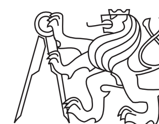
Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství



Diplomová práce

Vývojový nástroj pro generování VHDL kódu a správu projektů

Bc. Jan Matějů

Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

27. června 2012

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu Ing. Pavlu Kubalíkovi, Ph.D. za trpělivost a cenné připomínky. Dále chci poděkovat své přítelkyni, rodičům a kolegům za jejich podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Dobříši dne 27. června 2012 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2012 Jan Matějů. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Jan Matějů. *Vývojový nástroj pro generování VHDL kódu a správu projektů: Diplomová práce*. Praha: ČVUT v Praze, Fakulta informačních technologií, 2012.

Abstract

This master's thesis presents the development of application *Development tool for VHDL code generation and project management* (VHDT). The application simplifies the development and management of VHDL projects. The project is displayed in a well-arranged tree structure depending on the hierarchy of entities. It also helps to maintain projects in a consistent state. Other features include automatic generation of VHDL testbenches and structures based on user-defined templates. The NetBeans platform was used as a basis for the implementation. The thesis also includes a research of existing solutions.

Keywords VHDL, project hierarchy, consistency, generator, entity, NetBeans, parser, editor

Abstrakt

Tato práce mapuje vývoj aplikace *Vývojový nástroj pro generování VHDL kódu a správu projektů* (VHDT). Aplikace usnadňuje vývoj a správu VHDL projektů. Projekt je zobrazován v přehledné stromové struktuře závisející na hierarchii jeho entit. Nástroj zároveň pomáhá udržovat projekty v konzistentním stavu. Mezi další funkce patří automatické generování VHDL struktur a testbenchů na základě uživatelem definovaných šablon. Pro realizaci byla použita platforma NetBeans. Součástí práce je také rešerše stávajících řešení.

Klíčová slova VHDL, hierarchie projektu, konzistence, generátor, entita, NetBeans, parser, editor

Obsah

Úvod **15**
Struktura práce 16

1 Popis problému, specifikace cíle **17**
1.1 Deklarace záměru 17
1.2 Odborný článek 17

2 Rešerše stávajících řešení **19**
2.1 Nástroje umožňující generovat kód na základě šablon 19
2.2 Nástroje pro správu velkých projektů 21
2.3 Shrnutí rešerše 23

3 Analýza a návrh **25**
3.1 Analýza požadavků 25
3.2 Návrh šablon 27
3.3 Stromová hierarchie entit v projektu 32
3.4 Kontrola a oprava nekonzistencí 35
3.5 Analyzátor VHDL kódu, vyplňování šablon 36
3.6 Návrh uživatelského rozhraní 37
3.7 Volba implementačního prostředí 43

4 Realizace **47**
4.1 Platforma NetBeans 47
4.2 Struktura aplikace 50
4.3 Branding Module 50
4.4 JTattoo 51
4.5 VHDL Service API 51
4.6 VHDT Core 54
4.7 VHDL Support 54
4.8 VHDL Project Support 59

5 Testování **67**
5.1 Testování funkčních jednotek 67
5.2 Testování na uživatelích 67

Závěr **69**

Literatura **71**

A Seznam použitých zkratek **73**

B Obrázky **75**

C Instalační a uživatelská příručka **85**
C.1 Požadavky 85
C.2 Instalace a spouštění 85
C.3 Popis práce s aplikací 85

D Příručka programátora **91**
D.1 Požadavky 91
D.2 Rozšiřování aplikace 92

E Ukázky šablon **93**

F Obsah přiloženého CD **99**

Seznam obrázků

2.1 Xilinx ISE - vytvoření entity 20
2.2 Xilinx ISE - vytvoření testbenche 20
2.3 Notepad++ - generování testbenche 21
2.4 Xilinx ISE - generování komponenty a instance 22

3.1 Výsledná struktura ukázkové hierarchie (zobrazeno pomocí aplikace *VHDT*) 34
3.2 Task graph 42
3.3 JVM (převzato z (2)) 43

4.1 Architektura NetBeans platformy (převzato z (3)) 50
4.2 Schéma modulů, ze kterých se skládá aplikace VHDT 51
4.3 Třídní diagram struktur představujících entitu 53
4.4 Třídní diagram struktur představujících nekonzistence 55
4.5 Ukázka stromové struktury projektu zobrazené pomocí nástroje VHDT 61

B.1 Lo-Fi - Hlavní okno editoru 75
B.2 Lo-Fi - Menu File 76
B.3 Lo-Fi - Menu Edit 77
B.4 Lo-Fi - Kontextové menu na projektovém uzlu 78
B.5 Lo-Fi - Kontextové menu na uzlu VHDL souboru 79
B.6 Lo-Fi - Dialogové okno pro správu nekonzistencí 79
B.7 Hi-Fi - Hlavní okno programu 80
B.8 Hi-Fi - Menu Soubor 80
B.9 Hi-Fi - Menu Edit 81
B.10 Hi-Fi - Kontextové menu na projektovém uzlu 81
B.11 Hi-Fi - Dialogové okno pro správu nekonzistencí 82
B.12 VHDT - Dialog nekonzistencí 82
B.13 VHDT - Dialog pro výběr šablony 83

C.1 VHDT - Dialog nekonzistencí 87
C.2 VHDT - Dialog pro výběr šablony 88

Úvod

Jazyk VHDL je jedním z nejpoužívanějších jazyků pro popis hardwarových struktur hradlových polí. Je definován standardem IEEE 1076 z roku 1987. Umožňuje návrh jak logických tak i sekvenčních struktur(7). Jeho hlavní výhodou je jeho univerzálnost. Při návrhu hardwaru se v naprosté většině projektů používají základní struktury, a to především čítač a automat. Ty mají vždy stejný základ a funkčnost, která se v konkrétních implementací liší zpravidla pouze nepatrně. Například automat vždy obsahuje funkci přechodu mezi stavy a výstupní funkci.

Další typickou a nezbytnou věcí při návrhu logických obvodů před jejich fyzickou realizací je testování funkčnosti pomocí testbenchů. Tvorba testbenche je rutinní záležitost. V těle architektury testbenche musíme:

- podle testované entity vytvořit komponentu
- podle portů testované entity vytvořit výčet signálů
- namapovat porty na signály (vytvoření instance)

Nahlédneme-li na strukturu VHDL souborů jako na projekt, objevíme další možnosti jak usnadnit vývojářům hardwaru práci. Jednotlivé entity jsou shlukovány prostřednictvím komponent a instancí do složitějších architektur. Změní-li se nějaká entita, je třeba změnu promítnout do všech architektur, ve kterých je používána. Tento proces se dá plně automatizovat tak, aby se uživatel o udržování konzistence mezi jednotlivými architekturami nemusel vůbec starat.

Cílem této práce je vytvořit nástroj pro správu VHDL projektů a generování VHDL kódu. Výsledný produkt by měl zjednodušovat a zrychlovat práci s VHDL projekty. Bude vývojářům VHDL kódu umožňovat nahlížet na projekt jako na stromovou hierarchii VHDL entit. Dále by měl umět generovat nové entity, části VHDL kódu a testbenche existujících entit na základě uživatelem definovaných šablon. Pomocí šablon by mělo být možné vytvořit například základní strukturu entity, automatu, nebo čítače. Aplikace bude obsahovat vlastní editor VHDL kódu s podporou obarvení syntaxe. Součástí práce bude také rešerše stávajících řešení.

Struktura práce

Práce je členěna do čtyř kapitol:

- Kapitola 1 uvede čtenáře do problematiky vytváření VHDL entit a jejich testování. Dále se zaměřuje na správu VHDL projektů.
- Kapitola 2 se zabývá rešerší stávajících řešení. Tematicky je rozdělena na dvě části - rešerše nástrojů umožňujících generovat VHDL kód a rešerše nástrojů pro správu rozsáhlých projektů.
- Kapitola 3 se věnuje analýze a návrhu řešení. Detailně je popsán návrh UI. Diskutována je i volba implementačního prostředí.
- Kapitola 4 popisuje samotnou implementaci aplikace *Vývojový nástroj pro generování VHDL kódu a správu projektů (VHDT)*. Čtenář se seznámí především s technicky zajímavými částmi realizace, jako je obarvování syntaxe, parserování VHDL kódu, zobrazování stromové struktury či kontrola a oprava nekonzistencí. Podrobně je popsáno také zvolené implementační prostředí - platforma NetBeans.
- Kapitola 5 za zabývá testováním. Podrobněji rozebírá jednotkové testování a testování na uživatelích.

V příloze se nachází instalační a uživatelský manuál.

Kapitola 1

Popis problému, specifikace cíle

1.1 Deklarace záměru

Prostudujte existující řešení pro generování VHDL kódu na základě šablon a nástroje pro správu rozsáhlých projektů obsahujících velké množství VHDL zdrojových kódů. Důraz při porovnávání existujících řešení bude kladen na udržení konzistence rozhraní mezi jednotlivými architekturami.

Navrhněte a naprogramujte aplikaci umožňující jednoduše spravovat větší projekty obsahující více VHDL souborů a to tak, že při změně rozhraní nějaké architektury bude možné zkontrolovat konzistenci a uživateli navrhnout možné úpravy ostatních zdrojových kódů závislých na změněném rozhraní. Nástroj bude také umožňovat generovat části VHDL kódu na základě šablon. Aplikace bude obsahovat vlastní editor s podporou obarvení syntaxe VHDL. Dále bude možné zobrazit strom závislostí jednotlivých architektur. Aplikace bude řádně otestována na uživatelích.

Součástí řešení bude jednoduchý příklad a několik šablon. Text práce bude obsahovat i návod pro jednoduché rozšíření aplikace.

1.2 Odborný článek

Cílem práce je vytvořit nástroj pro správu VHDL projektů a generování VHDL kódu. Výsledný produkt má zjednodušovat a zrychlovat práci s VHDL projekty. Umožní vývojáři VHDL kódu nahlížet na projekt jako na stromovou hierarchii VHDL entit. Při změně rozhraní nějaké architektury bude možné zkontrolovat konzistenci ostatních entit závislých na změněném rozhraní. Uživateli budou navrženy možné úpravy zdrojových kódů. Pokud uživatel například přidá port entitě, která je komponentou jedné či více jiných entit, aplikace to pozná a označí nekonzistentní místa. Dále navrhne automatickou opravu nekonzistencí s možností ručního doladění detailů, jakými jsou např. odsazení, komentáře a podobně.

Další funkcí bude generování VHDL entit, částí VHDL kódu a testbenchů existujících entit na základě uživatelem definovaných šablon. Pomocí šablon bude možné vytvořit například základní strukturu entity, automatu nebo čítače. Tuto funkcionalitu aplikace částečně zdědí z mé bakalářské práce *Konfigurovatelný generátor základních struktur VHDL kódu (VHDL SGen)*. Generování bude nicméně zcela předěláno a uzpůsobeno novému VHDL parseru (který vzniká také v rámci této práce).

Aplikace bude obsahovat vlastní editor VHDL kódu s podporou obarvení syntaxe. Vzniklý produkt bude řádně otestován na uživatelích. Mezi požadavky patří také jednoduchá rozšiřitelnost aplikace. Součástí práce bude ukázkový projekt, několik šablon a rešerše stávajících řešení.

Kapitola **2**

Rešerše stávajících řešení

Kapitolu jsem rozdělil na dvě hlavní části. V první části se zabývám rešerší nástrojů umožňujících nějakým způsobem generovat běžné části VHDL kódu a testbenche existujících entit. Druhá část je věnována nástrojům pro správu rozsáhlých VHDL projektů.

2.1 Nástroje umožňující generovat kód na základě šablon

Pro rešerši jsem vybíral z několika programů a pluginů. Nakonec jsem zvolil dva projekty: komerční *Xilinx ISE* a nekomerční VHDL plugin do programu *Notepad++*. *Xilinx ISE* jsem vybral proto, že je to velmi silný a hojně užívaný nástroj určený k návrhu a testování hardwaru. Je na něm také vedena výuka na této škole. VHDL plugin do programu *Notepad++* vznikal se stejným cílem, jaký má část této práce. Podívejme se tedy na jednotlivé nástroje podrobněji.

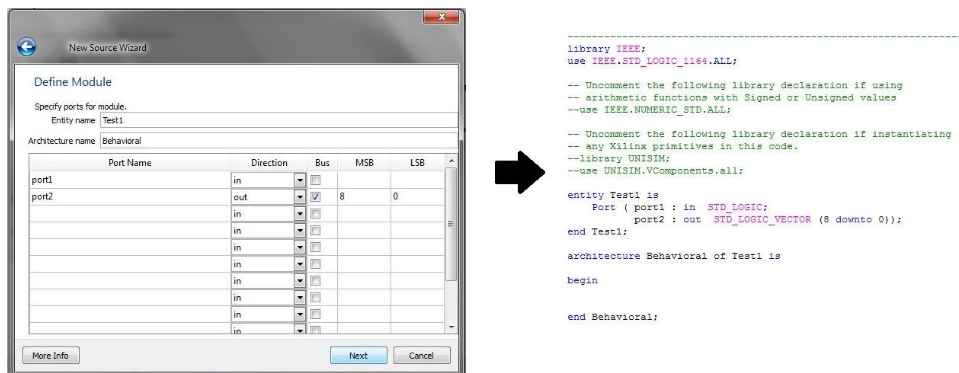
2.1.1 Xilinx ISE

Xilinx ISE je gigantický projekt firmy Xilinx. K datu vzniku této práce byla k dispozici již jeho 14. verze[1]. Je to sice komerční projekt, ale existuje možnost stáhnout omezenou verzi *WebPack* zdarma. Pro její stažení je třeba registrace. *Xilinx ISE* je vyvíjen jak pro OS Windows, tak pro OS Linux.

Ihned po zahájení stahování si všimneme první nevýhody - instalační soubory zabírají téměř 6GB místa na disku. Po instalaci *Xilinx ISE* zabírá místa ještě mnohem víc. Uživatelské prostředí je kvůli velkému množství funkcí zpočátku dost nepřehledné, nicméně si lze velmi rychle zvyknout. Nebudu rozebírat všechny jeho úžasné funkce, zaměřím se pouze na usnadňování práce při tvorbě často se opakujících entit a testbenchů. Ve druhé části rešerše budu zkoumat schopnost udržování konzistence rozhraní entit v projektu.

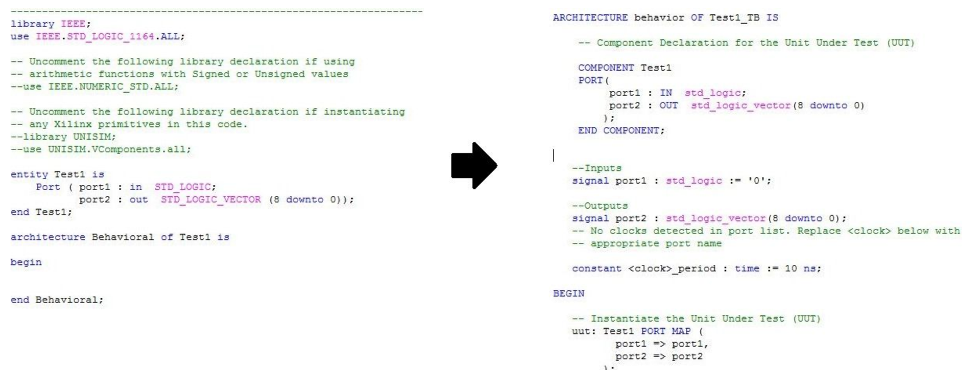
[1] Firma Xilinx nabízí na svých stránkách ke stažení ještě tři předchozí verze programu.

