VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EDITOR JAZYKA VHDL

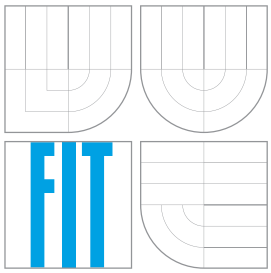
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FILIP BALAŠ

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EDITOR JAZYKA VHDL

EDITOR FOR VHDL LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FILIP BALAŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL KAJAN

BRNO 2013

Abstrakt

Tato práce popisuje návrh a vývoj editoru pro jazyk VHDL, který umožňuje práci s grafickou reprezentaci obvodu, pomocí které je možno provádět změny propojení komponent a jejich hierarchii. Tyto změny se následně promítnout zpět do textové podoby. Práce také popisuje základy členění VHDL kódu a některé konstrukce tohoto jazyka.

Abstract

This work describes design and development of editor for VHDL language, which allows to work with graphical representation of circuit, with which is possible make changes in connections of components and their hierarchy. The changes in in the graphical representation are reflected to the source code. This work also describes basics of VHDL code structure and some constructs of this language

Klíčová slova

VHDL, editor, hierarchie komponent, signály, Python, PyQt, QScintilla, PLY

Keywords

VHDL, editor, component hierarchy, signals, Python, PyQt, QScintilla, PLY

Citace

Filip Balaš: Editor jazyka VHDL, bakalářská práce, Brno, FIT VUT v Brně, 2013

Editor jazyka VHDL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Kajana. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Filip Balaš
15. května 2013

Poděkování

Rád bych poděkoval svému vedoucímu panu Ing. Michalu Kajanovi za jeho trpělivost, ochotu a odborné rady.

Obsah

Kapitola 1

Úvod

Vývoj hardware pomocí jazyka VHDL je založen na popisu jednotlivých komponent, jejich chování a také jejich propojení. Tyto části jsou vzájemně provázány pomocí tzv. signálů. Není neobvyklé, že některé tyto části jsou velmi rozsáhlé, či je jich velké množství a kód se tak stává méně přehledný. Problém nastává, když chce vývojář změnit propojení jednotlivých komponent. Tato operace bývá náročná a zdlouhavá a mnohdy dochází k chybám. Dále často je velice náročné si představit, jak se mezi sebou jednotlivé komponenty vzájemně ovlivňují.

Další možností je nejen komponenty mezi sebou propojovat, ale také vkládat jednu do druhé a tím vytvářet hierarchickou strukturu. Velmi často jsou tyto komponenty rozesety v mnoha různých souborech. Návrhář pak musí složitě prohledávat několik souborů a zjišťovat, jaké komponenty jsou použity a jak jsou připojeny.

Cílem této práce by tedy mělo být usnadnění těchto činností tím, že se vygeneruje grafická reprezentace obvodu a pak pomoci ní bude možno kód snadno upravovat a zároveň vidět jednotlivá propojení komponent či jejich hierarchii.

Tato práce je členěna následovně. V následující kapitole č.2 se čtenář dočte o konkrétních cílech projektu a také o současném stavu. Kapitola č.3 pak poskytne informace o jazyku VHDL a jeho struktuře. V kapitole č.4 se nachází popis použitých technologií. Kapitola č.5 popisuje návrh aplikace a poslední kapitola je věnována samotné implementaci a dosaženým výsledkům.

Kapitola 2

Formulace cíle a charakteristika současného stavu

Jak už bylo naznačeno v úvodu, tak hlavním cílem projektu je usnadnění práce návrháře při úpravách kódu a také pro přehledné zobrazení propojení. Úprava kódu je sice teoreticky velice jednoduchá, ale v případě ruční úpravy jde o velice zdlouhavou a nepříjemnou práci u které se lze velice snadno zmýlit nebo něco přehlédnout. Proto je vhodné takovou práci přenechat nějaké automatizované metodě, která to udělá rychle a bez chyb.

Proto bude cílem projektu vytvořit editor pro jazyk VHDL, který kromě běžných funkcí textového editoru umožní analyzovat kód a zobrazit jej v grafické reprezentaci jako obvod a pomoci tohoto zobrazení pak snadno upravovat.

2.1 Současný stav

Pro psaní kódu v jazyce VHDL je možno využít jakýkoliv textový editor, od nejzákladnějších editorů jako je třeba *Notepad*, přes pokročilé textové editory jako *vim, Geany, Notepad++, PSPad* atd. a nakonec až pokročilé editory a vývojová prostředí určené přímo pro jazyk VHDL, jako je například *Xilinx ISE, Sigasi* nebo *Emacs VHDL Mode*. Problém však je hlavně s funkčností a dostupností těchto editorů.

První skupina sice umí upravovat text, ale tím taky popis funkcí končí, nicméně jsou velmi rozšířené a hlavně zadarmo.

Druhá skupina je na tom o něco lépe. Jde o klasické textové editory, podporující funkce jako je zvýrazňování syntaxe, automatické odsazování, automatické napovídání, základní správu projektu apod. I tyto editory jsou většinou vydávány jako freeware či open-source, některé jsou multiplatformní a většinou podporují velké množství programovacích jazyků, včetně jazyka VHDL. Nicméně tyto editory většinou dokáží jen zvýraznit klíčová slova a literály, neprovádějí však žádnou analýzu kódu a nezjistí chybnou syntaxi ani nerozeznají význam identifikátorů.

Třetí skupina má největší funkčnost co se VHDL týče, avšak má i své nevýhody. Z toho důvodu bude nejlepší se seznámit s několika zástupci této skupiny.

2.1.1 Xilinx ISE

Xilinx ISE je velmi komplexní a pravděpodobně nejpokročilejší vývojové prostředí. Jde o balík nástrojů firmy Xilinx a má pokročilé editační schopnosti a možnosti generování

kódu. Kromě jazyka VHDL také podporuje jazyk Verilog, který je další představitel ze skupiny HDL jazyků. Xilinx ISE obsahuje velké množství součástí, například syntézní nástroj, nazvaný XST (Xilinx Synthesis Technology), simulátor a také za zmínku stojí možnost zobrazení schématické reprezentace na RTL (Register-Transfer Level) úrovni. Tato funkce se nejvíce blíží k tomu, co je účelem tohoto projektu. Nicméně, nejedná se o přímé zobrazení kódu jazyka VHDL, ale o zobrazení výsledku jedné z prvních úrovní syntézy.

Co by bylo možno považovat za nevýhodu, je skutečnost, že jde o komerční produkt a hlavně že se ceny plné verze pohybují v tisících dolarů. Další nevýhodou může být skutečnost, že s množstvím funkcí a součástí se pojí i velká náročnost na diskový prostor a samotné získání a zprovoznění zabere spoustu času. Poskytována je také volně dostupná verze, tzv. Xilinx ISE WebPACK, nicméně zde se nejedná o plnou verzi produktu a nese s sebou mnohá omezení[2].

V souhrnu by tedy bylo možno říct, že Xilinx ISE je velmi pokročilý nástroj pro vývoj hardware, ale jeho komplexnost a cena jej někdy mohou znevýhodňovat.

2.1.2 Sigasi

Sigasi je velmi pokročilý editor jazyka VHDL. Na rozdíl od Xilinx ISE se však nejedná o kompletní vývojové prostředí se syntetizérem a simulátorem, ale pouze o editor. Sigasi nabízí přehledné a uhlazené rozhraní založené na editoru Eclipse, který je určený pro vývoj software v širokém spektru programovacích jazyků. Editor Sigasi nabízí mnoho různých funkcí, kromě základních jako je zvýrazňování syntaxe podporuje také okamžitou kontrolu syntaxe a dokonce i sémantiky, během psaní kódu. Nabízí také velmi inteligentní napovídání a pokročilé možnosti refaktorizace kódu a také zobrazení hierarchické struktury [8].

Jednou z nevýhod je ale skutečnost, že zadarmo je dostupná jen studentská licence a tzv. Starter Edition. Sigasi Starter Edition je verze obsahující úplnou funkčnost plné verze, nicméně jen pro menší projekty. V případě většího projektu se automaticky omezí většina pokročilejších funkcí. Roční licence plné verze se pak pohybuje v ceně 600 Eur.

2.1.3 Emacs VHDL Mode

Za zmínku také stojí jeden z nejstarších editorů pro vývoj v jazyce VHDL. Jedná se o open-source rozšíření editoru Emacs. Toto rozšíření se nazývá VHDL Mode. Za vývojem tohoto rozšíření stojí Reto Zimmermann a Rod Whitby. Historie Emacs VHDL Mode se datuje zhruba roku 1992 a do dnes se pokračuje ve vývoji. Mezi funkce patří opět zvýrazňování kódu, napovídání, kontrola syntaxe, ale také generování mnoha částí kódu, či vkládání šablon nebo zobrazování hierarchie [1].

Tento editor byl také inspirací pro vznik dříve zmiňovaného editoru Sigasi [8].

Mezi výhody tohoto produktu patří skutečnost že je open-source a tedy volně k dostání a také to, že je naprosto přizpůsobitelný. Mezi nevýhody pak patří skutečnost, že vzhledem ke stáří, nejde o příliš uživatelsky přívětivé prostředí a ovládání a naučit se jej používat většinou trvá značný čas.

2.2 Formulace cílů

Výsledek tohoto projektu by se měl zařadit někde mezi druhou a třetí skupinu. Měl by být schopen základní editace a podporovat funkce jako zvýrazňování syntaxe apod. a zároveň být schopný částečné analýzy VHDL kódu a podporovat pokročilejší formu refaktorizace.

Dále by měl být kladen důraz na jednoduchost a přehlednost grafického rozhraní. Pak je zde požadavek na multiplatformost a cenu. Proto byly zvoleny prostředky, které umožní přenositelnost na jiné platformy a hlavně projekt je a nadále bude vyvíjen jako open-source.

Kapitola 3

Jazyk VHDL

3.1 Historie

Jazyk VHDL (VHSIC[1] Hardware Description Language) je jazyk určený pro popis hardware, který vznikl v r. 1983 jako součást VHSIC projektu. Tento projekt byl iniciován Ministerstvem obrany Spojených států amerických. Syntaxe jazyka byla odvozena z jazyka ADA [9].

V roce 1987 byl institutem IEEE standardizován a tím se zajistila kompatibilita mezi nástroji různých výrobců. Tato verze VHDL je nejčastěji označována jako VHDL-87. V roce 1993 vzniká další standard, s označením VHDL-93[11]. Tato verze jazyka je do dnes nejvíce rozšířená a podporována, proto se tato práce bude zabývat právě touto verzí.

Během následujících let bylo vydáno několik dalších standardů a rozšíření, které různě zjednodušovaly některé konstrukce a přidávaly nové vlastnosti s důrazem na zpětnou kompatibilitu s předchozími verzemi standardu. V současnosti je aktuální verze jazyka VHDL-2008.

3.2 Návrhové jednotky jazyka VHDL

Kód v jazyce VHDL je rozdělen do několika druhů jednotek, přičemž tyto jednotky jsou *entita* (**entity**), *architektura* (**architecture**), *balík* (**package**), *tělo balíku* (**package body**) a nakonec *konfigurace* (**configuration**). Tyto jednotky jsou rozdělené primární a sekundární, což znamená, že každá sekundární jednotka patří k nějaké primární jednotce. To znamená, že ke každé architektuře patří jedna konkrétní entita, a ke každému tělu balíku patří konkrétní balík. Nakonec je zde konfigurace, což je jediná primární jednotka která nemá žádnou sekundární jednotku [9].

3.2.1 Entita

Entita je primární jednotka, která je určena především k deklaraci rozhraní komponenty, nikoliv však k popisu jejího chování. Každá entita má své jméno, které je v dané knihovně unikátní. Všechny části deklarace entity jsou nepovinné, takže entita nemusí mít žádné rozhraní. V takovém případě se jedná o uzavřený systém.

