. 1.13: Príklad zapojenia [14]

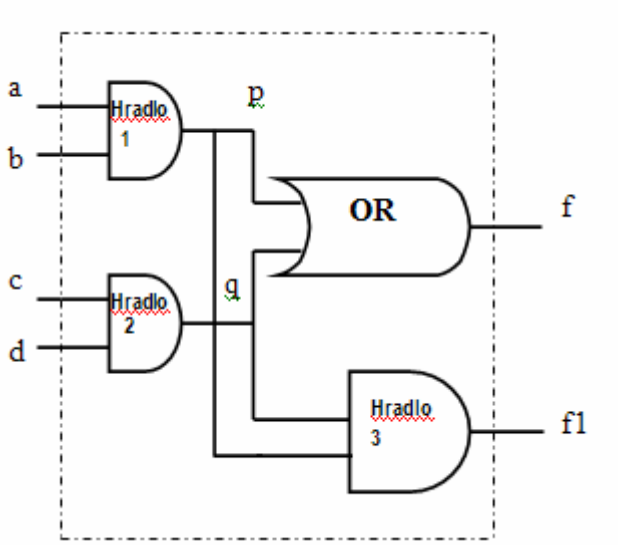
```
<pnml xmlns="http://www.example.org/pnml">
	<net id="n1" type="http://www.example.org/pnml/PTNet">
		<name>
			<text>An example P/T-net</text>
		</name>
		<place id="p1">
			<graphics>
				<position x="20" y="20"/>
			</graphics>
			<name>
				<text>ready</text>
				<graphics>
					<offset x="-10" y="-8"/>
				</graphics>
			</name>
			<initialMarking>
				<text>3</text>
			</initialMarking>
		</place>
		<transition id="t1">
			<graphics>
				<position x="60" y="20"/>
			</graphics>
			<toolspecific tool="PN4all" version="0.1">
				<hidden/>
			</toolspecific>
			</transition>
			<arc id="a1" source="p1" target="t1">
			<graphics>
				<position x="30" y="5"/>
				<position x="60" y="5"/>
			</graphics>
			<inscription>
				<text>2</text>
				<graphics>
					<offset x="15" y="-2"/>
				</graphics>
			</inscription>
		</arc>
	</net>
</pnml>
```

1.10 Ďalšie súborové formáty

Existujú aj ďalšie súborové formáty, ktoré slúžia na zápis logických obvodov, prípadne konečných stavových automatov. Medzi tieto formáty patria napríklad:

1. PLA
2. EQN
3. KISS

Je daný nasledovný obvod na obr. 1.14. Ďalej bude uvedený textový zápis tohto obvodu vo vyššie uvedených formátoch.



Obr. 1.14: Jednoduchý obvod

1.10.1 PLA

Počet vstupov je 4 (a,b,c,d) – sú definované pomocou príkazu *.i.*

Počet výstupov je 2 (f, f1) - sú definované pomocou príkazu *.o.*

Vo formáte PLA je to napísané nasledovne:

```
.i 4
.o 2
```

Pomenovanie vstupov a výstupov:

1. Zadefinujú sa názvy signálov na vstupe. Pre daný obvod sú tieto názvy: a, b, c, d
2. Zadefinujú sa názvy signálov na výstupe. Pre daný obvod sú tieto názvy: f, f1

```
.i  4
.o 2
.ilb a b c d
.ob f f1
```

Pravdivostná tabuľka

Po zadefinovaní vstupov a výstupov sa špecifikuje pravdivostná tabuľka daného obvodu. Počet riadkov pravdivostnej tabuľky je reprezentovaný príkazom *.p* v PLA súbore.

A	B	C	D	F	F1
1	1	-	-	1	0
-	-	1	1	1	0
1	1	1	1	1	1

Tab. 1: Pravdivostná tabuľka

Tabuľka zapísaná vo formáte PLA vyzerá nasledovne:

```
.i  4
.o 2
.ilb a b c d
.ob f f1
.p 3
 11-- 10
 --11 10
1111 11
```

Ukončenie súboru

Koniec súboru označuje príkaz *.e*.

1.10.2 EQN

EQN formát je jeden z najjednoduchších formátov, pretože priamo vyjadruje logické funkcie jednotlivých hradiel v danom obvode.

Počet vstupov je 4 (a,b,c,d) – sú definované pomocou príkazu *INORDER*.

Počet výstupov je 2 (f, f1) - sú definované pomocou príkazu *OUTORDER*.

Vo formáte EQN je to napísané nasledovne:

```
INORDER = a b c d;
OUTORDER = f  f1;
```

Špecifikácia signálov medzi hradlami

Signály medzi hradlami sú špecifikované nasledovným spôsobom:

[názov signálu] = logický výraz;

Napríklad pre daný obvod na obr. 1.14, signál p môže byť zapísaný ako:

*[p] = a * b;*

```
INORDER = a b c d;
OUTORDER = f  f1;
[p] = a * b;
[q] = c* d;
```

Špecifikácia výstupov

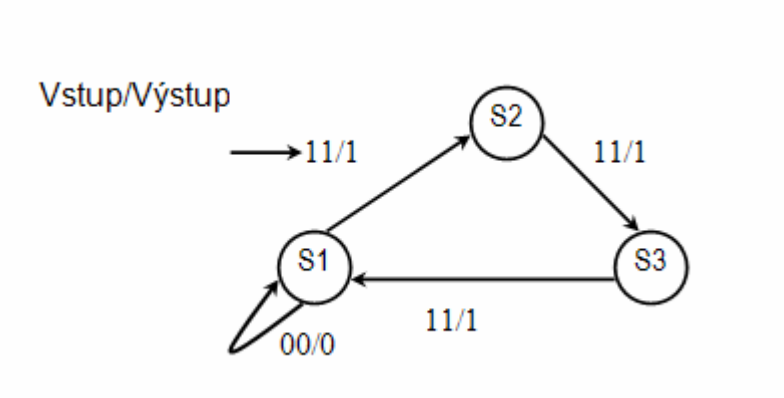
Výstupy obvodu môžu byť vyjadrené ako logické funkcie pozostávajúce zo signálov medzi hradlami alebo zo vstupných signálov. Napríklad výstupy f a f1 môžu byť vyjadrené nasledovne:

```
INORDER = a b c d;
OUTORDER = f  f1;
[p] = a * b;
[q] = c* d;
f= [p] + [q]
f1= [p]*[q];
```

1.10.3 KISS

KISS je používaný na minimalizáciu konečných stavových automatov.

Na obr. 1.15 je znázornený jednoduchý stavový automat. Jeho zápis vo formáte KISS je uvedený nižšie.



Obr. 1.15: Stavový automat

Počet vstupov je 2 (a,b,) – sú definované pomocou príkazu *.i.*

Počet výstupov je 1 (f) - je definované pomocou príkazu *.o.*

Počet stavov je 3 - definované pomocou príkazu *.s.*

Počet riadkov v tabuľke je 3 - definované pomocou príkazu *.p.*

Vo formáte KISS je to napísané nasledovne:

```
.i 2
.o 1
.s 3
.p 4
```

Tabuľka stavov

Po zadaní vstupov a výstup sa vytvorí tabuľka stavov – viď tab. 2.

