Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislava

Vizualizácia modelov digitálnych systémov

Tímový projekt I.

Bc. Hyben Martin
Bc. Jančiga Tomáš
Bc. Kardoš Martin
Bc. Maron Ľubomír
Bc. Süll Zsolt

Vedúci tímového projektu: Ing. Katarina Jelemenská, PhD.

Ročník: 1

Štúdium: Inžinierske

Obsah

1 Úvod 1

1.1 Zadanie projektu 1
1.2 Ciele projektu 1
1.3 Použité skratky 1

2 Analýza problémovej oblasti 2

2.1 Opisné jazyky pre dokumentáciu, návrh a verifikáciu digitálnych systémov 2
2.2 VHDL (VHSIC Hardware Description Language) 3
2.2.1 Opis štruktúry 3
2.2.2 Opis správania 7
2.2.3 Existujúce spôsoby vizualizácie a simulácie 7
2.3 Verilog HDL 10
2.3.1 Opis štruktúry 11
2.3.2 Modelovacie štruktúry 15
2.3.3 Existujúce spôsoby vizualizácie a simulácie 16
2.4 SystemVerilog 18
2.5 SystemC 18
2.5.1 Základné pojmy SystemC 19
2.5.2 Vizualizácia SystemC modelu 21
SystemC+Visualizer 22
2.5.3 Simulácia SystemC modelu 23
2.5.4 Vizualizácia simulácie SystemC modelu 23
2.6 Dôležité informácie v opisoch modelov 24
2.6.1 Informácie potrebné pre vizualizáciu simulácie opísaných modelov 24
2.6.2 Informácie potrebné pre vizualizáciu modelov 24
2.7 Extrakcia informácií opísaných v jazykoch 26
2.7.1 Syntaktické analyzátory 26
2.7.2 Lexikalne analyzatory 26
2.7.3 Sémantické analyzátory 27
2.7.4 Generátory analyzátorov 27
2.8 Formát extrahovaných informácií 31
2.8.1 XML 31
2.8.2 IP-XACT 32
2.8.3 Ostatné formáty 34
2.9 Možnosti vizualizácie extrahovaných informácií 35
2.10 VCD súbor 36
2.11 Vizualizátory VCD súborov 38

3 Existujúce riešenia 40

3.1 RTLVision PRO 40

3.2 Mentor Graphics HDL Designer 41
3.3 PLFire 42
3.4 Visual Elite Mentor Graphic 42

4 Špecifikácia požiadaviek 44

4.1 Funkcionálne požiadavky 44
4.2 Používateľské rozhranie 45
4.3 Systémové požiadavky 46

5 Hrubý návrh riešenia 47

5.1 Extrakcia informácií z opisov modelov do súboru XML 47
5.1.1 VHDL a Verilog 47
5.1.2 SystemC 48
5.2 Formát súboru XML 48
5.3 Simulácia opísaných modelov 54
5.3.1 VHDL 54
5.3.2 Verilog 55
5.3.3 SystemC 55
5.4 Návrh tried a objektov potrebných pre vizualizáciu extrahovaných informácií 55
5.4.1 Trieda Moduls 56
5.4.2 Trieda Ports 58
5.4.3 Trieda Connections 59
5.4.4 Trieda Points 60
5.5 Architektúra systému 61

6 Prototyp 64

6.1 Ciele prototypovania 64
6.2 Výsledky prototypovania 64
6.3 Testovanie prototypu 65

Literatúra 66

1 Úvod

1.1 Zadanie projektu

Analyzujte problematiku vizualizácie modelov digitálnych systémov, opísaných v dostupných HDL jazykoch. Analyzujte možnosti vizualizácie, ktoré poskytujú dostupné návrhové systémy. Na základe analýzy navrhnite a implementujte systém, ktorý transformuje zadaný HDL model na schematický zápis zodpovedajúci opisu štruktúry, resp. na vizualizáciu procesov zodpovedajúcich opisu správania. Vytvorený schematický zápis by mal zachovávať hierarchiu pôvodného modelu, umožňovať samostatné zobrazenie jednotlivých hierarchických úrovní, zmenu usporiadania objektov danej úrovne a export (tlač) jednotlivých hierarchických úrovní, prípadne ich výrezov. Systém by mal umožňovať vizualizáciu simulácie modelov digitálnych systémov, či už vo sfére štruktúry, alebo správania sa. Pri návrhu a implementácii systému sa zamerajte na jednoduchosť ovládania vytvorenej aplikácie s ohľadom na jej použitie vo forme učebnej pomôcky a podporného prostriedku na tvorbu dokumentácie a možnosť jednoduchého rozšírenia o podporu ďalších HDL jazykov.

1.2 Ciele projektu

Tento projekt si kladie za cieľ analyzovať existujúce možnosti vizualizácie, simulácie a vizualizácie simulácie opisných jazykov VHDL, Verilog a SystemC. Na základe tejto analýzy navrhnúť spôsob, akým bude možné vizualizovať modely opísané v týchto jazykoch, spôsob ako bude možné modely simulovať a spôsob, ako vizualizovať výsledky simulácie.
Pri návrhu riešenia spomínaných troch problémov je dôležité prihliadať na jednoduchú rozšíriteľnosť navrhnutého riešenia o ďalšie opisné jazyky.
Ďalším cieľom tohto projektu, je navrhnuté riešenie aj správne implementovať vo vývojovom prostredí Microsoft Visual Studio 2010 v programovacích jazykoch C# a C/C++ a vytvoriť takto plnohodnotnú aplikáciu určenú pre operačný systém Windows.

1.3 Použité skratky

ANTLR - ANother Tool for Language Recognition
HDL - Hardware Description Language
IEEE - Institute of Electrical and Electronic Engineers
ISP - Instruction Set Processor
VHDL - VHSIC Hardware Description Language
VHSIC - Very High Spead Integrated Circuits

2 Analýza problémovej oblasti

V analýze projektu sme sa zamerali na problematiku vizualizácie modelov digitálnych systémov opísaných v jazykoch VHDL, Verilog a SystemC. Najprv sme analyzovali jednotlivé opisné jazyky a diplomové práce vypracované na našej fakulte, ktoré sa zaoberali témami blízkymi problematike nášho zadania. Pri analýze diplomových prác sme zamerali na spôsoby vizualizácie modelov a vizualizácie výsledkov simulácie. Ďalej v práci sme definovali potrebné informácie na vizualizáciu modulov. Taktiež sme analyzovali syntaktické analyzátory, možné formáty na uloženie extrahovaných informácií a knižnice, ktoré by mohli slúžiť na vizualizáciu týchto informácií. Potom sme analyzovali aj súbor VCD, ktorý slúži na uloženie výstupov simulácie a nástroje, ktoré umožnia vizualizovať obsah tohto súboru. V poslednej časti práce sme analyzovali existujúce riešenia, ktoré slúžia na vizualizáciu VHDL, Verilog a SystemC modelov.

2.1 Opisné jazyky pre dokumentáciu, návrh a verifikáciu digitálnych systémov

Pre dokumentáciu, návrh a verifikáciu digitálnych systémov sa v súčasnosti používajú opisné jazyky HDL. Vývoj opisných jazykov začal okolo roku 1977, kedy boli súčasne predstavené dva opisné jazyky, a to ISP a KARL. Jazyk ISP viac pripomínal programovací jazyk používaný na opis vzťahov medzi vstupmi a výstupmi návrhu. Na druhej strane, opisný jazyk KARL podporoval štruktúrovaný návrh modelu, čo bolo aj základom interaktívneho opisného jazyka ABL.

Spomínané a ďalšie jazyky, ktoré boli predstavené na počiatku vývoja opisných jazykov, slúžili hlavne na verifikáciu architektúry návrhu. Neumožňovali modelovať návrhy s vysokým stupňom presnosti a obsahovali nepresný časový model.

V roku 1985 bol predstavený prvý moderný opisný jazyk Verilog. Nasledovaný bol jazykom VHDL, ktorý bol oficiálne uznaný za štandard v roku 1987. V priebehu niekoľkých rokov sa práve tieto dva stali najpoužívanejšími opisnými jazykmi na svete. V súčasnosti je predstavených množstvo opisných jazykov, napríklad SystemC, ktorého knižnica bola štandardizovaná organizáciou IEEE v roku 2005. Patrí medzi vyššie jazyky a bol vyvinutý za účelom zachovania celkového návrhu systému na jednej platforme, v tomto prípade pre používateľov programovacieho jazyka C++.

2.2 VHDL (VHSIC Hardware Description Language)

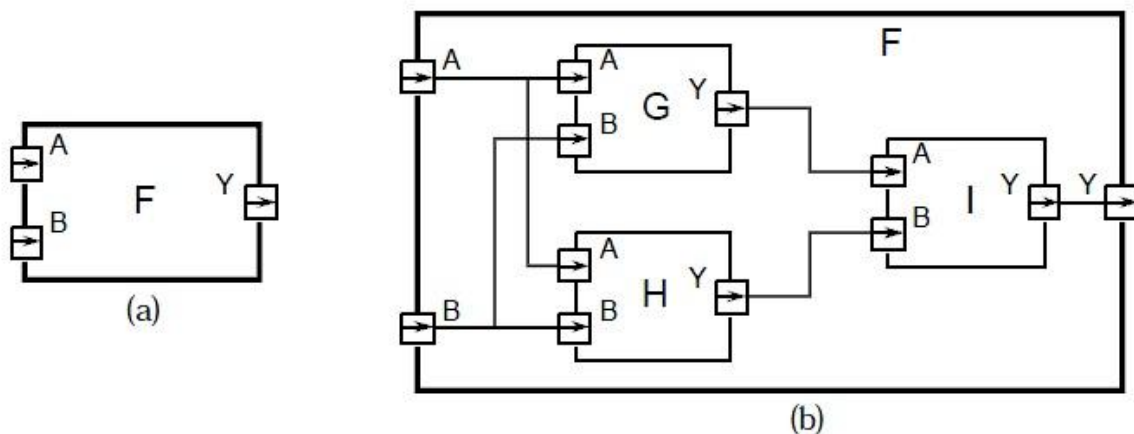
VHDL je jazyk určený na opis digitálnych systémov. Vznikol ako program vlády Spojených štátov amerických s názvom "Very High Speed Integrated Circuits" v roku 1980. Počas tohto programu vznikla potreba štandardizovaného jazyka pre opis digitálnych systémov, ktorej vyhovela inštitúcia IEEE, keď jazyk VHDL v roku 1987 štandardizovala.

VHDL ja navrhnutý tak, aby naplnil všetky požiadavky pri návrhu systémov. V prvom rade dovoľuje opis štruktúry návrhu, teda ako je navrhnutý systém rozdelený na jednotlivé funkčné moduly a taktiež ako sú tieto moduly prepojené. Ďalšou vlastnosťou jazyka je schopnosť špecifikovať správanie návrhu vďaka známym formulám programovacích jazykov. Po tretie, jazyk VHDL umožňuje simuláciu návrhu systému pred jeho fyzickým vytvorením a návrhári tak ľahko porovnajú alternatívy a otestujú systém bez jeho nákladného hardvérového prototypovania.

2.2.1 Opis štruktúry

Digitálny systém môže byť navrhnutý ako modul so vstupmi a výstupmi, pričom hodnoty na výstupe sú funkciou hodnôt na vstupe. Obrázok Obr. 2.1 ukazuje príklad takéhoto pohľadu na systém. Modul F má dva vstupy, A a B, a výstup Y. V terminológii VHDL nazveme modul F entitou, vstupy A a B vstupnými portami a výstup Y výstupným portom.

Jeden spôsob opisu modulu je opis zloženia modulu z modulov na nižšej úrovni opisu. Každý z týchto modulov je inštanciou entity, ktoré sú s inštanciami portov prepojené pomocou signálov. Obrázok 1 (b) znázorňuje spôsob, ako môže byť entita F zložená z inštancií entít G, H a I. Tento spôsob opisu sa nazýva opis štruktúry návrhu. Každá z entít G, H a I môže mať tiež svoj opis štruktúry.



Obr. 2.1: Príklad opisu štruktúry návrhu

Entity

Digitálny systém je väčšinou opísaný ako hierarchické usporiadanie modulov. Každý modul má svoj zoznam portov, ktoré predstavujú rozhranie pre komunikáciu s okolitým svetom. Vo VHDL sa takýto modul nazýva entitou, ktorá môže byť použitá ako komponent v návrhu, ale môže byť modulom na najvyššej úrovni návrhu.

Syntax deklarácie entity je nasledovná:

entity_declaration ::=
entity identifier **is**
entity_header
entity_declarative_part
[begin
entity_statement_part]
end [*entity*_simple_name] ;

entity_header ::=
[*formal*_generic_clause]
[*formal*_port_clause]
generic_clause ::= **generic** (generic_list) ;
generic_list ::= *generic*_interface_list
port_clause ::= **port** (port_list) ;
port_list ::= *port*_interface_list
entity_declarative_part ::= { entity_declarative_item }

Záhlavie entity je najdôležitejšou časťou deklarácie entity. Môže obsahovať špecifikáciu generických konštánt, ktoré sú použité na kontrolu štruktúry a správania entity, a porty, ktoré posielajú informácie dovnútra a von z entity.

Porty

Porty sú body prepojenia enity s okolím. Port je definovaný menom, režimom a typom.

Existuje 5 režimov, v ktorých môže port pracovať:

- in - tok informácií smerom dovnútra entity,
- out - tok informácií smerom von z entity,
- inout - tok informácií smerom dovnútra aj von z entity,
- buffer - tok údajov smerom von z entity so spätnou väzbou,
- linkage - používa sa iba v dokumentácii.

Architektúry

Jedna alebo viacero implementácií entít môžu byť opísané v telách architektúr. Každé telo architektúry môže opisovať odlišný pohľad na entitu. Jedna architektúra môže opisovať správanie, zatiaľ čo iná štruktúru entity.

Telo architektúry je deklarované pomocou nasledujúcej syntaxe:

architecture_body ::=
architecture identifier **of** entity_name **is**
architecture_declarative_part
begin
architecture_statement_part
end [architecture_simple_name] ;
architecture_declarative_part ::= { block_declarative_item }
architecture_statement_part ::= { concurrent_statement }
block_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| alias_declaration
| component_declaration
| configuration_specification
| use_clause
concurrent_statement ::=
block_statement
| component_instantiation_statement

Deklarácia v tele architektúry definuje položky, ktoré budú použité na vytváranie opisu návrhu. Vo všeobecnosti, v tele architektúry sú deklarované signály a komponenty, ktoré sú neskôr využité na vytvorenie štrukturálneho opisu.

Signály

Signály sa používajú na prepojenie modulov v návrhu.

Deklarujú sa pomocou nasledovnej syntaxe:

signal_declaration ::=
signal identifier_list : subtype_indication [signal_kind] [:= expression] ;
signal_kind ::= **register** | **bus**

Bloky

Moduly, ktoré sú súčasťou iných modulov, ďalej ako submoduly, môžu byť v tele architektúry opísané ako bloky. Blok je jednotka štruktúry modulov, so svojim vlastným rozhraním, prepojená s iným blokom alebo portami pomocou signálov.

Blok je špecifikovaný použitím nasledovnej syntaxe:

block_statement ::=
*block*_label :
block [(guard_expression)]
block_header
block_declarative_part
begin
block_statement_part
end block [*block*_label] ;
block_header ::=
[generic_clause
[generic_map_aspect ;]]
[port_clause
[port_map_aspect ;]]
generic_map_aspect ::= **generic map** (*generic*_association_list)
port_map_aspect ::= **port map** (*port*_association_list)
block_declarative_part ::= { block_declarative_item }
block_statement_part ::= { concurrent_statement }

Komponenty

V tele architektúry môžeme využiť aj entity, ktoré boli opísané samostatne a boli vložené do návrhových knižníc. Na využitie takejto entity musí architektúra deklarovať komponent, ktorý môže predstavovať aj šablónu definujúcu virtuálny návrh entity. Neskôr môže byť špecifikácia použitá na určenie danej knižnice.

Syntax deklarácie komponentu:

component_declaration ::=
component identifier
[local_generic_clause]
[local_port_clause]
end component ;

Architektúra môže obsahovať aj inštanciu menovaného komponentu s aktuálnymi hodnotami špecifikovanými pre generické konštanty a s portami prepojenými s aktuálnymi signálmi alebo portami entít.

Syntax vytvorenia inštancie komponentu:

```
component_instantiation_statement ::=
instantiation_label :
component_name
[ generic_map_aspect ]
[ port_map_aspect ] ;
```

2.2.2 Opis správania

Často je opis štruktúry návrhu nežiadúci, až nemožný. Je to najmä v prípade opisu entity na najnižšej úrovni opisu, úrovni hradiel. Ďalším príkladom je návrh systému, ktorého časťou je už navrhnutý integrovaný obvod. V takom prípade nás nezaujíma jeho vnútorná štruktúra, ale iba jeho správanie. Opis správania je potom vyjadrený Booleovskou funkciou. Na obrázku Obr. 2.2 je uvedený príklad takejto funkcie za predpokladu, že výstup Y entity F predstavuje exkluzívny súčet jej vstupov A a B.

$$Y = \overline{A} . B + A . \overline{B}$$

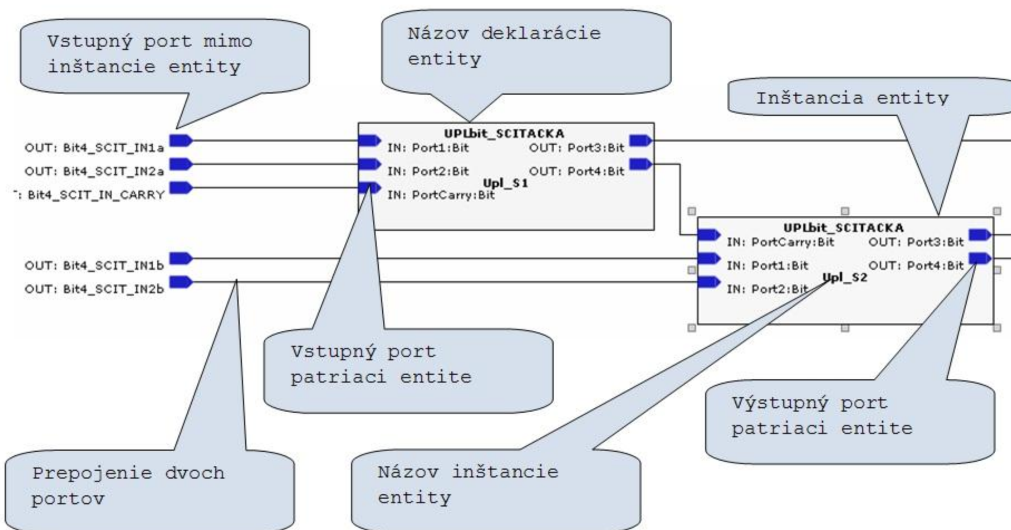
Obr. 2.2: Príklad opisu správania návrhu.

2.2.3 Existujúce spôsoby vizualizácie a simulácie

Bakalárska práca [5] sa zaoberá problematikou vizualizácie jazyku VHDL. Jej nadstavbou je diplomová práca [6], ktorá sa zaoberá simuláciou a vizualizáciou simulácie modelov opísaných v jazyku VHDL. Diplomová práca [7] a diplomová práca [8] sa tiež zaoberajú vizualizáciou modelov opísaných vo VHDL jazyku.