V hlavičce entity jsou dvě sekce, **generic** a **port**. Sekce **generic** slouží k definici generických konstant, které slouží k parametrizaci entity. Tímto způsobem lze například vytvořit

[1] VHSIC - Very high speed integrated circuit

generickou komponentu, která bude obsahovat paměť, kdy velikost této paměti bude parametrizovaná generickou konstantou a až v momentě použití dané komponenty se rozhodne jak veliká paměť bude vytvořena. Dále je zde sekce **port**, která má za účel definici portů, které budou sloužit jako rozhraní komponenty, pomoci kterého může komponenta komunikovat s okolím.

Porty jsou v podstatě signály, o kterých bude zmínka ještě později. Na rozdíl od klasických signálu ale mají ještě tzv. *režim*. Ten se používá k určení směru komunikace. Použitelné režimy jsou **in**, **out**, **inout**, **buffer** a **linkage**. Vstupní port (režim **in**) slouží pouze ke čtení uvnitř architektury. Výstupní port (režim **out**) slouží pouze k zápisu uvnitř architektury. Vstupně-výstupní port (režim **inout**) pak umožňuje jak zápis, tak čtení. Pak je zde port s režimem **buffer**, který slouží jako výstupní port, jen na rozdíl od výstupního portu z něj lze uvnitř architektury i číst. Nakonec je třeba zmínit také režim **linkage**, který označuje neznámý směr toku.

Dále lze v entitě deklarovat různé datové typy, konstanty, podprogramy apod., které budou dostupné ve všech architekturách dané entity a také je zde možné definovat operační sekci entity. Ta se však obvykle používá jen k různým kontrolám, protože z operační sekce entity nelze měnit hodnoty portů.

3.2.2 Architektura

Architektura je sekundární jednotka náležící k entitě. Pro jednu entitu je možno deklarovat libovolné množství architektur, přičemž každá může mít odlišné chování či jiný způsob implementace. Pro funkční program je třeba alespoň jedné dvojice *entita–architektura*, nebo lze použít architekturu jako komponentu v jiné architektuře. Architektura se skládá ze dvou částí, deklarační a operační.

Deklarační část je nepovinná, v takovém případě pracuje architektura pouze s deklaracemi z entity a z použitých balíků. Deklarační část architektury může deklarovat typy, podtypy, konstanty, podprogramy, signály, komponenty, soubory, aliasy a také umožňuje konfiguraci komponent. Tyto deklarace jsou pouze lokální a platí jen v rámci konkrétní architektury.

Operační část slouží k definici chování architektury. Může obsahovat pouze paralelní příkazy, neboli příkazy které jsou prováděny současně. Tyto příkazy jsou *přiřazovací příkaz*, *příkaz procesu*, *příkaz generate*, *příkaz volání paralelní procedury*, *příkaz assert*, *příkaz bloku* a nakonec *příkaz instancování komponenty*.

3.2.3 Balík

Balík a tělo balíku jsou jednotky určené pro sdílení deklarací pro opakované použití v jiných jednotkách. V balících lze uvést stejné deklarace jako například v architektuře, kromě konfigurace komponenty a těla podprogramů. Tělo balíku pak je určeno k implementaci deklarovaných podprogramů a deklaraci vnitřních objektů a typů. Toho lze využít tak, že je možno dodávat přeložené těla balíků spolu s deklarací balíku a například v případě standardních balíků je definováno, jak se mají podprogramy v balících chovat, ale není pevně určena přesná implementace. To znamená, že pak každý vývojář balíků může dodávat své optimalizované verze implementací.

Pro použití daného balíku je zapotřebí použití příkazů **library** a **use** pro zviditelnění balíku v návrhu.

Součástí VHDL je základní balík **standard** který je umístěn v knihovně **std**. Tuto knihovnu a její balík není třeba nijak uvést, protože knihovny *std*, *work* a balík *standard*

automaticky viditelné. Balík *standard* je obsahuje deklarace základních typů jako je například *bit, boolean, integer, time, character, string, bit_vector* a další. Spolu se standardem IEEE1076 [11] který popisuje jazyk VHDL, vznikl doprovodný standard IEEE1164 [9], který obsahuje definice několika velmi používaných balíků, které byly zařazeny do knihovny *ieee*. Tyto balíky obsahují například typy pro víceúrovňovou logiku či znaménkové a bezznaménkové typy a definici aritmetických operací využívající tyto typy.

3.2.4 Konfigurace

Konfigurace je speciální návrhová jednotka, která se využívá zaprvé k zvolení dvojice *entita–architektura*, která se použije a zadruhé se využívá na konfiguraci neosazených komponent uvnitř architektury. Toto řešení se používá k tomu, aby bylo možno vytvořit architekturu, uvnitř které je neosazená komponenta a kdykoliv je možné jednoduše určit jakou dvojicí *entita–architektura* se daná komponenta osadí.

3.3 Vybrané konstrukce jazyka VHDL

Protože se tato práce zabývá zobrazením hierarchické struktury a zobrazením obsahu a propojením uvnitř architektury, je třeba se věnovat hlavně paralelním příkazům a signálům kterými jsou propojeny. Obsah této kapitoly byl čerpán z [10][9].

Kromě paralelních příkazů obsahuje jazyk VHDL také sekvenční příkazy, které se od paralelních liší tím, že jsou prováděny postupně. Tyto příkazy nepopisují strukturu komponenty jako spíše její chování, které není pro nás při zobrazování příliš důležité. Sekvenční příkazy se vyskytují v podprogramech a v procesech. Podprogramy nás nebudou zajímat ani tak z pohledu definice chování, ale spíše z pohledu použití, protože jeden z paralelních příkazů je volání podprogramu. Procesy jsou pak také paralelní příkazy a opět, nebude nás ani tak zajímat, jaké obsahují příkazy, jako spíše jaké signály se uvnitř používají.

Z deklarací nás budou zajímat především signály a porty, pak deklarace komponent a nakonec lokální konfigurace komponent.

3.3.1 Signály

Signály jsou druh objektů, které představují abstrakci vodičů v reálných obvodech. Signály mohou být různých datových typů a mít přiřazené různé hodnoty. Jde o velice důležitý prvek jazyka VHDL, protože umožňuje propojení mezi paralelními příkazy. Tato práce se tedy bude zabývat především hledáním použitých signálů v jednotlivých paralelních příkazech.

3.3.2 Deklarace komponent

Použití komponent (objekt **component**) umožňuje vkládání dvojic *entita–architektura* do jiné architektury. Před tím než se ale dá použít, je zapotřebí ji nejdříve deklarovat. Co se deklaruje, je blok uvozený klíčovým slovem **component** a jedná se o typ komponenty. Typ komponenty obsahuje rozhraní, tedy sekce **port** a **generic**, stejně jako je to v případě entity.

Deklarace komponenty se může vyskytovat jak v deklarační částí architektury, kde se jedná pouze o lokální komponentu, tak v balíku, kde se jedná o komponentu která je určena pro opakované použití.

3.3.3 Konfigurace komponent

Konfigurace komponent je operace, která připojí konkrétní dvojici *entita–architektura* k použité komponentě v obvodu. Konfigurace může mít čtyři formy. První forma je pomocí návrhové jednotky **configure**, druhá je specifikace konfigurace (**configuration specification**) uvnitř deklarační části entity, třetí forma je přímá instalace entity uvnitř operační části architektury a poslední způsob je implicitní konfigurace, která probíhá v případě, že typ komponenty má stejné jméno i rozhraní jako některá z entit v knihovně.

Konfigurace komponent obsahuje také sekci **port map** a **generic map**, která slouží k namapování portů entity na porty komponenty.

3.3.4 Přiřazovací příkazy

Nejzákladnější z paralelních příkazů je přiřazovací příkaz. Jedná se o příkaz, který se používá pro nastavení hodnoty signálu. Přiřazení může mít několik forem. První je nepodmíněný přiřazovací příkaz. Jde o jednoduché přiřazení hodnoty výrazu do signálu. Druhá forma je podmíněný přiřazovací příkaz, který má dvě podoby, příkaz **when** a příkaz **with**.

Příkazy **with** a **when** umožňují vybrat který z výrazů bude vyhodnocen a jeho hodnota bude přiřazena do signálu. Příkaz **when** přiřadí hodnotu prvního výrazu pokud je splněna podmínka, nebo se provede druhá část příkazu. Druhá část příkazu může být opět příkaz **when**. Jde tedy o obdobu příkazu *if–elsif–else* ze sekvenčního prostředí, popřípadě podobného příkazu z klasických programovacích jazyků. Naproti tomu, příkaz **when** nejdříve provede testovaný výraz a podle výsledné hodnoty vybere, který z výrazů bude přiřazen.

3.3.5 Příkaz procesu

Příkaz procesu je příkaz který vytváří sekvenční prostředí pro provádění příkazů postupně, místo aby byly prováděny zároveň. Tento způsob se hodně blíží způsobu provádění příkazů v běžných procedurálních jazycích. Příkaz procesu patří mezi blokové příkazy, které mohou obsahovat vlastní lokální deklarace a také příkazy, které jsou prováděny. Nicméně sekvenční prostředí je mimo rámec této práce, proto se jimi nebudu více zabývat.

3.3.6 Příkaz bloku

Příkaz bloku je jeden z příkazů, který bude v rámci této práce velmi důležitý. Příkaz bloku nepatří mezi příliš často používané [9] příkazy z toho důvodu, že má podobnou funkčnost jako vložené komponenty. Jde o příkaz který umožní strukturované členění architektury. Jeho chování a použití je velmi podobné dvojici *entita–architektura*. V deklarační architektuře se vyskytují zaprvé deklarace portů a generických konstant, stejně jako v deklarační části entity. Dále je zde mapování portů bloku na signály architektury. Zbytek deklarační části a operační část je naprosto stejná jako v případě architektury.

3.3.7 Příkaz generate

Příkaz **generate** je speciální příkaz, který umožňuje podmíněně nebo opakovaně provádět určité příkazy. Má dvě formy *if–generate* a *for–generate*. Nejčastější využití *for–generate* formy je pro generaci pravidelného propojení různých elementů podle zadaných parametrů. Když se tedy použije v příkazu *for–generate* nějaký paralelní příkaz, parametrizovaný řídící proměnnou, pak se tento příkaz neprovede *N-krát* jako by to bylo v sekvenčním prostředí, ale místo toho se provede N paralelních příkazů, každý jinak parametrizovaný řídící proměnnou.

Podobně funguje i příkaz *if–generate*, který v případě splněné podmínky provede daný blok příkazů. Nejčastěji se příkaz *if–generate* využívá uvnitř příkazu *for–generate*.

3.3.8 Příkaz instancování komponenty

Poslední z příkazů, které jsou velmi důležité z pohledu této práce je příkaz pro instancování komponenty (zde použito ve významu objektu **component**, ne komponenta jako část většího celku). Jedná se o konstrukci, která vytvoří konkrétní instanci komponenty daného typu uvnitř architektury a poté se pomocí sekcí **port map** a **generic map** vytvoří mapování signálů architektury na porty entity.

Jazyk VHDL-93 umožňuje také tzv. přímou instanci, přičemž to dovoluje přímé použití konkrétní entity a její architektury bez použití systému komponent.

3.4 Způsoby popisu systému

Jazyk VHDL poskytuje prostředky pro tři základní způsoby popisu navrhovaného systému. Tyto popisy jsou nazývány *behaviorální* (popis chování), *data-flow* (popis toku dat) a *strukturovaný* (popis hierarchické struktury komponent). Rozdělení je čistě formální, ve skutečnosti lze tyto popisy libovolně kombinovat [9].