Vstup	**SS**	**NS**	**Výstup**
00	S1	S1	0
11	S1	S2	1
11	S2	S3	1
11	S3	S1	1

SS – Súčasný stav

NS – Nasledujúci stav

Tab. 2: Tabuľka stavov

Ukončenie súboru

Koniec súboru označuje príkaz *.e*.

Výsledný KISS súbor vyzerá potom nasledovne:

```
.i  2
.o 1
.s 3
.p 4
00 s1 s1 0
11 s1 s2 1
11 s2 s3 1
11 s3 s1 1
.e
```

1.11 Binárny rozhodovací diagram

BDD [15] je acyklický graf pre opis funkcie, pričom uzly grafu sú premenné danej funkcie, a z každého uzla grafu vystupujú len dve hrany ohodnotené hodnotami 0, 1. Modeluje všetky funkčné cesty od vrcholu grafu k jednotlivým listom grafu, kde vrchol stromu je samotná funkcia a listy grafu sú premenné funkcie.

BDD vytvárajú možnosť opísania funkcií jednotlivých modulov a celého číslicového systému, ktoré je nezávislé od implementácie systému.

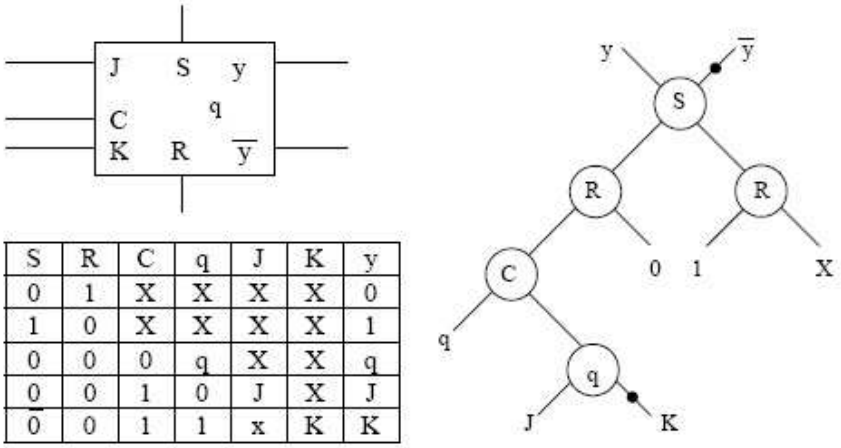
Generovanie testu je prechodom cez všetky cesty grafu, ktorý zabezpečí, že sa otestujú všetky špecifikácie funkcie.

Zovšeobecnenie binárnych rozhodovacích diagramov – opis pre funkčné bloky, pričom každý uzol môže byť reprezentovaný opäť binárnym rozhodovacím diagramom. Využitie pri hierarchickom generovaní testov.

Spôsob utvorenia binárneho rozhodovacieho stromu:

- z pravdivostnej tabuľky funkcie,
- aplikáciou klasickej Shannonovej expanznej rovnice.

Na nasledujúcom obrázku je znázornený príklad JK preklápacieho obvodu.



S	R	C	q	J	K	y
0	1	X	X	X	X	0
1	0	X	X	X	X	1
0	0	0	q	X	X	q
0	0	1	0	J	X	J
0	0	1	1	x	K	K

Obr. 1.16: Príklad JK preklápacieho obvodu

Kde, X označuje nelegálnu operáciu, v ktorej výstupná hodnota nemôže byť určená a označenie „bodky“ na hrane predstavuje negovanú premennú. Prechod cez binárny diagram implikuje vlastne nastavenie premenných na určitú hodnotu pozdĺž vybranej cesty, ktoré definuje určitý režim systému.

1.12 BDS

Program BDS je založený na novej technike rozkladu binárne rozhodujúcich diagramoch (BDD). Podporuje všetky typy rozkladu štruktúr, vrátane AND, OR, XOR, a MUX, tak algebricky ako aj logicky. Ako výsledok, metóda je veľmi efektívna v syntetizovaní funkcií AND/OR alebo XOR. Má tiež schopnosť zvládnuť veľké obvody, pretože ponúka BDD rozklad v rozdelenom logickom sieťovom prostredí. Výsledky experimentov ukazujú, že na BDD založený logický rozklad je sľubnou alternatívou k existujúcim logickým prístupom optimalizácie, tým pádom sa logická syntéza zrýchli. [16]

1.12.1 Implementácia systému BDS

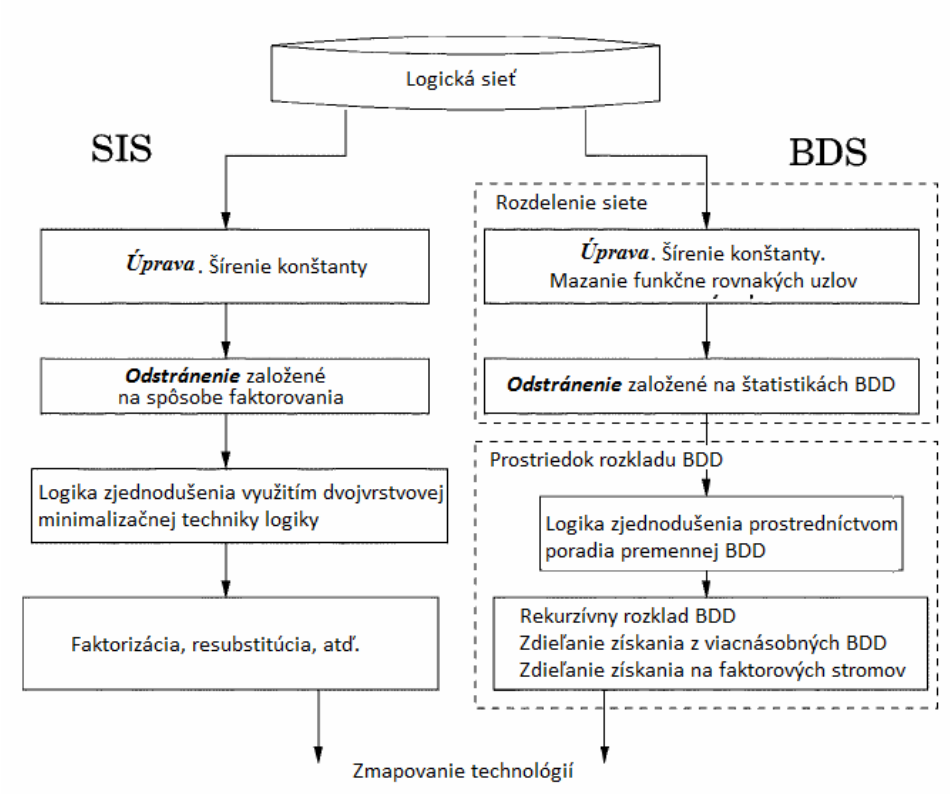
Aby bolo možné zvládnuť ľubovoľne veľké obvody, systém sa rozkladá na viac logických sieťových prostredí.