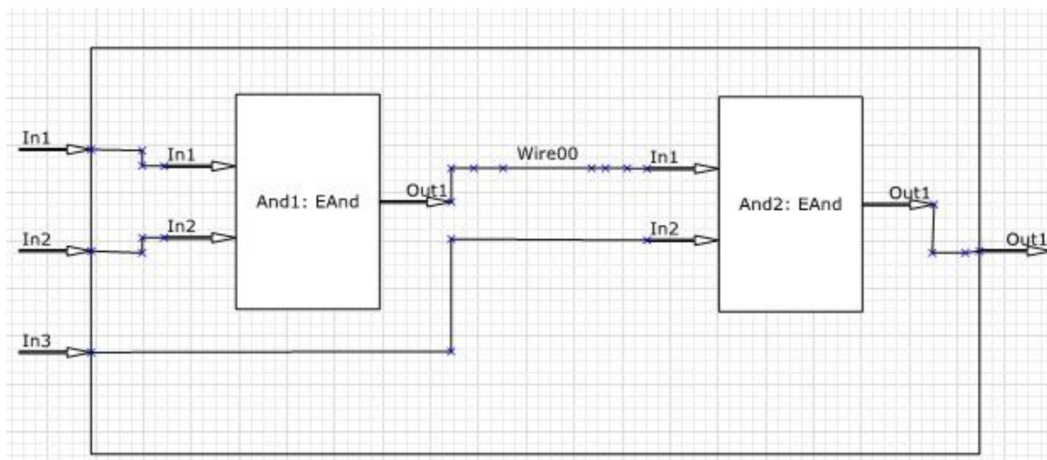
Vizualizácia

V bakalárskej práci [5] bol použitý spôsob vizualizácie entít, portov a prepojení modelov opísaných v jazyku VHDL, ktorý bol navrhnutý v diplomovej práci [7]. Tento spôsob reprezentácie je znázornený na obrázku Obr. 2.3.



Obr. 2.3: Prvý spôsob vizualizácie.

Iným spôsobom takejto reprezentácie je spôsob navrhnutý v diplomovej práci [8]. Tento spôsob je zobrazený na obrázku Obr. 2.4.



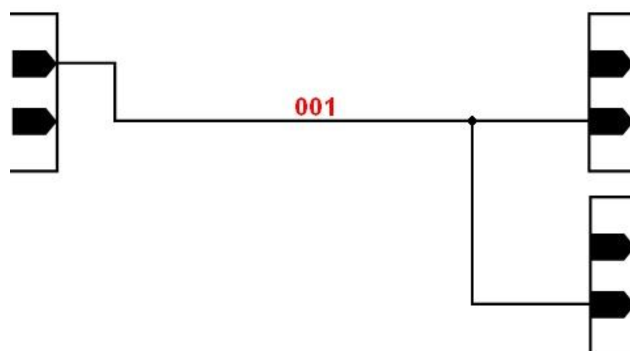
Obr. 2.4: Druhý spôsob vizualizácie.

Simulácia

Simulácia modelu opísaného v jazyku VHDL je možná prostredníctvom interných alebo externých simulátorov. V diplomovej práci [6] je na tento účel použitý externý simulátor GHDL [9]. Je to voľne dostupný simulátor jazyka VHDL, ktorý umožňuje kompilovať a simulovať VHDL kód. Jedným z výstupov tohto simulátora je aj súbor VCD, ktorý je kompatibilný s nástrojom GTKWave [10]. Nástroj GTKWave zobrazí simuláciu v tvare priebehu signálov v čase.

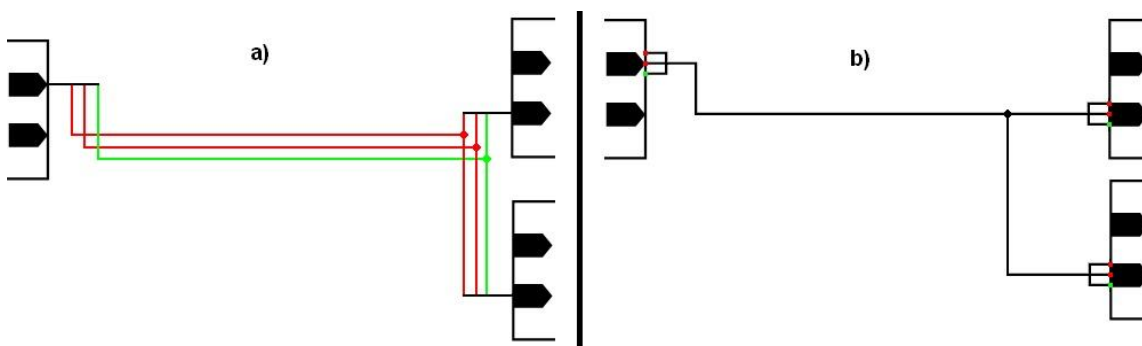
Vizualizácia simulácie

Vizualizácia simulácie je takisto problém, ktorý bol riešený v diplomovej práci [6]. Autor v rámci práce navrhol 3 spôsoby vizualizácie simulácie. Prvým spôsobom bolo nahradenie popisu signálu pri zmene jeho hodnoty práve touto hodnotou. Takýto spôsob reprezentácie simulácie je znázornený na obrázku Obr. 2.5.



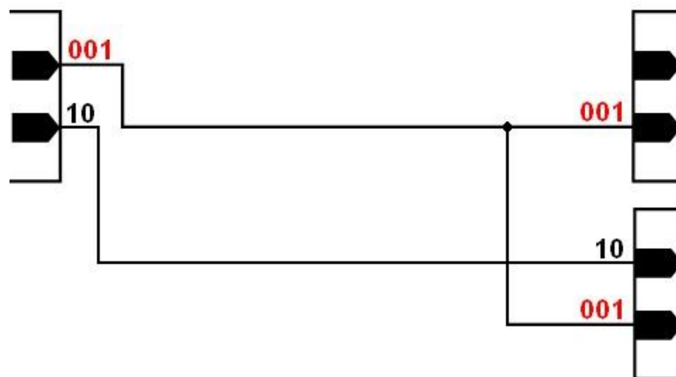
Obr. 2.5: Prvý spôsob vizualizácie simulácie.

Druhým spôsobom, zobrazeným na obrázku Obr. 2.6, je reprezentácia hodnôt signálov zmenou farby prepojenia alebo zmenou farby portov.



Obr. 2.6: Druhý spôsob vizualizácie simulácie.

Posledným a zároveň použitým spôsobom vizualizácie simulácie v diplomovej práci [6] je číselné zobrazenie hodnôt signálov pri portoch. Takéto zobrazenie je znázornené na obrázku Obr. 2.7.



Obr. 2.7: Tretí spôsob vizualizácie simulácie.

2.3 Verilog HDL

Verilog HDL je jeden z dvoch najpoužívanejších HDL jazykov. Bol vytvorený spoločnosťou Automated Integrated Design Systems v roku 1985, neskôr premenovaná na Gateway Design Automation. Tvorcom Verilog HDL bol Phil Moorby, ktorý bol neskôr hlavným vývojárom Verilog-XL. Vďaka úspechu s Verilog-XL sa spoločnosť v roku 1989 stala súčasťou spoločnosti Cadence Design Systems. Verilog bol navrhnutý ako jazyk určený na simuláciu. Až oveľa neskôr prišla myšlienka využiť Verilog pri syntéze obvodov. Najprv bol Verilog HDL licencovaným komerčným jazykom, vlastníctvom spoločnosti Cadence Design Systems.
Až v roku 1990 sa rozhodla spoločnosť otvoriť jazyk pre voľné používanie. Po vzniku OVI v roku 1991 niekoľko malých spoločnosti začalo vyvíjať Verilog simulátory. Prvé výsledky sa na trh dostali v roku 1992. Dnes už existuje veľa kvalitných Verilog simulátorov. V roku 1995 sa Verilog stal štandardom IEEE 1364, jeho aktualizovaná verzia je IEEE 1364-2001 Revision C [14].

Verilog umožňuje dva typy návrhu: zhora - nadol a zdola – nahor. Verilog možno používať na opis návrhu na štyroch úrovniach návrhu:

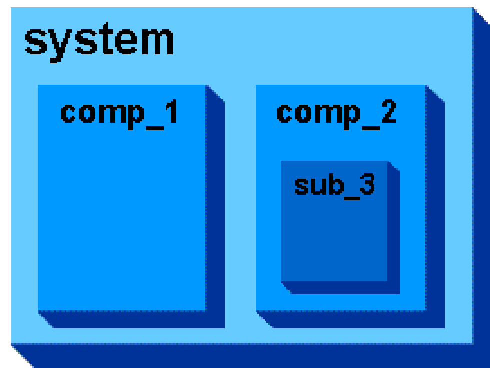
- Algorithmic Level - Algoritmická úroveň (podobne ako v programovacom jazyku C, pomocou podmienok a cyklov),
- Register Transfer Level (RTL používa registre prepojené s Boolean rovnicami),
- Gate Level (vzájomne prepojené AND, OR, atď.),
- Switch Level (Switch-e sú MOS tranzistory vo vnútri brán).

Jazyk definuje aj pojmy, ktoré môžu byť použité na ovládanie vstupov a výstupov simulácie.

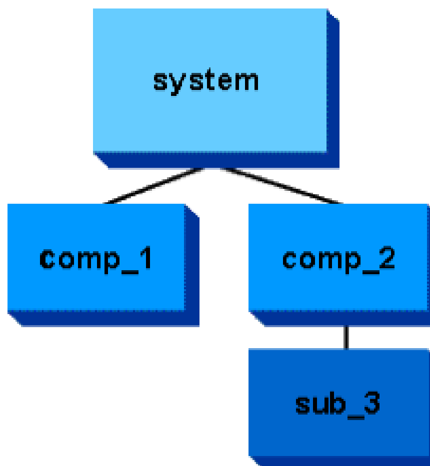
V poslednej dobe Verilog sa používa ako vstup pre syntézu programov, ktoré generujú Gate-level opis (netlist) pre obvod. Niektoré Verilog konštrukcie nie sú syntetizovateľné. Spôsob písania kódu taktiež ovplyvňuje veľkosť a rýchlosť syntetizovaného obvodu. Nesyntetizovateľné konštrukcie by mali byť použité len na testovanie. Jedná sa o programové moduly, ktoré slúžia na generovanie vstupov a výstupov na simulovanie zvyšku konštrukcie [14].

2.3.1 Opis štruktúry

Verilog umožňuje hierarchický prístup k návrhu. Na opis štruktúry používa dva prvky: moduly a porty. Model pozostáva z viacerých modulov. Modul je základný prvok a môže byť zložený z inštancií ďalších modulov. Modul, ktorý je zložený z inštancií ďalších modulov sa nazýva rodičovský (parent) modul a inštancia v ňom sa označuje ako potomok (child). Na obrázku Obr. 2.8 sa nachádzajú 4 moduly: System je rodičom modulov comp_1 a comp_2 a comp_2 je rodičom sub_3. Comp_1 a comp_2 sú potomkami modulu System a sub_3 je potomok modulu comp_2. Obrázok Obr. 2.9 znázorňuje tú istú hierarchiu modulov [15].



Obr. 2.8: Hierarchia modulov.



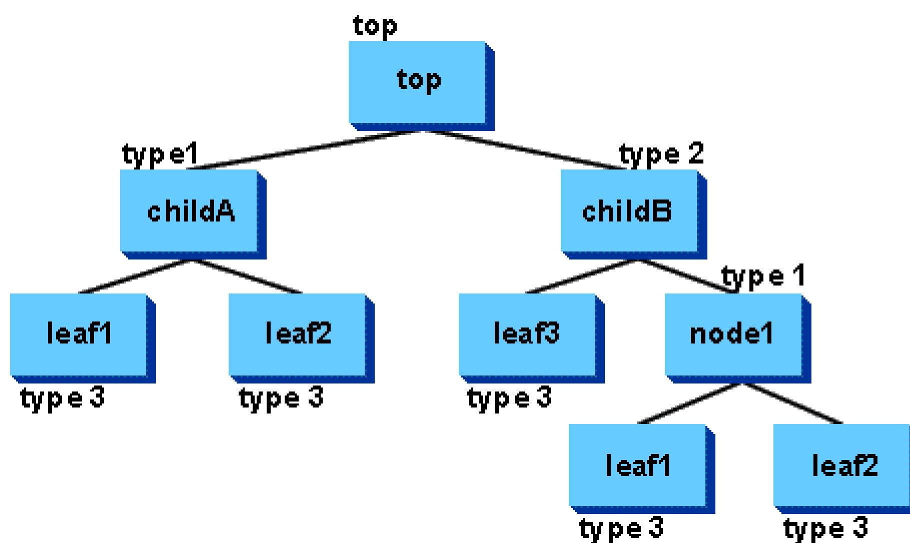
Obr. 2.9: Rovnaká hierarchia modulov ako na obrázku Obr. 2.8 znázornená iným spôsobom.

Hovoríme, že rodič inštanciuje (instantiates) potomok. To znamená, že vytvorí inštanciu modulu, aby modul sa stal potomkom daného modulu. V jazyku Verilog takúto štruktúru opíšeme nasledovne:

system instantiates comp_1, comp_2
comp_2 instantiates sub_3

Moduly

V jazyku Verilog každý modul má svoj typ a meno. Typy modulov sú definované vo Verilogu. V jednej hierarchii môžu nachádzať viacero inštancie modulov toho istého typu. Kvôli jedinečnosti, každá inštancia na jednej hierarchickej úrovni musí mať iné meno [15].



Obr. 2.10: Hierarchia modulov, ich názvy a typy.

V príklade na obr. 2.10 nachádzajú nasledovné moduly:

- 1 modul typu *top* s názvom *top*
- 2 moduly typu *type1* s názvom *childA* a *node1*
- 1 modul typu *type2* s názvom *childB*
- 5 modulov typu type3 s názvom leaf1, leaf2, leaf3, leaf1 a leaf2

Ako vidíte, niektoré moduly majú rovnaké meno a typ, ale nachádzajú sa na rôznych úrovniach. Na identifikáciu modulov preto nepoužívame iba meno a typ, ale pred jeho menom pridáme úplnú cestu v hierarchií, ktorá vedie k nemu.

Napríklad meno modulu v dolnom pravom rohu je: *top.childB.node1.leaf2*.

Moduly definujeme nasledovne:

```
module <typ_modulu> (<zoznam portov>);
        .
           .      // komponenty modulu
        .
 endmodule
```

<typ_modulu> je typ modulu, ktorý definujeme, *<zoznam portov>* je zoznam portov , ktoré umožňujú vstup a výstup dát.

Nasledujúci príklad ukazuje definíciu modelov, ktoré sa nachádzajú na Obr. 2.10:

```
module top;
        type1 childA(porty...);
        type2 childB(porty...);
endmodule

module type1(porty...);
        type3 leaf1(porty...);
        type3 leaf2(porty...);
endmodule

module type2(porty...);
        type3 leaf3(porty...);
          type1 node1(porty...);
endmodule

module type3(porty...);
        // neobsahuje inštancie ďalších modulov
 endmodule
```

Definícia modulu v samom sebe nevytvorí žiadny modul. Moduly sú vytvorené pri vytváraní inštancií, príklad:

```
module <typ_modulu_1> (<zoznam portov>);
        .
        .
    <typ_modulu_2> <meno_inštancie> (<zoznam portov>);
        .
        .
Endmodule
```

Výnimky tvoria iba moduly na najvyššej úrovni. Podľa definície, modul najvyššej úrovne nie je inštanciovaný s iným modulom. Pre modul najvyššej úrovne, meno modulu, ktorý je aj jeho typ, je taktiež používaný ako inštancia modulu. Na obrázku Obr. 2.11 je znázornený príklad, kde *foo* je modul najvyššej úrovne. Jeho potomok je modul *bee*, ktorý má typ *bar*.

```
module foo;
        bar bee (port1, port2);
endmodule

module bar (port1, port2);
        ...
endmodule
```

Obr. 2.11: Demonštračná hierarchia modulov.

Porty

Porty sú definované ako rozhranie medzi modulmi určené na prenos dát. Vo Verilogu existujú tri typy portu: vstupné (input), výstupné (output) a vstupno - výstupné (inout) [15].

Porty sú uvedené v zozname portov pri definovaní modulov a ich typ je deklarované v tele definície modulu.

Príklad:

```
module foo (in1, in2, out1, io1);
        input in1,in2;
          output out1;
          inout io1;
          …
Endmodule
```

Pri definícii portov, každý port musí mať meno. Inštancie modulov taktiež obsahujú zoznam portov. Jedná sa o spôsob prepojenia signálov rodiča so signálmi potomka.

Príklad:

```
module top;
        wire source1, source2;
        wire sink1;
        wire bus;
        foo f1(source1, source2, sink1, bus)
        …
Endmodule
```

V uvedenom príklade priradenie signálu k portu sa deje na základe poradia. Existuje ďalší spôsob, kde poradie portov nie je dôležité:

Príklad:

```
foo f1(.in1(source1), .out1(sink1), .io1(bus), .in2 (source1))
```

2.3.2 Modelovacie štruktúry

Verilog modely sa skladajú z modulov. Moduly pozostávajú z rôznych typov komponentov, medzi ktoré patria:

- Parametre (Parameters)
- Siete (Nets)
- Registre (Registers)
- Primitívy a inštancie (Primitives and Instances)
- Priebežné priradenie (Continuous Assignments)
- Procedurálne bloky (Procedural Blocks)
- Definície úloh a funkcií (Task/Function definitions)

Modul môže obsahovať ľubovoľný počet (aj 0) týchto komponentov. Poradie komponentov môže byť taktiež ľubovoľný.

- **Parametre:** konštanty, hodnota parametra je známa v čase kompilácie
- **Siete:** slúžia na prepájanie jednotlivých modulov, máju svoj názov, typ a môžu mať aj oneskorenie a silu. Sú riadené sieťovým ovládačom. Ovládač môže byť výstupný port inštancie modulu, výstupný port inštancie primitíva alebo ľavá strana priebežného priradenia.
- **Registre:** slúžia na uloženie hodnôt. Môžu byť použité, ako zdroje pre inštancie primitívov alebo modulov (môžu byť pripojené k vstupom), ale nemôžu byť riadené.. Existujú 4 typy registrov: Reg (umožňuje uložiť bity alebo bitové vektory), Integer, Time (64-bitový bezznamienkový Integer) a Real.