3.4.1 Behaviorální popis

Behaviorální popis sytému je popisem chování systému a je založeno hlavně na použití procesů a podprogramů. Tento popis se nejvíce blíží klasickému procedurálnímu paradigmatu programování, využívá se zde sekvenčních příkazů a lze tedy psát algoritmicky.

3.4.2 Data-flow popis

Data-flow popis, neboli popis toku dat je popis založený na používání paralelních příkazů. Data-flow popis se oproti behaviorálnímu popisu více podobá popisu hardware, tedy situace kdy se vytváří jednotlivé části obvodu, které pracuji zároveň a ty jsou propojeny pomocí vodičů.

3.4.3 Strukturální popis

Strukturální popis se popisem chování jednotlivých částí systému ale hlavně propojení menších systémů do větších celků. V tomto způsobu popisu systému se využívá především systému komponent, tedy vkládání existujících entit a architektur do jiných architektur a tím vytváření hierarchické struktury komponent.

3.5 Poznámky k analýze jazyka VHDL

Jak už bylo jednou uvedeno, hlavním účelem této práce není analýza funkčnosti či správnosti kódu, ale zobrazení propojení jednotlivých částí. Tedy co bude zapotřebí, je hlavně prohledat každý paralelní příkaz a najít které signály se v něm používají. Jednou z možností k dosažení tohoto cíle je využití regulárních výrazů, druhou je použít (zjednodušeného) syntaktického a sémantického analyzátoru.

Bohužel, vzhledem ke skutečnosti, že jazyk VHDL má složitá syntaktická pravidla, velmi často se spoustou nepovinných částí, například uvedení klíčového slova **architecture**

a jména architektury při ukončování definice architektury apod., tak není možné kód VHDL jednoduše prohledávat pomocí regulárních výrazů, protože takové konstrukce by byly příliš složité a náchylné na chyby. Z toho důvodu je jednodušší využít syntaktické analýzy. K tomu je vhodné využít kompletní gramatiky jazyka VHDL která je součástí standardu IEEE 1076-1993 [11], který definuje jazyk VHDL-93.

Kapitola 4

Použité technologie

Tato kapitola se bude zabývat použitými technologiemi při vývoji aplikace. Všechny tyto technologie jsou vybrány tak, aby byly multi-platformní a byly dostupné jako open-source.

4.1 Python

Python je objektově orientovaný interpretovaný programovací jazyk. Pro vývoj tohoto projektu byl zvolen z důvodů, že kód aplikace může být jednoduše šířen a používán na různých platformách v nezměněné podobě a bez potřeby překladu.

Kód v jazyce Python je většinou kratší a tím pádem i přehlednější než v jiných programovacích jazycích. Toho je dosaženo použitím širokého výběru standardních i nestandardních knihoven. Velikou výhodou tohoto jazyka je přítomnost tzv. *garbage collectoru*, neboli nástroje, který se automaticky stará o dealokaci nepoužívané paměti. Vývojář se tak nemusí příliš starat o odstraňování paměti, kterou již nepotřebuje [5].

Při vývoji byl využit Python verze 3.2.3, ale aplikace je kompatibilní s verzí 3.2.x a výše.

4.2 PLY

PLY je kompletní implementace nástrojů *Lex* a *Yacc* v jazyce Python. Nástroje *Lex* a *Yacc* jsou používány pro automatizované generování lexikálního a syntaktického analyzátoru. Pří vývoji aplikace byl použit pouze nástroj Lex, pro lexikální analýzu VHDL kódu. Důvodem výběru byla skutečnost, že vytvoření takového lexikálního analyzátoru je pomocí tohoto nástroje je velmi rychlé, přičemž pro specifikaci pravidel se používá klasických regulárních výrazů [3].

4.3 PyQt, Qt a QScintilla

4.3.1 Qt

Qt je velmi rozsáhlý framework určený pro vývoj přenositelných aplikací v jazyce C++. Jeho hlavním využitím je vývoj grafického uživatelského rozhraní, ale díky rozsáhlosti tohoto frameworku to není jeho jediný účel. Kromě modulů pro vývoj grafického uživatelského rozhraní jsou zde například moduly pro práci s XML soubory, multimédii, sítí, vlákny, souborovým systémem apod. Obsah této podkapitoly vychází z [7].

Signály a sloty

Qt přidává mimo jiné do jazyka C++ systém tzv. signálů a slotů. Tento mechanizmus umožňuje vysokoúrovňovou komunikaci mezi objekty. Hlavní využití těchto signálů je především pro upozorňování na události grafických objektů uživatelského rozhraní.

Slot je zvláštní druh metody, na kterou lze připojit signál. Propojení signálu se slotem se pak provádí voláním metody `connect()`, která je součástí všech objektů třídy **QObject**, která je předkem většiny objektů v Qt. Výhodou tohoto způsobu je, že volaný ani volající o sobě vůbec nemusí nic vědět.

QtDesigner

Součástí Qt je také *Qt Designer*, což je aplikace určena k návrhu a sestavení grafického uživatelského rozhraní pomoci grafického editoru. To umožní rychlý návrh a změny grafického rozhraní, bez nutnosti zasahovat do kódu. Tato aplikace funguje na principu *WYSIWYG*, což znamená, že to co je vidět v *Qt Designeru*, to bude také výsledkem. *Qt Designer* obsahuje spoustu grafických prvků, které lze do vytvářeného hlavního či dialogového okna přidat. Můžou to byt různá tlačítka, rámečky, textová pole, menu, záložky, plochu na kreslení a podobně. To dovolí vytvořit a rozvrhnout velice rychle grafické rozhraní a pak jej pomoci signálů a slotů propojit s třídami, které budou provádět konkrétní funkčnost a pokud je potřeba změnit například rozložení či velikosti prvků v okně, tak se to provede v *Qt Designeru* a kódu se to nijak nedotkne.

Graphics View Framework

Další velmi užitečnou součástí Qt je tzv. **Graphics View Framework**. Jde o komplexní systém určený pro zobrazování a interakci s velkým množstvím 2D prvků. Hlavní součástí jsou třídy **QGraphicsView**, **QGraphicsScene** a **QGraphicsItem**.

QGraphicsView je jeden z grafických prvků (*widgetů*), které lze přidávat do okna. Jeho účelem je zobrazování obsahu scény a propagaci událostí, jako například pohyby myši, stisk tlačítek myši nebo stisk kláves.

QGraphicsScene je třída, představující scénu, která má na starosti správu zobrazovaných prvků a jejich stavu, dále propaguje události přijaté z pohledu jednotlivým prvkům, pro které je událost určena (označené prvky, prvky na které se kliklo). Zároveň se chová jako úložiště těchto prvků a vytváří svou soustavu souřadnic.

QGraphicsItem je výchozí třída pro všechny prvky, které se zobrazují na scéně. Prvek také může obsahovat jiný prvek. V takovém případě se jedná o tzv. vztah *rodič–potomek*. Každý prvek má svou lokální soustavu souřadnic která má počátek v bodě, ve kterém je umístěn ve scéně. A také veškeré operace prováděné nad tímto prvkem jsou v souřadné soustavě prvku. Tudíž pokud vložíme do prvku A prvek B na souřadnice [10, 10], pak jsou tyto souřadnice v lokální soustavě souřadnic prvku A. Pokud pak změníme pozici prvku A v scéně, tak se změní i pozice prvku B v scéně, ale v lokální souřadné soustavě zůstává pořád na pozici [10, 10]. Z toho vyplývá, že se potomci pohybují zároveň s rodičovským prvkem. Takže jedna z možností jak vytvořit nový prvek je postavit jej z několika základních prvků.

V Qt je definováno několik základních tříd odvozených od **QGraphicsItem**, mezi které patří například základní tvary jako je čára (**QGraphicsLineItem**), obdelník (**QGraphicsRectItem**), elipsa (**QGraphicsEllipseItem**), text (**QGraphicsTextItem**) a další. Pokud tyto tvary nestačí, lze vytvořit novou třídu odvozenou od **QGraphicsItem** nebo jejich

potomků, a vytvořit si tak jakýkoliv prvek je potřeba.

4.3.2 PyQt

Jak už bylo zmíněno výše, Qt je určeno pro jazyk C++, nicméně protože tato práce je v jazyce Python, je potřeba použít knihovny PyQt, což je mapování Qt tříd do Pythonu. Toto spojení umožňuje zpřístupnění téměř všech možností Qt v jazyce Python. Většina rozhraní a metod je naprosto stejná jako v Qt. Výjimkou jsou například metody, které v C++ vrací hodnoty přes parametry metody, protože C++ nedokáže z funkcí a metod vracet více hodnot, na rozdíl od Pythonu, kde tato možnost je. U takových metod jsou hlavičky změněny aby metody nevracely nic přes parametry. Dále některé třídy, jako například **QString** nejsou dostupné, protože řetězce Pythonu verze 3 mají stejné vlastnosti.

PyQt také umožňuje také použití mechanizmu signálů a slotů. Rozdílem však je, že PyQt umožňuje použití jakékoliv metody či funkce jako slot. Také signály napsané v Pythonu nemusí mít definované žádné parametry, pokud se neposílají některému Qt objektu.[4]

Použití PyQt resp. Qt v této práci je z důvodu snadného a rychlého vytvoření uživatelského rozhraní a pro použití *Qt Graphics View Framework*, který umožňuje snadné a přehledné zobrazení 2D prvků — v tomto případě se bude jednat o komponenty obvodu a jejich propojení.

4.3.3 QScintilla

Poslední použitá knihovna v tomto projektu je QScintilla. Jedná se o port editační komponenty **Scintilla**, což je komponenta určena pro editaci zdrojových kódů. **Scintilla** je určena pro Windows a GTK+. Naproti tomu QScintilla, jak už napovídá její jméno je určena jako komponenta pro Qt. Mimo jiné také obsahuje mapování pro Python, takže je ji možné využít pomocí PyQt.

QScintilla je grafická komponenta určená k editaci textu a především zdrojových kódů. Mezi její vlastnosti patří např. zvýrazňování syntaxe, zobrazování čísel řádků, automatické odsazování, napovídání a další. QScintilla poskytuje dvě API [1]. První je nízkoúrovňové vycházející z API původní *Scintilly*. Druhé je vysokoúrovňové, připomínající více Qt API. QScintilla poskytuje širokou podporu jazyků, včetně jazyka VHDL [6].

4.4 Použité ikony

V grafickém uživatelském rozhraní je použito několika ikon, určených k zpřehlednění uživatelského rozhraní. Tyto ikony pochází z balíku ikon vydávaných pod licencí Public domain. Tento balík je součástí Open Icon Library [2].

[1] API = aplikační programové rozhraní
[2] Dostupné z odkazu: http://openiconlibrary.sourceforge.net/downloads.html

Kapitola 5

Návrh aplikace

Aplikace, která je výsledkem této práce se skládá z několika částí. První částí je grafické uživatelské rozhraní a jeho logika, druhou je zobrazovací komponenta určená pro vizualizaci výsledku analýzy a třetí je analyzátor jazyka VHDL. Tato kapitola se bude zabývat především návrhem grafického uživatelského rozhraní.

5.1 Návrh uživatelského rozhraní

5.1.1 Požadavky na uživatelské rozhraní

Hlavní nároky na uživatelské rozhraní jsou:

- přehlednost
- jednoduchost
- intuitivnost

Přehlednost

Uživatelské rozhraní aplikace musí být přehledné, aby uživatel nemusel zbytečně dlouho hledat funkci kterou potřebuje. Tohoto lze dosáhnout několika kroky.