Při zakládání nové struktury máme možnost vyplnit typy a názvy portů, které má budoucí entita obsahovat. V této fázi však nelze využít jinou, již dříve navrženou strukturu. Po vygenerování vznikne struktura obsahující vý-



Obrázek 2.1: Xilinx ISE - vytvoření entity

čet námi definovaných portů a prázdné tělo architektury. Nyní lze využít mnohem zajímavější funkci, a to vložení hotové struktury z množiny existujících šablon. Šablony si můžeme vytvářet i vlastní. Nicméně výraz „šablona“ není úplně na místě, při vkládání totiž nemáme možnost předpřipravenou strukturu upravit. Vloženou strukturu je možné upravovat pouze ručně v kódu, není zde žádná funkce usnadňující vyplňování.



Obrázek 2.2: Xilinx ISE - vytvoření testbenche

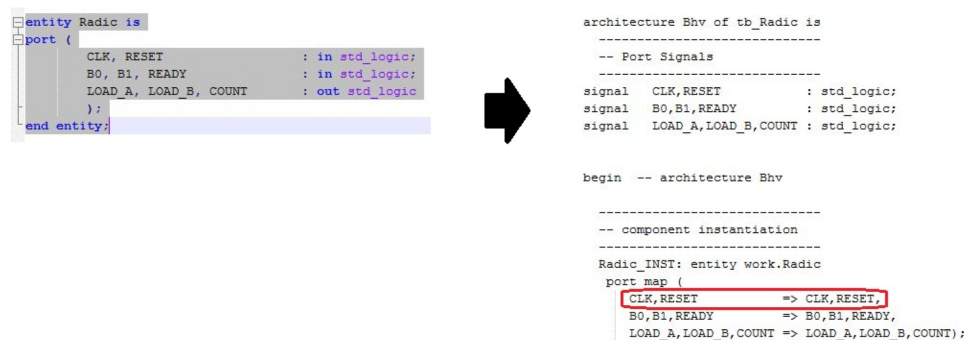
Při tvorbě testbenche z hotové entity nám *Xilinx ISE* sám vytvoří na základě původních portů komponentu. Dále podle portů vytvoří odpovídající testovací signály a také namapování portů na signály. Nicméně nám nedává žádnou možnost určit, jak bude předpřipravený testbench vypadat. Nemůžeme například zvolit, kam přesně se umístí komponenta nebo výčet signálů.

2.1.2 VHDL plugin do programu Notepad++

Na rozdíl od předchozího projektu je to extrémně malý a nenáročný program. Než ho začneme používat, musíme ho začlenit do programu *Notepad++*.

Notepad++ je kvalitní a velmi oblíbený textový editor. Jeho stažení a používání není zpoplatněno[2]. Po instalaci zabírá necelých 10MB prostoru na disku. Obsahuje podporu zvýrazňování syntaxe pro velké množství jazyků. Plně podporuje integraci pluginů. Po instalaci lze s pomocí jednoduchého dialogu v pár krocích přidat VHDL plugin.

Plugin podporuje pouze tvorbu testbenchů. Pokud chceme testbench vytvořit, musíme nejprve v okně *Notepadu++* otevřít soubor s testovanou strukturou. Ve struktuře musíme označit celou entitu a poté pomocí menubaru nebo klávesové zkratky provést příkaz *VHDL copy Entity*, který analyzuje a uloží



Obrázek 2.3: Notepad++ - generování testbenche

vybranou entitu. Následně můžeme v jakékoliv záložce *Notepadu++* pomocí jednoho ze tří příkazů vložit instanciaci, signály a nebo celý testbench. Princip je jednoduchý, nicméně potřeba označit celou entitu kvůli její následné analýze je nepohodlný krok navíc. Plugin také obsahuje chyby. Například pokud máme více portů stejného typu oddělených čárkou, jsou v testbenchi namapovány špatně (viz Obr. 2.3).

2.2 Nástroje pro správu velkých projektů

Jak již bylo popsáno v deklaraci záměru, důraz v této části rešerše je kladen na udržení konzistence rozhraní mezi jednotlivými architekturami. Zabývat se budu nástrojem *Xilinx ISE*. Původně jsem chtěl rozebrat také nástroj *HDL Designer*, ale nedokázal jsem se žádnou legální cestou dostat k testovací verzi. Z manuálu jsem se také nic ohledně udržování konzistence VHDL projektu

[2] Je vydáván pod veřejnou licencí *GNU General Public License*.

nedozvěděl. Na internetu se dá najít mnoho editorů, které mimo jiné podporují také obarvování VHDL syntaxe, či dokonce napovídání klíčových slov. Nástroje pro správu projektů jsem ale nenašel.

Ještě před rozebráním obou nástrojů se zmíním o nástroji *Verilog-Mode* pro textový editor *Emacs*. Jak již název napovídá, jedná se o nástroj pro editaci Verilog kódu, nikoliv pro VHDL kód. Zajímavá je ale jeho funkcionalita a vůbec myšlenka celého projektu. Tou se totiž velmi přibližuje nástroji vznikajícímu v rámci této práce. Případné zájemce proto odkazuji na web autora(9).

2.2.1 Xilinx ISE

V této části textu zkoumám nástroj *Xilinx ISE* se zaměřením na udržování konzistence mezi projekty. Vytvořil jsem jednoduchou hierarchii tří entit - entita *C*, obsahující jako komponentu entitu *A* a nezávislou entitu *B*. Náhled na stromovou strukturu kódu je zde řešen podobně, jako je plánován v nástroji VHDT.

Jedinou funkcí, která usnadňuje vytváření komponent a instancí, je možnost vytvořit z vybrané entity nový soubor obsahující komponentu a instanci vycházející z této entity. ISE uživatele v komentáři k souboru navádí, ať si generovaný kód ručně zkopíruje tam, kam je třeba. Porty v instanci jsou namapovány „do prázdna“. Podpůrné signály nejsou řešeny vůbec. Ve vygenerovaném kódu se navíc vyskytuje chyba - poslední port komponenty byl zakončen středníkem (viz Obr. 2.4).

```
entity A is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC);
end A;

architecture archA of A is

begin


end archA;
```

```
COMPONENT A
PORT(
   clk : IN std_logic;
   reset : IN std_logic;
   );
END COMPONENT;

Inst_A: A PORT MAP(
   clk => ,
   reset =>
);
```

Obrázek 2.4: Xilinx ISE - generování komponenty a instance

Pokud testovací entitu *A* modifikuji (například přidáním nových portů), ISE nenabízí žádnou možnost kontroly konzistence. Jediná cesta, kterou jsem objevil, je opět nechat vygenerovat nový kód s komponentou a instancí. To je pro uživatele jistě velmi nepohodlné.

2.3 Shrnutí rešerše

V rámci rešerše jsem prozkoumal funkcionalitu několika programů. Žádná volně dostupná aplikace nenabízí funkce, kterými by měl po dokončení disponovat nástroj vzniklý v rámci této práce.

Xilinx ISE je mocný pomocník návrhářů hardwaru. Nabízí obrovské množství funkcí. Jako správce VHDL projektů má ale velký prostor pro zlepšení. VHDL plugin do programu *Notepad++* je minimalistický ale nedotažený projekt, který se navíc zabývá pouze generováním kódu. Nejvíc mě zaujal nástroj *Verilog-Mode*. Neumožňuje sice pracovat s VHDL projekty, ale pro jazyk Verilog se zdá býti téměř obdobou nástroje, který vzniká v rámci této práce.

Kapitola 3

Analýza a návrh

V této kapitole sestavuji výčet funkčních a nefunkčních požadavků, navrhuji a rozebírám možná řešení a v závěru se zabývám diskuzí nad volbou implementačního prostředí.

3.1 Analýza požadavků

Cílem práce je navrhnout a implementovat aplikaci, která usnadňuje vývojářům hardwaru práci s VHDL projekty.

3.1.1 Funkční požadavky

Níže je uveden přehled funkčních požadavků. Stěžejní požadavky jsou dále rozebrány podrobněji.

- Práce s VHDL projekty
  - Otevření projektu
  - Zavření projektu
  - Vytvoření projektu
  - Importování projektu
  - Smazání projektu
  - Přidání nové entity
  - Úprava entity
  - Smazání entity
  - Zobrazení adresářové struktury projektu
  - Zobrazení projektu jako stromové hierarchie entit
- Kontrola konzistence entity v projektu

- Obarvování VHDL syntaxe
- Generování kódu
  - Generování nových kusů kódu na základě šablon
  - Generování nových entit na základě šablon
  - Generování testbenchů na základě šablon
- Jednoduchá správa šablon
  - Vytvoření nové šablony
  - Upravení šablony
  - Smazání šablony

Zobrazení projektu jako stromové hierarchie entit

Entity, které projekt obsahuje, bude možné zobrazit ve stromové hierarchii. Kořenovým uzlem je projekt, pod ním jsou uzly top entit (entity, které nejsou užity jako komponenty jiných entit). Vnitřními uzly jsou entity, které obsahují další komponenty, a přitom jsou samy komponentami jiných entit. Listy jsou entity, které ve své architektuře neobsahují žádné další komponenty.

V takovéto struktuře se některé entity mohou objevit vícekrát, pokud jsou jako komponenty definovány na více místech.

Kontrola konzistence entity v projektu

Entity se během vývoje projektu mění. Vývojář často upravuje jejich rozhraní. Tím rozumíme přidávání a odebírání generiků a portů. Po takovéto změně je třeba zjistit, zde není entita použita v jiné architektuře jako komponenta a/nebo instance. Pokud tomu tak je, musí být příslušné kusy kódu upraveny dle provedené změny.

Generování kódu

Na základě předem definovaných šablon půjde v aplikaci provádět následující operace:

- **Vytváření nových entit**

  Do projektu bude uživatel moci přidávat nové entity. Ty budou generovány spojením vybrané šablony a uživatelova vstupu. Vstupem se rozumí vyplnění všech customizovatelných položek šablony.

- **Vkládání kusů kódu**

  Do již hotové entity půjde vložit kus generovaného kódu. Ten bude generován opět z vybrané šablony (označené jako šablona fragmentu kódu) a z uživatelova vstupu.

- **Generování testbenchů**

  Testbench bude automaticky generován z testované entity a vstupní šablony označené jako šablona testbenche.

3.1.2 Nefunkční požadavky

Níže je uveden přehled nefunkčních požadavků. Ty stěžejní jsou opět rozebrány podrobněji.

- Rychlá přístupnost všech akcí
- Intuitivní uživatelské rozhraní
- Přizpůsobivost prostředí
- Platformní nezávislost

Rychlá přístupnost všech akcí

Aplikace má usnadňovat práci, její používání tedy musí být rychlé. Všechny důležité a často používané akce budou přístupné v hlavním toolbaru v podobě přehledných a jasně odlišených ikon. Dále budou všechny akce spustitelné pomocí klávesových zkratek.

Přizpůsobivost prostředí

Každý vývojář má svůj specifický styl formátování a strukturování kódu. Někdy bývá dokonce zvykem dodržovat jednotné styly na úrovni projektů či dokonce celých firem (programátoři pracující na stejném projektu mají předem určeno, jak musí struktura kódu vypadat). Je tedy důležité, aby bylo možné editor kódu vhodně nastavit (velikost tabulátoru, obarvení syntaxe a podobně).

Dále je většinou každý uživatel zvyklý na zažité zkratky. Ty by měly v aplikaci jít také jednoduše nastavit.

3.2 Návrh šablon

Volba šablony je důležitou součástí návrhu řešení. Šablona by měla být pro uživatele přehledná, pochopitelná a lehce osvojitelná. Správně navržená šablona může navíc zásadním způsobem usnadnit následnou analýzu a vyplňování šablony. Tato fáze návrhu tedy nesmí být podceněna.

Aplikace bude pracovat se třemi druhy šablon: šablona celé entity, šablona části VHDL struktury a šablona VHDL testbenche. První dvě šablony umožní rychlé vytváření opakujících se fragmentů VHDL kódu. Pomocí šablony VHDL testbenche bude uživatel mít možnost rychle vytvořit entitu určenou k testovaní existující struktury.

3.2.1 Šablony entit a částí VHDL struktur

Tyto šablony se od sebe budou lišit pouze informací, o kterou z nich jde. Tím bude možné rozlišit, zda lze konkrétní šablonu použít pro vložení kusu kódu, nebo pro vytvoření úplně nové VHDL entity. K rozlišení budou použity následující anotace:

- `@CodeTemplate` - označuje šablonu části kódu
- `@EntityTemplate` - označuje šablonu celé entity

Globální proměnné

Šablona bude uživateli umožňovat definovat univerzální proměnnou. Odkaz na tuto proměnnou (dále *globální proměnná*) bude možné použít místo názvu portů, signálů, typů atd. Dále ji bude možné použít jako část názvu jakékoliv jiné proměnné VHDL kódu. Pokud uživatel při vyplňování šablony změní název globální proměnné, bude tento název nahrazen na všech místech, kde byl použit odkaz na danou proměnnou. Tímto způsobem lze naráz přejmenovat velké množství záznamů. V tom spočívá hlavní smysl zavádění globálních proměnných.

Je třeba určit, kde a jak bude globální proměnná definována a jakým způsobem se na ni v šabloně budeme odkazovat. V první fázi návrhu se počítalo s tím, že proměnná bude zadefinována při prvním výskytu v šabloně. V takovém případě není třeba rozlišovat definici globální proměnné od odkazu na ni. První odkaz je zároveň i definicí. K označení globální proměnné byl vybrán znak '`$`', který ve VHDL kódu nemá žádný klíčový význam. Pokud by globální proměnné reprezentovaly pouze kompletní názvy proměnných VHDL kódu, stačilo by znakem '`$`' označit jenom začátek globální proměnné. To by ovšem bylo omezující. Je žádoucí, aby uživatel mohl definovat pomocí globální proměnné pouze část názvu skutečné proměnné. K tomu je nutné označit začátek i konec globální proměnné. Globálně závislé proměnné VHDL kódu potom budou v šabloně aplikace VHDL SGen vypadat následovně:

```
$cntr$
$cntr$_en
my_$cntr$
my_$cntr$_en
```

Globální část názvu může být ve formě prefixu, postfixu i uprostřed názvu. Počítá se také s tím, že aplikace bude nahrazovat i globální proměnné vyskytující se v komentářích.

VHDL parser, který byl implementován v rámci této práce (viz Kap. 4), čte komentáře jako jeden řetězec znaků. Neuměl by tedy najít globální proměnnou, která byla definována pouze v komentáři. Z tohoto důvodu (a v neposlední řadě také kvůli přehlednosti) byla zavedena *definice globální proměnné.* Umisťuje

se na začátek šablony, mezi anotaci označující typ šablony a klíčové slovo `entity`. Uživatel má lepší přehled o globálních proměnných. Zároveň se tím aplikaci usnadní parsování šablony. Globální proměnná je definována pomocí symbolu '`@`'. Ten se dává pouze před název globální proměnné. Definice globální proměnné `cntr` vypadá takto:

```
@cntr
```

Ve vygenerované struktuře nesmějí zůstat ani tyto definice, ani případné odkazy uzavřené v dolarech[3]. Při finálním vyplňování šablony jsou proto definice globálních proměnných odstraněny.

Na jednoduchém příkladu si ukážeme, jak bude vypadat práce s globálními proměnnými. Definujme jednu proměnnou, jejíž odkazy využijeme na několika místech kódu zároveň. V příkladu je použita šablona celé entity[4].

```
--Sablona ukazujici praci s globalnimi promennymi
@EntityTemplate --jedna se o sablonu cele entity
@citac    --definice globalni promenne

entity muj_$citac$ is   --odkaz pouzity k pojmenovani entity
port (
           clk             : in std_logic;
           --jmena portu pomoci odkazu
           $citac$_reset   : in std_logic;
           $citac$_enable  : in std_logic;
           $citac$_done    : out std_logic;
        );
end muj_$citac$;

--odkaz ve jmenu architektury
architecture muj_$citac$_arch of muj_$citac$ is
...
end muj_$citac$_arch;
```

Pokud uživatel vybere šablonu jakožto předlohu nové entity, zobrazí se na základě analyzovaných dat dialog s výčtem všech definovaných globálních proměnných (zde tedy pouze `citac`). Uživatel má možnost modifikovat název, nebo ponechat původní. Po stisknutí tlačítka *Generate* se ze šablony odebere řádek s definicí globální proměnné (`@citac`) a na místa všech odkazů `$citac$` se vloží nově zadané jméno (případně zůstane původní „citac“). Změníme-li tedy název např. na „counter“, bude výsledná struktura vypadat následovně:

```
--Sablona ukazujici praci s globalnimi promennymi
```

[3] Tedy odkazy na definované proměnné.