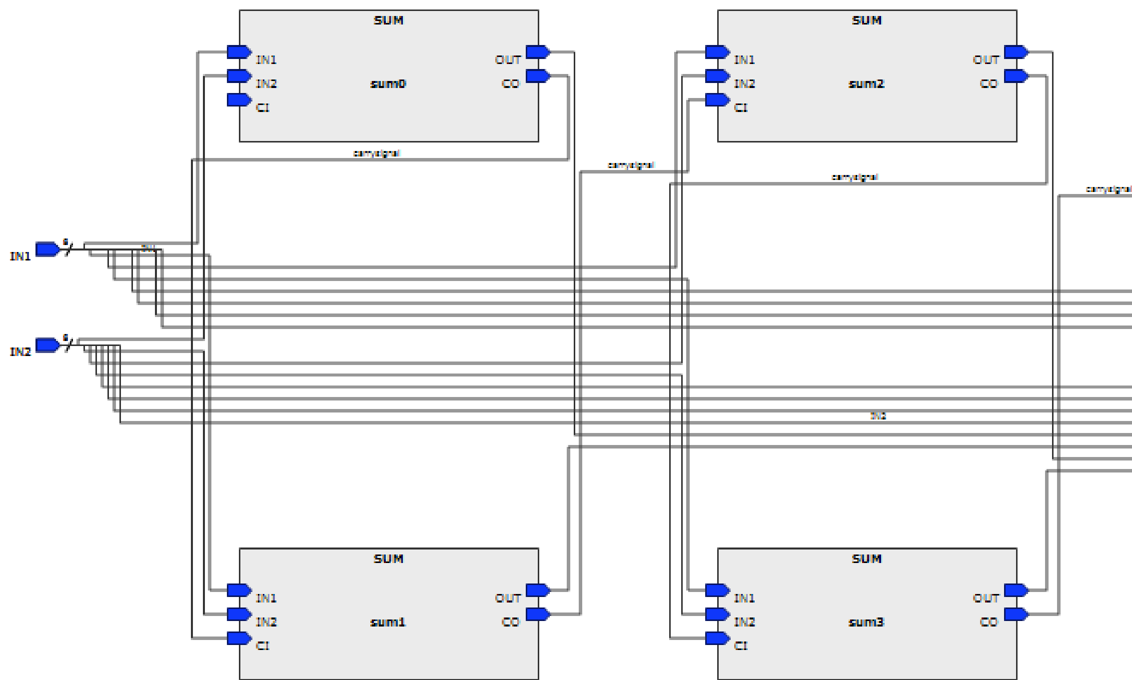
- **Primitívy a inštancie:** primitívy sú preddefinované typy modulov (and, nand, or, nor, xor, xnor, buf, not, bufif0, notif0, bufif1, notif1, pullup, pulldown), môžu byť inštanciované, ako ostatné typy modulov.
- **Priebežné priradenie:** často sa nazývajú ako tok dát, lebo opíšu ako sa presunú dáta z jedného miesta na druhú, teda vzťah medzi sieťami na ľavej strane a výrazom na pravej strane. Príklad: assign w1 = w2 & w3.
- **Procedurálne bloky:** umožňujú opísať sekvenčné správanie. Operácie v blokoch sa vykonávajú sekvenčne, ale jednotlivé bloky sa vykonávajú paralelne. Sú dva typy procedurálnych blokov: Initial (vykonávajú iba raz), Always (po dokončení sa začnú vykonávať odznova).
- **Úlohy a funkcie:** sú deklarované v moduloch, ale nesmú byť deklarované v procedurálnych blokoch. Úlohy môžu byť volané len v procedurálnych blokoch. Samotná úloha je príkaz, nemôže vystupovať ako operand vo výraze. Funkcie sú využívane ako operandy vo výrazoch, tiež môžu byť volané z procedurálnych blokov, iných funkcií alebo úloh [15].

2.3.3 Existujúce spôsoby vizualizácie a simulácie

Vizualizáciu modelov opísaných v jazyku Verilog a taktiež vizualizáciu simulácie Verilog kódu vo svojej diplomovej práce zaoberal aj Bc. Michal Nosáľ [16]. V nasledujúcej časti opíšem jeho riešenie.

Vizualizácia

Na vizualizáciu modelov je potrebné najprv získať potrebné informácie z Verilog opisu. Kód najprv teba kontrolovať a následne s nejakým parserom extrahovať potrebné informácie do nejakej štruktúry, ktorú potom môžeme vizualizovať. V tejto práci zameranej na syntaktickú analýzu bol použitý nástroj Icarus Verilog. Icarus pozostáva z niekoľkých častí, umožňuje syntézu a simuláciu jazyka Verilog. Ďalším krokom je extrahovať potrebné informácie z Verilog kódu.
Na extrahovanie informácií slúži komponent Verilog parser, ktorý bol vytvorený pomocou generátora parserov ANTLR v3. Potrebné informácie sa uložia do XML dokumentu. Následne treba spracovať XML dokument a vytvoriť grafickú reprezentáciu modelov. Na grafickú reprezentáciu bola použitá knižnica HDL Shapes Library, ktorá je upravená verzia knižnice VHDL Shapes Library, ktorú vytvoril vo svojej diplomovej práci Juraj Petráš [7] a neskôr ho modifikoval vo svojej diplomovej práce Dominik Macko [5]. VHDL Shapes Library je nadstavbou knižnice Netron Graph Library [23]. Tieto knižnice obsahujú aj algoritmy na optimalizáciu usporiadania objektov. Na Obr. 5 je znázornený časť vizualizovaného Verilog modelu.



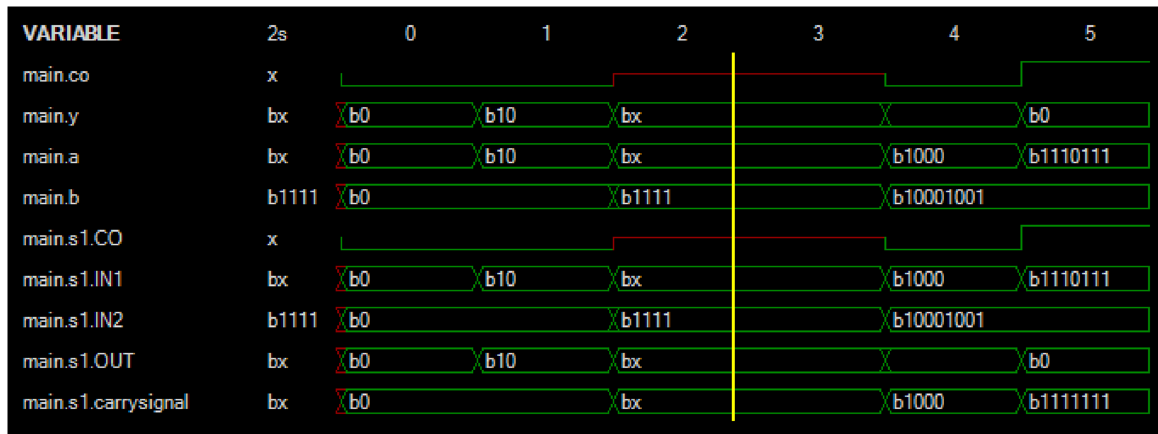
Obr. 2.12: Časť vizualizovaného modelu.

Simulácia

Na simuláciu opisu bol použitý externý simulátor, konkrétne vyššie spomínaný Icarus Verilog. Kompilátor kompiluje zdrojový kód a generuje prechodnú formu VVP preklad. Potom tento preklad je spustený simulátorom, ktorým výstupom je VCD súbor s výsledkami simulácie. Existujú aj ďalšie formáty (LXT, LXT2), ktoré sú v binárnej podobe, preto sú vhodnejšie pre veľké modely. V spomínanej práci bol generovaný VCD súbor, z ktorého potom bola spravená vizualizácia simulácie.

Vizualizácia simulácie

Ako som už to spomínal na vizualizáciu simulácie bol použitý VCD súbor. Obsah VCD súboru, teda výsledky simulácie sú spracované pomocou komponentu VCD parser. Po spracovaní súboru potrebné informácie sú uložené do XML dokumentu. Komponent Simulation Graph Library potom spracuje tento XML dokument a vizualizuje výsledky simulácie ako časový priebeh signálov. Na Obr. 6 je znázornený graf simulácie.



Obr. 2.13: Graf simulácie.

2.4 SystemVerilog

SystemVerilog je kombinovaný HDL a HVL (Hardware Verification Language) jazyk na základe rozšírenia Verilog-u.

Funkcie vo SystemVerilog-u môžeme rozdeliť na dve odlišné časti:

- SystemVerilog na návrh RTL – rozšírenie štandardného Verilogu
- SystemVerilog na verifikáciu – využíva rozsiahle objektovo orientované programovacie techniky a je viac blízky k Jave, ako k Verilogu

SystemVerilog poskytuje kompletné verifikačné prostredie, obsahuje v sebe nasledujúce verifikačné metódy: Constraint Random Verification, Assertion Based Verification a Coverage Driven Verification. Tieto metódy výrazne zlepšujú proces overenia. Taktiež poskytuje rozšírené funkcie na modelovanie hardvéru, ktoré zvyšujú produktivitu a zjednodušujú proces návrhu [20].

2.5 SystemC

Štandard IEEE Std 1666™-2005 definuje SystemC ako knižnicu tried jazyka ANSI C++ pre návrh hardvéru a systémov [1].
SystemC sa od ostatných špecifických a opisných jazykov líši tým, že umožňuje modelovať systémy a vnorené programové prostriedky súčasne na rôznych úrovniach abstrakcie [2]. Aplikácie SystemC môžu využívať možnosti jazykov C/C++ a môžu taktiež rozšíriť SystemC pomocou mechanizmov, ktoré poskytuje C++, a to bez toho, aby štandard SystemC nejako porušili [1].

V jazykoch VHDL alebo Verilog je opis štruktúry a opis správania sa pevne daný syntaxou jazyka. V prípade SystemC sa štruktúra modelu vytvára počas zavádzania programu [2].

Ak chceme v našej aplikácii používať knižnicu SystemC, je potrebné vložiť hlavičkový súbor systemc alebo systemc.h. Ak vložíme súbor systemc, sprístupníme tak iba menné priestory sc_core a sc_dt. Pre sprístupnenie jednotlivých názvov z týchto menných priestorov, ako aj z menného priestoru std musíme teda použiť deklaráciu alebo direktívu using. Pokiaľ sa nechceme zaoberať sprístupňovaním jednotlivých názvov môžeme použiť hlavičkový súbor systemc.h, ktorý za nás vloží všetky názvy, no odporúča sa radšej používať súbor systemc.

2.5.1 Základné pojmy SystemC

Ako sme už spomínali, SystemC je knižnicou jazyka C++. Táto knižnica v sebe implementuje triedy, ktoré delíme do štyroch skupín [2, 1]:

- Jadro jazyka(core languge)
- Predefinované kanály (predefined chanels)
- Dátové typy (data types)
- Nástroje (utilities)

Prvky jazyka definované v týchto triedach môžeme rozdeliť z hľadiska vizualizácie a simulácie modelu na tieto základné skupiny alebo typy:

- Moduly
- Dátové typy
- Kanály a porty
- Procesy

Moduly

Moduly sú hlavné štrukturálne bloky jazyka SystemC a sú implementované triedou sc_module. SC_MODULE je makro, ktoré sa môže použiť ako prefix pre definíciu modulu namiesto spôsobu C++ (verejné dedenie) class X: public sc_module.
Každý modul má svoj názov, ktorý sa predáva ako parameter konštruktora modulu. Pre predefinovanie (alebo prekonanie) tohto konštruktora je taktiež definované makro, konkrétne SC_CTOR(const sc_module_name&).

Ak by sme chceli SystemC porovnávať s jazykom VHDL, môžeme povedať, že moduly SystemC sú ekvivalentné s entitami VHDL.

Dátové typy

V SystemC, ako knižnici jazyka C++ je možné používať akékoľvek dátové typy C++, no samotná knižnica SystemC prináša dátové typy, ktoré sa viac hodia pre modelovanie hardvéru. Tieto dátové typy sa nachádzajú v tabuľke Tab. 1.1 [2].

Tab. 2.1: Najdôležitejšie dátové typy v SystemC, prebraté z [2].

Trieda alebo šablóna	Popis
sc_bit	Jeden bit
sc_logic	Štvorhodnotová logika
sc_bv<N>	Vektor N prvkov typu sc_bit
sc_lv<N>	Vektor N prvkov typu sc_logic
sc_int<N>	N-bitový znamienkový celočíselný typ, N <= 64
sc_uint<N>	N-bitový neznamienkový celočíselný typ, N<=64
sc_bigint<N>	N-bitový znamienkový celočíselný typ
sc_biguint<N>	N-bitový neznamienkový celočíselný typ
sc_signed	Znamienkový celočíselný typ
sc_unsigned	Neznamienkový celočíselný typ
sc_fixed<N>	N-bitové znamienkové číslo s pevnou rádovou čiarkou
sc_ufixed<N>	N-bitové neznamienkové číslo s pevnou rádovou čiarkou
sc_fix<N>	znamienkové číslo s pevnou rádovou čiarkou
sc_ufix<N>	neznamienkové číslo s pevnou rádovou čiarkou

Signály a porty

Rovnako ako signály vo VHDL, slúžia kanály na prenášanie informácií a porty pre prepájanie modulov. Porty a kanály sú implementované pomocou šablón jazyka C++, vďaka čomu môžu prenášať informáciu v podobe dátových typov jazyka C++, no vhodnejšie je použiť dátové typy opísané v predchádzajúcej kapitole. SystemC implementuje pre vytvorenie portov alebo kanálov nasledujúce šablóny [2]:

- **sc_signal<X>** - kanál dátového typu X
- **sc_in<X>** - vstupný port dátového typu X
- **sc_out<X>** - výstupný port dátového typu X
- **sc_inout<X>** - vstupno/výstupný port dátového typu X
- **sc_fifo<X>** - kanál FIFO dátového typu X
- **sc_mutex** - mutex
- **sc_semaphore** – semafór
- **sc_export** - export, umožňuje modulu poskytnúť rozhranie k jeho rodičovskému modulu.

Inštancie triedy sc_port, alebo tried odvodených, môžu byť spojené s inštanciou kanála, alebo s inou inštanciou triedy sc_port, alebo k inštancii triedy sc_export.
Inštancie triedy sc_export môžu byť pripojené k inštancii kanála, alebo k inej inštancii triedy export, ale nie k inštancii triedy sc_port. Z uvedeného vyplýva, že o prepojení portu a exportu môžeme povedať, že port je pripojený k exportu, no opačne to nie je pravda [1].

Porty môžu byť priradené pomocou mien, alebo pomocou , alebo pozícií. Menovité priradenie je implementované v členskej funkcii triedy sc_port, zatiaľ čo pozičné prepájanie je implementované členskou funkciou triedy sc_module. Takýmto spôsobom je možné priradiť maximálne 64 portov.
Oba spôsoby, aj menovité priradenie, aj pozičné priradenie predstavujú predefinovaný operátor (). V prípade menovitého priradenia je možné naviac použiť funkciu bind()[1].

Exporty môžu byť priradené iba menovitým priradením, pomocou členskej funkcie bind() alebo predefinovaného operátora () [1].

Procesy

Procesy sú z pohľadu C++ implementované ako metódy tried modulov. Popisujú chovanie obvodu a rozlišujeme tri druhy procesov [2]:

- **SC_METHOD** – typický kombinačný proces.
- **SC_CTHREAD** – typ procesu určený pre implementáciu sekvenčnej logiky, ktorá je citlivá na jeden hodinový signál. Čakanie na hodinový signál je realizované volaním funkcie wait().
- **SC_THREAD** – tiež určený pre implementáciu sekvenčnej logiky, môže volať funkciu wait(), ktorá bude čakať na takzvaný citlivostný zoznam.

2.5.2 Vizualizácia SystemC modelu

Pre vizualizáciu modelu je potrebné najskôr extrahovať informácie zo SystemC opisu. Tu sa ale dostávame do zaujímavej situácie v porovnaní s jazykmi VHDL a Verilog, pretože štruktúra tohto modelu sa zostavuje dynamicky pomocou jazyka C++, ktorý poskytuje definície makier, pretypovania, dynamickú prácu s pamäťou a ďalšie možnosti, kvôli ktorým nie je jednoduché (ak to nie je nemožné) extrahovať štruktúru modelu iba pomocou syntaktickej analýzy. Z tohto dôvodu nástroje, ktoré analyzujú opisy v SystemC syntakticky, bez zostavenia modelu, nemusia poskytovať vždy správny výstup.
V tejto kapitole sa pozrieme na existujúce riešenia extrakcie informácií alebo vizualizácie modelov opísaných v SystemC.

GSysC

Ide o knižnicu jazyka C++, ktorá bola vytvorená pre vizualizáciu a riadenú simuláciu SystemC modelov. Výhodou je, že schéma modelu je vytvorená počas spracovávania opisu. GSysC je pridaný ako vrstva medzi simulačné jadro SystemC a vytvorený model. Takto dokáže informovať používateľa o priebehu simulácie.
Nevýhodou GSysC je potreba doplniť do SystemC súboru direktívu pre pridanie knižnice GSysC a ďalej registráciu každého modulu modelu touto knižnicou [4].

SystemCXML

Tento nástroj extrahuje informácie potrebné pre vizualizáciu modelu z opisu SystemC a poskytne ich ako výstup vo formáte dokumentu XML. Používa program Doxygen, ktorý slúži na automatizované vytváranie dokumentácie k zdrojovým súborom. Výstup Doxygenu obsahuje ale nadbytočné informácie. Pre odstránenie týchto informácií a teda extrakciu tých potrebných sa používa knižnica Xerces-C++ (XML parser) a konečným výstupom je súbor XML obsahujúci už iba potrebné informácie pre vizualizáciu modelu. Nástroj pre vizualizáciu tohto formátu nepoznáme [4].

SystemC+Visualizer

Je výsledkom bakalárskej práce vytvorenej na FIIT STU v Bratislave[4]. Obsahuje upravenú knižnicu SystemC tak, aby bolo možné extrahovať potrebné informácie a tieto pomocou grafickej knižnice GLUT zobraziť v grafickej podobe. Toto riešenie nie je dostupné ako samostatná aplikácia, ale prestavuje súbor knižníc C++ a pre jeho využívanie je teda potrebné mať nainštalované nejaké vývojové prostredie pre C++.
Zdrojové kódy knižníc sú dostupné a je teda možné ich použiť a modifikovať podľa potreby.

SC_2VHDL

Ide o diplomovú prácu vypracovanú na ČVUT FEL v Prahe. Jej výsledkom je program, ktorý používa upravený kompilátor GCC pre preklad zdrojových súborov SystemC modelu do formy abstraktného syntaktického stromu (AST).
Kompilátor GCC samotný síce umožňuje uložiť AST do súboru, no bolo potrebné zmeniť jeho formát na XML, čo tvorca programu docielil spomínanou modifikáciou kompilátora.
Program SC_2VHDL potom z tohto XML súboru extrahuje potrebné informácie. Na parsovanie XML reprezentácie AST sa používa knižnica Expat.
Ďalej bol vytvorený zložitý emulátor pre AST zapísané v XML súbore, tento emulátor má za úlohu zaviesť celú reprezentáciu AST do pamäte, kde zostaví celý SystemC model aj s použitými knižnicami (vrátane SystemC).

Takýto zložitý emulátor bol implementovaný so zámerom vyhnúť sa modifikácii knižnice SystemC, no nakoniec musel autor pristúpiť aj k tomuto kroku a program SC_2VHDL preto používa modifikovanú knižnicu SystemC. Nakoniec program z emulovaného modelu extrahuje informácie týkajúce sa samotného modelu, ktoré prekonvertuje do podoby VHDL modelu [2].

2.5.3 Simulácia SystemC modelu

Simulácia modelu opísaného v SystemC znamená z pohľadu C++ vykonanie preloženého a zostaveného programu. Začína sa teda spracovaním opisu, kedy je model systému zostavený a zavedený do pamäte a pokračuje sa samotnou simuláciou.
Po vykonaní zadaného počtu cyklov sa simulácia ukončí a model sa z pamäte odstráni.
Knižnica SystemC štandardne obsahuje simulátor OSCI reference simulator. Pomocou neho je možné simuláciu spustiť a zastaviť, nie je možné krokovanie alebo návrat v čase. Výstup tohto simulátora je nutné pre každý model doprogramovať. Výsledky simulácie je možné ukladať buď v textovej forme alebo vo formáte súboru VCD.
Zdrojový kód tohto simulátora je dostupný a licencia umožňuje jeho modifikáciu na nekomerčné účely [3].

2.5.4 Vizualizácia simulácie SystemC modelu

Jedným z predstaviteľov takýchto vizualizátorov je program GSysC opísaný v kapitole 2.5.2. Tento program patrí medzi jednoduchšie riešenia s otvoreným zdrojovým kódom, ktorý ale nie je k dispozícii.
Pre vizualizáciu simulácie SystemC sa dajú tiež použiť komplexnejšie riešenia od komerčných výrobcov softvéru. Tieto programy budú popísané v ďalších kapitolách tejto práce.