1.12.2 Syntéza rozkladu

BDS prijíma celkový tok syntézy od SIS. Základným rozdielom medzi systémami SIS a BDS je v tom, ako oni reprezentujú logické uzly a vykonávajú jednotlivé optimalizačné postupy. SIS pracuje na algebrickej reprezentácií celej logickej siete, iteratívnom rozložení algebraických výrazov, zabránení kolapsu uzla a zjednodušením logiky. BDS najprv rozloží sieť do množiny uzlov, ktoré každé predstavuje miestny BDD, až potom vykoná rozklad BDD.

Všetky nasledujúce postupy sú vykonávané na miestnych BDD, využitím rozkladacích algoritmov nastavených pre BDD. Prvým krokom v syntéze použitého toku je odstránenie

pôvodnej redundancie logickej siete. Pokiaľ v procese nie je žiadna skutočná optimalizácia logiky, treba sieť pripraviť na ďalší rozklad. Všetky funkčne rovnaké uzly sú tiež odstránené z logickej siete. Odstránenie duplicitných uzlov v tejto skorej fáze zlepšuje prevádzkovú zložitosť BDS nad tradičnými prístupmi. [16]



Obr. 1.17: Porovnanie programu SIS s programom BDS

1.12.3 Rozdelenie siete podľa odstránených uzlov

Uplatnením logickej optimalizácie celej logickej siete pomocou globálnych BDD, nemusí byť praktické pre veľké plány. Naopak, uplatnením logickej optimalizácie na úplne miestne zastúpenie nemusí taktiež fungovať, pretože môže opustiť významné množstvo redundancií v sieti. Môže nastať čiastočné zrútenie logickej siete do súboru superuzlov. Každý superuzol potom môže byť reprezentovaný ako miestny BDD. Čiastočné zrútenie je kritické pre logickú syntézu systému. Pomáha zmazať logickú redundanciu spôsobenú napríklad miestnou zmenou uzlov v BDD. Čiastočné zrútenie môže byť implementované za cieľom odstránenia procedúry, ktorý sa snaží udržať správnu granularitu logickej siete. Správne navrhnutá

eliminačná schéma poskytuje dobrý východiskový bod pre algoritmy logickej optimalizácie. Dva prístupy boli navrhnuté pre odstránenie procedúry pomocou BDD. Prvý z nich je založený na postupnom eliminovaní. Druhý prístup je založený na opakovanom odstránení. [16]

1.12.4 Stroj rozkladu BDD

Najprv je BDD použitý na zoradenie premenných. To slúži ako prostriedok na dosiahnutie počiatočného zjednodušenia logiky, čo je dobrým východiskovým bodom k ďalšiemu rozkladu logiky. Rozklad daného BDD sa skladá z dvoch hlavných častí:

1. opakujúci sa rozklad BDD, kde je veľký BDD rekurzívne rozložený do menších častí
2. konštrukcia a spracovanie faktorových stromov. Faktorové stromy sú postavené spolu s rozkladom BDD a slúžia na uchovávanie záznamu výsledku rozkladu.

Opakujúci sa rozklad BDD je proces hľadania pre čo najúčinnejší rozklad. BDD dominátory sú empiricky (založený na predchádzajúcich skúsenostiach) usporiadané z hľadiska výslednej efektívnosti rozkladu nasledovne:

1. jednoduchý dominátory (1 -, 0 - a x-dominátor)
2. funkčné MUX
3. univerzálny dominator
4. univerzálny x-dominátor

Ak všetky vyhľadávania zlyhajú, BDD je rozložený pomocou obyčajného MUX s ohľadom na vrcholovú premennú v BDD. V praxi v tento posledný krok je dosiahnutý iba zriedka. Je potrebné zaistiť, aby BDD bolo rozložené, keď všetky ostatné pokusy zlyhajú. [16]

1.12.5 BDS-pga 2.0

BDS-pga je účinný program určený na logickú syntézu a optimalizáciu BDD pre LUT (pravdivostná tabuľka) založené FPGA. Je založený na programe BDS z University of Massachusetts Amherst. BDS-pga používa BDD manipulačné funkcie z balíka CUDD z University of Colorado Boulder. Dizajn obvodu sa načítava vo formáte BLIF a optimalizuje priestor a oneskorenie. Konečný syntetizovaný obvod zo súboru netlist blif z BDS-PGA môže byť pretransformovaný na pravdivostnú tabuľku FPGA pomocou FlowMap tech mapovacieho nástroja z UCLA VLSI CAD Lab. BDS-pga program je určený pre akademické účely a bol vyvinutý na Katedre elektrotechniky a výpočtovej techniky na University of Massachusetts Amherst. [17]

1.12.6 Rozklad založený na priestore

BDS-pga bol vytvorený úpravou základných syntéz algoritmov BDD prezentovaných v programe BDS. Nasledujúce kroky popisujú proces použitý na optimalizáciu priestore založeného BDS-pga.

- Schéma je v zadaná vo formáte BLIF a je analyzovaná s BDS-PGA
- MFFC (Maximum fanout free cone) sa používa na zistenie logiky na úrovni hradiel zo súboru netlist. Tento prístup vytvára buď globálne alebo lokálne uzly pre návrh logiky. Globálne uzol má jediný výstup, všetky súvisiace vstupy, a všetky logiky potrebné pre výstup. Miestne uzly obsahujú podmnožinu požadovaných logík pre výstup. Procedúra odstraňovania je riadená metrikami. Tieto metriky sú vypočítané pomocou počtu vstupných uzlov a veľkosti BDD.
- Uplatňované sú prístupy rozkladu, ktoré sú založené na rozklade BDS. BDD uzly, vytvorené eliminációu sa rozložia v opakovanom móde pokiaľ zostanú uzly typu 2-vstup, 1-výstup. Každý uzol BDD je opakovane umiestnený vo fronte čakajúc na rozklad. Základné kroky BDS sú aplikované na každý uzol BDD v tomto poradí:

- Sú použité jednoduché dominátory na získanie prístupu k algebraickým rozkladom funkcií AND (1-dominátor), OR (0-dominátor) a XNOR (x-dominátor)
- Aplikovaný je rozklad pomocou MUX
- Vykonáva sa vyvažovanie BDD
- Vykonáva sa rozklad BDD so zreteľom na vrcholovú premennú
- Vykonáva sa rozklad zovšeobecnenej logiky
- Rozklad zovšeobecnenej logiky x-dominátora

Možné voľby v programe BDS [17]:

-k k	na špecifikáciu K-vstupovej hodnoty LUT (predvolená hodnota je 5)
-useglb	na použitie len globálnej BDD (iba v prípade ak -useglb alebo -useloc nie sú zadané, tak lokálne ako globálne BDD predvolené)
-useloc	na použitie iba lokálnych BDD
-ethred	určenie prahovej hodnoty bežnej eliminácie (10 je vhodná hodnota na použitie)
-sharing	na rozdelenie extrakcie pred vykonaním rozkladu
-largefanout	povoliť kolaps viacerým uzlom v prvom opakovaní bežnej eliminácie
-heuristic	umožniť heuristickým premenných vymieňať rozklady
-xhardcore	povoliť logické rozklady založené na x-dominátorovi (XNOR)
-delay	oneskorenie ďalšej syntézy