Zaprvé by uživatelské rozhraní mělo mít vzhled podobný existujícím aplikacím, které jsou pro podobný účel, tedy poskytnout uživateli vzhled na který je zvyklý. To zahrnuje především rozložení prvků okna, pojmenování menu a jeho položek a používání vžitého názvosloví. V tomto případě se jedná například o umístění hlavního menu v okně úplně nahoře, pod menu nástrojový panel a pod ním pak ostatní prvky, jako například editor textu a podobně. V situaci, kdybychom hlavní menu nebo nástrojovou lištu umístili úplně dolů, mohlo by to uživatele zmást. Dalším příkladem můžou být položky menu. Většina aplikací má první položku menu pojmenovanou *Soubor* (nebo anglicky *File*) a toto menu obsahuje položky, týkající se například otevírání, ukládání a tisku souborů. Každý uživatel je tak zvyklý, že když potřebuje otevřít, či uložit soubor, musí použít právě toto menu. Pokud bychom ale toto menu pojmenovali nějak jinak, například *Dokument*, mohlo by se stát, že by uživatel chvíli tápal, než by nalezl funkci na ukládání.

Druhý krok je používání nástrojových panelů pro nejčastěji používané funkce. Nástrojový panel umožní, že tyto funkce budou kdykoliv rychle dostupné, bez toho aby uživatel musel složitě prohledávat menu. S tím souvisí používání ikon. Ikony totiž umožní uživateli

snadno najít funkci, jenž potřebuje, bez toho aby si musel číst popisky tlačítek. Opět, tyto ikony by měly být názorné, aby uživatel na první pohled poznal, co tlačítko dělá. Je mnoho symbolů, na které jsou uživatelé zvyklí a automaticky je spojují s funkcemi které představují. Jako například, pro vytvoření nového souboru se využívá symbolu listu papíru nebo pro uložení souboru se používá symbolu diskety či disku.

Jednoduchost

S přehledností uživatelského prostředí hodně souvisí také jeho jednoduchost. Jednoduché prostředí bývá většinou zároveň i přehledné a naopak, pokud je uživatelské prostředí příliš komplikované a obsahuje mnoho prvků, může to způsobit, že uživatel musí dlouho zjišťovat, který prvek má použít. Mezi doporučení, jak udržet uživatelské rozhraní jednoduché patří například používání pouze nezbytně velkého množství prvků, v případě menu nevytvářet mnohaúrovňová menu, do nástrojových panelů vkládat pouze často používané prvky apod.

Intuitivnost

Intuitivností je myšleno snadné a rychlé pochopení uživatelského rozhraní a jeho ovládání, bez přílišné nutnosti studovat manuál. Předpokladem pro intuitivní uživatelské rozhraní je splnění předchozích dvou nároků.

Další prvek který pomáhá snadnému pochopení je tzv. *místní nápověda* (anglicky *tooltip*). Tato nápověda se zobrazuje v případě, uživatel chvíli podrží kurzor myši nad prvkem. Účelem této nápovědy je získání krátkého popisu funkce prvku.

Velmi často je vhodné nastavit klávesové zkratky pro často používané akce. To umožní pokročilejším uživatelům rychlejší práci s programem. Nicméně i tady je nutné si dávat pozor na to, aby se dodržovala zásada používání zažitých zkratek tak, jak jsou uživatelé zvyklí. Není vhodné nastavit například klávesovou zkratku **Ctrl+s** na smazání souboru, protože většina uživatelů očekává pod touto klávesovou zkratkou akci uložení souborů. Na druhou stranu se ale od uživatele nesmí požadovat ovládání programu pomocí klávesových zkratek. Vždy je třeba umožnit uživateli aby si sám mohl vybrat zda bude používat klávesové zkratky, nebo ovládání myší.

5.1.2 Návrh hlavního okna

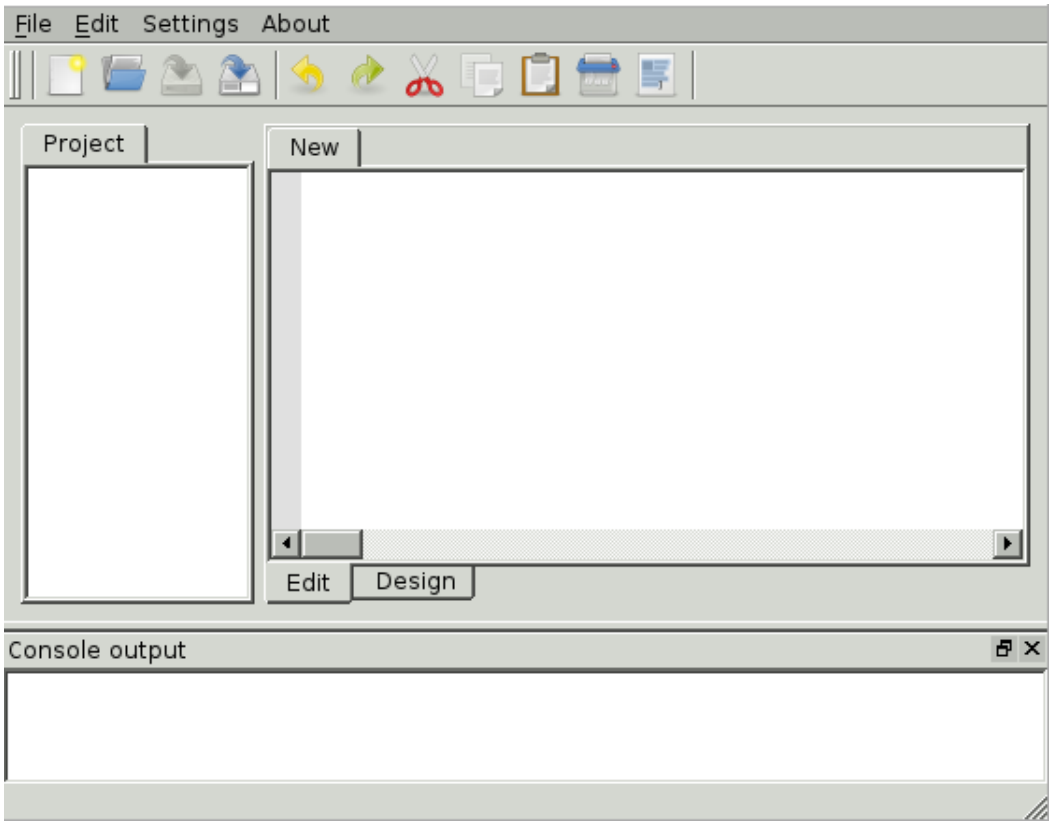
Pro návrh uživatelského prostředí bylo použito nástroje Qt Designer. Qt Designer umožňuje kompletní návrh uživatelského rozhraní v příjemném a přehledném prostředí. Výhodou tohoto návrhu je, že lze hned vidět výsledek, jak uživatelské rozhraní bude vypadat ve skutečnosti. Druhou výhodou je, že z výsledného návrhu lze vygenerovat kód, který navržené rozhraní vytvoří.

Na obrázku č.5.1 je vidět navržené rozhraní hlavního okna. V horní části je vidět hlavní menu a nástrojový panel. Ve spodní části je vidět malé dokovací okno určené pro výstup externích programů, a uprostřed nalevo je panel pro správu projektu a napravo je editor.

Hlavní menu a nástrojový panel

Hlavní menu se skládá ze čtyř částí: **File** (česky *Soubor*), **Edit** (česky *Upravit*), **Settings** (česky *Nastavení*) a **About** (česky *O aplikaci*).

Menu **File** obsahuje položky pro vytváření, otevírání a ukládání souborů. Mimo jiné obsahuje také vnořené menu **Project** s položkami pro práci s projektem. Umožňuje vytvářet,



Obrázek 5.1: Hlavní okno s textovým editorem

otevírat, přejmenovávat a uzavírat projekty, dále pak umí přidávat do projektu soubory a knihovny. Poslední položkou **File** menu je akce na ukončení aplikace.

Menu **Edit** obsahuje základní položky související s prací s textem. Mezi ně patří například akce **Undo** (česky *Zpět*) a **Redo** (česky *Vpřed*), které jsou určeny pro vrácení textu do podoby před poslední změnou a zpět.

Menu **Settings** momentálně obsahuje pouze položku **Preferences** (česky *předvolby*). Ta slouží k vyvolání dialogového okna pro nastavení aplikace.

Menu **About** obsahuje dvě položky. První je **About VVRE** (*O VVRE*), ta slouží pro vyvolání malého dialogového okna sloužícího k informacím o aplikaci. Druhou položkou je **About Qt**, která vyvolá informace o použité verzi Qt.

Pod hlavním menu je nástrojový panel. Ten obsahuje tlačítka určená pro několik často používaných akcí, jako například vytvoření, otevření a uložení souboru, akce **Undo** a **Redo** a akce pro práci s textem.

Okno konzole

Ve spodní části okna se nachází dokovací okno, které obsahuje textové pole. Toto textové pole je v režimu *pouze pro čtení* a je určené pro výpis obsahu standardního výstupu a standardního chybového výstupu externích aplikací, které jsou přes tento editor spouštěny. Toto okno se za normálních okolností nebude zobrazovat. Po spuštění bude zneviditelněno a zobrazí se až v okamžiku, kdy jej bude zapotřebí.

Toto dokovací okno je instancí třídy **QDockWidget** a je možné jej kdykoliv zavřít

kliknutím na křížek v pravé horní části dokovacího okna. Také je možné jej vytáhnout z hlavního okna, a pak se bude zobrazovat jako samostatné okno a pak je možné jej opět vrátit zpět.

Panel správy projektu

V levé části okna se nachází panel s listy (instance třídy **QTabWidget**). Tento panel má v současné době pouze jeden list, který obsahuje stromový pohled na soubory projektu. Tento pohled je instancí třídy **ProjectView**, což je třída odvozená od **QTreeWidget**. Třída **ProjectView** umožňuje zobrazení souborů projektu a hlavně práci s nimi. Může obsahovat tři druhy položek: **ProjectItem**, **FileItem** a **LibraryItem**. **ProjectItem** je hlavní položka představující projekt. Tato položka pak může obsahovat **FileItem** a **LibraryItem**. **FileItem** představuje položku zastupující VHDL soubor. Nakonec je zde položka **LibraryItem**, která představuje adresář knihovny.

Pohled třída **ProjectView** poskytuje mimo jiné svým položkám také kontextová menu. Pro **ProjectItem** je kontextové menu stejné jako vnořené menu **Project** uvnitř menu **File**. Pro **FileItem** je dostupné kontextové menu se šesti položkami. První tři jsou **Show file**, které otevře či zobrazí otevřený soubor, **Rename file**, sloužící k přejmenování souboru v projektu a **Remove file**, které slouží k odstranění souboru z projektu. Druhé tři pak slouží na analýzu souboru. První položka (**Check syntax**) slouží k volání externího programu pro kontrolu syntaxe, druhá (**Analyze file**) provádí analýzu kódu a třetí (**Display file**) zobrazí grafickou reprezentaci kódu.