Jednou z ďalších možností je aplikácia, ako výsledok diplomovej práce vypracovanej na našej fakulte. Táto aplikácia sa volá SystemC+Visualizer a ide o rozšírenie bakalárskej práce opísanej v kapitole 2.5.2 o možnosti zobrazenia výsledkov simulácie opísaného modelu. Použitý je už spomínaný OSCI refernce simulator ako súčasť knižnice SystemC a je pre neho dorobené používateľské rozhranie pre prehľadné zobrazenie výsledkov simulácie. Program taktiež umožňuje export výsledkov do súboru VCD.

2.6 Dôležité informácie v opisoch modelov

V tejto kapitole sa budeme venovať informáciám, ktoré sme identifikovali ako potrebné pre vizualizáciu modelov opísaných v jazykoch VHDL, Verilog a SystemC a taktiež informáciám potrebným pre vizualizáciu simulácie týchto modelov.

2.6.1 Informácie potrebné pre vizualizáciu simulácie opísaných modelov

Vizualizáciou simulácie rozumieme získanie hodnôt nachádzajúcich sa na portoch a signáloch modelu. Tieto hodnoty je potrebné poznať v každom čase. Pre splnenie tejto úlohy je vhodné zaznamenať iba časy, kedy došlo k zmenám hodnôt a taktiež zaznamenať aj samotné hodnoty.

Informácie, ktoré pre vizualizáciu simulácie potrebné môžeme teda zhrnúť na tieto tri:

- port alebo signál
- hodnota
- čas kedy bola (zmenená) hodnota zaznamenaná

2.6.2 Informácie potrebné pre vizualizáciu modelov

Na základe analýzy opisných jazykov VHDL, Verilog a SystemC sme identifikovali informácie, ktoré sú potrebné pre vizualizáciu modelov v jednotlivých jazykoch. Tieto informácie sú zobrazené v tabuľke Tab. 2.1, kde sa na jednom riadku nachádzajú vedľa seba ekvivalenty v každom jazyku a na konci riadka je uvedený podrobný popis a atribúty týchto informácií.

Tab. 1.2: Tabuľka dôležitých informácií potrebných pre vizualizáciu modelov.

VHDL	Verilog	SystemC	Popis	Dôležité atribúty
Entita + architektúra	Modul	Modul	Základné štrukturálne bloky. VHDL: Moduly môžeme chápať ako inštancie Entity + príslušnej architektúry, ktoré sú potom typom týchto inštancií. Verilog: Inštancie definícií modulov SystemC: Inštancie tried, ktoré dedia od triedy sc_module.	názov*, názov z pohľadu jazyka*, typ, cesta v rámci štruktúry modelu, zoznam portov modulu,
Port	Port	Port Export	Pomocou portov sú moduly prepájané prostredníctvom signálov (VHDL, SystemC) alebo bez signálov (Verilog).	názov, typ portu (in, out, inout,...), dátový typ, modul, ktorému patrí
Signál	Net**	Kanál**	Používajú sa pre prepájanie portov modulov.	názov, dátový typ, zoznam prepojených portov (porty musia byť identifikované názvom a názvom modulu) (taktiež je potrebné pridať nejaký atribút, ktorý konkrétny signál označí ako hodinový)

*V jazyku SystemC môžeme definovať pre inštancie modulov a hodín názvy z pohľadu C++, alebo v konštruktoroch týchto inštancií z pohľadu SystemC.

**V jazykoch SystemC a Verilog je možné prepájať porty pomocou kanálov, alebo bez kanálov. O portoch, ktoré sú prepojené bez kanálov môžeme uvažovať ako o portoch prepojených nepomenovaným kanálom bez akýchkoľvek parametrov.

2.7 Extrakcia informácií opísaných v jazykoch

2.7.1 Syntaktické analyzátory

Syntaktický analyzátor je program, ktorý podľa presne definovanej formálnej gramatiky transformuje vstupné dáta, väčšinou je to text, do vopred danej štruktúry. Tento komplexný proces sa v angličtine nazýva *parsing,* preto sa tieto programy tiež slangovo nazývajú parsermi. Parsery po transformácií dát, zachovávajú všetky dôležité hierarchické vzťahy. Vo všeobecnosti budujú syntaktické (*parse*) stromy.

V informatike sa najčastejšie vyskytujú parsery ako súčasť kompilátorov. Kompilátor si prevedie zdrojový text napísaného programu do internej podoby a až potom je ďalej spracovávaný. Väčšinou sú programovacie jazyky špecifikované za pomoci bezkontextových gramatík, čo veľmi uľahčuje tvorbu parserov.

Existujú dve základné metódy syntaktickej analýzy. Zakladajú sa na rozdielnych postupoch konštrukcie syntaktického stromu.

1. Metóda zhora nadol
   Pri tejto metóde sa najväčšie prvky "rozbíjajú" na menšie až vzniknú nedeliteľné časti, ktoré sa porovnajú zo vstupom.
   Príkladom tohto postupu sú LL gramatiky.

2. Metóda zdola nahor
   Táto metóda je opačná k predchádzajúcej. Pri tejto sa využíva konštrukcia derivačného stromu od listu, od najmenšej časti.
   Príkladom takejto metódy sú LR gramatiky

2.7.2 Lexikalne analyzatory

Tento druh analyzátorov využíva na svoju prácu lexikálnu analýzu (tiež sa používa výraz skenovanie, z angl. *scanning*).Pri tomto druhu analýzy sa reťazec znakov mení na prúd symbolov. Tieto symboly sa zvyknú nazývať lexémy.

Väčšina lexikálnych analyzátorov nedokáže rozpoznať chyby umiestnenia symbolov, rozoznáva len lexikálne chyby. Čiže logické usporiadanie symbolov v programe je analyzátoru neznáme. [11]

Takýto druh analýzy sa často používa pri programovacích jazykoch. Zdrojový kód programu je postupne nahrázaný symbolmy ktoré su daľej spracovávané. Lexikálne analyzátory majú väčšinou aj chybový výstup, teda zoznam chýb ktoré boli vo vstupnom texte nájdené. Lexikálne gramatiky obsahujú väčšinou malý počet lexém (v angl. používané aj *token*).

V praxi sa častou využíva spojenie syntaktických a lexikálnych analyzátorov. Použitie oboch analyzátorov nám zabezpečuje nielen kontrolu správnosti napísaného textu, teda či neobsahuje prvky ktoré nie sú definované pre danú gramatiku, ale aj transformáciu textu do presne danej digitálnej štruktúry (derivačný strom).

Príklady lexikálnych analyzátorov:

- lex/flex
- JavaCC
- Ragel

2.7.3 Sémantické analyzátory

Sémantická analýza patrí k najkomplexnejším postupom, ktoré slúžia na spracovanie textových informácií. Na rozdiel od syntaktickej analýzy, ktorá sa zaoberá len správnosťou syntaxe, sémantická analýzy berie do úvahy aj správnosti postupnosti a významu použitých znakov. Kontrola sa vzťahuje aj na správnosť a vykonateľnosť zadaných operácií.

Sémantický analyzátor prehľadáva a skúma syntaktický strom. V tejto fáze procesu sú tiež zahrnuté niektoré kontrolné operácie:

- Typová kontrola - overenie dátových typov. (napr.: int-to-string)
- Overenie všetkých premenných, tried, objektov, atď. .
- Kontrola zadaných funkcií.

Vstupom pre tento druh analyzátora je syntaktický strom. Tento strom bol vygenerovaný syntaktickým analyzátorom. Výstupom je tiež syntaktický strom, avšak obohatený o nové informácie.

2.7.4 Generátory analyzátorov

Keďže väčšina programovacích jazykov býva špecifikovaných podľa bezkontextových gramatík, je tvorba parserov značne zjednodušená. Program ktorý je schopný vygenerovať parser, teda syntaktický analyzátor sa nazýva generátor analyzátora (angl. parser generátor).

Vstupnými dátami pre generátor analyzátora je špecifikácia ktorej súčasťou sú je gramatika. Táto gramatika je najčastejšie zapísaná v BNF(Backus-Naur Form) alebo aj EBNF (Extende Backus-Naur Form). Niektoré experimentálne generátory berú za svoj vstup formálny opis sémantiky programovacieho jazyka. Tento postup sa tiež nazýva sémanticky založená kompilácia.
Výstupom generátora je väčšinou zdrojový kód parsera.

Analyzátor, ktorý bol takto vygenerovaný pozostáva z častí :

- Správa tokenov
- Výkonná jednotka
- Zásobník
- Akcie

Keďže tieto metódy nie sú žiadnou novinkou, vo svete vzniklo už veľa generátorov analyzátorov.

- ANTLR
- COCO/R
- GOLD
- SABLECC

ANTLR (ANother Tool for Language Recognition)

ANTLR je voľne šíriteľný jazykový nástroj, ktorý poskytuje prostredie pre tvorbu analyzátorov, kompilátorov a prekladačov z opisu gramatík obsahujúcich akcie do rôznych cieľových jazykov. Automatizuje konštrukciu vzoriek pre rozpoznanie jazyka. ANTLR z formálnej gramatiky generuje program, ktorý skontroluje zhodnosť viet gramatiky s cieľovým jazykom. Inými slovami, je to program, ktorý tvorí iné programy. Poskytuje podporu pre stromovú hierarchiu, prechádzanie takejto hierarchie, preklad a automatické hlásenie a zotavenie z chýb. Nástroj ANTLR je veľmi obľúbený, pretože je ľahký pre porozumenie, flexibilný, generuje výstupy čitateľné pre ľudí, má voľne dostupné zdrojové kódy a je aktívne podporovaný.

Vstupom pre nástroj ANTLR je vstupná gramatika, ktorá predstavuje presný opis jazyka rozšíreného o sémantické akcie. Na základe tejto gramatiky vytvorí zdrojové kódy, obsahujúce súbory tried, určené na analýzu vstupného jazyka. Zdrojové kódy sú vytvorené v jazyku špecifikovanom vo vstupnej gramatike. ANTLRv3 podporuje jazyky Java, C/C++, C#, Python a Ruby.

Vstupná gramatika

Na základe vstupnej gramatiky a definovaných pravidiel ANTLR vytvorí súbor tried, ktorý je potom použitý na analýzu voliteľného jazyka. Pre rôzne jazyky musia byť definované rôzne vstupné gramatiky. Nástroje slúžiace na spracovanie jazyka, ako je ANTLR, pozostávajú minimálne z dvoch dôležitých častí, a to:

- lexer, ktorého úlohou je spracovať tok znakov, ktorý následne rozdelí do značiek podľa definovaných pravidiel,
- parser, ktorý značky prečíta a interpretuje podľa svojich pravidiel.

Ako príklad vstupnej gramatiky definujme gramatiku jednoduchej kalkulačky. Začnime definovaním pravidiel jednoduchej aritmetickej operácie: 100+23.

```
grammar SimpleCalc;
add     : NUMBER PLUS NUMBER;
NUMBER: ('0'..'9')+ ;
PLUS  : '+';
```

Uvedený príklad obsahuje dve pravidlá pre lexer - NUMBER a PLUS - a jedno pravidlo pre parser - add. Pravidlá pre lexer začínajú vždy veľkým písmenom, zatiaľ čo pravidlá pre parser malým.

- NUMBER - definuje značku, ktorá je definovaná číslicou z rozsahu 0 - 9 vrátane (`'0'..'9'`), ktorá sa môže opakovať (+).
- PLUS - definuje značku s jediným znakom: +.
- `add` – definuje pravidlo pre parser, ktoré hovorí: “vyžadujem značky NUMBER, PLUS a NUMBER v uvedenom poradí.” Iné značky alebo značky v inom poradí vyústia do chybového hlásenia.

V príklade je definované, že sa môžu sčítať iba dve čísla. Ak chceme dosiahnuť neobmedzený počet sčítaných čísiel, pravidlo pre parser musíme upraviť nasledovne:

```
add: NUMBER (PLUS NUMBER)*
```

Symbol * znamená “nula alebo viac krát”. V prípade, ak chceme do operácie sčítania dodefinovať aj odčítanie, je potrebná nasledovná úprava pravidla pre parser a definovanie novej značky:

```
add: NUMBER ((PLUS | MINUS) NUMBER)*
MINUS : '-';
```

Symbol | reprezentuje “alebo”, teda “PLUS alebo MINUS”. Ak chceme analyzovať kompletné aritmetické operácie ako 1+2*3, existuje rekurzívny spôsob ako toho dosiahnuť.

```
expr    : term ( ( PLUS | MINUS )  term )* ;
term    : factor ( ( MULT | DIV ) factor )* ;
factor: NUMBER ;
MULT  : '*';
DIV   : '/';
```

Aby sme sa vyhli upozorneniam a chybovým hláseniam pri výrazoch, ktoré v sebe obsahujú “biele” znaky, teda medzery, tabulátory a podobne, musíme tieto znaky zadefinovať. V prípade výrazu 1 +2 * 3 by bol ale výstup lexer-u nasledovný: “NUMBER WHITESPACE PLUS NUMBER WHITESPACE WHITESPACE MULT WHITESPACE”. Nástroj ANTLR preto ponúka dva kanály, pomocou ktorých komunikuje lexer s parserom, štandardný a skrytý. Skrytý

kanál využijeme práve na to, aby výstup lexer-u bol pekne čitateľný. Kanál zadefinujeme pri definícii "bielych" znakov:

```
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ { $channel = HIDDEN; };
```

Pre čitateľnejší zdrojový kód vstupnej gramatiky môžeme použiť aj komentáre a pomocou nich rozdeliť pravidlá pre lexer a parser. Jednoriadkové komentáre sú definované pomocou "//" a viacriadkové komentáre pomocou "/* … */". Usporiadaná vstupná gramatika pre jednoduchú kalkulačku potom môže vyzerať nasledovne:

```
grammar SimpleCalc;

tokens {
	PLUS	= '+' ;
	MINUS	= '-' ;
	MULT	= '*' ;
	DIV	= '/' ;
}

/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/
expr	: term ( ( PLUS | MINUS )  term )* ;
term	: factor ( ( MULT | DIV ) factor )* ;
factor	: NUMBER ;

/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/
NUMBER	: (DIGIT)+ ;
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+	{ $channel = HIDDEN; } ;
fragment DIGIT	: '0'..'9' ;
```

Posledným krokom je uvedenie definovaného parseru do prevádzky, pričom toto uvediem v jazyku C#, v ktorom budeme vizualizátor HDL jazykov implementovať.

```
@members {
   public static void Main(string[] args) {
      SimpleCalcLexer lex = new SimpleCalcLexer(new ANTLRFileStream(args[0]));
         CommonTokenStream tokens = new CommonTokenStream(lex);

      SimpleCalcParser parser = new SimpleCalcParser(tokens);

      try {
         parser.expr();
```

```
        } catch (RecognitionException e)  {
          Console.Error.WriteLine(e.StackTrace);
        }
    }
}
```

Uvedený zdrojový kód predstavuje zvyčajnú postupnosť operácií: zoberie vstupný tok dát, vloží ich do vytvoreného lexeru, výstupný tok značiek z lexeru vloží do parseru a potom zavolá jednu z metód parsera. Jazyk, v ktorom bude parser pracovať je nutné v gramatike definovať:

```
grammar SimpleCalc;

options {
   language=CSharp2;
}
```

2.8 Formát extrahovaných informácií

2.8.1 XML

Pretože cieľom tohto projektu je vizualizovať digitálne systémy opísané rôznymi opisnými jazykmi, je potrebné zaviesť univerzálny, prechodový jazyk. Ideálnym riešením je použitie jazyka XML (eXtensible Markup Language). XML je univerzálny značkovací jazyk, ktorý je zovšeobecnením HTML jazyka. Bol štandardizovaný konzorciom W3C. Slúži na vecný opis štruktúry dokumentov, alebo na transformáciu z iného formátu. Syntax jazyka XML je jednoduchšia ako HTML, pričom nemá preddefinované značky.
Vďaka tomuto univerzálnemu jazyku je možné jednoducho doimplementovať podporu ďalších jazykov na opis digitálnych systémov. Každý jazyk bude následne preložený do univerzálneho XML formátu. Vizualizácia, alebo prípadne aj simulácia dostane na vstupe XML súbor a na základe tohto súboru vykreslí požadovaný výsledok. Výhoda jazyka XML spočíva v jeho dostupnosti, a taktiež existuje množstvo dostupných programov, knižníc a nástrojov na prácu s XML.
Pre účely transformácie kódu z opisných jazykov digitálnych systémov do XML existuje niekoľko špecifických XML schém. V rámci analýzy spomeniem XML schémy HDML, VXML, HXML, MoML a IP-XACT.

2.8.2 IP-XACT

IP-XACT je štandard, ktorý opisuje XML schému pre elektronické komponenty a ich dizajn. Tento štandard bol vytvorený konzorciom SPIRIT pre automatickú konfiguráciu a integráciu nástrojov. Cieľom konzorcia bolo vyvinúť XML schému na jednoduché zdieľanie hardvérových návrhov medzi IP poskytovateľmi a systémovými dizajnérmi.
Keď vytvárame hardvérový dizajn (vo VHDL, SystemC, ...), môžeme takýto dizajn opísať prostredníctvom XML súborov (použitím IP-XACT schémy), ktorá obsahuje informácie o vonkajších prepojeniach, o mape pamäte, o rozhraniach zberníc, ...ale taktiež obsahuje meta-dáta ako použitý jazyk alebo názov výrobcu. Hardvérový návrh môže byť jednoducho importovaný do projektu, ktorý využíva rovnaké IP-XACT funkcie.

Použitie spoločného formátu pre opis hardvéru je veľmi výhodné. Ponúka možnosť vytvorenia mnohých nástrojov, ako napríklad:

- Generovanie dokumentácie z XML
- Automatické vytvorenie XML opisných súborov z VHDL alebo SystemC
- Generovanie RTL kódu, ktorý spája IP
- Prekladanie hardvérového opisu do iného jazyka

SPIRIT konzorcium vyvíja a špecifikuje IP-XACT štandard od roku 2003. Počas vývoja bolo vydaných niekoľko verzií, pričom neustále narastal rozsah funkcií, ako napríklad spôsob vyjadrenia IP, nástroje pre import a integrovanie IP do návrhov, prepojenie, verifikácia a použitie modelov.
Štandardizovaná forma IP-XACT zahŕňa: komponenty, systémy, rozhrania pre zbernice a prípojky, abstrakcie zberníc, a detaily komponentov vrátane adresných máp, opisov registrov a polí, a popis súborov pre použitie v automatizovanom návrhu, overovaní, dokumentovaní. Súbor XML schém, ktorých formu opisuje World Wide Web Consorcium (w3c) a množina sémantických pravidiel konzistencie (SCR) sú zahrnuté.
Nakoľko forma IP-XACT je pomerne rozšírená a ponúka široké spektrum použitia, analyzujeme, aké sú možnosti zápisu dôležitých informácií potrebných pre vizualizáciu modelov.
Entita/Model – formát IP-XACT ju definuje ako komponent. Z dôležitých atribútov je možné použiť iba atribúty názov komponentu a porty.

Port – formát IP-XACT definuje port ako rozhranie. Porty sú rozdelené na káblové a transakčné. Káblové porty uchováva logické hodnoty, transakčné porty iné typy informácií. Z atribútov je možné uchovať názov, typ portu, ale aj dátový typ. Dátový typ je môže byť adresa, dáta, reset alebo clock.

Takto vyzerá ukážka zápisu portu v XML formáte:

```
<spirit:port>
<spirit:logicalName>Clock</spirit:logicalName>
<spirit:wire>
<spirit:qualifier>
<spirit:isClock>true</spirit:isClock>
```

```
</spirit:qualifier>
<spirit:onSystem>
<spirit:group>clk</spirit:group>
<spirit:width>1</spirit:width>
<spirit:direction>out</spirit:direction>
</spirit:onSystem>
</spirit:wire>
</spirit:port>
```

Signál/Net/Kanál – Tieto elementy sú definované v IP-XACT ako komponenty. Z dôležitých atribútov je zastúpený názov, dátový typ (podobne ako pri porte) a je zastúpený aj zoznam prepojených portov.