[4] Označena anotací `@EntityTemplate`

```
entity muj_counter is
port (
           clk             : in std_logic;
           counter_reset   : in std_logic;
           counter_enable  : in std_logic;
           counter_done    : out std_logic;
        );
end muj_counter;

architecture muj_counter_arch of muj_counter is
...
end muj_counter_arch;
```

3.2.2 Šablona VHDL testbenche

Vytváření komponent, generování signálů podle původního výčtu portů a mapování signálů na porty - to jsou tři nejdůležitější a zároveň dostačující věci k tomu, aby uživatel mohl být oproštěn od mechanického a stále se opakujícího procesu vytváření testovacích entit. Některé existující nástroje sice umí vytvořit testbench, ale uživatel nemá možnost ovlivnit jeho podobu. Aplikace VHDT bude pro šablonu VHDL testbeche definovat sadu příkazů, pomocí kterých bude uživateli umožněno vytvořit testovací entitu přesně podle jeho představ. Všechny příkazy jsou z důvodu přehlednosti a snadného parserování z obou stran ohraničeny znakem '$'. Následuje přehled příkazů a kódu, který se na jejich základě vygeneruje.

Pro snazší pochopení definujme jednoduchou entitu, z níž budou odvozeny jednotlivé části testbenche.

```
--Testovana entita
entity moje_entita is
port (
        clk, reset : in std_logic;
        vystup     : out std_logic;
     );
end moje_entita;

architecture ...
```

Pomocí analyzátoru[5] zjistíme, že entita má název „moje_entita“ a obsahuje porty „clk“, „reset“ a „vystup“. Rozeberme tedy příkazy, které může uživatel použít v šabloně testbenche:

[5] Princip viz Kap. 4

- `$entityName$` - Na všechna místa, na kterých se vyskytuje tento příkaz, bude vloženo jméno testované entity (i do komentářů). V našem případě se tedy všechny značky `$entity$` nahradí řetězcem `moje_entita`.

- `$tbName$` - Značka představuje souborové jméno nového testbenche. Vzniká spojením jména testované entity a postfixem „_tb“. Výhodou tohoto jména je, že je kontrolované (to znamená, že pokud v projektu existuje entita se stejným názvem, je k tomuto názvu přidán ještě řetězec „_x“, kde „x“ představuje první volné číslo).

- `$component$` - Na místě této značky se v testbenchi vytvoří celá komponenta obsahující generiky a porty testované entity. Komponenta je kopií původní entity 1:1[6]. Výsledný kód vypadá následovně:

```
component moje_entita is
port (
        clk, reset : in std_logic;
        vystup     : out std_logic;
     );
end component;
```

- `$signals$` - Příkaz pro vložení signálů. Ty se vytvářejí na základě znalosti portů testované entity. Výstup:

```
signal clk : std_logic;
signal reset : std_logic;
signal vystup : std_logic;
```

- `$instance$` - Vytvoření instance (včetně namapování generiků a portů na signály). Výstup:

```
uut : moje_entita
port map(
    clk => clk,
    reset => reset,
    vystup => vystup
);
```

  Defaultně aplikace vytvoří generiky stylem zápisu „v řadě“ a porty stylem zápisu „pod sebou“.

- `$generic_map$`, `$port_map$` - Vložení namapování generiků/portů na signály (styl zápisu „pod sebou“). Výstup:

[6]Požadavek vedoucího byl, aby se zachovalo formátování a všechny komentáře přesně tak, jak jsou použity v testované entitě.

```
port map(
    clk => clk,
    reset => reset,
    vystup => vystup
)
```

- `$generic_map_inline$`, `$port_map_inline$` - Vložení namapování generiků/portů na signály stylem „v řadě“.

```
port map(clk => clk, reset => reset, vystup => vystup)
```

Všechny generované řetězce dodržují odsazení svých původních značek. V kombinaci se značkou pro vložení jména původní entity si může uživatel vytvořit šablonu přesně podle svých potřeb. Ukázka reálně použitelné šablony je v příloze E.

Šablony testbenchů budou zastupovat ještě jednu užitečnou funkci, a to možnost vygenerovat pouze kus kódu vycházející z testované entity. Pokud uživatel po výběru šablony zvolí místo *Generate* možnost *Clipboard*, nevytvoří se nový soubor obsahující testbench. Aplikace pouze uloží vygenerovaný kus kódu do systémové schránky (clipboardu). Šablona testbenche tedy nemusí představovat pouze kompletní entitu, může jí být pouhá část kódu. Tímto způsobem půjde například vložit do clipboardu komponentu vygenerovanou na základě testované entity.

3.3 Stromová hierarchie entit v projektu

Uživatel zvolí kořenový adresář projektu, v jehož podsložkách aplikace vyhledá VHDL soubory. Z nich poté pro uživatele sestaví stromovou hierarchii.

3.3.1 Princip zobrazení entit

Princip zobrazení entit předvedu nejlépe na malém příkladu. Mějme dvě základní entity, které budou dále použity jako komponenty a instance jiné entity:

```
--entita A
entity A is
port (
        a1, a2    : in std_logic;
     );
end A;
architecture ...

--entita B
entity B is
port (
```

```
        b1, b2    : in std_logic;
     );
end B;
architecture ...
```

Nyní vytvořme entitu C, která ve své architektuře využívá entitu A a B.

```
--entita C
entity C is
...
end C;

architecture archC of C is

component A is
port (
        a1, a2    : in std_logic;
     );
end component;

signal a1_C, a2_C : std_logic;
signal b1_C, b2_C : std_logic;

begin

instA : A port map(
    a1 => a1_C,
    a2 => a2_C
);

instA2 : A port map(
    a1 => a1_C,
    a2 => a2_C
);

instB : B port map(
    b1 => b1_C,
    b2 => b2_C
);

end architecture;
```

Takto definovaná jednoduchá struktura by se v projektu měla zobrazit tak, jak je vidět na obrázku 3.1. Za zmínku stojí dvě skutečnosti:

Obrázek 3.1: Výsledná struktura ukázkové hierarchie (zobrazeno pomocí aplikace *VHDT*)

- Entita C může ve své architektuře definovat libovolné množství instancí jednoho typu entity (zde entita *A*, instance *instA* a *instA2*).
- Použitá instance nemusí být v architektuře definována jako komponenta (zde entita *B*, instance *instB*). Pokud bude naopak definována pouze jako komponenta, ve stromě architektur **nebude** zobrazena.

3.3.2 Úskalí, která mohou nastat

Je třeba počítat s nestandardními situacemi, které mohou v rámci projektu nastat. Uvádím jejich přehled a možnosti řešení:

- **V architektuře nějaké entity se objeví komponenta, k níž v projektu neexistuje odpovídající entita (tedy entita se stejným jménem).**

  Situace se dá řešit ignorováním dané komponenty. Elegantnější ovšem bude komponentu ve stromové struktuře zobrazit jako list neznámé entity. Tento list bude ve stromě entit barevně odlišen a nebude poskytovat žádné akce.

- **V projektu existuje více entit se stejným názvem.**

  Situace se dá korektně řešit dvěma způsoby. Aplikace může automaticky zvolit nejpozději vytvořený či modifikovaný soubor. Tato metoda má výhodu v tom, že nezatěžuje uživatele často zbytečnými dotazy. Nemusí ovšem každému vyhovovat. V některých systémech navíc neexistuje podpora data vytvoření souboru a dá se tedy spoléhat pouze na datum modifikace, které bývá u nově vytvořených a nemodifikovaných souborů prázdné.

  Jako korektnější řešení se mi jeví označit uzel dané entity v hierarchii například jinou barvou. Vyvoláním akce z kontextového menu uzlu uživatel může zvolit, která verze entity se použije.

3.4 Kontrola a oprava nekonzistencí

Stěžejní funkcí aplikace je udržování konzistence architektur jednotlivých entit v rámci projektu. Jedná se především o případy, kdy uživatel modifikuje výčet portů či generiků nějaké entity. Tuto změnu je třeba promítnout do všech entit, v jejichž architekturách se modifikovaná komponenta nachází.

Projekt bude reprezentován množinou entit, které obsahuje. Vyvolá-li uživatel nad uzlem některé z entit (dále „originální entita“) akci *Kontrola konzistence*, projde aplikace množinu všech entit a nalezne ty, ve kterých se originální entita vyskytuje jako komponenta. Nad těmito entitami následně proběhne kontrola konzistence. Nekonzistence může být tří typů a dvou podtypů. Typy nekonzistencí:

- **entita X komponenta** - pokud se deklarace komponenty liší od originální entity
- **entita X signály** - pokud výčet signálů v architektuře neobsahuje všechny porty originální entity
- **entita X instance** - pokud namapování generiků/portů neobsahuje všechny generiky/porty originální entity

Podtypy nekonzistencí:

- **lišící se** - pokud se komponenta/instance originální entity v kontrolované entitě nachází, ale je třeba je opravit
- **chybějící** - pokud komponenta/signály/instance v kontrolované entitě chybí

  Nastává v případě, že byla nalezena pouze komponenta nebo pouze instance, a druhá struktura z této dvojice chybí. Dále nastává pokud chybí ke konkrétní komponentě nějaké signály.

Nalezené nekonzistence budou zobrazeny v přehledné tabulce. Uživatel bude moci nahlížet na nabízené úpravy, případně automaticky generovaný kód ještě ručně modifikovat. Dále by bylo vhodné umožnit volbu úprav, které se provedou a které nikoliv.

Kontrola nekonzistence bude použitelná i pro usnadnění vkládání komponent do architektur entit. Vloží-li uživatel do architektury novou komponentu či instanci pouze jako prázdný blok kódu, uměle tím vytvoří nekonzistenci. Pokud si jí následně nechá automaticky opravit, ušetří si mnoho práce s ručním přepisováním.

3.5 Analyzátor VHDL kódu, vyplňování šablon

Všechny výše zmíněné funkce jsou založeny na znalosti struktury jednotlivých entit. Esenciální součástí této práce proto musí být kvalitní analyzátor VHDL kódu.

Výsledek kódové analýzy může mít několik podob. Lexikální analyzátor prochází kód a vrací jednotlivé lexikální symboly (tokeny), na které narazil. Na základě postupně získávaných tokenů syntaktický analyzátor identifikuje jednotlivé kódové struktury (např. porty, signály, procesy, componenty, entity a podobně). Jeho výstupem zpravidla bývá vnitřní podoba analyzovaného kódu.

Nejčastěji se vnitřní podoba ukládá ve formě derivačních stromů. Tato struktura se v praxi využívá pro překlad z jednoho jazyka do jiného (např. do již zmiňovaného byte-code, nebo do strojového jazyka). V případě této práce je ovšem zbytečné ukládat celý kód do nějaké vnitřní struktury. Bylo by to navíc obtížně realizovatelné. Při analýze a následném vyplňování šablon je totiž velmi důležité, aby výsledné struktury přesně zachovávaly formát a původní podobu šablon. Jaký výstup by tedy měl poskytovat syntaktický analyzátor této aplikace?

- **analýza šablon**

  Při analýze šablony je třeba získat množinu globálních proměnných a seznam všech výskytů v kódu (včetně souřadnic výskytu). Výstupem analyzátoru tedy může být obyčejná tabulka.

- **vyplňování šablon**

  Po získání vstupu od uživatele je třeba dle editované tabulky vyplnit šablonu. Pokud tabulka bude obsahuje proměnné spolu se souřadnicemi v kódu, není nutné šablonu opět parserovat. Je ovšem velmi důležité nahrazovat proměnné v šabloně **odspodu**, aby nemusely být po každém provedeném insertu přepočítávány pozice všech zbývající.

- **vyplňování šablony testbenche**

  Analýza šablony testbenche probíhá totožně jako analýza šablony entity či fragmentu kódu. Rozdíl v nárocích na parser je v tom, že je nutné poměrně detailně znát strukturu testované entity. Aby bylo možné vytvořit testbench, je třeba znát jméno entity, tabulku generiků a tabulku portů.

- **vytváření náhledu na stromovou strukturu projektu**

  Tato funkcionalita vyžaduje znalost architektur všech entit v projektu. Konkrétně jde o jména entit a výčty jejich komponent a instancí. Na základě těchto údajů se entity dají v rámci projektu zobrazit ve stromové hierarchii svých architektur.

- **hledání nekonzistencí**

  Kontrola nekonzistencí vyžaduje nejpodrobnější pohled do struktury každé entity. Kromě jména entity a výčtu komponent a instancí musíme znát také množinu generiků a portů entit (a komponent) a namapování jednotlivých generiků a portů v instancích. U generiků a portů nestačí jejich jména, je třeba znát i jejich typy (u portů i orientace). Dále je třeba získat výčet signálů definovaných v těle architektury.

Výstupem parseru by tedy měly být struktury obsahující všechny výše zmíněné informace. Jejich přehled se nachází v kapitole 4.

3.6 Návrh uživatelského rozhraní

V této části textu předkládám návrh uživatelského rozhraní v podobě, v jaké vznikl v rámci předmětu *Návrh uživatelského rozhraní (MI-NUR)*. Výsledný produkt se na základě uživatelských zkušeností a drobných změn v požadavcích na funkcionalitu lehce změnil. Rozdíly budou jasné z realizace (4) a uživatelské příručky (C).

3.6.1 Popis produktu

Aplikace VHDT umožňuje vývojářům hardwaru pokročilou správu VHDL projektů a dále umožňuje pracovat se šablonami struktur VHDL kódu.