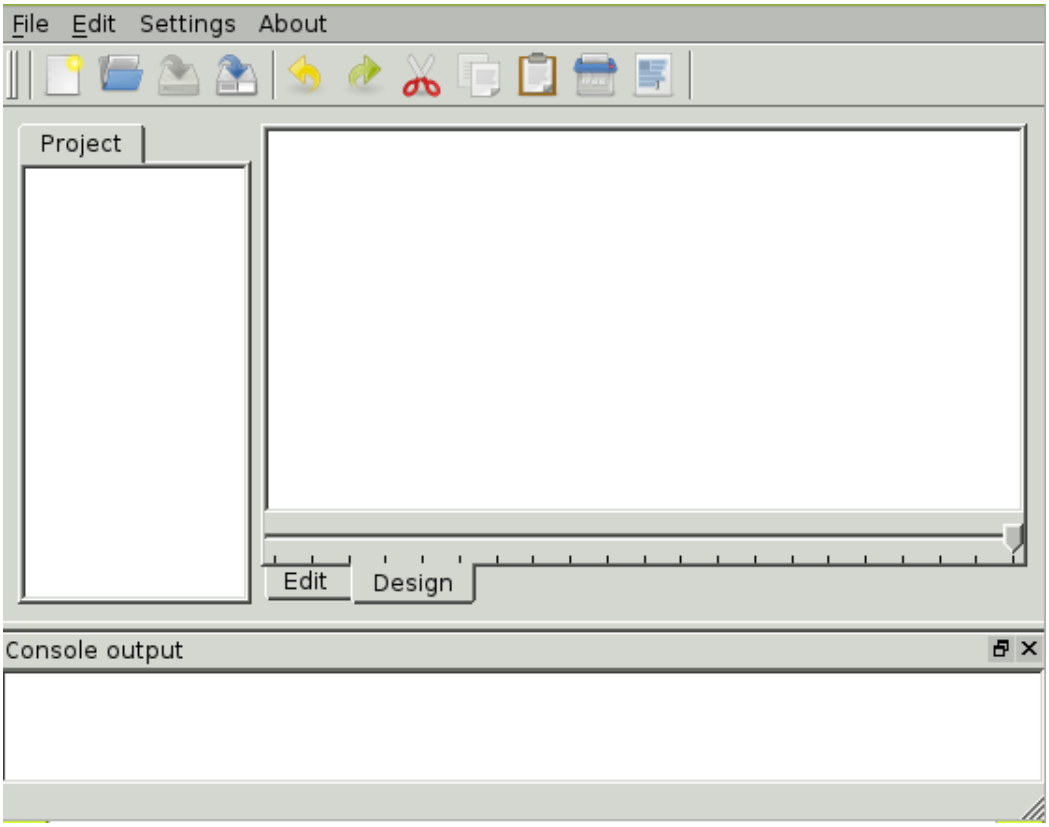
Editor

Napravo od panelu pro správu projektu je panel s listy, který je nejdůležitější v tomto programu, protože obsahuje oba editory, textový i grafický. Tento panel obsahuje dva listy. Jeden je označen **Edit** a druhý **Design**. Na obrázku č.5.1 lze vidět hlavní okno s aktivním listem **Edit**, zatímco na obrázku č.5.2 je vidět hlavní okno s aktivním listem **Design**. List **Edit** obsahuje další vnořený panel s listy, v tomto případě však každý list představuje jeden dokument. Uvnitř těchto panelů je už samotný textový editor. Tento textový editor je instancí třídy **VhdlDocument**, což je třída odvozená od třídy **QScintilla**. Druhý list obsahuje dva prvky: Posuvník a pohled (**QGraphicsView**). Tento list představuje grafický editor kódu. Posuvník slouží pro nastavení přiblížení či oddálení pohledu na vykreslovanou scénu.

Textový editor by měl umožňovat otevření více různých souborů, podporovat základní editační funkce a zvýrazňování syntaxe.

Grafický editor pak by měl umožnit zobrazit jednotlivé části kódu jako obvod skládající se z malých částí, přestavujících jednotlivé paralelní příkazy, propojených čárami, představujícími signály, které je propojují. Tyto části se zobrazí uvnitř většího kontejneru, představujícího architekturu. V případě použití příkazu bloku či příkazu instancování komponenty, se zobrazí také vnořený kontejner se zobrazením vnitřní struktury, v případě, že je k dispozici (komponenta je konfigurována). To pomůže k zobrazení hierarchie komponent. Dále v případě podržení kurzoru myši nad komponentou[1], by se měla zobrazit místní nápověda, která bude obsahovat dodatečné informace, jako například název, druh příkazu či komponenty a úryvek kódu.

[1] zde je použit pojem komponenta se významu části celku



Obrázek 5.2: Hlavní okno s grafickým editorem

Tyto jednotlivé části obvodu budou zobrazovány jako obdélníky s vystupujícími krátkými čárami, které budou představovat jednotlivé použité signály uvnitř komponenty (příkazu). Tento způsob zobrazení byl zvolen, aby bylo dosaženo podoby připomínající obvod elektrického schématu. Po stranách každého signálu je umístěn štítek s identifikátorem signálu. V případě kontejneru (představujícího architekturu, instalovanou komponentu či příkaz bloku) jsou po stranách umístěny dvě sady čar, jedna po vnější straně, druhá po vnitřní. Vnitřní sada čar představuje porty entity či bloku a vnější pak signály, které jsou na porty namapovány. Nad horní hranou komponenty se bude nacházet štítek, který bude obsahovat název příkazu, architektury či komponenty.

Kromě zobrazování by také měl grafický editor umožnit základní interakci, jako je přesouvání jednotlivých komponent po obrazovce, či jejich zvětšování v případě kontejnerů. Dále musí umožňovat úpravu propojení a názvů jednotlivých komponent. To bude umožněno pomocí změny názvu signálu, který z komponenty vystupuje. Změna signálu bude prováděna dvojklikem myši na štítek. Podobným způsobem se řeší také přejmenovávání komponent.

Jako druhou možnost změny propojení bylo zvažováno vytváření a úprava propojení pomocí myši, nicméně tento způsob není pro tuto situaci vhodný. Při odpojení signálu z příkazu by došlo k odstranění daného identifikátoru z kódu příkazu a tím by vznikl neplatný příkaz. Naproti tomu zvolené řešení zajistí, že vždy bude na daném místě příkazu použit nějaký identifikátor.

Samozřejmostí pak je promítnutí všech prováděných změn zpět do textové podoby. K tomu dojde při přepnutí na list **Edit**. Změny v kódu se provedou pouze v souboru který

byl zobrazen. Změny v zanořených komponentách, které se nachází v odlišných souborech nebudou provedeny. Pokud bude zapotřebí provést změny uvnitř architektury připojené ke komponentě, je třeba zobrazit si přímo soubor, který ji obsahuje.

Pro rychlý přechod na místo začátku příkazu či návrhové jednotky, bude možno také použít kontextové menu příkazu či jednotky. V tomto menu se bude nacházet položka **Jump to statement**, která způsobí přepnutí do textového režimu editoru, a přesun kurzoru na pozici začátku příkazu či jednotky. V případě, že se příkaz nachází v jiném souboru než je zobrazovaný soubor, pak se také příslušný soubor nejdříve otevře.

5.1.3 Okno nastavení aplikace

Okno nastavení aplikace je důležitou součástí každé aplikace s grafickým uživatelským rozhraním. Jedná se o různá nastavení, týkající se vzhledu i chování aplikace. Aplikace bez grafického uživatelského rozhraní většinou umožňují nastavení pomoci konfiguračního souboru či parametrů příkazové řádky. U aplikací s grafickým uživatelským rozhraním je takové řešení nastavení pro uživatele nepohodlné a neintuitivní.

Z toho důvodu je vhodné do aplikace přidat dialogové okno, které umožní uživateli upravit vhled a chování aplikace. V případě textového editoru se tak může jednat například o nastavení fontu, velikosti tabulátorů a podobně.

Návrh tohoto okna také probíhal v nástroji *Qt Designer*, který se v tomto případě ukázal jako velmi užitečný, protože potřeby nastavení aplikace se v průběhu vývoje značně měnily a rozšiřovaly. Nástroj *Qt Designer* pak umožnil velice rychlé přidání nové položky natavení bez toho, aby se musel příliš upravovat zdrojový kód aplikace. Rozšířit bylo tak zapotřebí jen logiku chování dialogu o nově přidanou položku, ale zbytek zůstal nezměněn.

Na obrázku č. 5.3 lze vidět finální podobu dialogu. Z pohledu návrhu rozhraní je pak důležité upozornit hlavně na uspořádání prvků. Většinu plochy okna zabírá panel s listy (**QTabWidget**), a pod ním pak se nachází trojice tlačítek, z nichž jedno slouží na uzavření okna s uložením změn, druhé na uzavření okna bez uložení změn a třetí na uložení změn bez uzavření okna.

Listy v panelu pak slouží pro logické rozčlenění různých druhů nastavení. To je zapotřebí zvláště v případě většího množství nastavení, protože v případě, že by všechno nastavení bylo součástí jednoho listu, pak by dialogové okno bylo veliké a nepřehledné. Další způsob členění je pak do skupin, aby se odlišilo, čeho se nastavení týká.

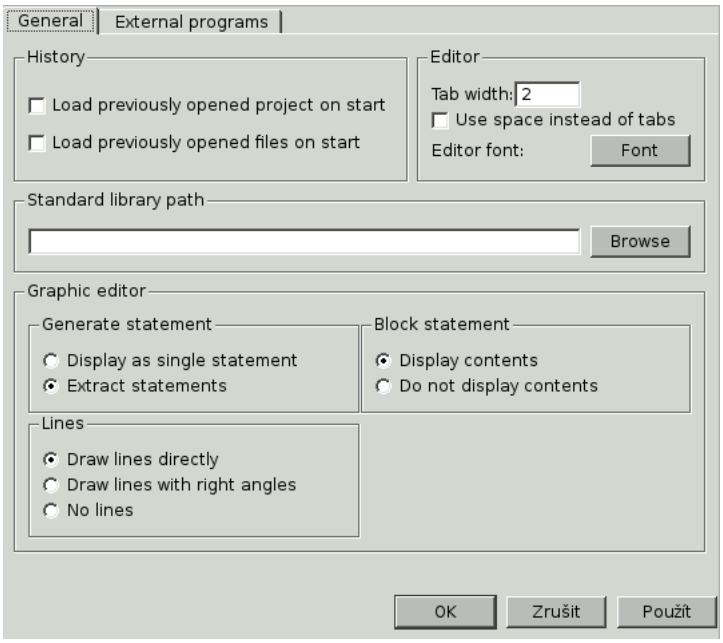
Toto okno lze vyvolat volbou **Settings** - **Preferences** v hlavním menu.

5.1.4 Dialog vytvoření nového projektu

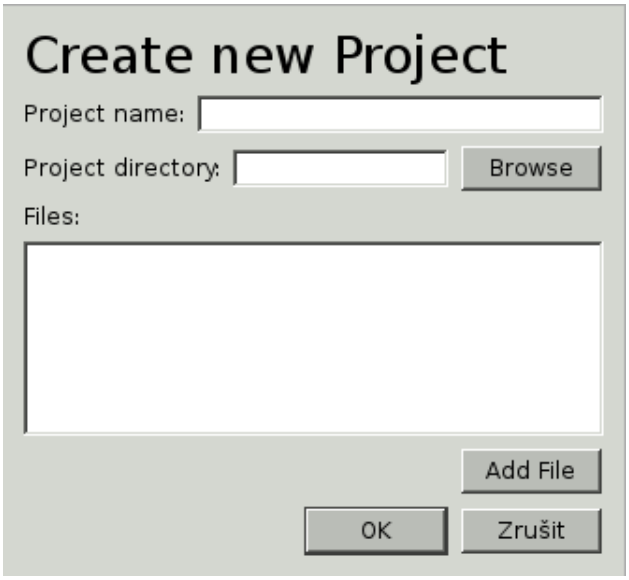
Tento dialog se zobrazí v případě vytváření nového projektu (hlavní menu **File** - **Project** - **New project**). Jeho rozvržení lze vidět na obrázku č. 5.4.

Dialog se skládá z tří textových polí. První (**Project name**) je určeno pro pojmenování projektu. Druhé pole (**Project directory**) je určené k nastavení umístění projektu. Cestu do adresáře projektu lze zapsat přímo, nebo po kliknutí na tlačítko **Browse** se zobrazí dialog pro výběr adresáře. Třetí pole je seznam souborů, které budou přidány do projektu. Přidat soubory lze tlačítkem **Add File**. Ve spodní části dialogu se nachází tlačítko pro potvrzení a pro zrušení tohoto dialogu.