Takto vyzerá ukážka zápisu signálu v XML formáte:

```
<spirit:channels>
<spirit:channel>
<spirit:name>masterChannel</spirit:name>
<spirit:displayName>Channel for Master communication</spirit:displayName>
<spirit:busInterfaceRef>InterfaceA</spirit:busInterfaceRef>
<spirit:busInterfaceRef>InterfaceB</spirit:busInterfaceRef>
</spirit:channel>
</spirit:channels>
```

Hodiny – hodiny sú v IP-XACT definované ako káblový port. Z dôležitých atribútov sú zastúpené všetky atribúty, teda názvy, perióda, pomer úrovní, oneskorenie prvej hrany, ako aj atribút či je prvá hrana nábežná alebo dobežná. Takto vyzerá ukážka zápisu atribútov pre hodiny, konkrétne perióda, pomer úrovní, oneskorenie prvej hrany a nábežnosť/dobežnosť prvej hrany.

```
<spirit:port>
<spirit:name>clk</spirit:name>
<spirit:wire>
<spirit:direction>in</spirit:direction>
<spirit:driver>
<spirit:clockDriver spirit:clockName="clk">
<spirit:clockPeriod>8</spirit:clockPeriod>
<spirit:clockPulseOffset>4</spirit:clockPulseOffset>
<spirit:clockPulseValue>1</spirit:clockPulseValue>
</spirit:clockDriver>
</spirit:driver>
<spirit:constraintSets>
<spirit:constraintSet spirit:constraintSetId=”timing”>
<spirit:timingConstraint
spirit:clockName=”hclk”>40</spirit:timingConstraint>
```

```
<spirit:timingConstraint spirit:clockName=”hclk”spirit:clockEdge=”fall”
spirit:delayType=”min”>30</spirit:timingConstraint>
<spirit:timingConstraint spirit:clockName=”hclk” spirit:clockEdge=”fall”
spirit:delayType=”max”>50</spirit:timingConstraint>
</spirit:constraintSet>
</spirit:constraintSets>
</spirit:wire>
</spirit:port
```

2.8.3 Ostatné formáty

AIRE / CE
AIRE/CE (The Advanced Intermediate Representation with Extensibility/Common Environment) je objektovo-orientovaný spôsob reprezentácie štruktúry, navrhnutý najmä pre reprezentáciu VHDL opisu. Jedná sa o množinu inštancií objektov, ktoré sú vzájomne prepojené. Tieto inštancie reprezentujú abstraktný syntaktický strom (AST) a metódy tried objektov reprezentujú aplikačné programové rozhranie (API). AIRE/CE využíva dva spôsoby reprezentácie, vnútornú prechodnú reprezentáciu (IIR) a súborovú prechodnú reprezentáciu (FIR).

HDML
HDML (Hardware Description Markup Language) je jednoducho editovateľná a rozšíriteľná špecifikácia XML, určená najmä pre jazyk VHDL, pričom dokáže reprezentovať celú štruktúru jazyka. Výhodou je jednoduchá rozšíriteľnosť.

VXML
VXML (VHDL Markup Language) slúži na univerzálnu reprezentáciu najmä pre VHDL jazyk. Definujú ho dve prechodné reprezentácie, VHDL XML Design Intermediate (VXDI) a VHDL XML Document Represaentation (VXDR). Prvá z nich slúži na reprezentáciu VHDL opisu, kým druhá na tvorbu dokumentácie, buď priamym spôsobom, alebo z VXDI.

MoML
MoMl (Modeling Markup Language) je XML špecifikácia slúžiaca na opis hardvérových návrhov. Slúži na definovanie prepojení medzi komponentmi. MoML disponuje abstraktnou syntaxou GSRC, ako aj podporou vizualizácie VHDL opisu. Výhodou tohto formátu je, že sa dá jednoducho rozšíriť, pretože je abstraktný a je potrebné ho konkretizovať. Ďalšie výhody jazyka MoML spočívajú v jeho rozšíriteľnosti, nezávislosti na implementačnom prostredí, taktiež je jednoducho použiteľný v internetovom svete (podobne ako XML).

HXML
HXML (Hardware eXtensible Markup Language) je univerzálny jazyk pre VHDL a Verilog.

Okrem spomenutých formátov existuje mnoho ďalších, ako napríklad MHDL, EDAXML, xADL a iné.

2.9 Možnosti vizualizácie extrahovaných informácií

Po extrahovaní informácií z jednotlivých jazykov do spoločnej reprezentácie je potrebné transformovať túto reprezentáciu do grafickej podoby. Na zobrazenie hierarchickej štruktúry musíme zvoliť nejakú knižnicu, pomocou ktorej vieme vykresliť a poprepájať objekty. Našli sme 4 knižnice, ktoré by sme mohli používať pri vizualizácií opisu:

Netron Graph Library – voľne dostupná a šíriteľná knižnica. Je postavená na platforme .NET Framework. Umožňuje základnú prácu s grafickými objektmi ako je napr. ovládanie pomocou myši, zmena rozmeru a umiestnenia. Táto knižnica bola použitá aj v diplomových prácach, ktoré boli vypracované na našej fakulte. V týchto prácach boli knižnice upravené podľa potreby a taktiež boli vytvorené aj knižnice VHDL Shapes Library a HDL Shapes Library. VHDL Shapes Library bola implementovaná ako nadstavba nad upravenou knižnicou Netron Graph Library. Boli v nej implementované objekty potrebné na vizualizáciu VHDL modelov a potrebné funkcionality na vykresľovanie týchto objektov. HDL Shapes Library je upravená verzia knižnice VHDL Shapes Library, boli nej upravené triedy tak, aby slúžili na grafickú reprezentáciu Verilog opisov. S malými úpravami knižnice VHDL Shapes alebo HDL Shapes by sme mohli veľmi ľahko implementovať knižnicu, ktorá by slúžila na vykreslenie objektov VHDL, Verilog a SystemC.

Okrem Netron Graph Library existujú aj ďalšie knižnice, ktoré umožňujú vykresľovanie a úpravu (premiestnenie, zmena rozmeru, atď.) rôznych grafov a objektov. Avšak pre tieto knižnice neexistujú žiadne pomocné knižnice, ktoré by boli vytvorené na vykreslenie HDL objektov, preto by sme ich museli vytvoriť my. Výhody niektorých z týchto knižníc sú, že sú novšie.

Diagram.NET Library – bezplatný a voľne šíriteľný nástroj pre tvorbu diagramov. Obsahuje WinForm kontrol, pomocou ktorého je možné ľahko ovládať vytvorené objekty v grafickom rozhraní. Taktiež umožňuje pridávanie a ovládanie objektov pomocou C# kódu. Jej nevýhodou je, že neexistuje pre ňu žiadna dokumentácia, takže vytvorenie potrebných objektov by bolo náročné [18].

NShape .NET Framework – voľne dostupný a šíriteľný framework pre .NET. Umožňuje vykreslenie, prezeranie a úpravu rôznych grafov, schém alebo diagramov. Taktiež umožňuje ovládať vytvorené objekty cez grafické rozhranie (premiestnenie, zmena rozmeru, atď.). Štýly vytvorených objektov tiež je možné zmeniť veľmi ľahko. NShape obsahuje veľmi dobrú dokumentáciu, preto vytváranie potrebných objektov by bolo jednoduchšie, ako v prípade knižnice Diagram.NET [19].

Problémom pri vykresľovaní grafov, teda aj pri vizualizácii je zobrazené prvky vhodne rozmiestniť. Pri vykresľovaní grafov existuje niekoľko prístupov, ako postupovať, aby vykreslený graf vyzeral dobre. Neexistuje žiadny algoritmus, ktorý by vykresľoval grafy najlepšie, lebo sa nedá jednoznačne určiť, ktoré vykreslenie je dobré a ktoré nie. Avšak existujú niektoré kritéria, na základe ktorých sa dá orientačne posúdiť kvalita vykreslenia.

Medzi tieto kritéria patrí [21]:

- Počet prekrížení hrán – čo najmenej
- Veľkosť plochy – čo najmenšia
- Suma dĺžok všetkých hrán – čo najmenej
- Maximálny rozdiel medzi dĺžkami jednotlivých hrán – čo najmenej
- Počet zlomov hrán – čo najmenej

Niektoré postupy vykresľovania grafov [21]:

- Topológia – Tvar – Metrika – výsledkom je pravouhlý graf
- Hierarchický prístup – výsledkom je graf vykreslený v tvare stromu, kde je možné identifikovať jednotlivé vrstvy
- Prírastkový prístup
- Silový prístup – jednotlivé hrany v grafe predstavujú sily pôsobiace na vrcholy, výsledkom je graf s najmenšou „energiou".

2.10 VCD súbor

VCD (value change dump) je ASCII súbor generovaný nejakým simulátorom. Obsahuje informácie o zmenách hodnôt vybraných premenných v modeli. VCD formát bol definovaný v štandarde IEEE 1364-2001 pre jazyk Verilog, ale aj simulátory ostatných jazykov dokážu vytvoriť VCD súbor.

Existujú dva typy VCD súborov:

- Štvorstavový: reprezentuje zmeny premenných pomocou štyroch stavov 0, 1, x, z, neobsahuje informácie o sile.
- Rozšírený: reprezentuje zmeny a silu premenných.

VCD súbory generujú VCD systémové úlohy, ktoré treba definovať v zdrojovom kóde. Následne spustíme simuláciu a počas simulácie systém vytvorí VCD súbor.

VCD systémové úlohy, ktoré môžeme pridať do zdrojového kódu sú nasledovné:

- $dumpfile – určí súbor, do ktorého sa uložia výsledky simulácie
- $dumpvars – určí ktoré premenné budú uložené do súboru
- $dumpoff, $dumpon - slúžia na zastavenie a na pokračovanie v uložení
- $dumpall – príkaz predstavuje kontrolný bod, kedy sa hodnoty všetkých premenných sa uložia do súboru
- $dumplimit – môže určiť veľkosť VCD súboru
- $dumpflush – pred ukončením simulácie vytvorí VCD súbor, aby sme mohli sledovať výsledky už počas simulácie

Nasledujúci príklad ilustruje formát štvorstavového VCD súboru:

```
$date Sept 10 2008 12:00:05 $end
$version Example Simulator V0.1 $end
$timescale 1 ns $end
$scope module top $end
$var wire 32 ! data $end
$var wire 1 @ en $end
$var wire 1 # rx $end
$var wire 1 $ tx $end
$var wire 1 % err $end
$var wire 1 ^ ready $end
$upscope $end
$enddefinitions $end
#0
b10000001 !
0@
1#
0$
1%
0^
#1
1@
#2
0@
#3
1@
#4
0@
#11
b0 !
0#
#16
b101010101010110101010101010101 !
1#
#20
0%
#23
```

2.11 Vizualizátory VCD súborov

Po simulácií VHDL, Verilog alebo SystemC modelov, výsledky simulácia so uložia do VCD súboru. Na vizualizáciu VCD súborov môžeme používať externý nástroj alebo môžeme vytvoriť vlastný komponent, ktorý bude slúžiť na vizualizáciu simulácie. V nasledujúcej časti uvediem niektoré voľne dostupné nástroje, ktoré dokážu vizualizovať VCD súbory.

GTKWave
Nástroj na vizualizáciu výsledkov simulácie vo forme časového priebehu signálov. Dokáže prečítať súbory typu LXT, LXT2, VZT, FST, GHW, VCD a EVCD. Pridávanie signálov je jednoduché, zobrazuje aj hierarchiu modelov. Prostredníctvom TCL skriptu je možné vynútiť určité prednastavenia nástroja, napr.: môžeme nastaviť, ktoré porty chceme vizualizovať pri otvorení, môžeme priblížiť nejakú časť priebehu signálov, atď. Je implementovaný v programovacom jazyku C, preto by bolo možné jednoducho implementovať do nášho systému.

WaveViewer
Voľne dostupný nástroj na vizualizáciu VCD, EVCD, TDML a BTIM súborov vo forme časového priebehu signálov. Medzi jeho funkcie patria: možnosť vybrať porty, ktoré chceme zobraziť, priblížiť alebo vzdialiť zobrazené priebehy, triediť signály podľa mena, vyhľadať signály a hodnoty signálov, atď. Ovládanie nástroja je dosť náročné a nezobrazuje ani hierarchiu modelov.

Wave VCD Viewer
Ďalší nástroj na vizualizáciu VCD súborov. Taktiež vizualizuje formou časového priebehu signálov. Podporuje iba VCD súbory. Poskytuje podobné funkcie, ako predchádzajúce produkty. Pridávanie signálov je jednoduchý, zobrazuje aj hierarchiu modelov.

VCD View
Jednoduchý nástroj na vizualizáciu VCD súborov vo forme časového priebehu signálov. Nepodporuje žiadne iné formáty. Má podobné funkcie, ako predchádzajúce nástroje. Pridávanie signálov nie je tak jednoduchý, ako u ostatných vizualizátorov a nezobrazuje ani hierarchiu modelov.

Michal Nosáľ vo svojej diplomovej práci[16] vytvoril knižnicu **Simulation Graph Library**, ktorá spolu s komponentom VCD Parser (taktiež vytvorený v rámci tejto práce) umožňujú vizualizovať VCD súbory. Obidva komponenty boli implementované v programovacom jazyku C#.

Simulation Graph Library + VCD Parser
Po simulácii modelov, výsledky simulácie sú najprv uložené do VCD súboru. Komponent VCD Parser slúži na extrahovanie potrebných informácií z VCD súboru a tieto informácie potom uloží do XML dokumentu. Knižnica Simulation Graph Library sa potom spracuje tieto informácie a vizualizuje výsledky simulácie vo forme časového priebehu signálov.

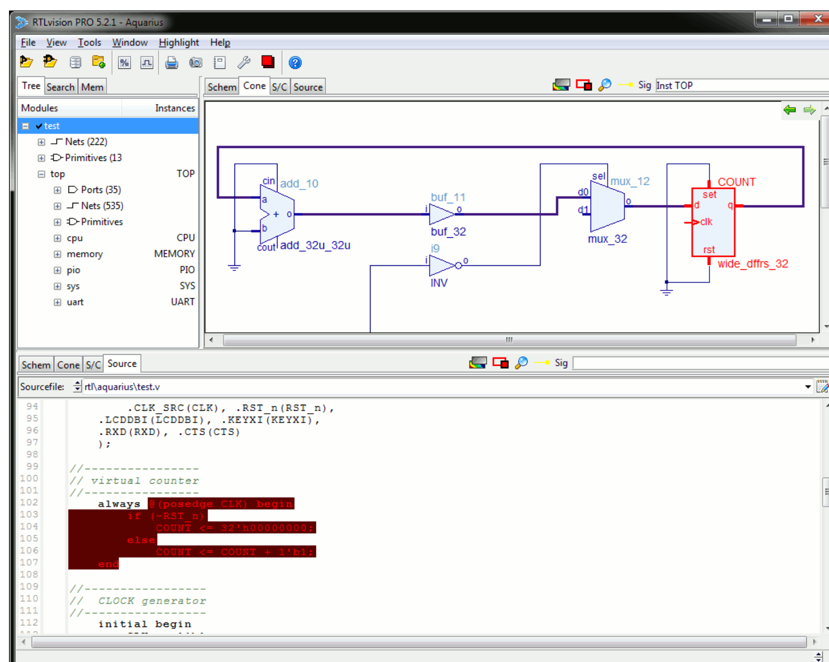
Nevýhodou vyššie uvedených nástrojov a knižníc je, že výsledky simulácie zobrazia iba vo forme časového priebehu signálov. Pre nás takáto vizualizácia nie je postačujúca. Potrebujeme vytvoriť takú vizualizáciu simulácie, kde hodnoty signálov a portov bude možné sledovať aj vo vizualizovanej štruktúre modulov. Časový priebeh signálov môže slúžiť ako doplnok vizualizácie simulácie.

3 Existujúce riešenia

Myšlienka samotnej vizualizácie nie je vôbec nová. Už pri vznikoch týchto opisných jazykov sa formulovali prvé nástroje, ktoré zjednodušujú a sprehľadňujú tisíce riadkov kódu. Vizualizáciou kódu sa tiež predišlo chybám, ktoré by sa v množstve kódu stratili. V nasledujúcich kapitolách sú opísané niektoré existujúce riešenia a konkrétne programy, ktoré slúžia na vizualizáciu kódov jazykov VHDL, Verilog a SystemC.

3.1 RTLVision PRO

Aplikácia RTLVision je komplexným riešením vizualizácie pre opisné jazyky od firmy Concept Engineering GmbH. Podporuje opisné jazyky ako VHDL, Verilog, System Verilog. Aplikácia vizualizuje kód v prehľadnej forme. K dispozícií je tiež presné navádzanie zobrazovaných prvkov na kód ktorý reprezentujú. RTLVision tiež podporuje presné zobrazovanie hodinového signálu, ktoré býva často neprehľadné. Za jeho pomoci je tiež možné vypracovať komplexné stromy priebehov signálov. Aplikácia tiež obsahuje nástroje na vykreslenie a prácu s krivkami signálov, ktoré prehľadne vykresľuje Podporuje tiež inkrementálnu kompiláciu a aj rekompiláciu iba vybraných oblastí, čo skracuje dobu vývoja. Ako ďalším podporným nástrojom na prácu je automatický tvorca systémovej dokumentácie. Na obrázku Obr. 3.1 je zobrazené grafické používateľské rozhranie aplikácie.



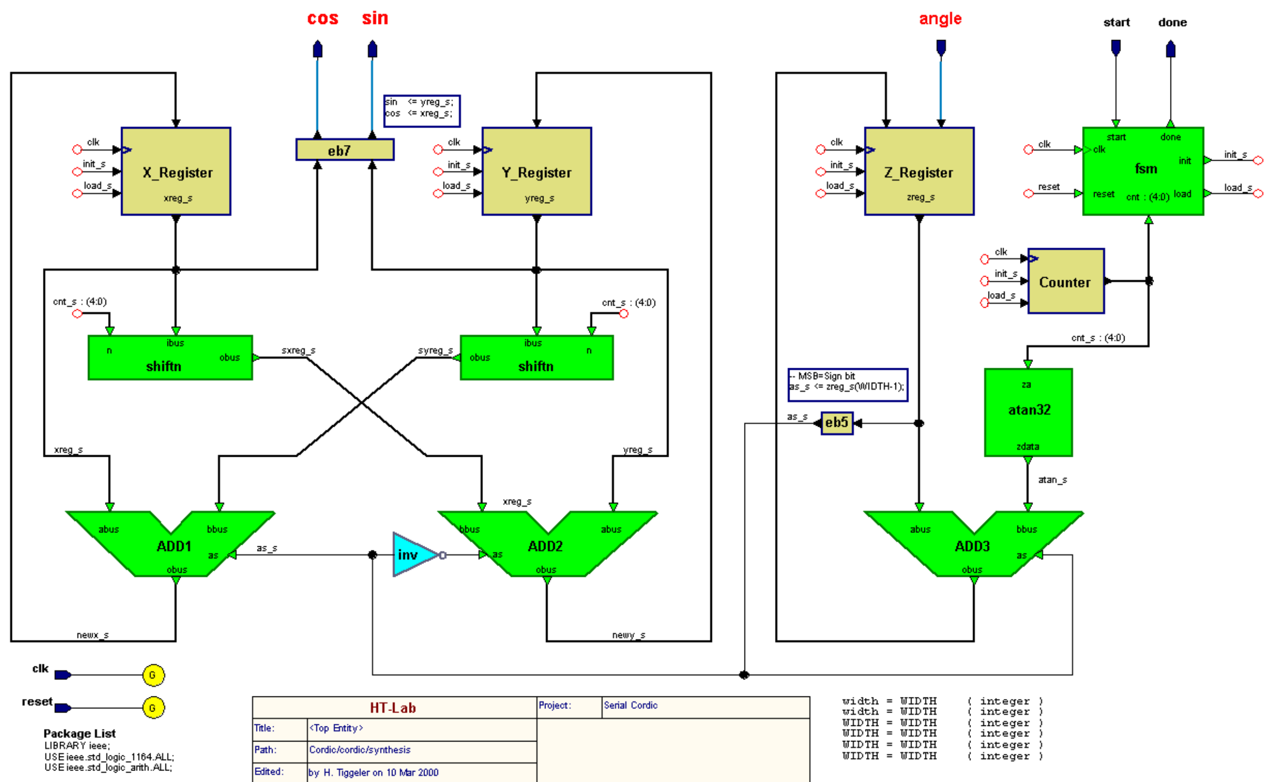
Obr. 3.1: Grafické používateľské rozhranie RTL Vision PRO.