Výstup z programu BDS-pga 2.0

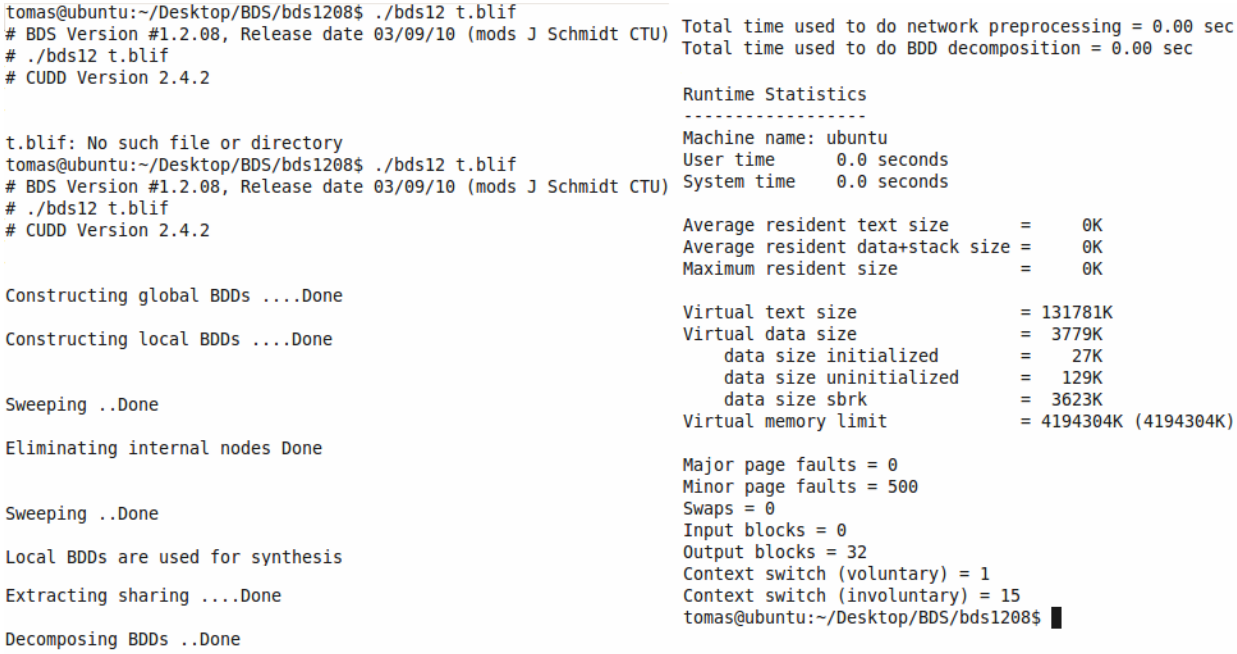
Pre analýzu sme získali súbor (**t.mv**) jednoduchého logického obvodu vo formáte BLIF.MV. Tento súbor sme prekonvertovali pomocou programu VIS do formátu BLIF. Po konvertovaní sme dostali súbor **t.blif**, ktorý sme použili ako vstup do programu BDS na spracovanie.

Obsah súboru **t.mv**:

```
.model t
.inputs a b
.outputs f
.mv f 4
.table a b ->f
.default 0
0 1 1
1 0 2
1 1 3
.end
```

Obsah súboru **t.blif**:

```
.model t
.inputs a0 b0
.outputs f0 f1
.names a0 b0 f0
01 1
11 1
.names a0 b0 f1
10 1
11 1
.exdc
.end
```



Obr. 1.18: Obrazovka zo spusteného programu BDS po spracovaní súboru **t.blif**

1.12.7 Zhodnotenie analýzy

Po skončení analýzy vybraných oblastí, ktorá nám objasnila smer, ktorým sa máme uberať, sme si určili, že budeme požívať súborový formát BLIF aj preto, že je rozšírený, používajú ho analyzované systémy ako SIS, VIS a MVSIS, a aj preto, že má pomerne nenáročný syntax. Existujúce riešenia boli brané do úvahy také, ktoré podporujú formát PNML vychádzajúci z XML pre zápis Petriho sietí, ktorý chceme využiť v našom programe. Našim cieľom je teda podpora týchto dvoch súborových formátov a aj vytvorenie grafického editora na návrh obvodov.

2 Špecifikácia riešenia

Táto kapitola obsahuje špecifikáciu navrhovaného systému.

2.1 Funkcionálne požiadavky

Cieľom Tímového projektu je vytvorenie základu pre daný modulárny systém. Keďže problematika danej oblasti je veľmi široká, rozhodli sme sa, že naša aplikácie bude v prvom rade podporovať rozšírený súborový štandard BLIF pre kombinačné a sekvenčné obvody. Petriho siete sme sa rozhodli implementovať pomocou formátu PNML. Do nášho projektu chceme zahrnúť grafický editor pre hradlové obvody a Petriho siete a tiež možnosť simulácie daných obvodov.

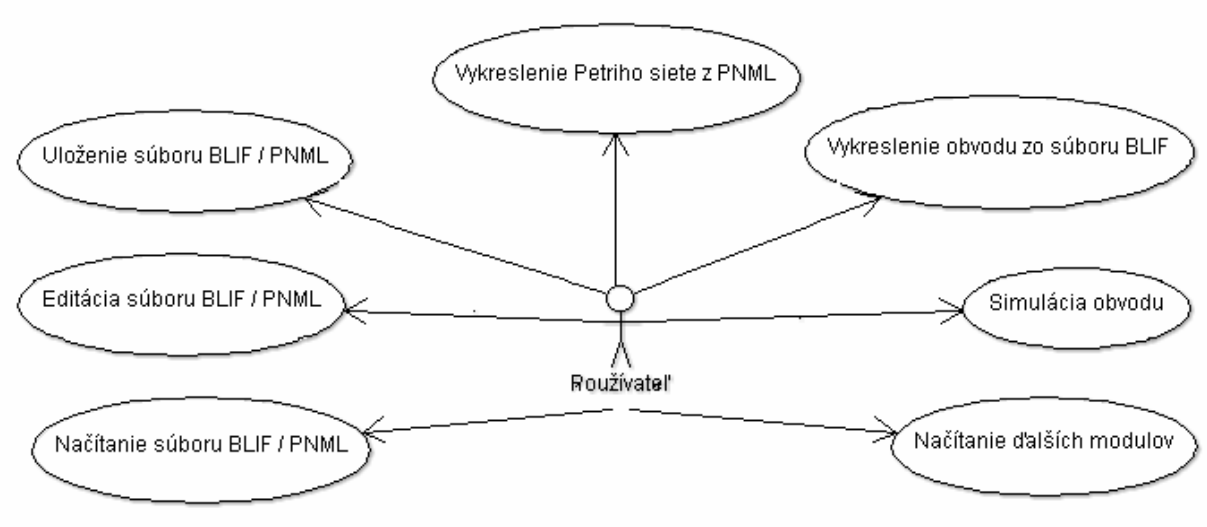
Ako vyplýva z uvedeného hlavným cieľom implementovať do systému prácu so súbormi vo formáte BLIF a PNML. Súbory v týchto formátoch bude možné otvárať, editovať a ukladať. Z načítaného súboru vo formáte BLIF bude vykreslený obvod. Tento obvod bude možné upraviť a previesť späť do formátu BLIF. Chceme dosiahnuť, aby vykreslený obvod bolo možné aj funkčne overiť pomocou simulácie.