Po vytvoření nového projektu se tento projekt automaticky otevře.



Obrázek 5.3: Dialogové okno s nastavením



Obrázek 5.4: Dialogové okno pro vytvoření nového projektu

5.1.5 Ostatní dialogy

Pro účely projektu byly vytvořeny ještě další dva jednoduché dialogy. První je **EditDialog**, který slouží pro zadání vstupu od uživatele a druhý je **ListDialog**, který se používá výběru jedné položky ze seznamu. **EditDialog** se v projektu využívá například pro změnu jmen komponent nebo změnu jmen souborů a projektu. **ListDialog** je určen primárně pro výběr, která návrhová jednotka bude zobrazena.

Kapitola 6

Implementace aplikace

6.1 Grafické prvky pro zobrazení obvodu

Pro zobrazení obvodu se využívá *Qt Graphics View Framework*. Jde o systém, který umožňuje jednoduše vytvářet jednotlivé grafické prvky. Zobrazování obvodu je založeno na principu návrhového vzoru *Model-View-Controller*, s tím rozdílem, že grafické prvky obvodu zde fungují zároveň jako *View* i *Controller*. Tento návrhový vzor funguje na principu oddělení dat od zobrazovací části. Konkrétně v případě tohoto programu, analyzátor nezobrazuje přímo výsledek analýzy, ale vytváří strukturu, která je poté zobrazena na obrazovce pomocí grafických prvků.

Pro účely zobrazování obvodu byly vytvořeno několik tříd, které reprezentují jednotlivé součásti obvodu. V příloze B.1 je vidět diagram zobrazující dědičnost tříd grafických prvků. Třída **VhdlBaseItem** je výchozí třída všech grafických prvků tohoto projektu, kromě třídy **VhdlGraphicsContainerManipulatorItem** a jejich potomků, které nepotřebují stejné rozhraní.

Třída **VhdlBaseItem** poskytuje svým potomkům především možnost pojmenování, připojení libovolného jiného prvku (tyto prvky budou pak sdílet některé vlastnosti), připojení štítku a přístup k globálnímu nastavení aplikace. Tato třída je odvozena od třídy **QGraphicsItem**.

Rozhraní pro pojmenování se skládá ze tří metod. První metoda je `name()`, která vrací název objektu. Druhá metoda je `setName()`, která nastaví jméno prvku a všech připojených prvků. Nakonec je zde metoda `setOnlyMyName()`, což je metoda, která je volána metodou `setName()` pro tento i připojené prvky. Pokud je tedy zapotřebí upravit pojmenovávání prvku, je preferovaný způsob reimplementovat pouze metodu `setOnlyMyName()`. Výchozí implementace neumožňuje pojmenovat prvek jménem, které patří mezi klíčová slova jazyka VHDL.

Třída **VhdlGraphicsLabelItem** je určena jako štítek pro ostatní prvky. Rozhraní třídy **VhdlBaseItem** pro pojmenování umožňuje, že při změně jména štítku se změní i jméno prvku, ke kterému je štítek přiřazen. Příliš dlouhé jména je lepší omezit, aby pak štítek nezabíral příliš velký prostor. Děje se tak automaticky, podle nastavené hodnoty maximální délky textu štítku. V případě, že uživatel chce znát celé jméno, pak stačí podržet kurzor myši nad štítkem a zobrazí se místní nápověda s plným jménem. **VhdlGraphicsLabelItem** mimo jiné dědí od třídy **QGraphicsSimpleTextItem**, díky které může zobrazovat text.

Třída **VhdlGraphicsPinItem** představuje signál použitý v paralelním příkazu. Předkem této třídy je **QGraphicsLineItem**, díky které má tvar čáry. Od této třídy jsou pak odvozeny třídy **VhdlGraphicsPortItem** a **VhdlGraphicsMappedItem**, které se liší

pouze tím, že poskytují více informací. **VhdlGraphicsPortItem** se používá pro reprezentaci portu entity či bloku a **VhdlGraphicsMappedItem** představuje signál, který je na tento port mapován.

Třída **VhdlGraphicsRectItem** je výchozí třídou pro prvky, které mají reprezentovat zobrazované příkazy a návrhové jednotky. Přidává do rozhraní metody pro přidávání (`addPin()`), vyhledávání (`pin()`), a odebírání (`removePin()`) výstupů signálů, které se budou zobrazovat po stranách. Dále je potřeba, aby uměla zpracovat data, které jsou výsledkem analýzy a zobrazit je. Další důležitou vlastností je, aby v případě, že je prvek této třídy umístěn v kontejneru, si hlídal svou pozici a neopustil prostor kontejneru. Tím se zajistí, aby uživatel mohl prvkem pohybovat, avšak ne mimo kontejner. Mezi další metody určené jako rozhraní patří metody na stanovení umístění popisku signálu vůči jeho výstupu,a kontextové menu, k němuž patří položky na rotaci prvku (**Rotate item**), a na skok do souboru, kde daný příkaz či návrhová jednotka začíná (**Jump to statement**). Třída **VhdlGraphicsRectItem** dědí od třídy **QGraphicsRectItem**, čímž získává obdélníkový tvar. Potomky této třídy jsou **VhdlGraphicsContainer** a **VhdlGraphicsStatementItem**, přičemž druhý zmiňovaný má za účel zobrazovat jednoduché příkazy.

Třída **VhdlGraphicsContainerItem** je výchozí třídou pro prvky, které představující kontejner obsahující ostatní prvky. Tyto prvky mohou být například architektura nebo příkaz bloku. Používá stejné rozhraní jako **VhdlGraphicsRectItem**, jen pro přidávání a vyhledávání vstupů a výstupů jsou zapotřebí rozlišit vnitřní porty (**VhdlGraphicsPortItem**) a vnější namapované signály (**VhdlGraphicsMappedItem**). Od toho jsou přístupné metody pro přidávání (`addInnerPin()`, `addOuterPin()`), vyhledávání (`outerPin()`, `innerPin()`) a mapování (`addPinMapping()`). Mimo jiné si musí kontejner hlídat, zda je dostatečně velký pro všechny vložené prvky, pokud ne, tak se automaticky zvětší na potřebnou velikost.

Jednou z nejdůležitějších úloh kontejneru je propojovat prvky, které obsahuje. Děje se tak na základě identifikátoru signálu. Výsledkem tak bude propojení všech signálů se stejným identifikátorem. Výjimkou pak je v případě mapování komponent a příkazů bloku, když je port namapován jako **open**, což znamená, že daný port není nikam připojen. V takovém případě k propojení nedochází.

Propojení může mít tři formy. První forma je propojení přímo, kdy je čára reprezentující spojení vedena přímo z bodu na konci čáry jednoho signálu do bodu na konci čáry druhého signálu. Druhý způsob je čáru vést pouze v pravých úhlech. Nakonec samozřejmě je možnost nekreslit propojení vůbec. Tato volba je výhodná obzvláště v případě, že je obvod příliš složitý a propojení nepřehledné. Z toho důvodu je přidána do kontextového menu položka **Toggle connection visibility**, která umožní vypnout zobrazování propojení uvnitř kontejneru. Druhá možnost je využití globálního nastavení přes dialogové okno nastavení aplikace, kde jsou na výběr všechny tři způsoby propojování.

Třídy odvozené od **VhdlGraphicsContainerItem** jsou **VhdlGraphicsArchitectureItem** a **VhdlGraphicsBlockStatementItem**. Jedná se o třídy, které jsou určené pro zobrazování architektury, resp. příkazu bloku a instance komponenty. Na rozdíl od svého předka poskytují dodatečné informace o příkazu a několik prvních řádků příkazu, v případě architektury zobrazují také entitu, ke které architektura patří.

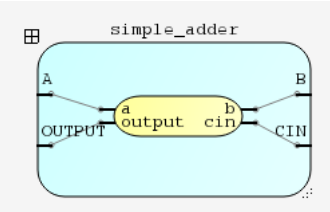
Vzhledem k tomu, že je zapotřebí umožnit uvnitř kontejneru některé funkce myši jako je hromadné označování prvků, je zapotřebí, aby kontejner nereagoval na akce myši. Problém však nastává v případě potřeby kontejner přesunout nebo zvětšit. Z toho důvodu byly zavedeny manipulátory kontejneru. Jde o třídy prvků odvozené od třídy **VhdlGraphicsContainerManipulatorItem**. Mezi ty patří prvek třídy **VhdlGraphicsContai-**

nerResizeGripItem a prvek třídy **VhdlGraphicsContainerMoveGripItem**.

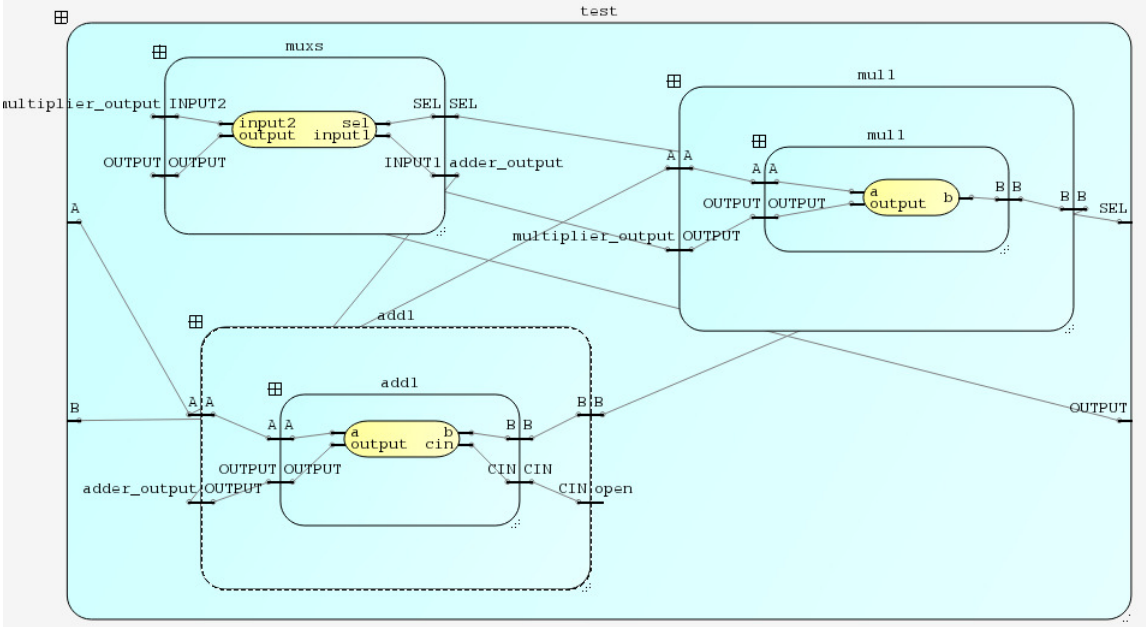
VhdlGraphicsContainerMoveGripItem je třída poskytující rukojeť, která při pohybu přenáší tento pohyb na rodičovský prvek. Když je tedy tato rukojeť součástí kontejneru, dá se použít pro přesun kontejneru po obrazovce. Tato rukojeť se nachází v levém horním rohu kontejneru a má tvar čtverce s křížem uprostřed.

VhdlGraphicsContainerResizeGripItem pak poskytuje rukojeť, která umožní měnit velikost kontejneru. Ta se běžně nachází v pravém dolním rohu kontejneru.

Na obrázku č. 6.1 je vidět příklad zobrazení jednoduché architektury, obsahující pouze jediný příkaz. Na obrázku č. 6.2 je vidět příklad strukturované architektury, přičemž jsou zde vidět dvě konfigurované komponenty a jednu přímou instanci komponenty.