3.2 Mentor Graphics HDL Designer

Táto aplikácia v sebe spája možnosti hĺbkovej analýzy, pokročilého editovania a kompletného projektového manažmentu. Podporuje opisné jazyky ako Verilog, VHDL a System Verilog a to aj FPGA a ASCI dizajn. Príklad vizualizovaného modelu sa nachádza na obrázku Obr. 3.2.

Hlavné výhody aplikácie :

- Podrobný dizajn navrhovania a overovania sad pravidiel.
- Interaktívna HDL vizualizácia a tvorba pomocných nástrojov.
- Automatická pomoc pri generovaní dokumentácie.
- Inteligentné ladenie a analýza



Obr. 3.2: Príklad modelu vizualizovaného v aplikácii Mentor Graphics HDL Designer.

Ďalšou výhodou tejto aplikácie je spolupráca s IP-XACT. Pri tvorbe dizajnu je možné použiť a modifikovať IP-XACT modely.

HDL Designer odporuje niekoľko editorov, za pomoci ktorých môže byť práca na tvorbe kódu výrazne uľahčená. Medzi tieto editory patrí :

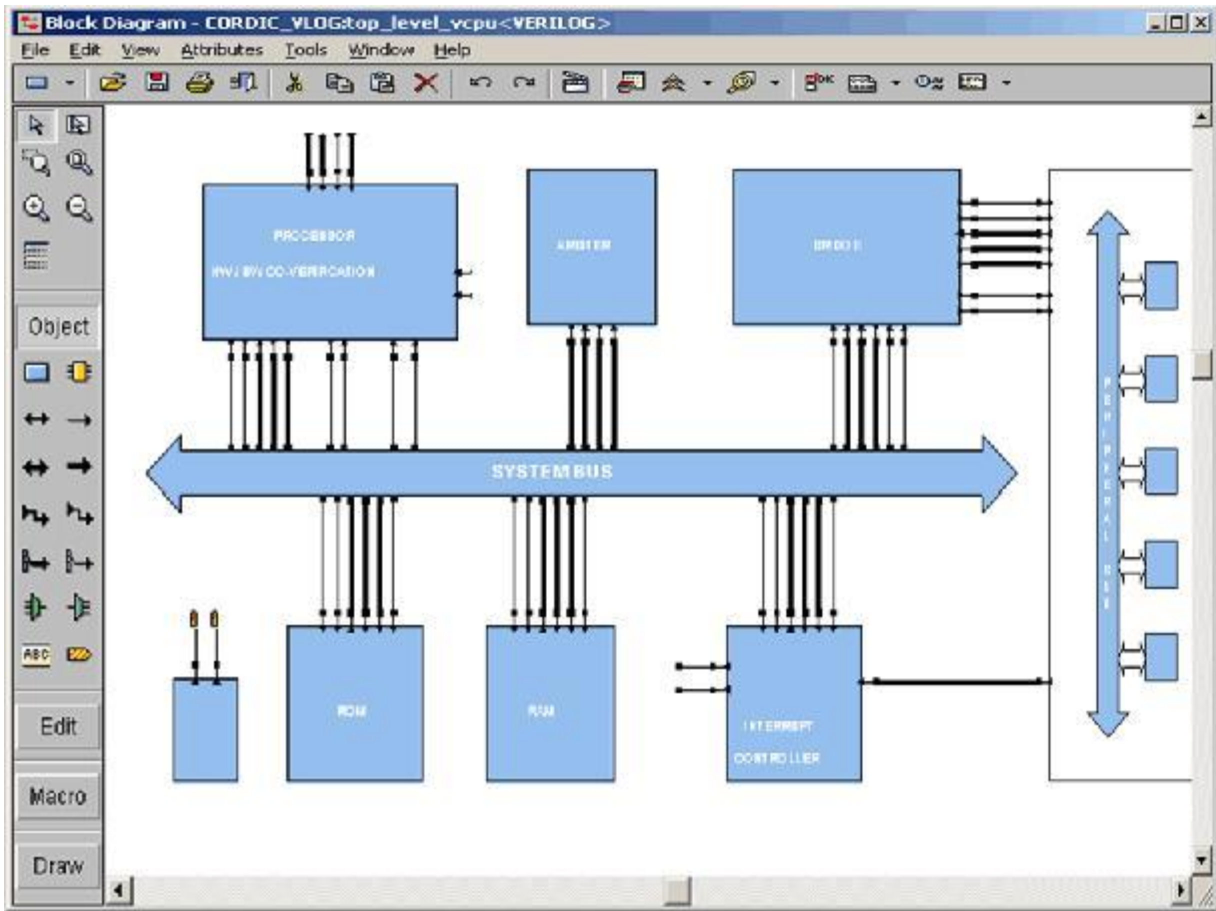
- Editor dizajnu rozhraní založený na tabuľkovom editore (IBD)
- Blokový diagram
- Stavový stroj
- Pravdivostná tabuľka
- Diagramy toku
- K doplneniu HDL Designer obsahuje tiež EMACS/vi HDL-aware textový editor.

3.3 PLFire

Vizualizačný nástroj PLFire je pomerne jednoduchý nástroj len s obmedzenou možnosťou vizualizácie. Je určený na vizualizáciu jazyka VHDL a to len Phase Logic (PL) obvodov. Tento nástroj je však zaujímavý preto lebo vznikol na pôde univerzity, ako výsledok práce skupiny študentov. Jedná sa o univerzitu Southern Metodist University of Dallas. Program PLFire je napísaný v jazykoch C/C++ a na zobrazovanie používa knižnice OpenGL.

3.4 Visual Elite Mentor Graphic

Jedná sa o aplikáciu, ktorá je pokročilou platformou pre TLM a RTL dizajn. Je vyvíjaná spoločnosťou Mentor Graphic, tak ak už spomenutý nástroj HDL Designer. Táto platforma umožňuje dizajnérom a systémovým architektom aby boli schopný intuitívne prepojiť SystemC, TLM 2.0 a HDL systémy do komplexnej SoC architektúry.
Visual Elite je postavený na silnej HDL infraštruktúre, ktorá poskytuje najmodernejší elektronický systém (ESL) a mechanizmy transakčnej modelovacej úrovne (TLM). Nástroj výrazne zjednodušuje a zrýchľuje návrh v hierarchickom tvare a tiež uľahčuje návrh algoritmických systémov. Prináša dizajn a integračné platformy v rámci všetkých abstraktných úrovní a domén.
Tento nástroj je možné prepojiť z niektorými simulátormi, ako napríklad : Vista , Questa. Príklad vizualizovaného modelu, ako aj grafické používateľské rozhranie aplikácie sa nachádza na obrázku Obr. 3.3.



Obr. 3.3: Príklad modelu vizualizovaného v aplikácii Visual Elite.

4 Špecifikácia požiadaviek

Táto kapitola obsahuje požiadavky, ktoré sú kladené na vytváraný systém. Tieto požiadavky sú podľa charakteru rozdelené na funkcionálne požiadavky, požiadavky na používateľské rozhranie, ako aj systémové požiadavky. Zároveň je spomenutým požiadavkám pridelená priorita rozdelená do troch úrovni nasledovne:

1. **Nutné:** Hovorí o tom, že táto požiadavka má najvyššiu prioritu, je dôležitá tak pre celkovú funkčnosť výsledného systému, ako aj pre splnenie zadania tohto projektu.

2. **Povinné:** Nesplnenie takejto požiadavky nemá za následok nefunkčnosť systému, no má za následok nesplnenie kompletného zadania tohto projektu, teda bola by implementovaná iba nejaká funkčná časť systému.

3. **Rozširujúce:** Tieto požiadavky nevyplývajú zo zadania a ich splnenie nie je preto nutné a taktiež nie je nutné pre funkčnosť samotného systému, ale môže výsledný systém rozšíriť o zaujímavé alebo užitočné funkcie.

4.1 Funkcionálne požiadavky

Nutné:

1. Systém musí vedieť transformovať model zapísaný v jazyku VHDL , Veriog a SystemC do formátu , ktorý umožní jeho vizualizáciu.
2. Systém musí vedieť transformovaný model korektne vizualizovať a zobraziť všetky jeho časti .
3. Všetky zobrazené časti modelu musia byť správne prepojené podľa zadaného opisu.
4. Systém musí vedieť vizualizovať aj hierarchiu štruktúry modelu.
5. Systém musí umožňovať jednoduché vnáranie sa a vynáranie v hierarchii štruktúry opísaného modelu.
6. Systém musí umožňovať zmenu usporiadania zobrazených častí systému na jednej úrovni.
7. Systém musí dokázať zobraziť dostatočné množstvo informácií na to, aby bolo možné jednoznačne identifikovať jednotlivé časti systému. Názvy týchto častí sa musia zhodovať s názvami v opise modelu.

Povinné:

1. Systém by mal umožňovať vizualizáciu simulácie modelov opísaných v jazykoch VHDL, Verilog a SystemC.
2. Systém by mal automaticky určiť vhodné rozmiestnenie vizualizovaných prvkov modelu tak, aby sa jednotlivé prvky neprekrývali, aby bol každý viditeľný a aby došlo k čo najmenšiemu počtu prekrížených prepojení medzi časťami systému.
3. Systém by mal byť jednoducho rozšíriteľný o ďalšie HDL jazyky.
4. Systém by mal umožňovať ukladanie vizualizovaných modelov.
5. Systém by mal umožňovať exportovanie výsledkov simulácie.

Rozširujúce:

1. Systém by mohol umožňovať exportovanie vizualizovanej podoby modelu ako aj jeho jednotlivých úrovní.
2. Systém by mohol umožňovať exportovanie výrezov vizualizovaného modelu, ako aj jeho jednotlivých úrovní.
3. Systém by mohol umožňovať zobrazenie viacerých vizualizovaných modelov, vizualizovaných simulácií, viacerých modelov opísaných v rôznych jazykoch v jednom okne programu.
4. Systém by mohol umožňovať vytváranie a editáciu testovacích vstupov pre simulované modely.
5. Systém by mohol zobrazovať textovú podobu opísaného modelu.
6. Systém by mohol umožňovať vykonávanie zmien v textovej podobe opisu modelu a jej následné uloženie.
7. Systém by mohol poskytovať editor súborov s opismi móde.
8. Systém by mohol zvýrazňovať syntax podporovaných jazykov v editore.

4.2 Používateľské rozhranie

Aby bolo možné vizualizovať modely digitálnych systémov, je potrebné používať grafické rozhranie. Užívateľ by mal mať možnosť formou dialógového okna otvoriť jeden z podporovaných VHDL, Verilog alebo SystemC súborov. Následne je možné užívateľa informovať napríklad o priebehu parsovania. Prípadné chyby, či už vo vstupnom súbore, alebo obmedzenia týkajúce sa napríklad rozmiestnenia komponentov na obrazovke budú zobrazené užívateľovi. Následne bude vykreslené rozmiestnenie jednotlivých komponentov na obrazovke. Je veľmi dôležité, aby boli rozmiestnené čo najlogickejšie, to znamená aby sa neprekrývali alebo aby bolo čo najmenej prekrížených signálov. Samozrejme, že v prípade príliš veľkých systémov nebude možné všetky komponenty zobraziť ideálnym spôsobom, preto je potrebné definovať

politiku rozmiestňovania komponentov na obrazovke. Výsledná schéma by nemala ostať nemenná, preto by malo byť možné jednotlivé komponenty jednoduchým podržaním myši presúvať.

Celé grafické rozhranie by malo byť jednoduché a intuitívne. Prvky na používanie programu musia byť čo najrýchlejšie dostupné, najlepšie formou panela. Pri návrhu takéhoto panelu je vhodné jednotlivé ovládacie prvky zastúpiť ikonami, ktoré urýchlia používanie. Takéto ikony musia graficky čo najlepšie opísať danú funkciu. Zároveň by mali byť všetky, alebo väčšina prvkov dostupných aj z menu, kde je dôležité pamätať na logické usporiadanie jednotlivých položiek menu. Hlavnú časť grafického rozhrania bude tvoriť okno s vizualizovaným systémov. Tu je dôležitých viacej faktorov, ako napríklad farba pozadia, ako aj farba komponentov, ich veľkosť, rozloženie, možnosť posúvania sa po obrazovke a iné. Pri návrhu používateľského rozhrania je dôležité uvažovať aj o zjednodušení ovládania. To je možné docieliť napríklad možnosťou používania skratiek, ale taktiež aj poskytnutím čo najväčšieho množstva nápovied.

4.3 Systémové požiadavky

Navrhovaný nástroj by mal byť dostupný pre čo najväčšie množstvo užívateľov. To znamená, že musíme systémové požiadavky špecifikovať tak, aby spustenie programu bolo jednoduché a nevyžadovalo napríklad špeciálny operačný systém. Preto sa zameriame na platformu Windows, ktorá je najrozšírenejšia a ponúka veľa možností pri implementácii nástroja. Vzhľadom na to, že nástroj bude vyvíjaný v jazyku C#, bude potrebné použiť vhodný framework, napríklad .NET verzie 3,5.

5 Hrubý návrh riešenia

5.1 Extrakcia informácií z opisov modelov do súboru XML

V tejto časti opíšeme nástroje a postupy toho, ako budú extrahované potrebné informácie z jednotlivých jazykov. V prípade jazykov VHDL a Verilog je to skoro ten istý postup, preto sme ich zhrnuli do jednej kapitoly.

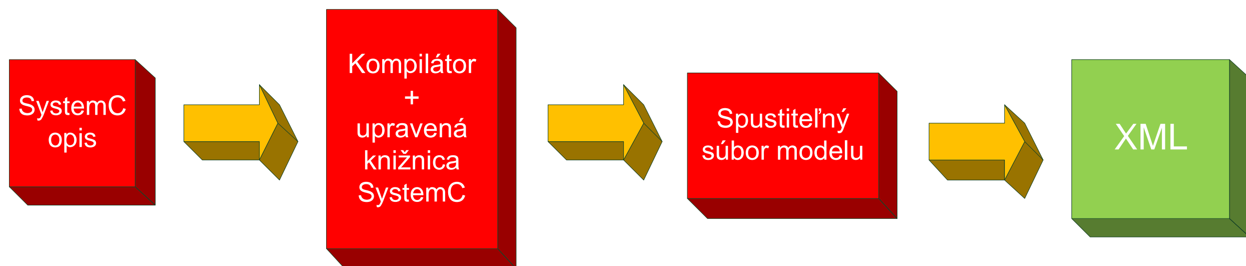
5.1.1 VHDL a Verilog

Pre jazyky VHDL a Verilog je postup extrakcie informácií z opisov modelov do súboru XML rovnaký. Pre analýzu jazykov využijeme generátor analyzátorov ANTLRv3. Ako bolo popísané v analýze, vstupom pre generátor je vstupná gramatika, ktorá musí byť definovaná zvlášť pre každý jazyk. Vstupnú gramatiku pre jazyk VHDL využijeme z diplomovej práce [6], pre jazyk Verilog z diplomovej práce [16]. Úprava vstupných gramatík je potrebná preto, aby boli zo zdrojových kódov jednotlivých jazykov extrahované rovnaké, teda podstatné informácie. Tieto informácie boli analyzované v kapitole 2.6. Vstupné gramatiky potom vložíme do generátora ANTLRv3, pričom jeho výstupom budú súbory, pomocou ktorých budeme analyzovať zdrojové kódy. Úprava gramatík a generovanie súborov pomocou generátora analyzátorov ANTLRv3 je teda jednorazová činnosť. Ďalej treba navrhnúť také moduly navrhovaného nástroja, ktoré budú mať za úlohu analýzu vstupného zdrojového kódu daného jazyka a jeho transformáciu do súboru XML. Pre jazyk VHDL nazveme tento modul VHDL2XML a pre jazyk Verilog Verilog2XML. Tieto moduly teda na základe vygenerovaných súborov budú analyzovať vstupné zdrojové kódy, transformovať ich do XML súboru a následne ich ukladať pre potrebu budúceho zobrazenia. Na obrázku Obr. 5.2 je znázornený diagram činností, ktoré je treba vykonať pre transformáciu zdrojových súborov jazykov VHDL a Verilog do súboru XML.

Obr. 5.1: Diagram činností potrebných na transformáciu a uloženie zdrojových súborov do súboru XML.

5.1.2 SystemC

Pri návrhu riešenia extrakcie informácií z modelov opísaných v tomto jazyku sme sa najskôr pokúšali nájsť nejaké vhodné existujúce riešenie, ktoré by sme mohli použiť prípadne modifikovať, alebo použiť aspoň niektoré jeho časti. Najlepším kandidátom sa najskôr javila aplikácia SystemCXML. Ako sme ale po kompilácii zistili, táto aplikácia nebola dokončená a preto sme sa jej využitím ďalej nezaoberali.
Ostatné aplikácie analyzované v kapitole 2.5.2 sa pre naše potreby taktiež použiť nedali, preto sme sa rozhodli, že zvolíme postup ako v prípade aplikácie SystemC+Visualizer a taktiež upravíme knižnicu SystemC, tak aby po skompilovaní modelu a jeho spustení, poskytla aplikácia výstup vo forme XML súboru. Tento návrh riešenia je graficky zobrazený na obrázku Obr. 5.3.



Obr. 5.2: Návrh spôsobu extrakcie informácií zo SystemC opisu.

5.2 Formát súboru XML

Aby bolo možné vizualizovať všetky tri jazyky, ktoré sú predmetom tejto práce, je potrebné používať jeden spoločný, univerzálny jazyk ako prechodový. Ten bude tvorený len tými konštrukciami, ktoré sú pre jazyky spoločné a zároveň dôležité. To je obzvlášť výhodné, pretože jednotlivé jazyky obsahujú množstvo konštrukcií, ktoré sú pri vizualizácii nevyužiteľné.
V časti analýzy sme rozobrali, aké sú možnosti použitia prechodného formátu. Najvhodnejší spôsob pre naše potreby je použiť XML formát. Taktiež sme analyzovali rôzne formáty a špecifikácie, avšak buď boli určené len pre jeden jazyk, alebo boli príliš univerzálne a obsahovali množstvo nepotrebných a zložitých konštrukcií. Navyše nepodporovali všetky požadované atribúty pre vizualizáciu, ktoré sú spomenuté v tabuľke č. 2.1. Preto sme sa rozhodli vytvoriť vlastný formát XML, ktorý bude umožňovať zapísať všetky požadované atribúty, s prihliadnutím na prípadné doplnenie.
Teraz si opíšeme, ako sme definovali prechodný XML formát. Tento opis budú tvoriť uzly a atribúty. Kým atribúty sú značky, ktoré nie sú ďalej rozvinuté, uzly rozvinuté sú, pričom ich opis bude rozobratý neskôr. Pri atribútoch je vždy spomenutý dátový typ atribútu, alebo možnosti. Výpis uzlov je zoradený podľa ich vnorenia.