3.6.2 Business požadavky

- Usnadnění práce při programování HW
- Snadná propagace změn základních entit stromem projektu
- Rychlé vytváření opakujících se entit pomocí šablon
- Generování testbenchů existujících entit
- Přehledný editor kódu
- Platformová nezávislost prostředí
- Intuitivní a rychlá práce s aplikací

3.6.3 Omezení

- Aplikace je jednookenní
- Předpokládáme funkční editor textu zvládající základní operace typu *open, save, undo, redo, copy/paste, find, replace, ...* (platforma NetBeans)

- Předpokládáme funkční prostředí umožňující přidávat, odebírat a přesouvat prvky hlavního okna (platforma NetBeans)

3.6.4 Brainstorming

- Musí to být rychlé
- Každá akce musí jít spustit nějakou klávesovou zkratkou
- Důležitý je toolbar, odkud půjdou rychle volat akce
- Není důležité, aby uživatel uměl s aplikací ihned intuitivně pracovat
- Není totiž na „jedno použití“ typu e-shop s nějakým úzce specializovaným zbožím
- Uživatel bude mít čas se s aplikací seznámit a naučit se jí používat
- Poté ji bude používat pravděpodobně každý den, většinu času stráveného při návrhu hardwaru
- Používání zaučenou osobou musí být velmi rychlé, dialogy nesmí obsahovat zbytečné kroky navíc a podobně
- Vše ovladatelné pouze pomocí klávesnice

3.6.5 Use cases

- Obecné užití
  - Hlavní okno aplikace vždy obsahuje menubar umožňující vyvolat všechny kontextově dostupné akce
  - Aplikace umožňuje otevřít více souborů v záložkách
  - Hlavní okno obsahuje toolbar pro rychlou práci s aktuálně vybraným souborem
  - Hlavní okno obsahuje toolbar s nejpoužívanějšími kontextově závislými akcemi
  - Editor zvýrazňuje syntaxi VHDL kódu
  - Spodní část okna obsahuje status bar
- Práce s projekty
  - Hlavní okno obsahuje náhled na stromovou strukturu aktuálně otevřených projektů
  - Možnost vyvolání kontextově závislé akce na uzlu projektového stromu

  - Vytvoření projektu
  - Otevření projektu
  - Uložení projektu
  - Uzavření projektu
  - Otevření entity
  - Editace entity
  - Uložení entity
  - Vygenerování entity na základě šablony
  - Vložení kusu kódu generovaného na základě šablony
  - Generování testbenche dané entity na základě šablony
  - Nalezení entit v projektu, které jsou ovlivněny změnou entity (nalezení nekonzistencí)
  - Automatická úprava vybraných ovlivněných entit (oprava nekonzistencí)
  - Ruční úprava vybraných ovlivněných entit

- Práce se šablonami struktur
  - Vytvoření šablony
  - Editace šablony
  - Vyplnění šablony a vygenerování kódu
  - Uložení kódu
  - Smazání šablony

3.6.6 Personas

- František
  - Návrhář hardwaru, 32 let
  - Zajímá se o počítače, komixy, miluje houbaření
  - Člen klubu hráčů CS 1.6
  - Trpí nadváhou, má zvýšený tlak
  - Každý den navrhuje v rámci svého zaměstnání hardware
  - Jelikož se při návrhu hardwaru opakovaně využívají méně či více modifikované základní struktury (například čítače a automaty), musí František velmi často dělat „opičí“ práci.
  - Práci s IDE (jakým je Eclipse nebo NetBeans) zvládá na velmi pokročilé úrovni

3.6.7 Task list

- Vytvoř novou entitu
- Otevři entitu
- Zobraz šablony
- Ulož soubor
- Smaž soubor
- Založ nový projekt
- Otevři projekt
- Smaž projekt
- Zavři projekt
- Přidej do projektu entitu
- Odeber entitu z projektu
- Zkontroluj projekt a zobraz seznam všech nekonzistencí s možností jejich multi-výběru
- Označ nekonzistenci
- Automaticky oprav vybrané nekonzistence
- Otevři vybranou nekonzistenci v editoru (pro ruční úpravu)
- Vyplň šablonu VHDL kódu a vygeneruj novou entitu
- Vlož generovaný kus kódu
- Vyber šablonu testbenche a vygeneruj testbench
- Vyber šablonu testbenche, vygeneruj kód a vlož ho do systémové schránky

3.6.8 Task groups

PROJEKT, ENTITA, ŠABLONA

- Menubar
  - File
    - Vytvoř novou entitu [ENTITA]
    - Otevři entitu [ENTITA]
    - Zobraz šablony [ŠABLONA]

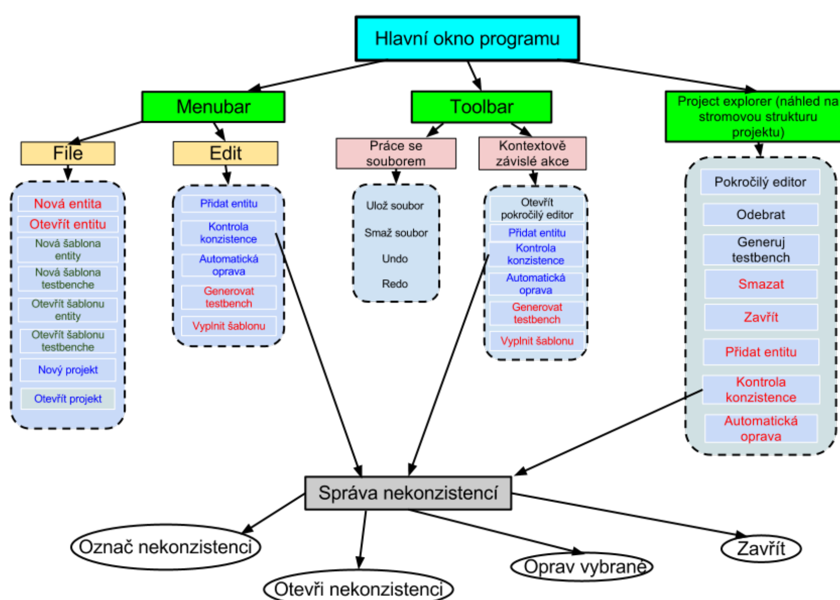
    * Založ nový projekt [PROJEKT]
    * Otevři projekt [PROJEKT]
  - Edit
    * Přidej do projektu entitu [PROJEKT]
    * Vyplň šablonu VHDL kódu a vygeneruj novou entitu [PROJEKT]
    * Zkontroluj projekt a zobraz seznam všech nekonzistencí s možností jejich multi-výběru [PROJEKT]
    * Vyber šablonu testbenche a vygeneruj testbench [ENTITA]
    * Vyber šablonu testbenche, vygeneruj kód a vlož ho do systémové schránky [ENTITA]
    * Vlož generovaný kus kódu [ENTITA]

- Toolbar pro práci se souborem
  - Ulož soubor [ENTITA/ŠABLONA]
  - Smaž soubor [ENTITA/ŠABLONA]
  - Undo [ENTITA/ŠABLONA]
  - Redo [ENTITA/ŠABLONA]
- Toolbar kontextově závislých akcí
  - Přidej do projektu entitu [PROJEKT]
  - Zkontroluj projekt a zobraz seznam všech nekonzistencí s možností jejich multi-výběru [PROJEKT]
  - Vyber šablonu testbenche a vygeneruj testbench [ENTITA]
  - Vyber šablonu testbenche, vygeneruj kód a vlož ho do systémové schránky [ENTITA]
- Náhled na stromovou strukturu projektu (kontextově závislé akce dostupné z nabídky jednotlivých uzlů)
  - Smaž projekt [PROJEKT]
  - Zavři projekt [PROJEKT]
  - Přidej do projektu entitu [PROJEKT]
  - Zkontroluj projekt a zobraz seznam všech nekonzistencí s možností jejich multi-výběru [PROJEKT]
  - Automaticky oprav všechny nekonzistence [PROJEKT]
  - Odeber entitu z projektu [ENTITA]
  - Vyber šablonu testbenche a vygeneruj testbench [ENTITA]

- Dialogové okno pro správu nekonzistencí
  - Označ nekonzistenci [PROJEKT]
  - Automaticky oprav vybrané nekonzistence [PROJEKT]
  - Ručně doupravit vybranou nekonzistenci [PROJEKT]
  - Zavřít beze změn [PROJEKT]

3.6.9 Task graph

Viz obrázek 3.2.



Obrázek 3.2: Task graph

3.6.10 Wireframe

Wireframe (neboli Lo-Fi prototyp) se nachází v příloze B, obrázky B.1 - B.6.

3.6.11 Hi-Fi prototyp

Hi-Fi prototyp se nachází rovněž v příloze B, obrázky B.7 - B.11.

3.7 Volba implementačního prostředí

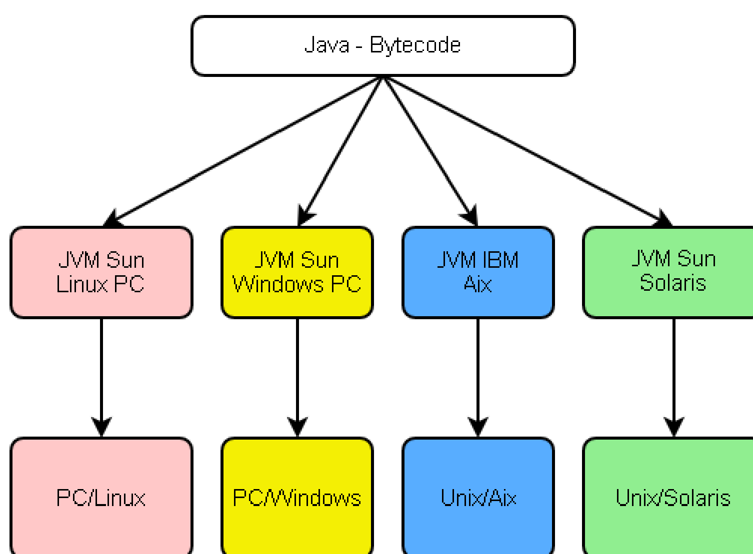
3.7.1 Volba programovacího jazyka

Vzhledem k požadavku nezávislosti na platformě jsem jako implementační jazyk zvolil jazyk Java. Díky JVM je minimálně závislý na OS. Navíc má vlastní grafickou knihovnu SWING, která se skládá z takzvaných *lehkých komponent*[7], tedy komponent nezávislých na operačním systému. O jejich podobu a vykreslování se stará samotná Java. Aplikace potom vypadá na každém operačním systému téměř totožně. Knihovna SWING je tedy pro multiplatformní aplikaci ideální.

Na Javě je také postaveno několik známých a uživateli prověřených vývojových prostředí, jejichž tvůrci poskytují platformu (RCP) všem uživatelům[8].

3.7.2 O jazyce Java

Java je odpočátku vyvíjena jako multiplatformní jazyk. Přenositelnost je zajištěna díky tomu, že je Java jazykem interpretovaným. Interpretovaný znamená, že překlad zdrojového kódu neprobíhá přímo do spustitelného souboru, tj. do strojového jazyka počítače. Místo toho se zdrojový kód přeloží do pseudojazyka zvaného byte-code. Tento jazyk je nezávislý na cílovém počítači. Spouš-



Obrázek 3.3: JVM (převzato z (2))

tění programu probíhá pomocí interpreteru (v Javě je jím JVM). Program

[7]Na rozdíl od starší knihovny AWT postavené na *těžkých komponentách*. Ta sice dosahuje vyššího výkonu, ale vypadá na každé platformě jinak(5).

[8]O dostupných RCP bude referováno dále.

lze spustit na každém počítači, pro který je připravena Java platforma. Ta se skládá z Java Core API a již zmiňovaného JVM.

Problém interpretovaných jazyků ve srovnání s kompilovanými jazyky je jejich rychlost. Vše se vytváří dynamicky a uvolňování paměti probíhá automaticky. To je pohodlné pro programátora, pokud píše rozsáhlejší aplikaci. Pohybuje se na vyšší úrovni abstrakce, nemusí řešit alokování a uvolňování paměti. Za to ovšem platíme snížením rychlosti aplikace. Java se tento problém snažila částečně řešit pomocí takzvaných JIT kompilátorů. Ty během spouštění programu překládají byte-code do strojového jazyka konkrétního počítače. Tato metoda ovšem velmi brzdí zavádění programu do paměti. Java tedy přišla ještě s jedním vylepšením, technologií hot-spot. Pomocí technologie hot-spot se do strojového kódu konkrétního počítače překládají pouze kritická místa programu. Existují vyumělkované příklady[9], ve kterých je Java rychlejší než kompilované jazyky (konkrétně jazyk C). V praxi je Java ovšem stále pomalejší.

Bezesporu velkou výhodou Javy je mocná podpora v knihovnách. Java nabízí silné nástroje pro práci s textem. Dále umožňuje pohodlnou práci s I/O proudy. Poslední knihovnou, kterou zde zmíním, je knihovna obsahující kolekce. Kolekce jsou hotové datové struktury všech možných druhů. Jsou v nich realizovány tabulky, stromy, fronty atd. Jsou-li tyto kolekce používány správně, dokážou být velmi rychlé a efektivní.

3.7.3 Volba platformy

Aplikace umožňující pokročilou správu projektů, která navíc obsahuje sofistikovaný editor kódu, se nedá v rozumném čase implementovat „na zelené louce“. V dnešní době existuje množství ověřených platforem, na jejichž knihovnách se dá postavit robustní desktopová aplikace.

Já jsem se při výběru platformy zaměřil na dvě známá a uživateli oblíbená API, a to *NetBeans Platform* a *Eclipse RCP*.

3.7.4 Eclipse RCP vs. NetBeans Platform

Obě platformy jsou si velmi podobné. Z pohledu uživatele se výsledná aplikace nemusí lišit téměř vůbec. Z pohledu vývojáře se platformy liší především v rozdílném způsobu implementace správy životního cyklu modulů. To jsou ale věci „pod kapotou“, které ve většině případů nemusejí běžného programátora zajímat. Na tomto místě by mohlo být rozsáhlé porovnání všech použitých konceptů a implementovaných funkcí, které by ve výsledku došlo k závěru, že až na drobnosti jsou si obě platformy velmi podobné.

Eclipse RCP má výhodu v existenci velkého množství pluginů a rozšíření. Jejich průzkumu jsem již dříve věnoval poměrně dost času. Nakonec jsem ale došel k závěru, že pro tuto práci by žádný z nich neměl znatelný přínos.

[9] Viz např. pokus pana Herouta(6).

V čem ovšem vidím podstatný rozdíl, který nakonec nejvíce ovlivnil volbu platformy, je grafické prostředí. Eclipse RCP využívá knihovny SWT, zatímco platforma NetBeans je postavená na knihovně Swing. O výhodách a nevýhodách těchto dvou grafických knihoven se v minulosti vedly vášnivé diskuze. Zmíním zde hlavní výhody a nevýhody obou knihoven.

- SWT
  - \+ množství příkladů na webu
  - \+ používá nativní knihovny cílového systému, uživateli připadá „známý“
  - \- používá nativní knihovny cílového systému, vyžaduje nativní knihovny pro každý cílový systém
  - \- nemusí podporovat každé chování na všech systémech
- Swing
  - \+ součást knihovny Java - nepotřebuje žádné další knihovny
  - \+ vypadá a chová se stejně na všech platformách
  - \+ dobrá online dokumentace
  - \+ podporován oficielními rozšířeními Javy
  - \- nativní look and feel se může chováním lišit od skutečného nativního systému
  - \- lehké komponenty jsou mírně náročnější na systémové zdroje

Swing z porovnání vychází vzhledem k nefunkčním požadavkům aplikace lépe než SWT. Velkou roli hraje také můj subjektivní pocit (podložený zkušeností z praxe), že komponenty Swingu se dají přizpůsobit prakticky jakkoliv. Knihovny SWT takovou volnost nenabízejí. Pokud má být navíc zajištěna funkčnost na více platformách zároveň, vzniká těžko řešitelný problém.

Z výše zmíněných důvodů jsem nakonec jako základ pro svoji aplikaci zvolil platformu NetBeans.

Kapitola 4

Realizace

V této kapitole rozebírám charakteristiku zvolených technologií a jednotlivé moduly vzniklé aplikace. Zaměřuji se na stěžejní funkce aplikace a nestandardní či zajímavé části implementace.

4.1 Platforma NetBeans

4.1.1 Rich Client

V architektuře klient-server se termín „Rich Client“ používá pro klienty, když se data zpracovávají většinou na straně klienta. Klient také poskytuje grafické uživatelské prostředí (GUI). Většinou se jedná o aplikace, které se dají rozšířit pomocí zásuvných modulů.

Rich Client je obvykle postaven nad nějakým základem - základním rámcem (frameworkem). Tento základní rámec nabízí určitou infrastrukturu, na které může uživatel postavit aplikaci z logických částí, kterým se říká moduly.

Přehled vlastností Rich Client aplikace:

- flexibilní a modulární architektura aplikace
- nezávislost na platformě
- přizpůsobení uživateli
- možnost pracovat on-line i off-line
- jednoduchá distribuce k uživateli
- jednoduchá aktualizace klienta

Pro tuto diplomovou práci jsou stěžejní první tři body.

4.1.2 Rich Client platforma (RCP)

Rich Client platforma je program, který poskytuje prostředí pro životní cyklus aplikace a je základem desktopové aplikace. Mnoho desktopových aplikací má stejné nebo podobné vlastnosti. Jsou to například nabídky (menu), nástrojové lišty, stavový řádek, zobrazení průběhu zpracování, zobrazení dat, přizpůsobení nastavení, ukládání a načtení konfigurace a uživatelských dat, úvodní obrazovka (splash screen), okno O aplikaci (About box), nastavení národního prostředí (internacionalizace), systém nápovědy atd. Framework už všechny tyto funkce a komponenty obsahuje a není třeba je programovat.