Obrázek 6.1: Zobrazení jednoduché architektury



Obrázek 6.2: Zobrazení strukturované architektury

Zobrazení příkazu generate a příkazu bloku

Jak již bylo dříve zmíněno, tak příkaz **generate** slouží k generování příkazů. Problém je, že bez pokročilé analýzy je velice těžké rozhodnout jaké příkazy přesně budou výsledkem. Z toho důvodu se nabízejí dvě možnosti jak zobrazit příkaz **generate**. První možností je zobrazit celý příkaz **generate** jako jediný příkaz, tak jak se to děje v případě příkazu

procesu, nebo je možné příkazy obsažené v příkazu **generate** zobrazit na stejné úrovni jako ostatní paralelní příkazy.

Z toho důvodu byla vytvořena volba v dialogu pro nastavení aplikace, která umožní uživateli si vybrat, které chování použije.

Podobně lze také nastavit zobrazování příkazu bloku. K dispozici je zobrazení jako jednoduchý příkaz a zobrazení včetně vnitřního obsahu.

6.2 Analyzátor jazyka VHDL

Pro analýzu jazyka je potřeba použít (alespoň částečný) syntaktický a sémantický analyzátor. Analýza bude probíhat na úrovni návrhových jednotek a paralelních příkazů. Pro účely tohoto projektu je tedy zapotřebí vědět pouze z jakých návrhových jednotek soubor skládá, jaké signály a komponenty jsou v nich deklarovány, jaké paralelní příkazy obsahují a jaké signály jsou použity v těchto příkazech. Z toho důvodu je součástí projektu zjednodušený syntaktický analyzátor. To, že jde o zjednodušený analyzátor znamená například, že některé konstrukce budou úplně přeskočeny a tedy může se jednoduše stát, že nebude odhalena chyba v syntaxi. Jedná se především o výrazy, některé deklarace a podobně.

6.2.1 Lexikální analýza

První krok, který je zapotřebí před syntaktickou analýzou je lexikální analýza. Pro lexikální analýzu je použit nástroj *Lex*, z knihovny PLY [3]. Tento nástroj umožňuje jednoduché vytvoření lexikálního analyzátoru.

Stačí vytvořit modul nebo třídu, v ní vytvořit proměnnou **tokens**, do které se uloží seznam názvů všech typů tokenů a pak je třeba nadefinovat buď proměnné, nebo funkce (v případě třídy nadefinovat metody), které popisují lexémy, které se mají zpracovávat. Popis lexému je formou regulárního výrazu. Tento regulární výraz se pak uloží buď přímo do proměnné, nebo pokud se jedná o funkci, tak do dokumentačního řetězce funkce. Tyto proměnné či funkce musí mít jméno ve tvaru **t_XXX**, kde **XXX** je typ tokenu.

Takto vytvořená třída či modul se pak předá konstruktoru třídy Lex a vytvořená instance je hotový lexikální analyzátor. Ten se pak používá tak, že se mu nastaví řetězec se vstupem a pak při každém volání metody `token()` je vrácen jeden token. Token je objektem třídy **LexToken**, která obsahuje čtyři atributy: **value**, **type**, **lineno** a **lexpos**. Atribut **value** obsahuje původní lexém, atribut **type** obsahuje název typu tokenu, **lineno** obsahuje číslo řádku, na kterém se lexém nacházel a **lexpos** pak pozici od začátku.

Pro naše řešení úprav textu je vhodné si uchovávat všechny tokeny, které při analýze vznikly a uchovávat si je v seznamu. Tento seznam se pak využije při úpravách textu pomocí grafického editoru. Tímto způsobem totiž vždy jistota na které místo v textu se zapisuje a při vkládání textu se nemusí nijak složitě zbytek textu posouvat apod. A jakmile jsou všechny úpravy hotovy, všechny položky tohoto seznamu se pak spojí a vytvoří se upravený kód.

Většina lexikálních analyzátorů provádí přeskakování bílých znaků a komentářů. To však není pro účely tohoto projektu příliš vhodné, protože pokud se budou uchovávat tokeny v seznamu, aby se pak z něj později mohl opět zrekonstruovat kód, tak je zapotřebí uchovávat také všechny bílé znaky a komentáře. To by ale později komplikovalo syntaktickou analýzu, protože v té nehrají bílé znaky a komentáře žádnou roli. Z toho důvodu bylo zvoleno jako řešení zapouzdření celého lexikálního analyzátoru do vlastní třídy.

Třída **VhdlLexer** slouží jako zapouzdření lexikálního analyzátoru a zároveň obsahuje všechny pravidla pro lexémy. Konstruktor této třídy přijímá jeden parametr, kterým je kód, nad kterým se má provést lexikální analýza. Tato třída má metodu `token()`, která vrací objekt třídy **LexToken** tak, jak by to prováděl přímo objekt třídy **Lex**, ale kromě toho si vnitřně sestavuje seznam všech tokenů a do jednotlivých tokenů přidává další tři atributy. Prvním je atribut **index**, což je index tokenu v seznamu tokenů a druhý atribut je **list**, což je odkaz na seznam tokenů, ve kterém je tento token obsažen, třetí atribut je **objectType**, který hned nastaví na **None** a slouží pro ukládání typu objektu u tokenů obsahujících identifikátor. Výhodou ukládání seznamu do tokenu je to, že pro rekonstrukci kódu stačí kterýkoliv token. Všechny bílé znaky a komentáře jsou přeskočeny, ale před tím jsou uchovány v seznamu tokenů.

6.2.2 Syntaktická analýza

Pro syntaktickou (a z části také sémantickou) analýzu byla vytvořena třída **VhdlParser**. Tato třída implementuje syntaktickou analýzu shora dolů pomocí rekurzivního sestupu. Syntaktická pravidla vychází z standardu VHDL-93 [11], kde jsou popsány pomocí EBNF[1].

Třída **VhdlParser** prochází postupně kód, přičemž k tomu používá třídy **VhdlLexer**. Tato třída obsahuje několik pomocných metod a pak metody zastupující některé neterminály.

Mezi důležité pomocné metody patří metoda `token()`, která získá token z lexikálního analyzátoru a ten pak vrátí. Kromě toho, pokud je volán s parametrem `trackTypes=True`, pak v případě že narazí na identifikátor, tak prohledá deklarované objekty a zkontroluje jestli není deklarován objekt se stejným identifikátorem. Pokud ano, pak nastaví typ deklarovaného objektu do atributu **objectType**, který je součástí tokenu.

Další důležitá metoda je `_getTokensUntil()`, která slouží k získávání tokenů tak dlouho, než se narazí na token jehož typ byl metodě předán jako druhý parametr. Metoda `_getTokensUntil()` má také nepovinný parametr **trackTypes**, jehož hodnotu pak předává metodě `token()`. Využití této metody je pro přeskočení některých částí kódu, které není třeba zvláště analyzovat. Toho se například u některých deklarací, jako je třeba deklarace podtypu. O podtypu nepotřebujeme příliš vědět, jen nám stačí znát identifikátor a vědět že se jedná o podtyp. Jakmile tedy narazíme na identifikátor podtypu, stačí si jej uložit a pak jen přeskočit na nejbližší středník, kde deklarace končí. Podobně jsou řešeny některé příkazy, kdy např. příkaz přiřazení se také analyzuje pouze tak, že se přeskočí pomocí metody `_getTokensUntil()` až na konec příkazu, přičemž pokud byl předán parametr `trackTypes=True`, tak během toho došlo poznačení všech typů objektů s odpovídajícím identifikátorem.

Velmi používané jsou taky pomocné funkce `expectType()` a `isType()`. Tyto funkce jsou určeny k ověřování typu tokenu. Obě funkce mají stejné parametry. První parametr je token, jehož typ je třeba ověřit, druhým parametrem je buď řetězec, pak se porovnává, zda typ tokenu je shodný s předaným řetězcem. Druhá možnost je předat jako druhý parametr seznam či n-tici řetězců a pak se porovnává, zda token je alespoň jednoho ze zadaných typů. Funkce `isType()` vrací pravdivostní hodnotu, podle toho, zda token je či není zadaného typu. Naproti tomu funkce `expectType()` nevrací nic, ale v případě že token není zadaného typu, vyvolává výjimku **VhdlSyntaxError** s upozorněním na chybu syntaxe, včetně čísla řádku a porovnáním typu tokenu s očekávanou hodnotou.

[1] EBNF = Extended Backus-Naur Form (česky rozšířená Backus-Naurova forma). Jedná se o způsob popisu syntaxe.

Metody zastupující neterminální symboly mají pojmenování vycházející z pojmenování neterminálu, který představují, navíc mají ještě prefix `p_`, který je odlišuje od obyčejných metod. Metoda neterminálu **entity_declaration** má pak název **p_entityDeclaration**.

VhdlParser si v průběhu analýzy neustále uchovává kontextovou informaci o tom, v jaké návrhové jednotce nebo příkazu[2] se nachází a kde má hledat a vkládat deklarace objektů.

Analyzátor tedy vždy vkládá nové deklarace do objektu představující jednotku, ve které se nachází. To funguje na principu zásobníku. K tomu aby získal tento objekt, používá k tomu pomocnou metodu `_scope()`, která vrátí objekt na vrcholu zásobníku. Výchozí objekt je objekt třídy **VhdlProject**. K tomu, aby se analyzátor na začátku nové jednotky zanořil do jejího prostoru platnosti jmen, používá pomocnou metodu `_diveIntoScope()`. Naopak, po ukončení jednotky se použije `_raiseFromScope()`, která způsobí vynoření se z prostoru platnosti jmen ukončené jednotky do prostoru jmen nadřazené.

Analyzátoru je zapotřebí při inicializaci objekt **VhdlProjekt**, do kterého bude zapisovat analyzované informace. Pokud se analyzátoru předá také objekt třídy **VhdlLibrary**, pak budou všechny deklarace prováděny v rámci knihovny, kterou představuje. V případě, že se objekt třídy **VhdlLibrary** nepředá, pak se budou překládané jednotky ukládat do knihovny *work*.

Analýza souboru se spouští metodou `parse()`, přičemž pokud běh této metody skončí bez vyvolání výjimky, pak to znamená, že analýza proběhla úspěšně.

6.2.3 Výstupní struktura dat

Syntaktický analyzátor ukládá veškeré informace o analyzovaných jednotkách, příkazech a deklarovaných objektech uvnitř stromové struktury. Kořenem této struktury je objekt třídy **VhdlProjekt**, který obsahuje objekty třídy **VhdlLibrary**, představující použitou knihovnu.

Tyto knihovny pak obsahují návrhové jednotky (třídy odvozené od **VhdlDesignUnit**) a ty už pak různé deklarace a příkazy. Tím se vytváří kompletní struktura projektu. V příloze C.1 lze vidět kompletní diagram dědičností tříd.

Deklarace běžných objektů jako jsou signály, proměnné, typy a podobně, jsou uloženy jako objekty třídy **VhdlObject**. Deklarace typu komponenty pak jako **VhdlComponentType**, což je třída odvozená od třídy **VhdlInterface**, která poskytuje ukládání portů a generických konstant a jejich porovnávání.

Jednoduché příkazy jsou ukládány jako objekty **VhdlStatement**, ty poskytují několik důležitých metod. Nejdříve jsou zde metody na prohledávání prostoru platnosti jmen `findinScope()`(). Dále jsou zde metody `setStatementInfo()` a `statementInfo()`. Ty mají za úkol najít komentář, který se nachází o řádek výše, než začátek příkazu a pokud obsahuje text "VVRE (Do NOT change this line)", tak podle toho pozná že to jsou dodatečné informace o příkazu. Tyto informace jsou používány pro zobrazování, protože se v nich ukládá pozice, rotace a velikost grafického prvku. Toto řešení zajišťuje, aby pokaždé při zobrazení grafické reprezentace souboru se zobrazily komponenty na stejných místech. Pokud se před příkazem tento komentář nenachází, bude před něj vložen, pokud budou zapotřebí některé informace nastavit. Podobně pak funguje metoda `setName()`, která v případě, že příkaz nemá jméno, a je požadováno toto jméno nastavit, pak vloží před příkaz sekvenci "LABEL: "kde LABEL je jméno příkazu. Nakonec je zde metoda `usedSignals()`,

[2] Zahrnuje deklaraci, konfiguraci a instancování komponenty, deklaraci těla podprogramu, příkazu bloku a příkazu procedury

která vrací seznam slovník všech použitých signálů v příkazu, přičemž každá položka toho slovníku obsahuje seznam všech tokenů s výskyty těchto signálů.