Modules
Uzol *Modules* obsahuje všetky komponenty alebo moduly. Uzol nemá žiadne atribúty.

Vnorené uzly:
- *Module* – jednotlivé komponenty alebo moduly

Module
Uzol *Module* zodpovedá jednému komponentu (v prípade VHDL), alebo modulu (Verilog a SystemC).

Vnorené uzly:
- *PortList* – množina pripojených portov

Atribúty:
- *Name* – názov entity (String)
- *NameLang* – názov z pohľadu jazyka (String)
- *Type* - typ entity (bit, bit_vector, ...)
- *Path* – cesta v rámci štruktúry modelu (String)
- *PosX* – pozícia modulu na osi x
- *PosY* – pozícia modulu na osi y

PortList
Uzol *PortList* obsahuje všetky porty, ktoré obsahuje daná entita. Uzol neobsahuje žiadne vnorené uzly.

Atribúty:
- *PortName* – názov portu modulu (String)

Ports
Uzol *Ports* obsahuje všetky využívané porty. Uzol neobsahuje žiadne atribúty.

Vnorené uzly:
- *Port* – jednotlivé porty

Port
Uzol *Port* definuje jeden port patriaci entite. Neobsahuje žiadne vnorené uzly.

Atribúty:
- *Name* – názov portu (String)
- *Orientation* – orientáciu portu (in, out, inout, ...)
- *DataType* – typ portu (bit, bit_vector, ...)
- *Width* – voliteľné, počet bitov portu

- ModuleName – Názov modulu, ktorému port patrí (String)
- *PosX* – pozícia portu na osi x
- *PosY* – pozícia portu na osi y

Signals
Uzol *Signals* obsahuje všetky používané signály. Uzol neobsahuje žiadne atribúty.

Vnorené uzly:
- *Signal* – jednotlivé signály

Signal
Uzol *Signal* definuje objekt signál, ktorý opisuje prepojené porty. *Signal* obsahuje taktiež flag, či sa jedná o hodinový signál.

Vnorené uzly:
- *PortMap* – zoznam prepojených portov
- *Route* – množina prepojených uzlov a ich súradníc

Atribúty:
- *Name* – názov objektu (String)
- *DataType* – dátový typ signálu (bit, bit_vector, ...)
- *IsClock* – flag o prítomnosti hodinového signálu (Boolean)

PortMap
Uzol *PortMap* mapuje porty, ktoré sú prepojené jedným signálom. Neobsahuje žiadne atribúty.

Vnorené uzly:
- *Map* – mapovanie názvov jednotlivých prepojených portov s názvami entít

Map
Uzol *Map* obsahuje mapovaný názov portu a názov modulu. Uzol nemá žiadne vnorené uzly.

Atribúty:
- *PortName* – názov portu, ktorý je prepojený signálom (String)
- *ModuleName* – názov modulu, ktorý je prepojený signálom cez port (String)

Route
Uzol *Route* obsahuje uzly, ktoré tvoria signál. Uzol *Route* neobsahuje žiadne atribúty.

Vnorené uzly:
- *Node* – jednotlivé uzly tvoriace signál (prvý a posledný uzol, rohové uzly a rázcestia)

Node
Uzol *Node* opisuje jeden uzol, ktorý vykresľuje signál. Obsahuje pozíciu uzla a počet bitov signálu od predchádzajúceho uzla. Zároveň definuje rázcestie, odkiaľ vedú rôzne trasy signálu.

Vnorené uzly:

- *Cross* – definuje jednu vetvu rázcestia signálu

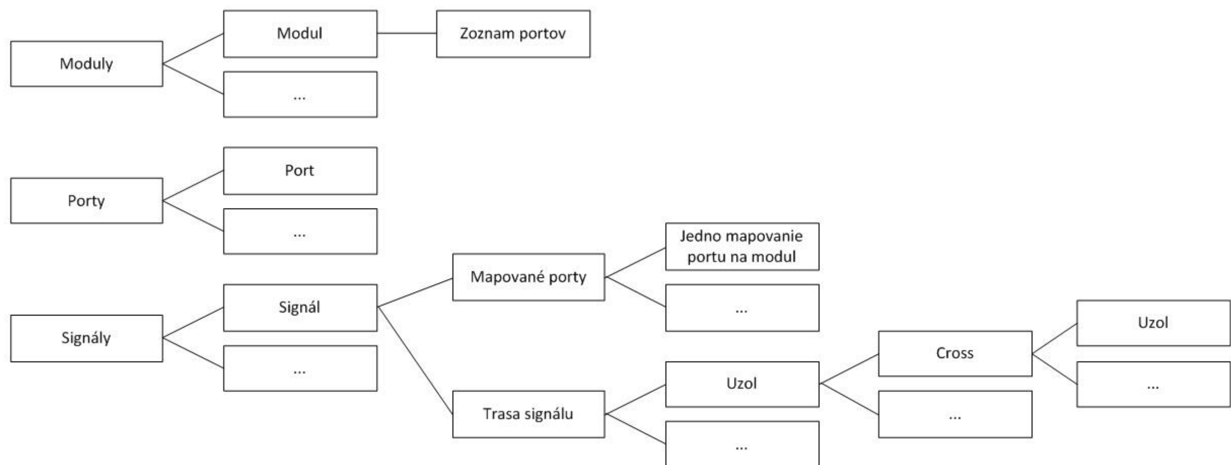
Atribúty:

- *PosX* – pozícia uzla signálu na osi x
- *PosY* – pozícia uzla signálu na osi y
- *Width* – šírka signálu od predchádzajúceho uzla

Cross
Uzol *Cross* obsahuje jednu vetvu trasy signálu od rázcestia.

Vnorené uzly:

- *Node* – jednotlivé uzly tvoriace vetvu signálu



Obr. 5.3: Štruktúra uzlov v XML schéme.

Navrhnutý formát XML schémy môžeme demonštrovať na časti VHDL kódu. Jedná sa o jednu entitu, ktorá tvorí XOR modul. Výpis príkladu v prechodnom XML formáte je obsiahly, prete sme vynechali niektoré, opakujúce sa konštrukcie.

```
entity Stage_1 is
  port ( In1:in Bit; In2:in Bit;
     Out1:out Bit; Out2:out Bit );
```

```
end Stage_1;
...

architecture Struktura_Q of Obvod_XOR is
  signal Internal1, Internal2: Bit;
  component Stage_1
  port ( In1:in Bit; In2:in Bit;
       Out1:out Bit; Out2:out Bit );
  end component;
...
begin
    S1:Stage_1
    port map ( In1=>In1, In2=>In2, Out1=>Internal1, Out2=>Internal2  );
    S2:Stage_2
    port map ( In1=>Internal1, In2=>Internal1, Out1=>Out1  );
end Struktura_Q;
```

```
<Modules>
  <Module>
      <Name>Stage1</Name>
      <NameLang>Stage_1</NameLang>
      <Type>typ</Type>
      <Path>cesta</Path>
      <PosX>100</PosX>
      <PosY>150</PosY>
      <PortList>
        <PortName>In1</PortName>
        <PortName>In2</PortName>
        <PortName>Out1</PortName>
        <PortName>Out2</PortName>
      </PortList>
  </Module>
</Modules>

<Ports>
  <Port>
      <Name>In1</Name>
      <Orientation>in</Orientation>
      <DataType>bit</DataType>
      <ModuleName>Stage_1</ModuleName>
      <PosX>75</PosX>
      <PosY>175</PosY>
  </Port>
  <Port>
      <Name>In2</Name>
```

```
      ...
    </Port>
    ...
  </Ports>

  <Signals>
    <Signal>
      <Name>Internal1</Name>
      <DataType>bit</DataType>
      <IsClock>false</IsClock>
      <PortMap>
        <Map>
          <PortName>Out1</PortName>
          <ModuleName>Stage_1</ModuleName>
        </Map>
        <Map>
          <PortName>In1</PortName>
          <ModuleName>Stage_2</ModuleName>
        </Map>
        <Map>
          <PortName>In2</PortName>
          <ModuleName>Stage_2</ModuleName>
        </Map>
      </PortMap>
      <Route>
        <Node>
          <PosX>125</PosX>
          <PosY>175</PosY>
          <Width>0</Width>
        </Node>
        <Node>
          <PosX>175</PosX>
          <PosY>175</PosY>
          <Width>1</Width>
          <Cross>
            <Node>
              <PosX>200</PosX>
              <PosY>175</PosY>
              <Width>1</Width>
            </Node>
          </Cross>
          <Cross>
            <Node>
              <PosX>175</PosX>
              <PosY>200</PosY>
              <Width>1</Width>
```

```
        </Node>
        <Node>
          <PosX>200</PosX>
          <PosY>200</PosY>
          <Width>1</Width>
        </Node>
      </Cross>
    </Node>
   </Route>
  </Signal>
  <Signal>
   <Name>Internal2</Name>
    ...
  </Signal>
</Signals>
```

5.3 Simulácia opísaných modelov

Nasledujúca časť obsahuje opis spôsobu simulácie pre jednotlivé jazyky. Cieľom bolo, aby sme výsledky simulácie uložili do spoločného formátu, ktorý potom môžeme vizualizovať.

5.3.1 VHDL

Pre jazyk VHDL využijeme externý simulátor GHDL. Simulátor GHDL je prístupný pomocou príkazového riadku a teda je vhodný pre naše potreby. VHDL súbor je najprv skompilovaný, potom je z neho vytvorený vykonateľný súbor a nakoniec sa tento súbor vykoná. Pri zadaní správnych argumentov bude výstupom GHDL simulátora VCD súbor, ktorý potrebujeme pre ďalšiu vizualizáciu.

Prvým využitím tohto simulátora je kompilácia a teda aj kontrola syntaxe zdrojového VHDL kódu. Túto kompiláciu nazývame aj analýzou návrhu opísaného v jazyku VHDL. Kompiláciu spustíme príkazom spustenia simulátora GHDL s prepínačom -a, ktorého argument je názov kompilovaného VHDL súboru.

$ ghdl -a test.vhdl

Tento príkaz vytvorí alebo obnoví súbor *work-obj93.cf*, ktorý opisuje knižnicu *work*. V systémoch Windows sa ale následne nevytvorí žiadny objektový súbor. Potom nasleduje vytvorenie vykonateľného súboru, teda druhé využitie simulátora.

$ ghdl -e top_entity

Prepínač -e znamená *elaborate*, teda spracovať. Takto simulator GHDL vytvorí kód v poradí spracovania návrhu, s entitou s názvom top_entity na vrchole hierarchie. Simulácia entity je následne vykonaná s prepínačom -r. Keďže potrebujeme výstup v súbore VCD, definujeme aj túto skutočnosť, čo je aj posledným navrhovaným využitím simulátora GHDL

$ ghdl -r top_entity --vcd=top_entity.vcd

5.3.2 Verilog

Pre simuláciu Verilog opisov využijeme externý simulátor Icarus Verilog. Icarus Verilog je nástroj na simuláciu a syntézu Verilog opisu. Verilog zdrojový kód najprv kompilujeme, kompilátor generuje VVP preklad. Tento VVP preklad je potom simulovaný pomocou simulátora. Výstup simulácie je VCD súbor, ktorý budeme potrebovať pre vizualizáciu simulácie.

Kompiláciu Verilog kódu spustíme pomocou nasledujúceho príkazu:

$ iverilog -o vystup.vvp vstup.v

Prepínač -o znamená output a za ním ide názov výstupného súboru, v tomto prípade vystup.vpp, typ výstupného súboru je vvp. Ďalším argumentom príkazu je názov vstupného verilog zdrojového kódu (vstup.v). Po vytvorení vvp súboru spustíme simulácie pomocou príkazu:

$ vvp vystup.vvp

Simulátor nám vygeneruje výstupný VCD súbor s názvom vystup.vcd.

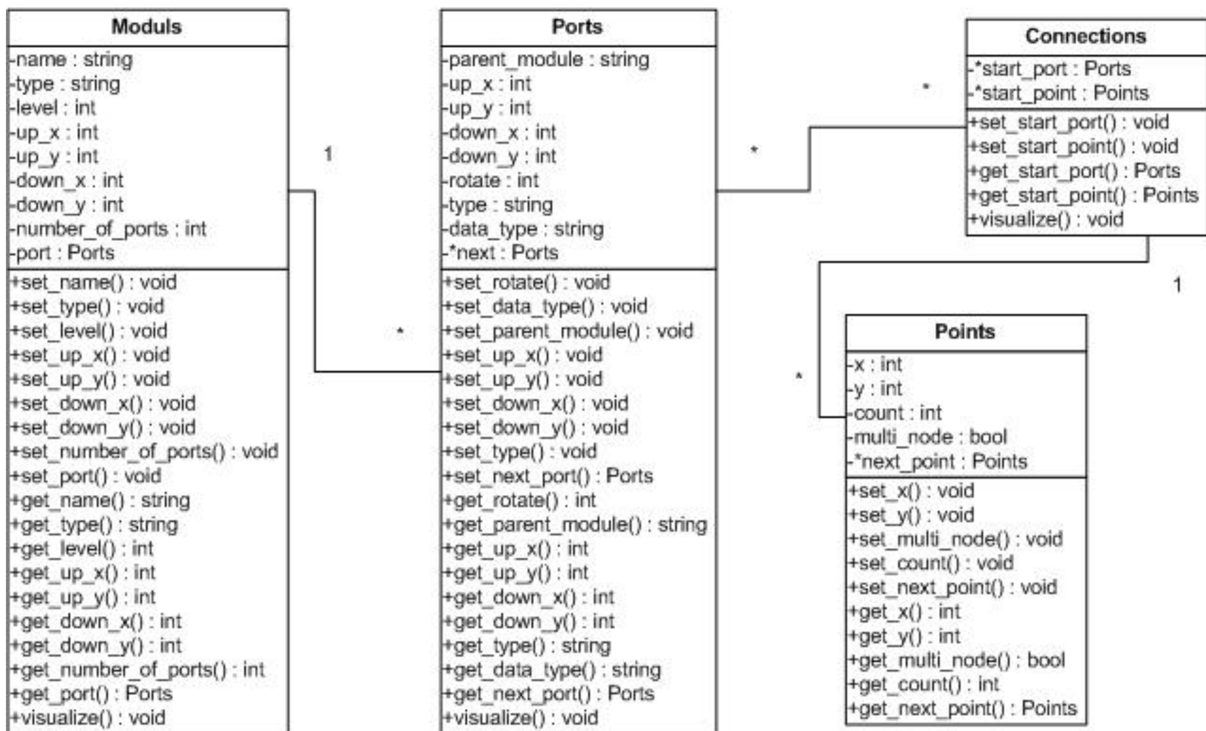
5.3.3 SystemC

Štandardná knižnica SystemC obsahuje simulátor OSCI reference simulator opísaný v kapitole 2.5.3. Tento simulátor sa hodí presne pre naše potreby, pretože poskytuje výstup v podobe VCD súboru, ktorý následne vizualizujeme.

5.4 Návrh tried a objektov potrebných pre vizualizáciu extrahovaných informácií

Vizualizačný nástroj ktorý navrhujeme bude čerpať všetky informácie zo súboru formátu XML. Avšak je nutné rátať s tím, že program bude potrebovať reprezentáciu všetkých potrebných informácií aj v pamäti, teda vo forme objektov a tried. Tieto triedy a objekty budú reprezentovať informácie, ktoré boli extrahované zo zdrojových kódov jazykov VHDL, Verilog a SystemC. Objekty ktoré budú vytvorené ako inštancie navrhnutých tried, budú v sebe obsahovať informácie potrebné na vizualizáciu.

Navrhnuté triedy sú zobrazené na obrázku Obr. 5.4 vo forme diagramu tried.



Obr. 5.4: Diagram navrhnutých tried pre reprezentáciu extrahovaných informácií.

5.4.1 Trieda Moduls

Pre triedu entita je potrebné uchovávať nasledujúce informácie :

```
class Moduls
{
      private string name;
      private string type;
      private int level;
      private int up_x;
      private int up_y;
      private int down_x;
      private int down_y;
      private int number_of_ports;
      private Ports *port;

      public void set_name(string name);
      public void set_type(string type);
      public void set_level(int level);
      public void set_up_x(int x);
      public void set_up_y(int y);
      public void set_down_x(int x);
      public void set_down_y(int y);
      public void set_number_of_ports(int number);
      public void set_port(Ports port);
```

```
    public string get_name();
    public string get_type();
    public int get_level();
    public int get_up_x();
    public int get_up_y();
    public int get_down_x();
    public int get_down_y();
    public int get_number_of_ports();
    public Ports get_port();

    public void visualize();

}
```

Atribúty

- name –meno entity vzhľadom na jazyk.
- type – typ entity.
- level – uchováva v sebe číslo hierarchickej vrstvy, na ktorej bude modul vykresľovaný.
- up_x – x súradnica ľavého horného rohu modulu.
- up_Y – y súradnica ľavého horného rohu modulu.
- down_x – x súradnica dolného pravého rohu modulu.
- down_y – y súradnica dolného pravého rohu modulu.
- number_of_ports – počet portov daného modulu.
- port – pointer na prvý port modulu.

Funkcie

Funkcie get a set budú poskytovať a nadstavovať atribúty daného objektu, prístup k nim nebude priamy.

Ďalšie funkcie :

- visualize() – funkcia slúži na vykreslenie daného modulu. Bude použitý príkaz na vykreslenie obdĺžnika, ktorý sa vykreslí na základe up_x, up_y, down_x a down_y. Vykresľovanie sa bude vykonávať do pracovného okna.

5.4.2 Trieda Ports

Pre triedu ports je potrebné uchovávať nasledujúce informácie :

```
class Ports
{
     private string parent_module;
     private int up_x;
     private int up_y;
     private int down_x;
     private int down_y;
     private int rotate;
     private string type;
     private string data_type;
     private Ports *next;

     public void set_rotere(int rotate);
     public void set_data_type(string data_type);
     public void set_parent_module(string name);
     public void set_up_x(int x);
     public void set_up_y(int y);
     public void set_down_x(int x);
     public void set_down_y(int y);
     public void set_type(string type);
     public void set_next_port(Ports *next_port);

     public int get_rotate();
     public string get_parent_module();
     public int get_up_x();
     public int get_up_y();
     public int get_down_x();
     public int get_down_y();
     public string get_type();
     public string get_data_type();
     public Ports get_next_port();

     public void visualize();
  }
```

Atribúty

- parent_module –meno modulu, ku ktorému je port priradený.
- up_x – x súradnica ľavého horného rohu portu.
- up_Y – y súradnica ľavého horného rohu portu.
- down_x – x súradnica dolného pravého rohu portu.
- down_y – y súradnica dolného pravého rohu modulu.
- rotate – pootočenie portu .
- type – typ portu (vstupný, výstupný).

- data_type –dátový typ portu .
- *next – pointer na ďalší port modulu.

Funkcie
Funkcie set a get poskytujú a nadstavujú parametre objektu.

Ďalšie funkcie:

- visualize() – funkcia slúži na vykreslenie portu. Bude použitý príkaz na vykreslenie obdĺžnika, ktorý sa vykreslí na základe up_x, up_y, down_x a down_y. Vykresľovanie sa bude vykonávať do pracovného okna, k príslušnému modulu.

5.4.3 Trieda Connections

Táto trieda vyjadruje prepojenie dvoch a viacerých portov. Pre triedu Connections je potrebné uchovávať nasledujúce informácie :

```
class conections
{
      private Ports *start_port;
      private Points *start_point;

      public void set_start_port(Ports port);
      public void set_start_point(Points point);

      public Ports get_start_port();
      public Points get_start_point();

      public void visualize();
}
```

Atribúty
- *start_port – pointer na port z ktorého vychádza signál.
- *start_point – pointer na prvý zlomový bod spojenia.