Možnost konfigurace a rozšířitelnost aplikace jsou pak hlavním znakem takového frameworku. Výsledkem například je, že položky nabídky zapíšete deklarativně do textového souboru, odkud si je framework automaticky nahraje. To znamená, že se zdrojový kód soustředí do jednoho místa a vývojáři se mohou starat jen o skutečné problémy budoucí aplikace.

Nejdůležitější vlastností Rich Client platformy je její architektura. Aplikace postavené na takové platformě se vytvářejí ve formě modulů, ve kterých jsou izolovány jednotlivé logicky ucelené části aplikace. Modul je popsán deklarativně a platforma ho automaticky načte(3).

Rich Client platforma osvobodí vývojáře od úkolů, které nesouvisí s obchodní logikou aplikace. Z vlastní zkušenosti[10] mohu potvrdit, že vytvořit například editor kódu pouze ze standardních knihoven Javy je opravdu náročná práce, při které vývojář „znovu vynalézá kolo“.

Výhody Rich Client platformy:

- zkrácení doby vývoje
- konzistence uživatelského rozhraní
- aktualizace
- nezávislost na platformě
- možnost opakovaného použití a spolehlivost

4.1.3 Charakteristika platformy NetBeans

Platforma NetBeans nabízí, kromě obecných předností Rich Client platformy, množství frameworků, API (programových balíčků) a několik specifických vlastností, které mohou být vývojářům užitečné. Následuje výčet a popis vlastností, které jsou užitečné v kontextu této práce:

- **podpora uživatelského prostředí**

  K dispozici jsou okna, nabídky, nástrojové lišty a další komponenty. Programátor se věnuje jen vytvoření vlastních akcí, do kterých soustředí svůj

[10] Viz moje bakalářská práce *Konfigurovatelný generátor VHDL kódu*(8).

kód. Jak již bylo zmíněno dříve, NetBeans je založeno plně na knihovně Swing. V této aplikaci jsou hojně využívány například vylepšené uživatelské dialogy.

- **editor**

  Výkonný editor NetBeans, který je v NetBeans IDE[11], lze použít ve vlastní aplikaci. Nástroje a funkce editoru jsou snadno rozšiřitelné a přizpůsobitelné. Vyžil jsem především možnost přidat podporu obarvování syntaxe vlastního jazyka (VHDL).

- **podpora pro vytváření průvodců**

  NetBeans obsahuje jednoduché nástroje pro vytvoření rozšiřitelných a uživatelsky přívětivých průvodců.

- **datové systémy**

  Načítaná data mohou být lokální nebo přístupná přes FTP, CVS, z databáze nebo ve formě XML souboru. V NetBeans existuje abstrakce, která jedním modulem transparentně zpřístupní data ostatním modulům. Skutečný původ dat už není důležitý, protože se o něj postaralo API platformy. Díky tomuto API a tutoriálu(12) bylo jednoduché přidat do aplikace podporu rozeznávání a otevírání VHDL souborů.

4.1.4 Architektura platformy NetBeans

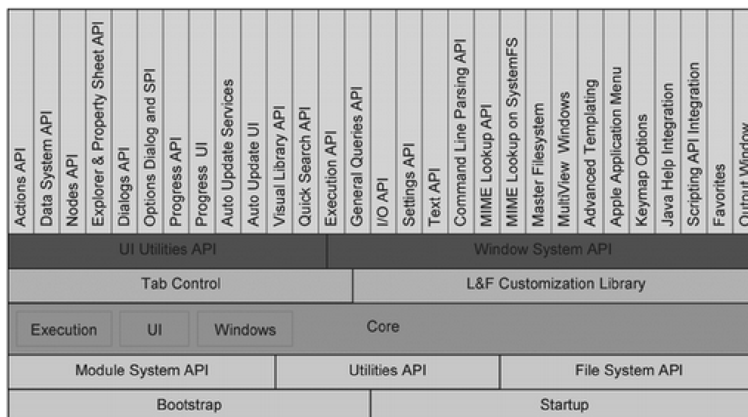
Základním stavebním kamenem platformy NetBeans je *modul*. Modul je sada funkčně svázaných tříd a jiných zdrojů, popisu rozhraní, které modul vystavuje, a také popis ostatních modulů, které modul potřebuje pro svoji funkci. Celá platforma NetBeans je rozdělená do modulů. Ty jsou načteny jádrem platformy, které je známo jako běhový kontejner NetBeans (*NetBeans Runtime Container*). Běhový kontejner načítá moduly aplikace dynamicky a automaticky, přičemž je zodpovědný za běh celé aplikace(3). Architektura NetBeans platformy je výstižně zachycena na obrázku 4.1.

4.1.5 Lookup

Návrhový vzor Lookup je stěžejním prostředkem pro vzájemnou komunikaci modulů aplikace postavené na platformě NetBeans. Když něco nejde vyřešit klasickým způsobem, většinou to jde pomocí Lookupu.

Lookup je v podstatě mapa, ve které je klíčem pro vyhledávání třída (`Class`) objektu a hodnota je instance objektu. Moduly poskytují prostřednictvím Lookupu objekty k vyhledávání. Výhodou je typová bezpečnost, protože se místo názvu třídy používá přímo třída (`Class`). Typem klíče je předem

[11] Vývojové prostředí Javy, ale i mnoha jiných jazyků.



Obrázek 4.1: Architektura NetBeans platformy (převzato z (3))

určen i typ získané instance. Zároveň není možné získat instanci typu, který modul nezná. Pomocí Lookupu se dá předávat i více instancí jednoho typu.

Pomocí Lookupu se dají předávat instance i z modulu do modulu, aniž by se navzájem znaly. Lookupů se v jedné aplikaci může vyskytovat několik. Platforma NetBeans poskytuje jeden základní, a to globální Lookup. Dále poskytují Lookup například třídy `TopComponent`, `DataObject` a další. Princip Lookupu se v platformě NetBeans využívá také k vyhledávání a poskytování služeb. Modul může využívat funkcionalitu služby, i když neví nic o její implementaci.

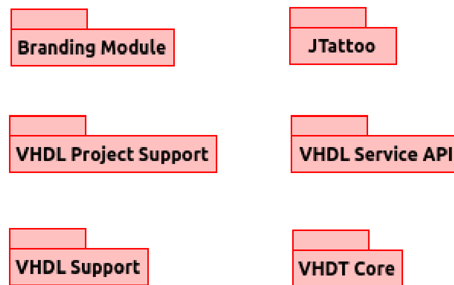
4.2 Struktura aplikace

Jak již bylo zmíněno, platforma NetBeans je modulární. Aplikace VHDT se tedy skládá z modulů. Schéma modulů popisuje obrázek 4.2. Jednotlivým modulům se věnují další podkapitoly práce.

Modul `VHDL Support` lze v kombinaci s modulem `VHDL Service API` (na nějž má modul `VHDL Support` definovanou závislost) použít jako samostatný plugin do NetBeans IDE. Díky tomuto pluginu NetBeans IDE dokáže rozeznávat VHDL soubory a zvýrazňovat jejich syntaxi v editoru kódu.

4.3 Branding Module

Branding module je pomocný modul, díky kterému se dá nastavovat celá platforma NetBeans. Dají se zde přemisťovat, přejmenovávat či promazávat akce z menu, toolbarů, kontextových nabídek, dialogů nastavení a podobně. Dále je zde možno nastavovat chování platformy. Vše je proveditelné skrze globální



Obrázek 4.2: Schéma modulů, ze kterých se skládá aplikace VHDT

xml soubor *layer*. Každý nový modul aplikace má svůj lokální *layer* soubor, který se do globálního nepromítá.

Pomocí tohoto modulu jsem z aplikace odebral nepotřebná menu, nepotřebné položky některých menu a nepotřebné položky toolbarů. Dále jsem customizoval okno „Favorites“. To bylo přejmenováno na „VHDL Templates“. Stejně tak byly přejmenovány všechny související akce, jako např. „Add to Favorites“ na „Add to VHDL Templates“.

4.4 JTattoo

Tento obalový modul obsahuje knihovnu třetí strany JTattoo(1). Knihovna JTattoo je oblíbeným prostředkem ke customizaci vzhledu[12] aplikací postavených na knihovnách Java Swing. Obsahuje množství vyladěných, méně či více oku lahodících schémat. Knihovna je k dispozici zdarma.

V aplikaci VHDT knihovnu používá modul `VHDT Core`. Ve třídě `Installer` pro nastavování defaultního či uživatelem dříve vybraného vzhledu a v balíčku `cz.ctu.fit.mateju.vhdt.core.actions.laf` v akcích pro nastavení vzhledu aplikace.

4.5 VHDL Service API

Modul obsahuje rozhraní služeb, podpůrné struktury a statické proměnné používané v ostatních modulech aplikace. Díky tomuto modulu nevznikají mezi moduly aplikace cyklické závislosti.

4.5.1 Služby

V balíčku `cz.ctu.fit.mateju.vhdt.vhdlserviceapi.services` se nachází rozhraní služeb užívaných ve více modulech aplikace. Jedná se o následující služby:

[12] LaF

- `ConsistencyCheckerService` - Poskytuje metody ke kontrole konzistence projektu.
- `EntityService` - Poskytuje akce na VHDL entitách.
- `InconsistencyService` - Definuje operace s objekty nekonzistencí.
- `VHDLProjectService` - Nejrozsáhlejší služba. Umožňuje provádět akce nad projektem.

Všechny služby jsou implementovány v modulu `VHDL Project Support` a budou podrobně probírány v kapitole věnující se tomuto modulu. Jejich implementace (jsou ji k dispozici) může získat kterákoliv třída libovolného modulu prostřednictvím globálního Lookupu[13].

4.5.2 Podpůrné struktury

Balíček `cz.ctu.fit.mateju.vhdt.vhdlserviceapi.structures` obsahuje datové struktury představující VHDL entity a nekonzistence.

Reprezentace VHDL entity

Struktura tříd reprezentujících VHDL entitu je znázorněna v diagramu tříd na obrázku 4.3. Třídy obsahují metody pro snadné tisknutí logu. Kvůli udržování některých objektů v množinách (např. v kolekci `HashMap`) byly přepsány metody `hashCode()`, `equals(Object o)` a `compareTo(<T> o)`. Implementovány jsou podle klasických Java zvyklostí tak, aby fungovaly na základě přirozené podobnosti/rovnosti objektů (zde VHDL entit).

Třída `Signal` (a od ní odvozené třídy `Generic` a `Port`) definují ještě metodu `getComparableType()`. Ta spojí celý typ signálu („vše za dvojtečkou“) v jeden řetězec, odstraní všechny bílé znaky a převede řetězec na malá písmena (*lower case*). Jak název metody napovídá, pomocí takto upraveného řetězce se dají porovnávat typu signálů, portů a generiků.

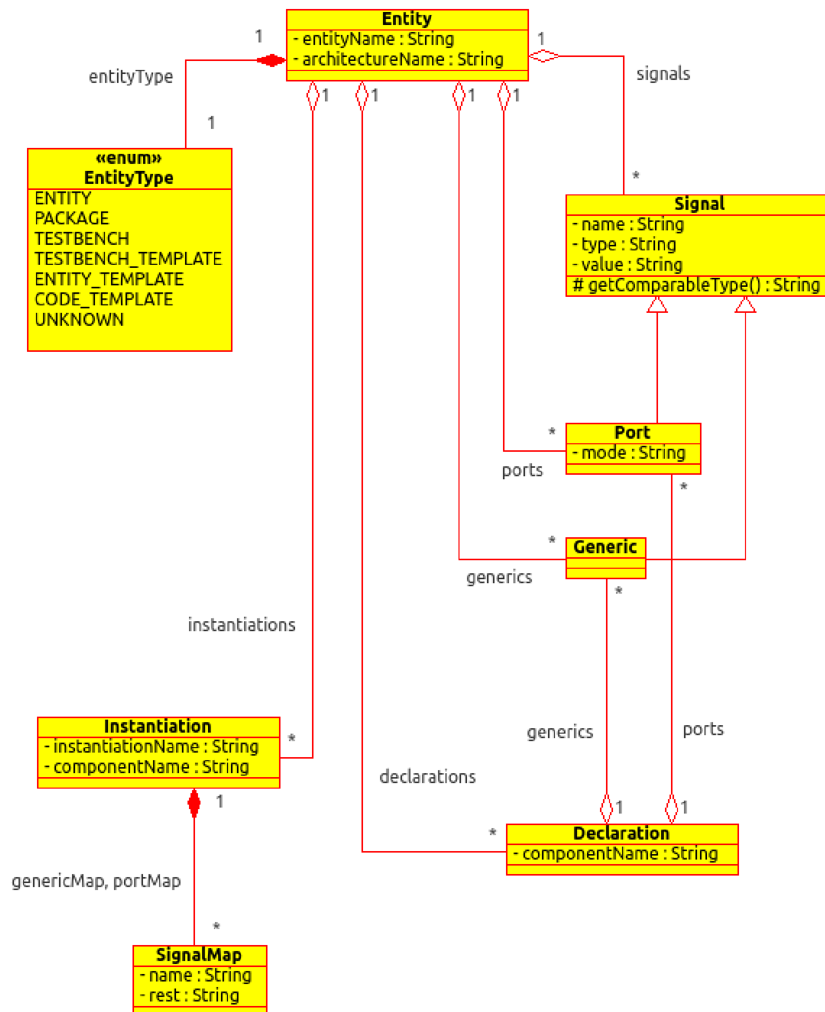
Třída `Instantiation` a třída `Signal` obsahují metody pro tisk své reprezentace jako VHDL kód. Toho je využíváno při opravě nekonzistencí a při vyplňování šablon, kdy se objekty instancí a signálů nechají pomocí těchto metod snadno vytisknout.

Reprezentace nekonzistence

Na základě provedené analýzy vznikly nekonzistence tří typů:

- **ED** (*Entity X Declaration*)

[13] Modul musí pochopitelně být závislý na modulu definujícím rozhraní služby, zde tedy `VHDL Service API`.



Obrázek 4.3: Třídní diagram struktur představujících entitu

- **EI** (*Entity X Inconsistency*)
- **ES** (*Entity X Signals*)

Každá nekonzistence obsahuje objekt entity, ke které náleží (tedy ve které byla objevena). Dále obsahuje `FileObject` dané entity (ten představuje přímo soubor s kódem), boolovskou hodnotu informující o tom, zda byla nekonzistence označena k opravení a „opravný balíček“ `FixPack`. `FixPack` je struktura uchovávající opravený kus kódu a pozici, kam má být vložen.

Třída `Position` se využívá při opravě nekonzistencí a při generování kódu ze šablon. Obsahuje pozici v souboru, odsazení dané pozice (kvůli dodatečným úpravám odsazení víceřádkové struktury) a případně značku (což je typ značky, jejíž pozici objekt `Position` označuje, je-li použit při práci se šablonami).

Strukturu tříd reprezentujících nekonzistence zobrazuje diagram 4.4.

4.6 VHDT Core

4.6.1 Installer

Balíček `cz.ctu.fit.mateju.vhdt.vhdtcore` obsahuje třídu `Installer`, která se stará o nastavení základního či uživatelem dříve zvoleného LaF po startu aplikace.