Třída **VhdlDeclarative**, která představuje příkaz či blok, který může obsahovat deklarace, poskytuje metody na přidávání deklarací (`addToDeclarations()`) a vyhledávání deklarovaných objektů(`findInDeclarations()`). Od této třídy odvozená třída **VhdlBlockStatement**, představující příkaz, který může obsahovat jiné příkazy, pak přidává možnost ukládání i vyhledávání příkazů.

Třída **VhdlMapping** představuje příkaz, který obsahuje mapování portů a generických konstant. Tato třída obsahuje metody pro mapování poziční, tak přiřazováním.

Třída **VhdlBlock**, která dědí od tříd **VhdlMapping**, **VhdlInterface** a **VhdlBlockStatement**, slouží k reprezentaci příkazu bloku.

Třída **VhdlComponent**, která dědí od **VhdlMapping** je třídou představující konfiguraci komponenty. Umožňuje přiřadit architekturu k entitě a namapovat ji na typ komponenty.

Třída **VhdlComponentInstantiation** dědí od tříd **VhdlMapping** a **VhdlStatement** a jde o reprezentaci příkazu instance komponenty. Kromě toho, že umožňuje připojit konfiguraci komponenty, tak lze provést přímou instanci entity.

Třída **VhdlDesignUnit** dědí od třídy **VhdlBlockStatement** a je výchozí třídou pro všechny návrhové jednotky. Od ní jsou pak odvozeny třídy **VhdlArchitecture**, **VhdlPackage** (tělo balíku a hlavička balíku se zde nerozlišuje) a **VhdlEntity**, která navíc dědí od třídy **VhdlInterface**. Účel této třídy je pro uchování názvu souboru, ve kterém je návrhová jednotka uložena.

Hotová struktura je pak předaná zobrazovací komponentě a ta obsah zobrazí.

6.2.4 Známé nedostatky implementace

V této kapitole uvedu několik známých nedostatků implementace.

- Nefunkční podpora návrhové jednotky **configuration** — V současné verzi programu použití návrhové jednotky **configuration** není dokončeno. Celý blok se přeskočí. A komponenty zůstanou neosazeny. Konfigurace komponent je momentálně možná pouze přímo v architektuře.
- Nefunkční podpora klíčových slov **all** a **others** při konfiguraci komponent — V případě specifikace konfigurace uvnitř deklarační části architektury není funkční podpora klíčových slov **all** a **others**.
- Propojování signálů v grafické reprezentaci je trochu nepřehledné — Pro propojení signálů v současné době není použito žádného algoritmu pro nalezení stromu s nejkratšími hranami

Kapitola 7

Závěr

Cílem této práce bylo vytvořit editor, který by umožňoval pohodlnější práci vývojáři hardware při úpravách kódu za pomocí grafické reprezentace obvodu. A cíl se mi podařilo částečně splnit. Editor je funkční a použitelný pro úpravu kódu VHDL, umožňuje základní správu projektu a také umožňuje zobrazit grafickou reprezentaci kódu. Nicméně některé vlastnosti nebyly z důvodu špatného rozvržení času dokončeny.

V tomto projektu mám v úmyslu dále pokračovat, dokončit všechny nedodělky a poté editor vydat veřejně na internetu jako open-source projekt.

Přínos této práce je pro mne značný, protože jsem si jednak zlepšil dovednost s jazykem Python, dále jsem si vyzkoušel návrh uživatelského rozhraní a práci s Qt frameworkem a také jsem si oživil své znalosti ze studia jazyků a překladačů.

7.1 Možnosti rozšíření

Možností pro rozšíření je mnoho a vypsat je všechny by vydalo nadlouho a u některých by jejich implementace klidně vystačila na další Bakalářskou práci. Nicméně zmíním takové, které mám v úmyslu časem doplnit do této aplikace.

- **Úplná kontrola syntaxe** - Závislost na externím programu pro kontrolu syntaxe není úplně ideální, vhodné rozšíření by byla implementace kompletního syntaktického analyzátoru jazyka VHDL, aby se odstranila tato závislost.
- **Zobrazování obsahu souborů** - Užitečné rozšíření by bylo pro zobrazení stromové struktury návrhových jednotek v postranním panelu. Například zobrazení deklarovaných objektů uvnitř balíku apod.
- **Sémantické napovídání** - Napovídání klíčových slov a identifikátorů podle kontextu. Například by se při psaní konfigurace komponenty automaticky napovídaly existující entity, nebo při deklaraci signálu automaticky nabízely existující datové typy.
- **Vkládání šablon** - Jednalo by se o nástroje pro vytváření a vkládání šablon kódu.
- **Podpora novějších standardů** - Byť je VHDL-93 do dnes hodně používané, už se jedná o 20 let starý standard a od té doby už VHDL pokročilo. Proto by asi bylo vhodné se podívat na vlastnosti novějších verzí standardu a zkusit je zapracovat.

Literatura

[1] Emacs VHDL Mode. [online], [cit. 2013-05-12].
URL http://www.iis.ee.ethz.ch/~zimmi/emacs/vhdl-mode.html

[2] ISE Design suite. [online], [cit. 2013-05-12].
URL
http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm

[3] PLY (Python Lex-Yacc). [online], [cit. 2013-05-12].
URL http://www.dabeaz.com/ply/ply.html

[4] PyQt4 Reference Guide. [online], [cit. 2013-05-12].
URL http://pyqt.sourceforge.net/Docs/PyQt4/

[5] Python v3.3.2 documentation. [online], [cit. 2013-05-12].
URL http://docs.python.org/3/

[6] QScintilla - a Port to Qt v4 and Qt v5 of Scintilla. [online], [cit. 2013-05-12].
URL http://pyqt.sourceforge.net/Docs/QScintilla2/index.html

[7] Qt 4.8 Documentation. [online], [cit. 2013-05-12].
URL http://qt-project.org/doc/qt-4.8/

[8] Sigasi. [online], [cit. 2013-05-12].
URL http://www.sigasi.com/

[9] Douša, J.: *Jazyk VHDL*. České vysoké učení techické v Praze, 2003, iSBN 80-01-02670-1.

[10] Pedroni, V. A.: *Circuit Design with VHDL*. MIT Press, 2004, iSBN 0-266-16224-5.

[11] Society, I. C.: *IEEE Standard VHDL language reference manual*. New York : Institute of Electrical and Electronics Engineers, 1994, iSBN 1-55937-376-8.

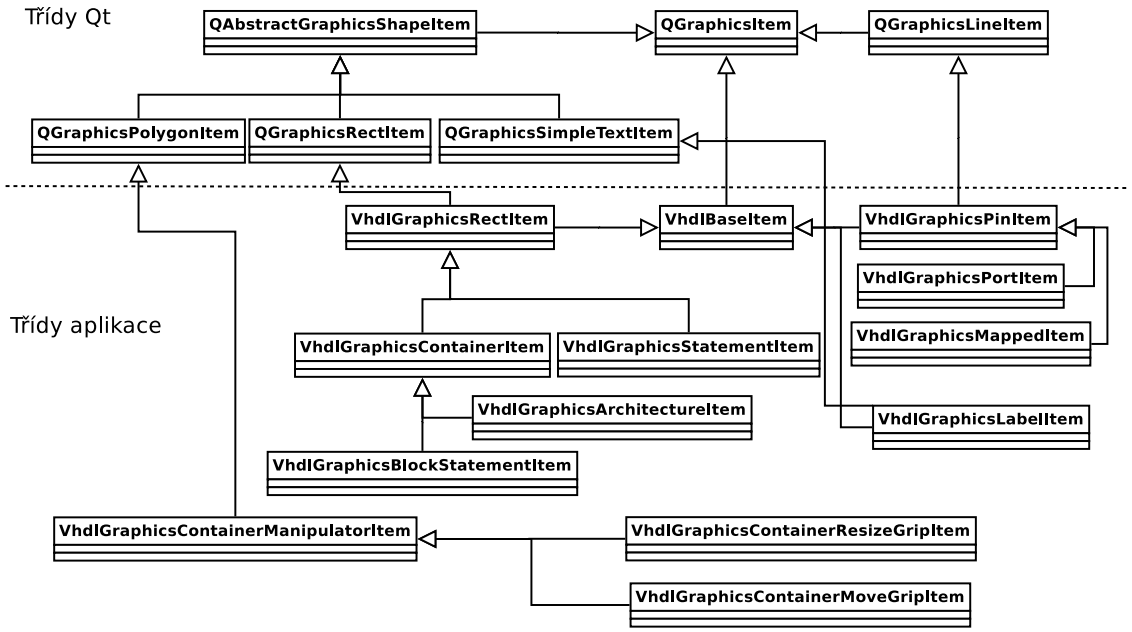
Příloha A

Obsah CD

- **examples** - ukázkové zdrojové soubory
- **stdlibs** - standardní knihovny VHDL
- **README.txt** - Manuál k programu
- **src** - zdrojové soubory programu
- **thesis.pdf** - zpráva ve formátu pdf
- **thesis-src** - zdrojové texty zprávy

Příloha B

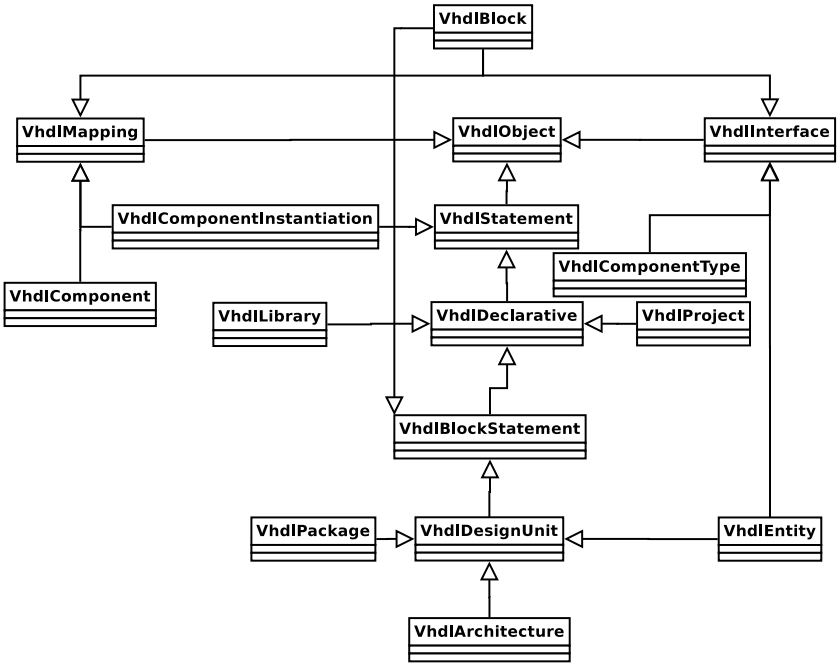
Diagram tříd grafických prvků



Obrázek B.1: Diagram tříd grafických prvků zobrazující dědičnosti

Příloha C

Diagram tříd pro ukládání výsledků analýzy



Obrázek C.1: Diagram dědičností tříd pro ukádání vysledků anylýzy