Pre Petriho siete chceme dosiahnuť podobnú funkčnosť. Podobne ako pri súborovom formáte BLIF rovnako aj pri PNML chceme dosiahnuť možnosť načítania obvodu zo súboru, vykresliť topológiu, editovať ju a spätne uložiť. Rovnako chceme v grafickom editore vytvárať rozsiahlejšie Petriho siete, testovať ich a ukladať v súborovom formáte PNML.

Medzi funkcionálne požiadavky patrí aj požiadavka na modularitu systému. Táto je dôležitá z hľadiska jednoduchého budúceho rozširovania funkčnosti systému pre ďalšie metodiky návrhu.

2.2 Prípady použitia

Na obr. 2.1 sú zobrazené prípady použitia navrhovaného systému, ktoré sú následne bližšie popísané. V systéme vystupuje jediný aktér a to samotný používateľ.



Obr. 2.1: Diagram Prípadov použitia

Jednotlivé prípady použitia sú nasledovné:

Číslo prípadu použitia: UC1
Názov: Načítanie súboru BLIF/PNML
Hráč: Používateľ
Popis: Používateľ chce načítať obvod zo súboru BLIF alebo Petriho sieť zo súboru PNML. V menu vyberie položku otvoriť súbor a vyberie daný súbor. Vybraný súbor sa načíta do systému.

Číslo prípadu použitia: UC2
Názov: Editácia súboru BLIF/PNML
Hráč: Používateľ

Popis: Používateľ chce editovať načítaný obvod zo súboru BLIF alebo Petriho sieť zo súboru PNML. Bude fungovať ako obyčajný textový editor.

Číslo prípadu použitia: UC3
Názov: Uloženie súboru BLIF/PNML
Hráč: Používateľ
Popis: Používateľ chce uložiť obvod do súboru BLIF alebo Petriho sieť do súboru PNML. V menu vyberie položku uložiť súbor a vyberie miesto uloženia. Následne sa súbor uloží a bude možné ho opätovne načítať.

Číslo prípadu použitia: UC4

Názov: Vykreslenie Petriho siete z PNML
Hráč: Používateľ
Popis: Po otvorení PNML súboru sa Petriho sieť načíta a vykreslí do grafického editoru. V tomto editore bude možné zmeniť načítanú Petriho sieť.

Číslo prípadu použitia: UC5
Názov: Vykreslenie obvodu zo súboru BLIF
Hráč: Používateľ
Popis: Po otvorení BLIF súboru sa načíta obvod a vykreslí do grafického editoru. V tomto editore bude možné zmeniť načítaný obvod.

Číslo prípadu použitia: UC6
Názov: Simulácia obvodu
Hráč: Používateľ
Popis: Po otvorení BLIF súboru sa načíta obvod a vykreslí do grafického editoru. Správanie obvodu bude možné odsimulovať pomocou simulácie.

Číslo prípadu použitia: UC7
Názov: Načítanie ďalších modulov
Hráč: Používateľ
Popis: Používateľ bude mať možnosť načítať si externé rozšírenie k programu. Tým sa rozšíri aj samotná funkcionalita, napr. podpora ďalšieho súborového formátu, rozšírené možnosti simulácie atď.

2.3 Nefunkcionálne požiadavky

Na vytváraný systém sú kladené nasledovné nefunkcionálne požiadavky a to:

- Prehľadné používateľské prostredie
- Intuitívna práca s programom
- Ľahký návrh obvodov a ich simulácia

Hlavným cieľom je, aby bol program intuitívny a ľahko použiteľný. Musí spĺňať požadovanú funkcionalitu.

3 Hrubý návrh riešenia

3.1 Výber implementačného prostredia

Výber implementačného prostredia je dôležitý krok pred samotnou implementáciou systému. Keď sa zvolí zle, môže to ovplyvniť nielen náročnosť implementácie, ale aj konečný program. Treba si zvážiť a porovnať, ktoré implementačné prostredie je najvhodnejšie.

Program musí byť intuitívny a používateľsky príjemný – tzv. user-friedly. Keďže bude obsahovať aj grafický editor, jednoznačne musí mať grafické rozhranie. V analyzovaných programoch ako napr. VIS, alebo SIS stačila aj konzolová aplikácia, avšak tieto programy ponúkali výsledky len v textovej podobe. Aplikácia musí byť modulárna, primárnym cieľom projektu je urobiť základ tohto programu, aby sa neskôr dali do neho doplniť ďalšie moduly. Na vytváranie modulov je najvhodnejšie použiť externé knižnice. Tie sa len nakopírujú do dopredu daného adresára, napr. „x:\nas_program\plugins“ a potom program ich už automaticky rozozná pri spustení, a načíta si tieto knižnice. Je to už len na programátorovi, akú funkcionalitu implementuje do jednotlivých knižníc.

3.1.1 Java

Objektovo orientovaný jazyk od spoločnosti Sun (teraz patrí už pod spoločnosť Oracle). Je platformovo nezávislý – takisto beží pod operačným systémom Windows ako pod Linuxom. Existuje veľa foriem javy, napr. na webovské aplikácie, mobilné zariadenia, desktopové aplikácie atď. Vychádza z jazyka C++. Jeho nevýhoda je, že kvôli multiplatformovej podpore je vykonávanie programov pomalšie a neefektívne.

3.1.2 C++

Je rozšírením jazyka C o triedy a objekty, ale zachoval si sčasti aj procedurálnu stránku jazyka C. Poskytuje širokú škálu možností, ako napr. dedenie, zapuzdrenie, šablóny, predlohy

apod. Poskytuje podporu grafického rozhrania pomocou triedy MFC. Avšak programovanie takýchto aplikácií je náročné.

3.1.3 Platforma .NET

V skutočnosti sa ani nejedná o jeden programovací jazyk, ale skôr o programovaciu techniku. Jeho základ tvorí .NET Framework, ktorý poskytuje možnosť objektovo orientovaného programovania. Podporuje viacero programovacích jazykov ako napr. C++, C#, Visual Basic atď. Výhodou je, že tieto programovacie jazyky sa líšia len syntaxou, ale v podstate používajú rovnaké knižnice, dátové typy. Preto aj všetky tieto jazyky sú rovnako výkonné a efektívne. Automaticky podporuje triedy, metódy, vlastnosti, udalosti, polymorfizmus atď.