4.6.2 Akce

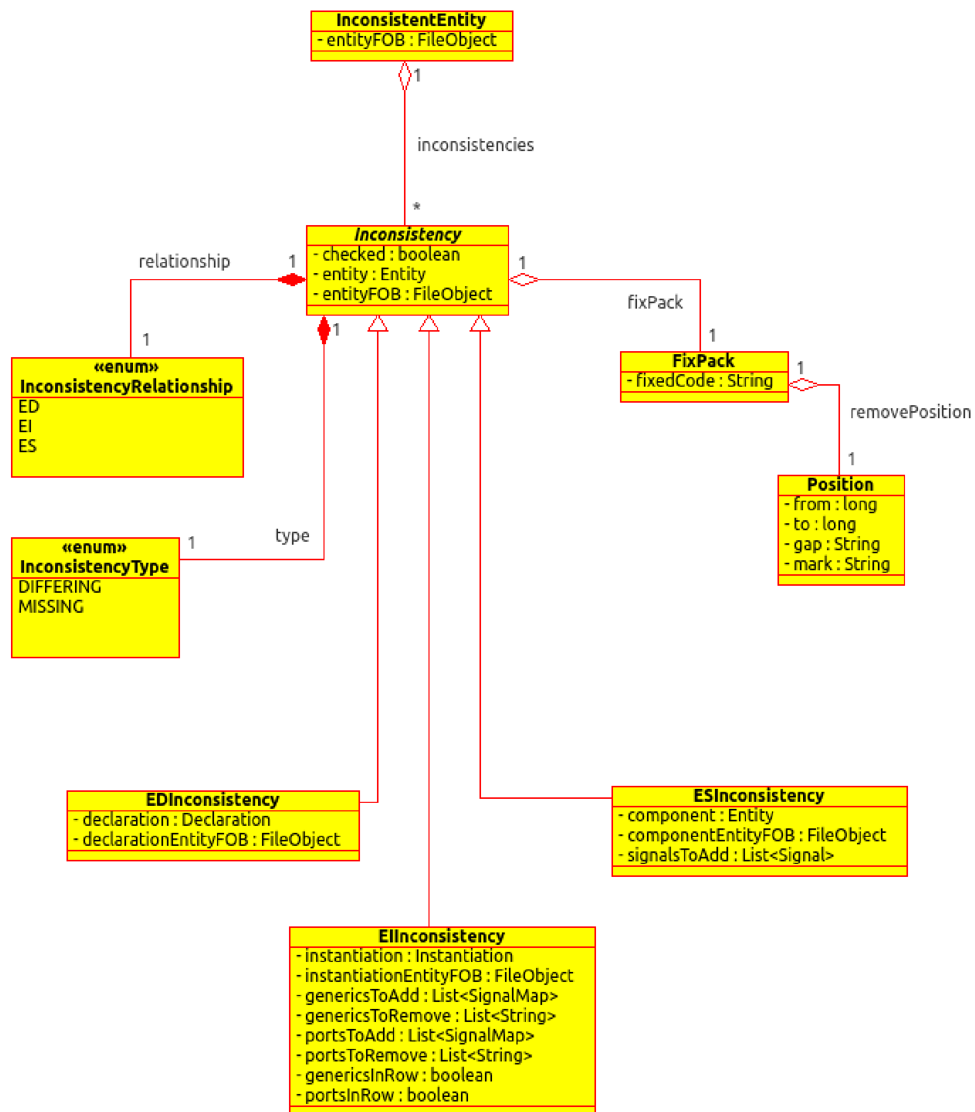
V balíčku `cz.ctu.fit.mateju.vhdt.vhdtcore.actions` se nacházejí tři druhy akcí:

- **akce pro změnu vzhledu aplikace**
- **akce asociované s entitami** - kontrola konzistence entity, generování testbenche z entity a vložení kusu generovaného kódu do entity
- **akce asociované s projektem** - kontrola konzistence celého projektu, nalezení entity v projektu, generování nové entity do projektu, refresh projektu, force refresh projektu a import existujícího projektu

Akce pomocí anotací udávají, v jakém kontextu mají být aktivní a ze kterých menu a kterými klávesovými zkratkami se dají vyvolávat. Po svém vyvolání delegují požadavky na služby, které ze svých metod volají.

4.7 VHDL Support

Modul `VHDL Support` se stará o to, aby aplikace uměla rozeznávat VHDL soubory a analyzovat jejich strukturu. Dále implementuje a integruje platformní



Obrázek 4.4: Třídní diagram struktur představujících nekonzistence

rozhraní `Lexer SPI`, díky němuž umí standardní NetBeans editor zvýrazňovat syntaxi VHDL kódu.

4.7.1 Rozpoznávání VHDL souborů

Vytvoření nového MIME typu[14] je v NetBeans díky přehlednému průvodci a tutoriálu(12) záležitostí pár kliků myší. Takto byl v aplikaci vytvořen nový MIME typ „*text/x-vhdl*“ reprezentující soubory s příponami „.vhd“, „.VHD“, „.vhdl“ a „.VHDL“. Díky tomu aplikace VHDT rozpozná VHDL soubory. Pomocí průvodce byla přidána i ikonka, díky které uživatel např. v dialogovém okně pro výběr souboru pozná, že jde o VHDL soubor.

Důležitou částí balíčku pro rozpoznávání kódu je třída `VHDLDataObject`, která umožňuje definovat chování a operace nad VHDL soubory. V aplikaci získala mimo jiné také následující důležité vlastnosti:

- `getFromProject(Class<?> clazz)`

  Datový objekt se pomocí Lookupu pokusí najít projekt, v němž se nachází. Povede-li se mu to, pokusí se v Lookupu projektu najít a vrátit hledanou třídu.

- `getStructure()`

  Vrátí objekt `Entity` reprezentující strukturu datového objektu. Pokud objekt ještě neexistuje, je vytvořen pomocí parseru. Pokud existuje, nevytváří se a je pouze vrácen. Toto muselo být ošetřeno kvůli refreshování projektu. Kdyby se každá entita znovu parserovala, velké projekty by se refreshovaly znatelně déle. Před refreshem se proto obnovují pouze struktury entit, v jejichž souborech došlo ke změnám[15].

- `propertyChange()` (třída je sama sobě posluchačem změn)

  Prostřednictvím této metody je při přejmenování souboru, při změně obsahu souboru (po uložení pomocí akce *Save*[16].) a při smazání soubor zavolána pomocí služby `VHDLProjectService` metoda `refreshProject()`.

4.7.2 Analyzátor VHDL kódu a obarvování syntaxe

Během implementace lexeru a parseru jsme čerpal informace z přehledného VHDL tutoriálu(14) a z hyperlinkované VHDL BNF syntaxe(4).

[14]Standard používaný k popisu obsahu souboru.

[15]Výjimku tvoří akce `forceRefresh()`, při které se znovu parserují všechny entity projektu.

[16]To vyvolá změnu hodnoty vlastnosti *modified*.

Lexer

Základem nové lexeru[17] se stal původní lexer z mé bakalářské práce(8). Prošel pouze několika málo úpravami, a to především kvůli možnosti implementovat platformní rozhraní `Lexer SPI`[18].

Hlavní logika je implementována v abstraktní třídě `AbstractVHDLLexer`, kde je odstíněna od závislosti na vstupním zdroji. Načítání znaků ze vstupu, indexování vstupu a další problémy s tím spojené řeší konkrétní implementace lexeru, třídy `VHDLSPILexer` a `VHDLLexer`. Lexer využívá flyweight[19] tokenů. První načtení do paměti trvá déle, neboť jsou vytvořeny objekty všech definovaných tokenů. Každý další přístup je již realizován jako přístup to tabulky tokenů (dle jména konkrétního tokenu). Vše tedy probíhá velmi rychle.

`VHDLSPILexer` implementuje platformní rozhraní `Lexer` a dědí od třídy `AbstractVHDLLexer`. Jako vstup používá speciální třídu platformy NetBeans, `LexerRestartInfo`. Díky integraci tohoto lexeru do výsledné aplikace může být v textovém editoru obarvována syntaxe VHDL kódu. Je třeba akorát nastavit vše potřebné pomocí souboru *layer.xml* a nadefinovat způsob zobrazení jednotlivých skupin tokenů v souboru *FontAndColors.xml*. Skupiny tokenů jsou definované ve třídě `VHDLSPILanguageHierarchy`. Pro více podrobností doporučuji projí tutoriál přidání podpory pro nový jazyk na stránkách NetBeans(12).

`VHDLLexer` také dědí od třídy `AbstractVHDLLexer`. Jako vstup používá klasický soubor (třídu `File`). O navigaci v textovém souboru se stará rychlá třída `FileChannel` z balíčku `java.nio`[20]. `VHDLLexer` přidává ještě jednu užitečnou metodu - `getPosition()`. Ta vrací pozici (počáteční a konečný znak) aktuálně načteného tokenu.

Parser

Parser[21] byl implementován od základu znovu. S parserem v aplikaci *VHDL SGen*(8) nemá společný ani kousek kódu. Zvolil jsem vyšší úroveň abstrakce, aby parserování probíhalo co nejjednodušeji a nejbezpečněji.

K získávání tokenů parser používá instanci třídy `VDHLLexer`. Tomu se předá vstupní soubor, ze kterého se bude číst. Základem parseru je abstraktní třída `AbstractVHDLParser`, která definuje následující metody:

- `closeStream()`

  Zavírá vstupní stream (deleguje na lexer).

- `reset()`

[17] Neboli lexikální analyzátor.
[18] Viz (11).
[19] Návrhový vzor Flyweight, neboli „muší váha“. Pro více informací viz (13).
[20] Speciální knihovna jazyka Java sloužící k rychlé práci se vstupními proudy. Viz (10).
[21] Neboli syntaktický analyzátor.

Nastaví pozici v souboru na začátek a vymaže aktuálně načtený řetězec a pozici.

- `nextToken()`

  Načte další token. Pomocí přepínačů `ignoreWhitespace` a `ignoreComments` umožňuje (jak již název napovídá) přeskakovat bílé znaky a/nebo komentáře. To je velmi důležitá volba například kvůli zachovávání komentářů u některých upravovaných částí kódu.

- `getString()`

  Delegováno lexeru. Vrací například jméno načteného identifikátoru či stringovou podobu načtené číselné hodnoty.

- `getPosition()`

  Opět delegováno lexeru. Vrací pozici aktuálně načteného tokenu (pozice počátečního a koncového znaku).

- `accept(int expected)`

  Pokud číselná hodnota aktuálně načteného tokenu odpovídá hodnotě parametru `expected`, načte další token a vrátí `true`.

- `isToken(int ordinal)`

  Vrací `true`, pokud číselná hodnota aktuálně načteného tokenu odpovídá hodnotě parametru `ordinal`.

- `isEOF()`

  Vrátí `true`, pokud jsme se při parserování dostali na konec souboru.

Od třídy `AbstractVHDLParser` dědí třída `VHDLParser`. Je to pravděpodobně nejdůležitější třída celé aplikace. Pomocí ní se dá z VHDL souboru získat celá řada nezbytných informací. Jednotlivé veřejně přístupné metody si probereme podrobněji:

- `getEntity()`

  Nejdůležitější a nejkomplexnější metoda. Stará se o vytvoření objektu `Entity` z parserovaného souboru. Načítá jméno entity, výčet generiků a portů, jméno architektury, seznam komponent, signálů a instancí. Na základě dat poskytnutých touto metodou staví aplikace VHDT stromovou hierarchii entit v projektu.

- `getEntityPosition()`

  Vrátí pozici těla entity parserovaného VHDL souboru počínaje začátkem tokenu `IDENT` (představující jméno entity) a konče koncem tokenu `END`.

- `getArchitecturePosition()`

  Vrátí pozici těla architektury počínaje koncem tokenu `IS` a konče začátkem tokenu `END`.

- `getDeclarationPosition(String decName)`

  Vrátí pozici těla deklarace zadaného jména (detaily viz programová dokumentace[22]).

- `getDeclarationEndPosition(String decName)`

  Vrátí pozici konce těla deklarace zadaného jména, nebo pozici začátku architektury. To má význam při vkládání nových signálů souvisejících s danou deklarací. Pokud deklarace v architektuře je, vloží se signály pod ní. Pokud tam není, vloží se signály na začátek těla architektury.

- `getInstantiationPosition(String instName, String decName)`

  Vrátí pozici těla instance zadaného jména a typu.

- `getMarkPositions(List<Position> positions)`

  Naplní zadaný seznam pozicemi a názvy všech „šablonových“ značek vyskytujících se ve vstupním souboru. Hledají se všechny „globální“ a „dolarové“ proměnné[23].

- `getGlobalPositions(List<Position> positions)`

  Naplní zadaný seznam pozicemi a názvy všech „globálních“ proměnných.

Většina zmíněných metod řeší zároveň také vzdálenost hledané struktury od začátku řádku (odsazení, v aplikaci nazýváno „GAP“).

4.8 VHDL Project Support

Modul `VHDL Project Support` je nejrozsáhlejším modulem aplikace. Obsahuje balíčky definující VHDL projekt, speciální uživatelské dialogy, továrny a datové struktury sloužící k vytvoření a zobrazení stromové hierarchie entit a také konkrétní implementace služeb definovaných v modulu `VHDL Service API`.

4.8.1 Definice projektu

Aby aplikace postavená na platformě NetBeans uměla pracovat s nově definovaným projektem, je třeba implementovat několik důležitých rozhraní. Těmi jsou:

[22] Javadoc přiložený na CD.

[23] Viz kapitola 3, sekce *Návrh šablon.*

- `Project`

  Nese informace o projektovém adresáři a akcích, které se dají s projektem provádět. Udává obsah projektového Lookupu (třídy, které mají být z Lookupu dostupné). Stará se také o ukládání a načítání projektových properties[24] a poskytuje informace o projektu (`ProjetInformation`).

- `ProjectLogicalView`

  Stará se o náhled na projekt. Definuje kořenový uzel projektu a akce z něho dostupné. Z této třídy se také spouští tvorba projektového stromu entit (viz vytváření stromové hierarchie dále).

- `ProjectFactory`

  Určuje, zda konkrétní adresář je či není projektovým adresářem. Umožňuje nahrávat a ukládat projekt.

Zájemce najde detailní návod v tutoriálech NetBeans(12) a také v dokumentaci k NetBeans API(11).

4.8.2 Zobrazení stromové struktury entit

Základním předpokladem je znalost vnitřní struktury každého VHDL souboru. O tu se stará třída `VHDLParser`[25]. Strom se staví od kořene směrem dolů. Základním uzlem je `ProjectNode`, kterému je v konstruktoru předán objekt typu `VHDLProjectChildFactory`.

`VHDLProjectChildFactory` pracuje s objektem typu `ProjectStructure`, který v sobě v podobě tabulek uchovává všechny VHDL soubory nalezené v projektu (jejich objektové reprezentace - objekty typu `VHDLDataObject`). Rozděluje je do tří kategorií:

- **balíčky** - VHDL soubory typu *package*, v projektové stromu zobrazeny modrou barvou, forma „*jmenoBalicku (jmenoSouboru)*“
- **testbenche** - v projektovém stromu zobrazeny zelenou barvou, forma „*jmenoEntity - jmenoArchitektury (jmenoSouboru)*“
- **entity** - černá barva, forma

  „*jmenoEntity - jmenoArchitektury (jmenoSouboru)*“

Rozdělení probíhá na základě typu každé entity (ten je získán z objektu `Entity`, který poskytuje třída `VHDLDataObject`[26]).

Po získání tabulek všech entit jsou mezi potomky projektového kořenu automaticky přidány všechny balíčky a testbenche (vždy to jsou top entity).

[24] Uživatelem definované vlastnosti projektu.