Funkcie

Funkcie typu get a set nadstavujú a poskytujú atribúty objektu.

Ďalšie funkcie:

- Visualize – Funkcia vykreslí dané spojenie. Vykreslenie sa bude realizovať vykreslením čiary, ktorej začiatočný a konečný bod bude mať x-ovú a y-ovú súradnicu. Tieto súradnice sa budú uchovávať v objekte point.

5.4.4 Trieda Points

Trieda Points uchováva zlomové body spojenia. Ak sa spojenie láme, tak údaje o tomto zlome sa uložia do zlomového bodu, ktorý bude objektom tejto triedy. Pre triedu Points je potrebné uchovávať nasledujúce informácie :

```
class Points
{
      private int x;
      private int y;
      private bool multi_node;
      private int count;
      private Points *next_point;

      public void set_x(int x);
      public void set_y(int y);
      public void set_multi_node(boolean multi_node);
      public void set_count(int count);
      public void set_next_point(Points *next_point);

      public int get_x();
      public int get_y();
      public boolean get_multi_node();
      public int get_count();
      public Points get_next_point();

      public vizualize();
}
```

Atributy

- x – x-ová súradnica bodu zlomu;
- y – y-ová súradnica bodu zlomu;
- multi_node – ak sa bude jednať o bod zlomu kde prichádza k vetveniu tak bude hodnota true.
- count – vyjadruje početnosť spojenia.
- *next_point – pointer na ďalší zlomový bod.

Funkcie

Funkcie typu get a set nadstavujú a poskytujú atribúty objektu.

Ďalšie funkcie:

- Visualize – Funkcia vykreslí spojenie dvoch po sebe idúcich zlomových bodov.

5.5 Architektúra systému

Na obrázku Obr. 5.4 je uvedený blokový diagram navrhovanej architektúry systému.

Zdrojové súbory jazykov VHDL, Verilog a SystemC
Tieto zdrojové súbory opisných jazykov predstavujú tie súbory, ktoré požaduje používateľ vizualizovať. Sú to teda vstupy do navrhovaného systému.

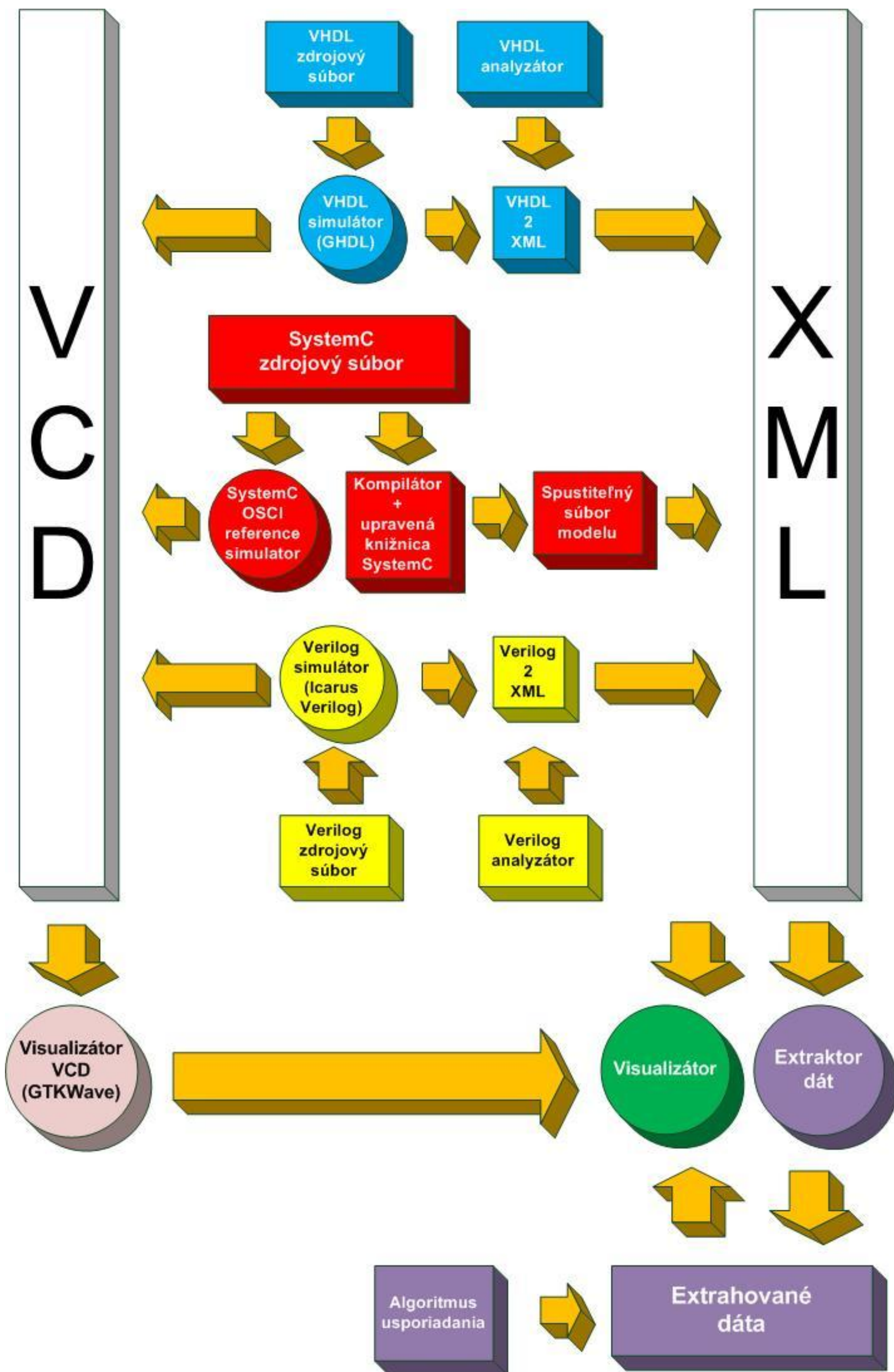
Simulátory jazykov VHDL, Verilog a SystemC
Simulátory jednotlivých jazykov sú navrhnuté externé simulátory, ktoré využijeme v systéme. Využijeme ich na kontrolu syntaxe, kompiláciu, či simuláciu zdrojových súborov. Pre jazyk VHDL je navrhnutý externý simulátor GHDL, pre jazyk Verilog simulátor Icarus Verilog a pre opisný jazyk SystemC - SystemC OSCI reference simulator.

VHDL a Verilog analyzátor
Tieto bloky diagramu predstavujú súbory tried vygenerovaných generátorom ANTLRv3, pričom pre jazyk VHDL sa blok nazýva VHDL analyzátor a predstavuje súbor tried vygenerovaný na základe vstupnej gramatiky pre jazyk VHDL. Blok Verilog analyzátor predstavuje súbor tried vygenerovaný na základe vstupnej gramatiky pre jazyk Verilog.

Moduly VHDL2XML a Verilog2XML
Tieto bloky diagramu majú za úlohu transformácie vstupného zdrojového kódu daného jazyka (VHDL alebo Verilog) do súboru XML a jej následné uloženie. Transformácia je vykonaná na základe analyzátoru pre daný jazyk, teda pomocou súboru tried preň vygenerovaných.

Kompilátor + upravená knižnica SystemC
Pomocou kompilátora bude opísaný model zostavený spolu s upravenou knižnicou, výsledkom čoho bude spustiteľnú súbor modelu.

Spustiteľný súbor modelu
Spustiteľnú súbor modelu sa nakoniec spustí a vykoná samotnú extrakciu dát, a poskytne ich vo forme výstupného XML súboru.



Obr. 5.5: Blokový diagram hrubého návrhu systému.

Súbor XML
Súbor XML je blok diagramu, ktorý reprezentuje prechodný formát informácií získaných zo zdrojových kódov. Tieto informácie sú prostredníctvom súboru XML uchované na neskoršie využitie, no na ich základe sa aj opísaný návrh systému vizualizuje.

Súbor VCD
VCD je súbor obsahujúci informácie o priebehoch signálov, teda o simulácií opísaného návrhu systému. Tieto informácie sú získané na základe výstupu navrhnutých externých simulátorov.

Extraktor dát
Extraktor dát slúži v systéme na extrahovanie dát z formátu XML. Bude vytvárať všetky potrebné objekty a dátové štruktúry, aby program mohol s nimi pracovať. Druhov funkciou extraktora bude naplnenie atribútov vytvorených objektov a dátových štruktúr.

Etrahované dáta
Jedná sa o všetky dátové štruktúry a objekty, ktoré vytvoril extraktor na základe XML súboru.

Algoritmus usporiadania
Tento algoritmus bude pracovať s extrahovanými dátami. bude ich dopĺňať o súradnice, na ktorých budú jednotlivé objekty vykresľované, a tiež sa bude starať o prehľadnosť vykreslených informácií.

Vizualizátor
Objekt ktorý bude postupne vykresľovať všetky potrebné objekty do používateľského prostredia.

Vizualizátor VCD
Vizualizátor VCD je funkčný celok systém, ktorý slúži na zobrazenie priebehu simulácie na základe VCD súboru. Rozhodli sme sa využiť existujúci vizualizátor GTKWave, ktorý má voľne dostupné zdrojové kódy a teda využijeme tú časť nástroja, ktorá vykresľuje priebehy signálov.

6 Prototyp

6.1 Ciele prototypovania

Prototyp našej aplikácie by mal vedieť hlavne parsovať súbory jazykov VHDL a Verilog, ktoré prevedie do nami navrhnutého XML formátu, a z tohto formátu zobraziť základnú grafickú reprezentáciu opísaných modelov.

Cieľom vytvorenia tohto prototypu bolo otestovať zvolené časti architektúry systému týkajúce sa jazykov VHDL a Verilog. Chceli sme zistiť, či budú pracovať spoľahlivo, keď budú zakomponované do našej aplikácie a či bude pomocou nich možné zo súborov týchto jazykov vygenerovať súbor v našom navrhnutom XML formáte. Implementáciu jazyka SystemC sme odložili na neskoršie etapy projektu, pretože našim zámerom nebolo otestovať čo najväčší počet jazykov, ale otestovať čo najväčšiu časť architektúry systému navrhnutú v hrubom návrhu.
Zrejme najdôležitejším cieľom bolo otestovanie navrhnutého XML formátu, a jeho prípadné doplnenie o ďalšie atribúty, ktoré budú potrebné pre uloženie vizualizovaného modelu.
Nakoniec prototyp aplikácie poslúžil aj pre osvojenie si zvolenej grafickej knižnice a pre prvé experimenty s grafickou podobou vizualizovaných modelov.

6.2 Výsledky prototypovania

Výsledný prototyp nám pomohol odhaliť rad nedostatkov XML formátu súboru pre ukladanie vizualizovaných modelov. Nedostatky boli opravené a budú zahrnuté v podrobnom návrhu.

Pre implementáciu parsera jazyka Verilog sme používali už existujúci parser, ktorý bol navrhnutý a implementovaný v diplomovej práci Michala Nosáľa. Museli sme zmeniť funkciu, ktorá generuje XML súbor, pretože sme použili iný formát tohto súboru. Boli nutné aj ďalšie zmeny v niektorých triedach, napríklad bolo potrebné pridať informácie o typoch signálov (bit alebo bitvector). Následne sme mohli otestovať správnosť konverzie do XML formátu a mohli sme opraviť prípadné chyby. Počas testovania sme zistili, že použitý parser neparsuje inštancie primitívov (primitívy sú preddefinované moduly, ako napr. not, and, nand, or, nor, atď.). Oprava parsera, aby parsoval aj primitívy je dosť náročná a preto to bude úloha na ďalší semester. Okrem spomínanej chyby parser fungoval správne a pri parsovaní Verilog kódu, ktorý neobsahoval inštancie primitívov, XML súbor obsahoval už všetky požadované informácie o moduloch, portoch a signáloch.

Jednou z častí prototypu je modul, ktorého úlohou je analýza a transformácia zdrojového kódu v jazyku VHDL do navrhnutého XML formátu. Využitý je modul vytvorený Ing. Dominikom Mackom, ktorý bol upravený tak, aby výstupný XML súbor zodpovedal navrhnutému formátu. Úprava spočívala vo vytvorení novej funkcie, ktorá XML súbor generuje. Táto funkcia využíva ďalšiu funkciu, ktorá prehľadáva komponenty v opísanom systéme a bola vytvorená na základe algoritmu prehľadávania architektúr a v nich vytvorených komponentoch. Funkcia prehľadáva rekurzívne architektúry štruktúry pokiaľ nenatrafí na komponent opísaný

architektúrou správania. Výsledkom je modul, ktorého vstupom je zdrojový kód v jazyku VHDL a výstupom XML súbor zodpovedajúci navrhnutému formátu.

Významnou časťou prototypu je grafické používateľské rozhranie, ktoré spája jednotlivé moduly prototypu do celku. Jeho súčasťou je aj primitívny textový editor, v ktorom si používateľ môže prezerať a editovať zdrojové kódy podporovaných opisných jazykov. Rozhranie umožňuje používateľovi otvoriť niekoľko zdrojových súborov, pričom ich prezeranie je umožnené pomocou záložiek. Zmeny v súboroch si môže používateľ uložiť. Súčasťou používateľského rozhrania prototypu je aj kontrola syntaxe zdrojového súboru v jazyku VHDL pomocou externého simulátora GHDL. Umožnená je aj akási prvá verzia simulácie modelu opísaného v jazyku VHDL. Spočíva vo vytvorení VCD súboru simulátorom GHDL a jeho následným prezeraním prostredníctvom nástroja GTKWave.

Pri spájaní modulov, zaoberajúcich sa transformáciou jazykov VHDL a Verilog nastal problém týkajúci sa využitého generátora analyzátorov ANTLR. VHDL modul využíval inú verziu knižníc nástroja ANTLR ako modul Verilog. Problém bol vyriešený použitím knižníc najnovšej verzie nástroja ANTLR a vygenerovaním nových súborov tried určených na analýzu nástrojom AntlrWorks 1.4.3.
Modul v súčasnej verzii simuluje iba systémy, ktoré vo svojom opise majú entitu "testbench".

Nakoniec bola na prototype otestovaná zvolená grafická knižnica, ktorá bola otestovaná na skúšobnej vzorke grafických objektov. Grafická knižnica splnila naše očakávania a bude teda možné ju použiť vo výslednej aplikácii nášho projektu.

6.3 Testovanie prototypu

Nevyhnutnou súčasťou implementácie prototypu je aj jeho testovanie. To bolo vykonávané v dvoch fázach podľa toho, ako boli jednotlivé súčasti prototypu vyvíjané.

Prvá fáza sa týkala testovania jednotlivých parserov. V prípade parsera XML sme mali na vstupe niekoľko ručne vytvorených XML súborov, presne podľa navrhnutého formátu. Výstupom boli textové výpisy atribútov jednotlivých objektov, ktoré vznikli parsovaním XML. Na základe vizuálnej kontroly sme overili správnosť parsera. Aby sme mohli otestovať všetky obmedzenia týkajúce sa navrhnutého XML formátu, vstupné XML súbory boli modifikované o chybné údaje. Následne sme overili, že parser v prípade takýchto chýb vyhodí výnimku, čo je korektné. Takýto postup bol aplikovaný aj na zvyšné parsery.

Druhá fáza sa zameriavala na testovanie celého prototypu, teda aplikácie pozostávajúcej z VHDL a Verilog parserov, XML parsera a grafickej knižnice. Vstupom tohto testu boli VHDL a Verilog zdrojové kódy. Tieto zdrojové súbory boli parserom prevedené do univerzálneho XML súboru, ktorý zároveň slúžil na vizuálnu kontrolu správnosti formátu. Následne v prvej fáze otestovaný parser XML vytvoril objekty a vizualizátor ich vykreslil, čo sme overili opäť vizuálnou kontrolou.

Literatúra

[1] IEEE Standard SystemC Language Reference Manual: IEEE Computer Society, 2006

[2] Sýkora. J.: Metody extrakce modelu z jazyka SystemC, Diplomová práca,: ČVUT v Prahe, Fakulta elektrotechnická, máj 2009. 89 str., Dostupné na: http://necago.ic.cz/prj/sc2vhdl/dip-20090512-rev3.pdf (2011-10-17).

[3] Turoň. J.: Vizualizácia simulácie SystemC modelu, Diplomová práca,: STU v Bratislave. FIIT, 2010. 100s., FIIT-13428-17022.

[4] Gerbian.P.:Unicode Compiler-Compiler, Diplomová práca,:Univerzita Karlova v Prahe. MFF,2006. 75s

[5] Macko D.: Vizualizácia VHDL modelu. Bakalárska práca. Slovenská technická univerzita, Fakulta informatiky a informačných technológií, Bratislava. 2009. 50s.

[6] Bc. Macko D.: Vizualizácia VHDL modelov digitálnych systémov. Diplomová práca. Slovenská technická univerzita, Fakulta informatiky a informačných technológií, Bratislava. 2011. FIIT-13428-35524. 60s.

[7] Bc. Petráš J.: VIZUALIZÁCIA VHDL MODELU. Diplomová práca. Slovenská technická univerzita, Fakulta informatiky a informačných technológií, Bratislava. 2008. 85s.

[8] Bc. Zubal M.: Vizualizácia VHDL opisu. Diplomová práca. Slovenská technická univerzita, Fakulta informatiky a informačných technológií, Bratislava. 2008. 115s.

[9] Gingold, T.: GHDL. Dostupné na: http://ghdl.free.fr/ (26.10.2011)

[10] GTKWave. Dostupné na: http://gtkwave.sourceforge.net/ (26.10.2011)

[11] Johnson, M., Zelenski J.: Lexical analysis,Handout 03, 2008

[12] Turoň, J.: Vizualizácia opisu v jazyku SystemC, Bakaláska práca,: STU v Bratislave, FIIT, 2008, 72s., FIIT-5214-17022.

[13] IEEE Computer Society: IEEE Standard for IP-XACT. Dostupné na: http://standards.ieee.org/getieee/1685/download/1685-2009.pdf (27.10.2011)

[14] Nyasulu, P. M. : Introduction to Verilog, 2001, http://www.facweb.iitkgp.ernet.in/~anupam/verilog_2.pdf (27.10.2011)

[15] Sanguinetti, J. : Verilog Tutorial, 2002, http://staff.ustc.edu.cn/~han/CS152CD/Content/Tutorials/Verilog/VOL/main.htm (27.10.2011)

[16] Bc. Nosáľ, M. :Vizualizácia Verilog modelov digitálnych systémov, Diplomová práca, FIIT STU, Bratislava, 2010, FIIT-13428-17072

[17] NShape .NET Framework, http://code.google.com/p/nshape/ (27.10.2011)

[18] Diagram.NET, http://code.google.com/p/diagramnet/ (27.10.2011)

[19] IEEE Computer Society: IEEE Standard Verilog Hardware Description Language, 2001. ISBN 0-7381-2827-9 SS94921. http://allhdl.ru/pdf/ieee_std_1364_2001.pdf (27.10.2011)

[20] Rao, A. : What is SystemVerilog, http://electrosofts.com/index.html (27.10.2011)

[21] Germano, T. : Graph drawing, 1999, http://davis.wpi.edu/~matt/courses/graphs (27.10.2011)

[22] Johnson, M., Zelenski J.: Semantic Analysis,Handout 10, 2007

[23] Vanderseypen, M. F.: Netron Graph Library v2.1 White Paper, 2004 http://mirror.transact.net.au/sourceforge/n/project/ne/netron-reloaded/documentation/v2.1%20white%20paper/Netron_Graph_Library_V2.1_White_Paper.pdf