Tento systém je možné implementovať vo viacerých programovacích jazykoch. Medzi vhodné implementačné jazyky patrí Java, C++, C#. Pre implementáciu sme si vybrali jazyk C#, keďže s tento jazyk poskytuje:

- potrebnú podporu pre modularitu systému pomocou knižníc
- prístup k platforme .NET
- jednoduchosť vyššieho programovacieho jazyka
- jednoduchú tvorbu používateľského prostredia
- jednotlivý členovia tímu majú s týmto jazykom najviac skúseností

Vývojové prostredie, v ktorom sa bude aplikácia vyvíjať, bude Visual Studio 2008.

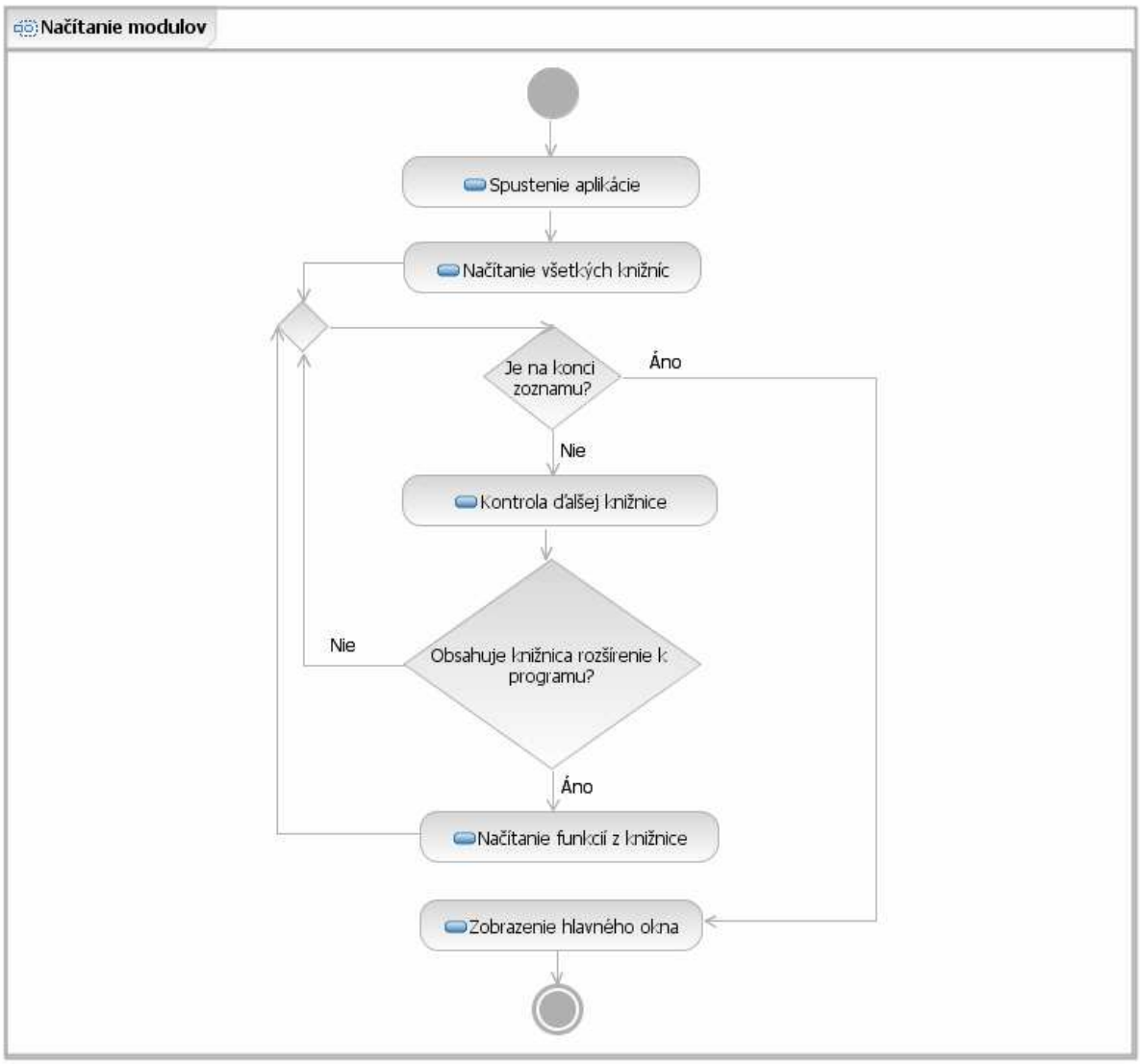
3.2 Architektúra systému

Ako už bolo spomenuté, program bude modulárny. To znamená, že bude existovať základná aplikácia s pevnými funkciami, a ďalšie funkcie sa do nej doplnia pomocou knižníc. Tým sa zabezpečí, že program bude ľahko rozšíriteľný a bude dynamický.

3.2.1 Načítanie modulov

Pri spustení sa skontroluje adresár, kde by mali byť uložené knižnice, či sa vôbec nejaké tam nachádzajú. Program si načíta zoznam týchto knižníc, a potom sa skontroluje či sú to

rozšírenia pre danú aplikáciu. Ak áno, pridajú sa funkcie z knižníc do hlavnej aplikácie a zobrazí sa hlavné okno.