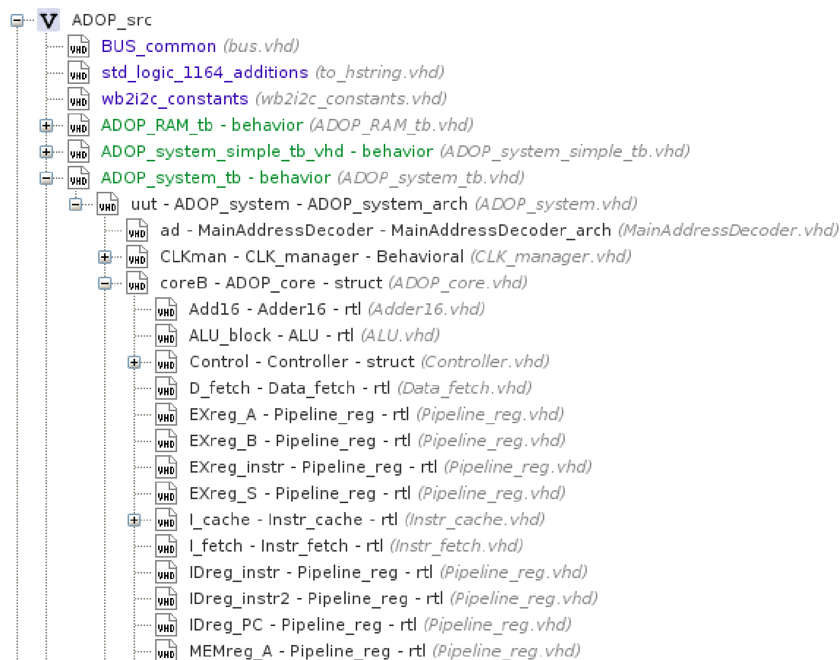
[25] Viz 4.7.2

[26] Viz metoda `getStructure()`.

Ostatní entity musí projít procesem, ve kterém se vyberou ty, které nejsou použity jako instance jiných entit či testbenchů. Pokud takové v projektu existují, jsou přidány do seznamu top entit. Je důležité zdůraznit, že entita nebude mezi top entitami **pouze** pokud se vyskytuje v architektuře jiné entity jako **instance**. Důvod je jasný - potomky každého uzlu (entity) projektu jsou instance v něm použité. Komponenty, které sice jsou v architektuře deklarované, ale nejsou od nich vyrobené žádné instance, nejsou zobrazovány jako potomci dané entity.

Každá top entita je v projektovém stromu reprezentována přímo jako uzel `VHDLDataNode`. Vnitřní uzly projektového stromu jsou vždy instance nějakých architektur. A těch může být více (dokonce i v rámci jediné architektury). Nemohou proto být reprezentovány přímo svým `DataNode`m. Místo toho je pro ně vytvořen zástupný objekt typu `VHDLEntityShadow`, který je v projektovém stromu zastupován uzlem `VHDLInstantiationDataNode`. Název instančního uzlu je zobrazován ve formátu „*jmenoInstance - jmenoOrigEntity - jmenoArchitektury (jmenoSouboru)*“. O vytváření potomků uzlů představujících entity (tedy o vytváření instančních uzlů) se stará třída `VHDLEntityChildFactory`.

Na obrázku 4.5 je příklad stromové struktury projektu zobrazené v aplikaci VHDT.



Obrázek 4.5: Ukázka stromové struktury projektu zobrazené pomocí nástroje VHDT

4.8.3 Služby

`ConsistencyCheckerServiceImpl`

Obsahuje jedinou veřejnou metodu `checkConsistency(Project project, String entityName)`. Tato metoda získá z projektu seznam všech entity (objekt `ProjectStructure`). V nich hledá takové entity a testbenche, v jejichž architekturách se objevuje kontrolovaná entita jako komponenta či instance.

Výstupem metody je seznam nekonzistentních entit (`InconsistentEntity`). Nekonzistence zpočátku nebyly vkládány do objektů `InconsistentEntity`. Potom ale vznikala těžko řešitelná situace se zapisováním změn do souborů. K jednomu VHDL souboru se může vázat několik nekonzistencí. Opravovány (případně uživatelem modifikovány) jsou zvlášť, nezávisle na sobě (protože předem není známo, které uživatel označí k opravě a které nikoliv). Proto bylo třeba seskupit nekonzistence patřící ke stejnému souboru dohromady. Ve třídě `InconsistencyServiceImpl` jsou nekonzistence určené k opravě seřazeny sestupně podle pozic (v souboru), na které jsou zvolené úpravy vkládány. Vkládání probíhá od konce souboru, díky čemuž není třeba přepočítávat pozice oprav, které ještě nebyly provedeny.

`InconsistencyServiceImpl`

Stará se o opravy nekonzistencí. Pro potřeby dialogu přehledu nekonzistencí (viz sekce 4.8.4) má metodu `fixInconsistency(Inconsistency inc)`, která rozpozná typ vložené nekonzistence a připraví její opravu prostřednictvím třídy `VHDLCodeBuilder` (v podobě objektu třídy `FixPack`).

Po stiknutí tlačítka *Ok* v dialogu přehledu nekonzistencí je zavolána metoda `fixAndStoreInconsistencies(List<InconsistentEntity> incs)`, jež vybere ze seznamu nekonzistencí ty, jež mají být opraveny. Poté opraví ty znich, které ještě nebyly opraveny v rámci volání metody `fixInconsistency()` a nakonec provede samotný zápis změn do souboru.

Před zápisem změn je nutno nekonzistence v rámci jednoho souboru ještě seřadit (viz předchozí text).

`VHDLCodeBuilder`

Singleton[27], jehož prostřednictvím ostatní služby provádějí opravy nekonzistencí, vyplňování šablon a vytváření testbenchů.

Metody sloužící k opravě nekonzistencí vrací vždy objekt typu `FixPack` obsahující přidaný/opravený fragment kódu a pozici, na kterou má být vložen. Metody sloužící k vyplňování šablon a generování testbenchů vracejí celý řetězec reprezentující opravenou entitu. Metody jsou dobře zdokumentovány, nebudu zde tedy uvádět jejich vyčerpávající výčet.

[27] Česky „Jedináček“ - instance tohoto objektu se v aplikaci vyskytuje vždy pouze jednou. To samé platí i pro všechny služby definované pomocí mechanismů platformy NetBeans.

`VHDLProjectServiceImpl`

Služba poskytuje metody úzce související s VHDL projektem. Každý projekt obsahuje vlastní instanci této služby, takže nemusí být každé metodě předáván projekt, na kterém se má daná operace provést. Metody jsou volány především z globálních akcí a akcí na projektovém uzlu. Operace, které by mohly trvat delší dobu, jsou spouštěny ve vlastním vlákně, aby neblokovaly grafické prostředí. O tom, že aplikace provádí nějakou akci, je uživatel informován ukazatelem průběhu ve stavové liště.

- `refreshProject()`

  Obnovuje projektový strom a seznam všech VHDL souborů projektu. U entit, kterým byla úmyslně vymazána naparserovaná struktura (objekt `Entity`), se během refreshe opět provede jejich analýza.

- `forceRefreshProject()`

  Stejné jako refresh, akorát se všem VHDL souborům před refreshem vymaže jejich naparserovaná struktura. To se hodí především, pokud byl některý soubor změněn v nějaké externím editoru. Aplikace to totiž detekuje až při pokusu do souboru zapisovat nebo z něj číst.

- `findAllVHDLFiles()`

  Využívá se při načítání projektové struktury. Vyhledá všechny VHDL soubory, které jsou obsaženy v projektovém adresáři a jeho podsložkách.

- `findAndOpenEntity(String entityName)`

  Vyhledává v projektu entitu podle jejího názvu. Pokud entitu nalezne, otevře jí v editoru kódu.

- `checkConsistency(String entityName)`

  Kontrola konzistence vybrané entity v rámci celého projektu a oprava nalezených nekonzistencí. Skrze službu `ConsistencyCheckerService` nalezne všechny nekonzistence, které následně zobrazí v okně správy nekonzistencí.

- `generateNewEntity()`

  Zobrazí dialog pro výběr šablony entity. Pokud uživatel nějakou vybere a vyplní, je pomocí služby `VHDLCodeBuilder` vygenerována a uložena v projektu jako nová entity.

`EntityServiceImpl`

Obsahuje metody, které nejsou přímo závislé na projektu, v němž se entita vyskytuje. Stejně jako u předchozí služby jsou metody volány pomocí globálních či kontextově závislých akcí definovaných v modulu `VHDTCore`.

- `generateTestbench(DataObject entity)`

  Metoda zobrazí dialog sloužící k výběru šablony a poté pomocí služby `VHDLCodeBuilder` vygeneruje nový testbench, který uloží do složky s testovanou entitou.

- `insertCode(int position, Document document)`

  Stejně jako předchozí metoda i tato zobrazí dialog pro výběr šablony. Pokud uživatel nějakou šablonu vybere a zvolí *Generate*, vygeneruje se na základě šablony pomocí služby `VHDLCodeBuilder` nový fragment kódu, který je následně vložen do kódu stávajícího. Kód se objeví v editoru na vybrané pozici. Uživatel má poté možnost změny v editoru uložit, nebo vše vrátit do původního stavu tlačítkem *Undo*.

4.8.4 Dialogová okna

Tato okna nejsou v aktuální podobě zmíněna v původním návrhu UI. Vznikla (nebo byla upravena) na základě připomínek a požadavků testujících uživatelů.

Okno správy nekonzistencí

Okno pro správu nekonzistencí přehledně zobrazuje nalezené nekonzistence s možností jejich výběru. Je členěno na tři hlavní části:

- Tabulka nalezených nekonzistencí s možností zvolit ty, které mají být opraveny.
- Náhled na kód upravované entity **před** opravou. Je zvýrazněna pozice nebo úsek, kam bude vložen generovaný kód.
- Náhled na fragment vkládaného kódu s možností dodatečné ruční úpravy.

Okno správy nekonzistencí je ukázáno na obrázku B.12.

Okno pro výběr šablony

Okno pro výběr šablony se vzdáleně podobá klasickému oknu pro výběr souboru. Odlišují ho čtyři skutečnosti:

1. Neposkytuje možnost vybírat ze všech složek a disků v počítači, nýbrž pouze ze souborů a složek přidaných v aplikaci mezi *VHDL Templates*.
2. Filtruje vybírané šablony a zobrazuje pouze ty, které odpovídají aktuálně prováděné akci. Například při generování testbenche tedy nabízí pouze šablony testbenchů.

3. Zobrazuje náhledy šablon.
4. Obsahuje panel pro vyplnění globálních proměnných (pokud se nejedná o šablonu testbenche).

Okno je ukázáno na obrázku B.13.

Kapitola 5

Testování

5.1 Testování funkčních jednotek

Testování funkčních jednotek[28] jsem aplikoval převážně na třídy analyzující kód. Vytvořil jsem sadu poloautomatických testů. Testovalo se například, zda je lexikální analyzátor schopen analyzovat jakýkoliv VHDL kód, či zda se v kódu nevyskytují lexikální symboly, které ještě nezná.

Podobným způsobem se testoval syntaktický analyzátor. Zkoušely se různé druhy generiků, portů, signálů, typů a konstant. Byly testovány neúplné a chybné vstupy a následně se zkoumaly reakce analyzátoru. Bylo vytvořeno několik testovacích entit, jejichž naparserované struktury jsou poté porovnávány se skutečností. V testovacích entitách jsem se snažit nasimulovat co nejvíce nestandardních a jinak ošemetných situací. Postupně byly odhalovány a přidávány další situace, které by mě dříve vůbec nenapadly. Nyní jsou testy již poměrně rozsáhlé a dokáží případného upravovatele syntaktického analyzátoru včas upozornit na mnoho záludností.

Jednotkové testy jsou napsány také pro službu kontroly konzistence objektu. Byly vytvořeny modelové situace nekonzistencí, na jejichž základě musí služba nalézt všechny očekávané nekonzistence.

Tímto způsobem testování byly odhaleny chyby, které by se zpětně dohledávaly jen velmi obtížně. Zcela určitě se vyplatí investovat čas do psaní unit testů. Pro parser a lexer je dle mého názoru psaní testů naprosto nezbytné.

5.2 Testování na uživatelích

Uživatelské testování aplikace probíhalo ze dvou důvodů:

1. **Ověření funkčnosti aplikace.** V softwarovém inženýrství se tento způsob testování nazývá *Black-box testing.* Aplikace je testována jako celek.

[28] Tzv. *White-box testing.* V Javě se dá s výhodou využít JUnit testů.

Kontroluje se, zda z konkrétních akcí a vstupů dostáváme očekávané výsledky. Zda se aplikace chová tak, jak je očekáváno. U aplikace VHDT probíhalo testování pouze manuálně. Nebyly použity žádné automatické testy. Jednotlivé prototypy aplikace byly testovány po každé iteraci.

2. **Zjišťování, zda je aplikace uživatelsky přívětivá.** U aplikace typu editor kódu je velmi důležité, aby ovládání bylo pochopitelné, intuitivní a rychlé. Uživatel, který bude aplikaci používat, si bude chtít usnadnit práci. Nestojí o složitý neohrabaný nástroj. Toto testování mi tedy přišlo nejdůležitější. Povedlo se získat dva testery z řad kolegů, kteří zkoušeli navržená prostředí a poskytovali cennou zpětnou vazbu. Na základě těchto testů bylo grafické prostředí několikrát méně či více předěláno.

Pomocí těchto způsobů testování bylo v průběhu vývoje aplikace odhaleno a následně opraveno velké množství chyb a nedostatků. Zároveň se povedlo navigaci mezi komponentami GUI dostat do přehledného a uživatelsky přívětivého stavu.

Uživatelské testy probíhaly na několika platformách. Testovalo se především v OS *Windows 7, Ubuntu 10.10* a *Kubuntu 12.04*. Aplikace byla od začátku vyvíjena tak, aby na žádném ze systémů nenastal problém[29]. Aplikace na všech systémech běžela dle očekávání. Projevil se pouze problém s kódováním zdrojových souborů. Jeden z testovacích projektů byl ve zpětně neurčitelném editoru uložen v kódování, ve kterém měla aplikace VHDT problém s českými znaky. V prostředí Linuxu aplikace tuto skutečnost uživateli oznámí a české znaky zobrazí jako „rozypaný čaj“. Ve Windows 7 ale aplikace problém nezahlásí a některé neznámé znaky zobrazí jako zalomení řádku, čímž vzniká problém při parserování. V rozumném čase se mi tento problém nepodařilo odstranit.

[29]Typické jsou například problémy se souborovými cestami.

Závěr

V rámci této diplomové práce vznikla aplikace *Vývojový nástroj pro generování VHDL kódu a správu projektů (VHDT).* Nástroj usnadňuje vývojářům hardwaru práci při běžných a často se opakujících činnostech, jakými jsou vytváření základních entit (automaty, čítače a pod.), generování testbenchů a udržování konzistence rozhraní mezi jednotlivými architekturami. V rámci práce byla provedena rešerše stávajících řešení, na jejímž základě jsem došel k závěru, že volně dostupný nástroj požadovaných vlastností v současné době na trhu chybí.

Aplikace umožňuje vytvářet a spravovat rozsáhlé VHDL projekty. Ty jsou zobrazovány ve stromové hierarchii vycházející ze závislostí mezi jednotlivými architekturami. K dispozici je pokročilý editor VHDL kódu s podporou obarvování syntaxe. V rámci práce vznikl také poměrně pokročilý analyzátor VHDL kódu.

Nástroj byl během svého vývoje podroben uživatelskému testování. Na jeho základě byly průběžně prováděny menší či větší změny oproti původnímu návrhu. Ty se projevily především v návrhu UI.

V příloze práce se nachází několik ukázkových šablon. Dále zde čtenář nalezne příručku programátora, která obsahuje návod na otevření, úpravu, sestavení a spuštění aplikace. Po přečtení této práce (zvláště kapitoly *Realizace*) a příručky programátora by případný pokračovatel neměl mít problém navázat v práci na tomto projektu.

Prostor pro rozšíření aplikace vidím například v realizaci a začlení parseru, který implementuje třídy z platformní knihovny `Parsing API`. Díky takovému parseru by se radikálně rozrostly možnosti usnadnění, které nabízí dosavadní editor. Jde například o zvýrazňování syntaktických chyb, nabídka automatického doplnění klíčových slov během psaní a pod. Potenciál k rozšíření mají také šablony, ve kterých může vzniknout množství nových příkazů podporujících často opakované činnosti.