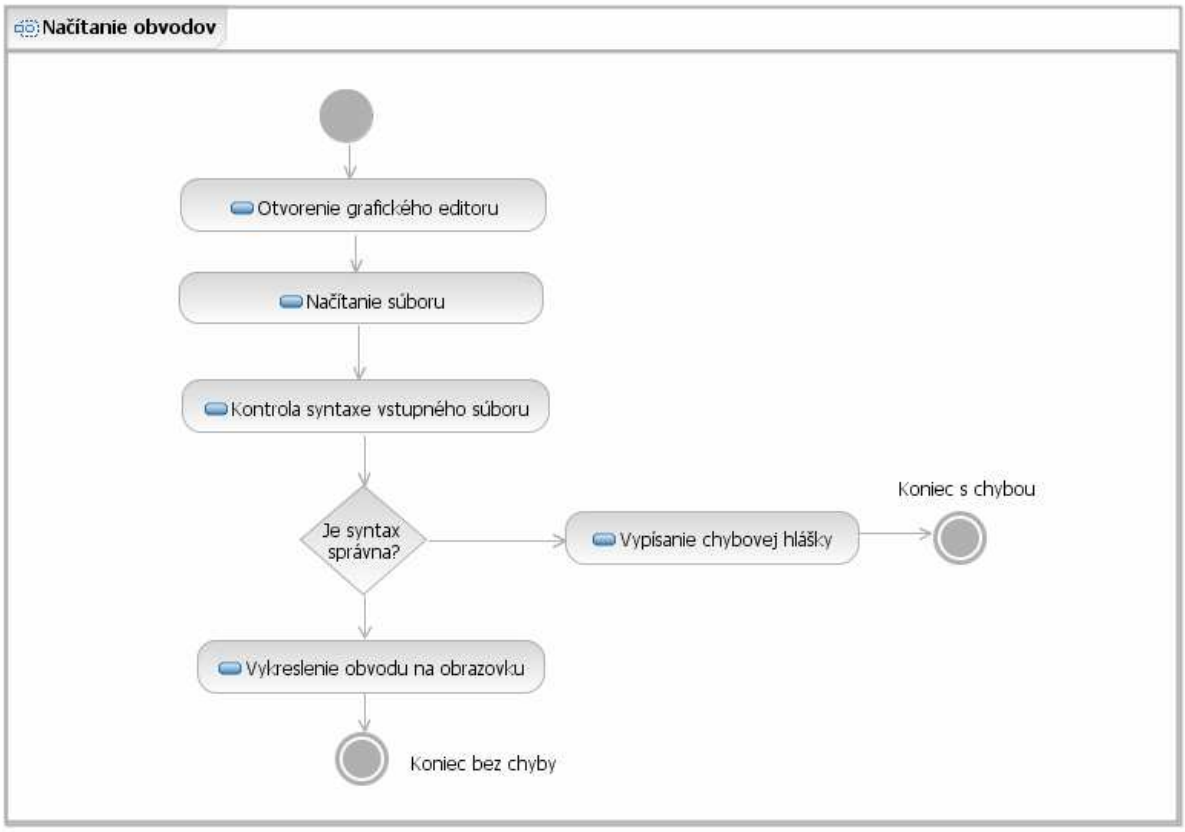


Obr. 3.1: Načítanie modulov

3.2.2 Grafický editor

Ak sa načíta BLIF alebo PNML súbor, skontroluje sa syntax týchto súborov, a následne sa na obrazovku vykreslí obvod, ktorý je opísaný v danom súbore. Nezáleží na tom, či ide, o Petriho sieť, kombinačné alebo sekvenčné obvody, či stavové automaty, program to automaticky rozozná. Takto sa naskytne používateľovi možnosť vidieť daný obvod/sieť aj graficky.

Ďalšou možnosťou bude takéto obvody vytvárať pomocou grafického editora. Používateľ si zvolí typ obvodu/siete a následne z dostupných prvkov si poskladá obvod, ktorý bude možné uložiť vo formáte BLIF alebo PNML.



Obr. 3.2: Načítanie súboru

3.2.3 Simulácia obvodov

Druhoradým cieľom je implementovať možnosť simulácie obvodov. Nastavili by sa jednotlivé parametre na vstupe, a sledoval by sa výstup. Výsledky by mohli byť reprezentované ako graficky tak i textovo. V prípade Petriho sietí by sa sledovali stavy jednotlivých uzlov, alebo pri stavových automatoch by sa vyhodnocovali jednotlivé stavy.

3.3 Požiadavky na systém

Aplikácia bude mať minimálne požiadavky na systém na ktorom sa bude spúšťať. Stačí počítač s operačným systémom Windows XP a vyššie. Hardvérové nároky budú totožné s hardvérovými nárokmi operačného systému, t.j. minimálne:

- Procesor Intel alebo AMD s taktovacou frekvenciou 300 MHz a viac
- 128 MB pamäte RAM a viac
- Monitor s rozlíšením 800x600 a vyššie
- Klávesnica a myš
- 100MB voľného miesta na pevnom disku

Softvérové nároky :

- Operačný systém Windows XP , Vista, Windows 7
- Platforma .NET 2.0 a vyššia

Platforma .NET [22] je súčasťou automatických aktualizačných balíkov, dá sa však doinštalovať aj manuálne. Bude súčasťou inštalačného balíka aplikácie.

4 Záver

V tejto etape riešenia projektu sme pomocou analýzy mohli vybrať oblasti problematiky, ktorými sa budeme ďalej zaoberať pri návrhu prototypu. Vybrané oblasti sme špecifikovali stanovením prípadov použitia a požiadaviek na systém. Nakoniec sme v kapitole Hrubý návrh riešenia uviedli ako má vyzerať architektúra systému pre prototyp projektu.

5 Použitá literatúra

[1] VIS, http://vlsi.colorado.edu/~vis/whatis.html. Posledný prístup: 30.10. 2010

[2] VILLA, T., SWAMY, G. , SHIPLE, T.: *VIS User's Manual*

[3] VIS download, http://web.cecs.pdx.edu/~alanmi/research/soft/ports/vis.exe. Posledný prístup: 30.10. 2010

[4] Chai, D., Jie-Hong, J., Jiang, Y, Li, Y, Mischenko, A., Brayton, R.: MVSIS 2.0 User's Manual, http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/doc/mvsis_20_manual.pdf ,2003, 10s.

[5] Chai, D., Jie-Hong, J., Jiang, Y, Li, Y, Mischenko, A., Brayton, R.: MVSIS 2.0 Programmer's Manual, http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/doc/mvsis_20_prog.pdf, 2003, 8s.

[6] MVSIS: Logic Synthesis and Verification. Posledný prístup: 10.11.2010

http://www-cad.eecs.berkeley.edu/Respep/Research/mvsis

[7] Active-HDL Lattice Edition online documentation and tutorials, http://www.latticesemi.com/documents/an8079.pdf. Posledný prístup: 10.11.2010

[8] Active-HDL 8.3 http://www.aldec.com/activehdl/, Posledný prístup: 10.11. 2010

[9] Petri .NET Simulator 4.6.21 http://www.popsnail.com/science/petri-net-simulator-4-6-21.html. Posledný prístup: 10.11.2010

[10] Zimmermann, A.; Freiheit, J.; German, R. Hommel, G.: *Petri Net Modelling and Performability Evaluation with TimeNET*. 11th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'2000), LNCS 1786, pp. 188-202, ISBN 3-540-67260-5, Springer-Verlag, Schaumburg, Illinois, USA, 2000

[11] TimeNet 4.0 Manual http://user.cs.tu-berlin.de/~timenet/ManualHTML4/overv.html. Posledný prístup: 30.10. 2010

[12] Getting started with CPN Tools http://wiki.daimi.au.dk/cpntools-help/getting_started_with_cpn_.wiki. Posledný prístup: 10.11. 2010

[13] ISO/IEC/JTC1/SC7. Subdivision of project 7.19 for a Petri net standard. http://www.jtc1-sc7.org/. Posledný prístup: 10.11. 2010

[14] Best, E., Devillers, R., Koutny, M.: Petri Net Algebra. EATCS Monographs on Theoretical Computer Science Series. Springer-Verlag. http://elib.tu-darmstadt.de/tocs/95312013.pdf Posledný prístup: 10.11. 2010

[15] GRAMATOVÁ, E.: Diagnostika a spoľahlivosť. Bratislava: FIIT STU, 2007. 8-9 s. Prednáška č.8

[16] YANG, C., CIESIELSKI, M.: BDS: a BDD-Based Logic Optimization System. Amherst: Dept. of Electrical & Computer Engineering, University of Massachusetts, July 2002. ISBN:1-58113-187-9. Bds-tcad02.pdf.

[17] BDS-pga version 2.0. August 2004. http://www.ecs.umass.edu/ece/tessier/rcg/bds-pga-2.0/ Posledný prístup: 9.11.2010

[18] Berkeley Logic Interchange Format (BLIF)

http://www1.cs.columbia.edu/~cs4861/s07-sis/blif/index.html Posledný prístup: 9.11.

2010

[19] Berkeley Logic Interchange Format (BLIF)
http://www.cs.uic.edu/~jlillis/courses/cs594/spring05/blif.pdf Posledný prístup: 9.11. 2010

[20] Rudell, L. Richard.: MULTIPLE-VALUED LOGIC MINIMIZATION FOR PLA SYNTHESIS, Memorandum No. UCB/ERL M86/65, Electronics Research Laboratory Electrical Engineering and Computer Science Department University of California Berkeley, California 94720, 5 June 1986

[21] LOG: The LOG system. Posledný prístup: 10.11. 2010
http://www.cs.berkeley.edu/~lazzaro/chipmunk/describe/log.html

[22] Microsoft .NET http://www.microsoft.com/net/ Posledný prístup: 10.11.2010

6 Prílohy

Táto kapitola obsahuje prílohy dokumentácie.

6.1 Príloha A1 – zdrojový kód súboru max.mv

```
# find the max of 8 numbers
.model max
.inputs a1 a2 a3 a4 a5 a6 a7 a8
.outputs maximum
.mv a1 8
.mv a2 8
.mv a3 8
.mv a4 8
.mv a5 8
.mv a6 8
.mv a7 8
.mv a8 8
.mv s1 8
.mv s2 8
.mv s3 8
.mv s4 8
.mv s5 8
.mv s6 8
.mv maximum 8

.table a1 a2 s1
0 - =a2
1 (0,1) =a1
1 (2,3,4,5,6,7) =a2
2 (0,1,2) =a1
2 (3,4,5,6,7) =a2
3 (0,1,2,3) =a1
3 (4,5,6,7) =a2
4 (0,1,2,3,4) =a1
4 (5,6,7) =a2
5 (0,1,2,3,4,5) =a1
5 (6,7) =a2
6 (0,1,2,3,4,5,6) =a1
6 (7) =a2
7 - =a1

.table a3 a4 s2
0 - =a4
1 (0,1) =a3
1 (2,3,4,5,6,7) =a4
2 (0,1,2) =a3
2 (3,4,5,6,7) =a4
3 (0,1,2,3) =a3
3 (4,5,6,7) =a4
4 (0,1,2,3,4) =a3
4 (5,6,7) =a4
5 (0,1,2,3,4,5) =a3
5 (6,7) =a4
6 (0,1,2,3,4,5,6) =a3
6 (7) =a4
7 - =a3
.table a5 a6 s3
0 - =a6
1 (0,1) =a5
1 (2,3,4,5,6,7) =a6
2 (0,1,2) =a5
2 (3,4,5,6,7) =a6
3 (0,1,2,3) =a5
3 (4,5,6,7) =a6
4 (0,1,2,3,4) =a5
4 (5,6,7) =a6
5 (0,1,2,3,4,5) =a5
5 (6,7) =a6
6 (0,1,2,3,4,5,6) =a5
6 (7) =a6
7 - =a5
.table a7 a8 s4
0 - =a8
1 (0,1) =a7
1 (2,3,4,5,6,7) =a8
2 (0,1,2) =a7
2 (3,4,5,6,7) =a8
3 (0,1,2,3) =a7
3 (4,5,6,7) =a8
4 (0,1,2,3,4) =a7
4 (5,6,7) =a8
5 (0,1,2,3,4,5) =a7
5 (6,7) =a8
6 (0,1,2,3,4,5,6) =a7
6 (7) =a8
7 - =a7


.table s1 s2 s5
0 - =s2
1 (0,1) =s1
1 (2,3,4,5,6,7) =s2
2 (0,1,2) =s1
2 (3,4,5,6,7) =s2
3 (0,1,2,3) =s1
3 (4,5,6,7) =s2
4 (0,1,2,3,4) =s1
4 (5,6,7) =s2
5 (0,1,2,3,4,5) =s1
5 (6,7) =s2
6 (0,1,2,3,4,5,6) =s1
```

```
6 (7) =s2
7 - =s1

.table s3 s4 s6
0 - =s4
1 (0,1) =s3
1 (2,3,4,5,6,7) =s4
2 (0,1,2) =s3
2 (3,4,5,6,7) =s4
3 (0,1,2,3) =s3
3 (4,5,6,7) =s4
4 (0,1,2,3,4) =s3
4 (5,6,7) =s4
5 (0,1,2,3,4,5) =s3
5 (6,7) =s4
6 (0,1,2,3,4,5,6) =s3
6 (7) =s4
7 - =s3
```

```
.table s5 s6 maximum
0 - =s6
1 (0,1) =s5
1 (2,3,4,5,6,7) =s6
2 (0,1,2) =s5
2 (3,4,5,6,7) =s6
3 (0,1,2,3) =s5
3 (4,5,6,7) =s6
4 (0,1,2,3,4) =s5
4 (5,6,7) =s6
5 (0,1,2,3,4,5) =s5
5 (6,7) =s6
6 (0,1,2,3,4,5,6) =s5
6 (7) =s6
7 - =s5
.end
```

6.2 Príloha A2 – zdrojový kôd súboru adder_mod4.mv

```
.model adder_mod4
.inputs carryin
.inputs x
.inputs y
.outputs sum
.outputs carry
.mv x 4
.mv y 4
.mv sum 4
.mv carry 3
.mv carryin 3
.table x y carryin->
carry
0 0 0  0
2 2 0  1
1 3 0  1
3 1 0  1
1 0 0  0
0 1 0  0
2 3 0  1
3 2 0  1
0 2 0  0
1 1 0  0
2 0 0  0
3 3 0  1
0 3 0  0
1 2 0  0
2 1 0  0
3 0 0  0

0 0 1  0
2 2 1  1
1 3 1  1
3 1 1  1
1 0 1  0
0 1 1  0
2 3 1  1
3 2 1  1
0 2 1  0
```

```
1 1 1  0
2 0 1  0
3 3 1  1
0 3 1  1
1 2 1  1
2 1 1  1
3 0 1  1

0 0 2  0
2 2 2  1
1 3 2  1
3 2 2  1
0 2 2  0
1 1 2  0
2 0 2  0
3 3 2  2
0 3 2  1
1 2 2  1
2 1 2  1
3 0 2  1
.table x y carryin ->
sum
0 0 0  0
2 2 0  0
1 3 0  0
3 1 0  0
1 0 0  1
0 1 0  1
2 3 0  1
3 2 0  1
0 2 0  2
1 1 0  2
2 0 0  2
3 3 0  2
0 3 0  3
```

```
1 2 0  3
2 1 0  3
3 0 0  3

0 0 1  1
2 2 1  1
1 3 1  1
3 1 1  1
1 0 1  2
0 1 1  2
2 3 1  2
3 2 1  2
0 2 1  3
1 1 1  3
2 0 1  3
3 3 1  3
0 3 1  0
1 2 1  0
2 1 1  0
3 0 1  0

0 0 2  2
2 2 2  2
1 3 2  2
3 1 2  2
1 0 2  3
0 1 2  3
2 3 2  3
3 2 2  3
0 2 2  0
1 1 2  0
2 0 2  0
3 3 2  0
0 3 2  1
1 2 2  1
2 1 2  1
3 0 2  1
.end
```