Literatura

(1) JTattoo.
http://www.jtattoo.net/.

(2) Wikimedia Commons.
http://commons.wikimedia.org/wiki/Main_Page.

(3) Böck, H.: *Platforma NetBeans - Podrobný průvodce programátora*. Computer Press, 2010.

(4) Gerhard Petrowitsch: Hyperlinked VHDL-93 BNF Syntax.
http://tams-www.informatik.uni-hamburg.de/.

(5) Herout, P.: *Java - grafické uživatelské prostředí a čeština*. Kopp, 2007.

(6) Herout, P.: *Učebnice jazyka Java*. Kopp, 2008.

(7) J. Pinker, M. Poupa: *Číslicové systémy a jazyk VHDL*. BEN, 2006.

(8) Matějů, J.: *Konfigurovatelný generátor základních struktur VHDL kódu*. ČVUT v Praze, Fakulta elektrotechnická, 2010.

(9) Michael McNamara: Verilog Mode for Emacs.
http://www.verilog.com/verilog-mode.html.

(10) Oracle: Dokumentace k Java 6 SE API.
http://docs.oracle.com/javase/6/docs/api/.

(11) Oracle: Dokumentace k NetBeans API.
http://bits.netbeans.org/dev/javadoc/.

(12) Oracle: NetBeans Platform Learning Trail.
http://netbeans.org/features/platform/all-docs.html.

(13) Pecinovský, R.: *Návrhové vzory*. Computer Press, 2007.

(14) Wolfram Glauert: VHDL Tutorial.
http://www.vhdl-online.de/tutorial/.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

AWT Abstract Windowing Toolkit

GUI Graphical User Interface

I/O Input/Output

JFC Java Foundation Classes

JIT Just In Time

JRE Java Runtime Environment

JVM Java Virtual Machine

LaF Look and Feel

MIME Multi-purpose Internet Mail Extensions

OS Operační Systém

RCP Rich Client Platform

SWT Standard Widget Toolkit

UI User Interface

VHDL VHSIC Hardware Description Language

VHDL SGen VHDL Structure Generator

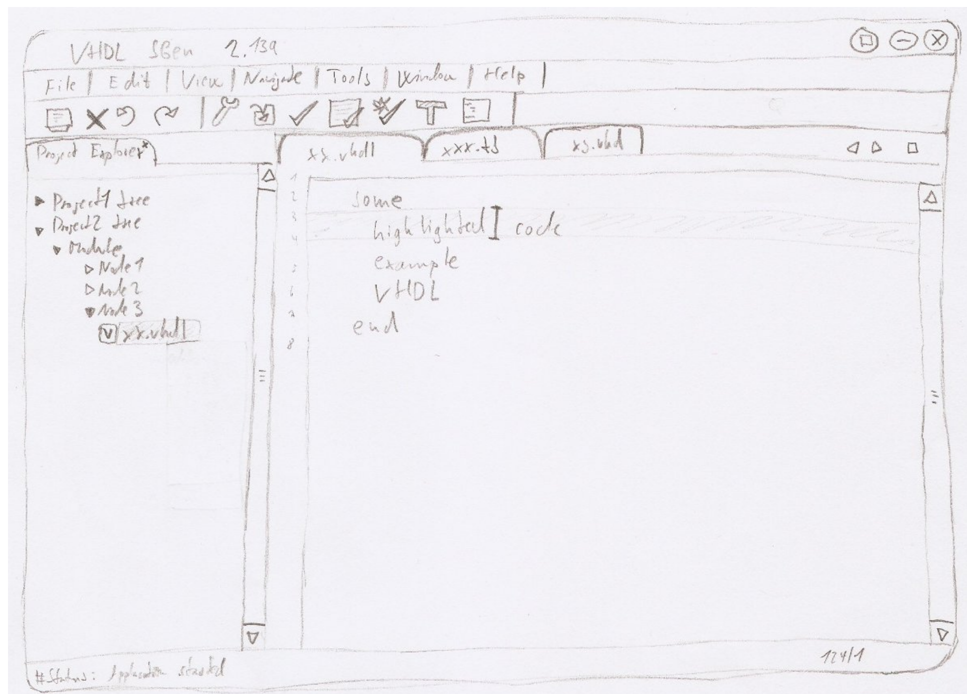
VHDT VHDL Design Tool

VHSIC Very High Speed Integrated Circuits

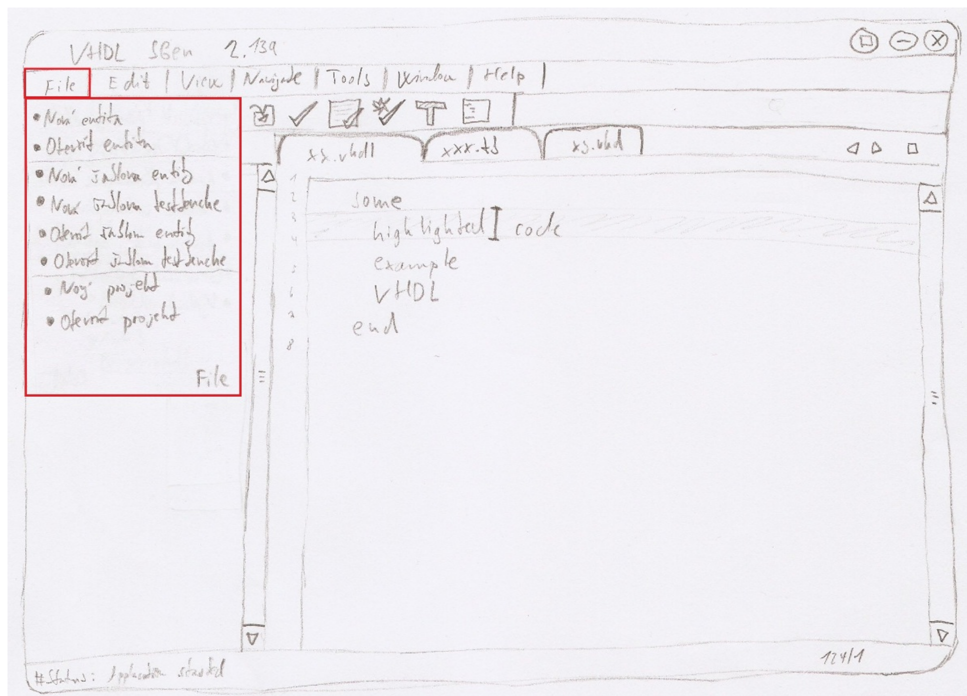
XML Extensible Markup Language

Příloha B

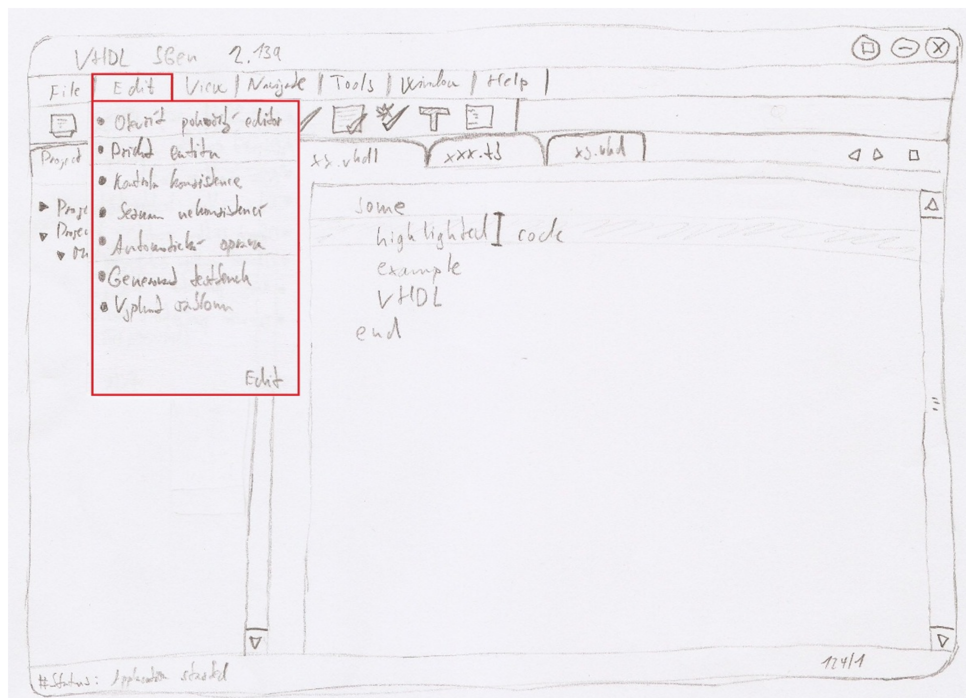
Obrázky



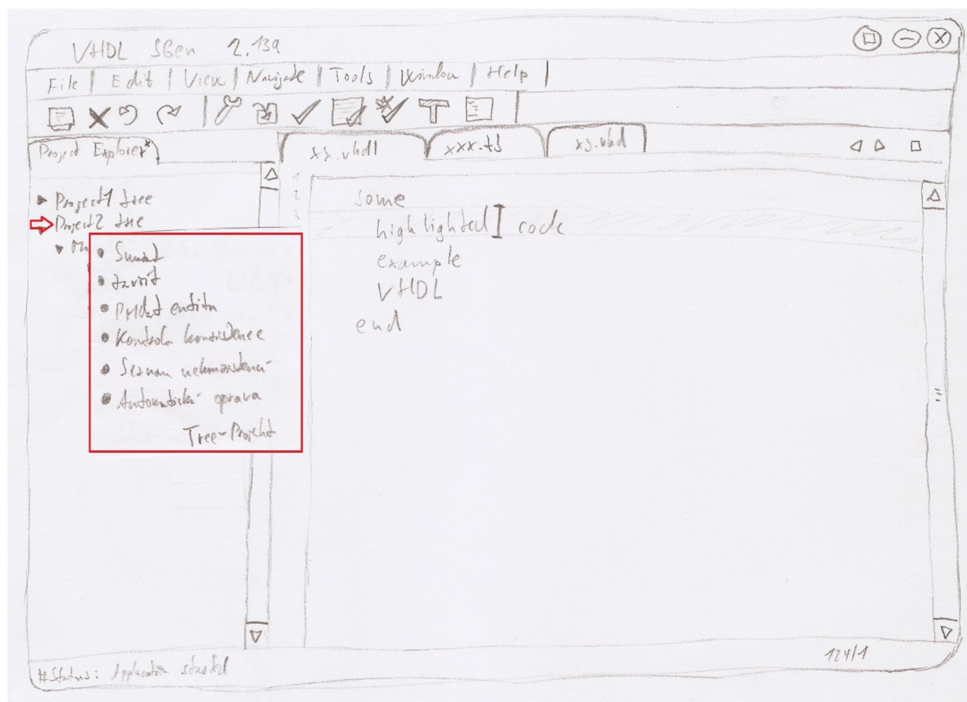
Obrázek B.1: Lo-Fi - Hlavní okno editoru



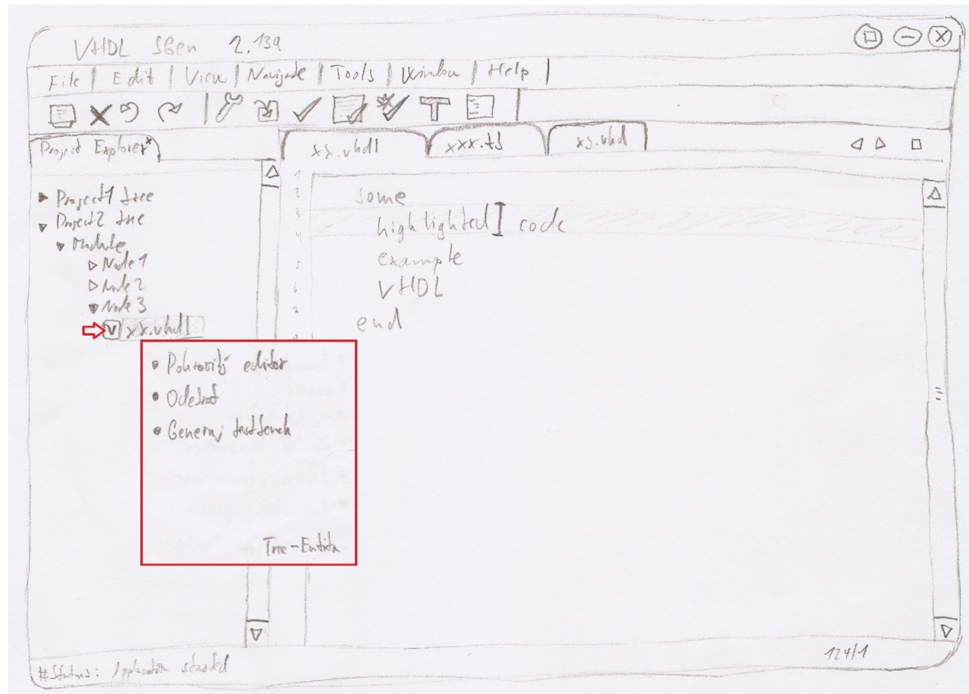
Obrázek B.2: Lo-Fi - Menu File



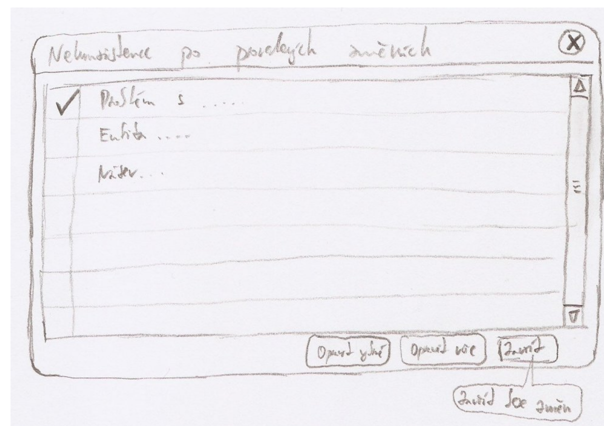
Obrázek B.3: Lo-Fi - Menu Edit



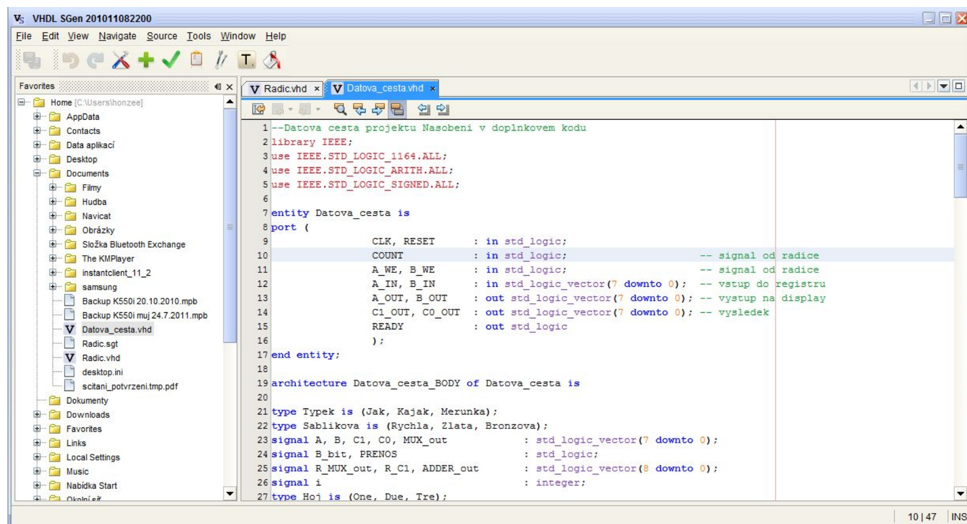
Obrázek B.4: Lo-Fi - Kontextové menu na projektovém uzlu



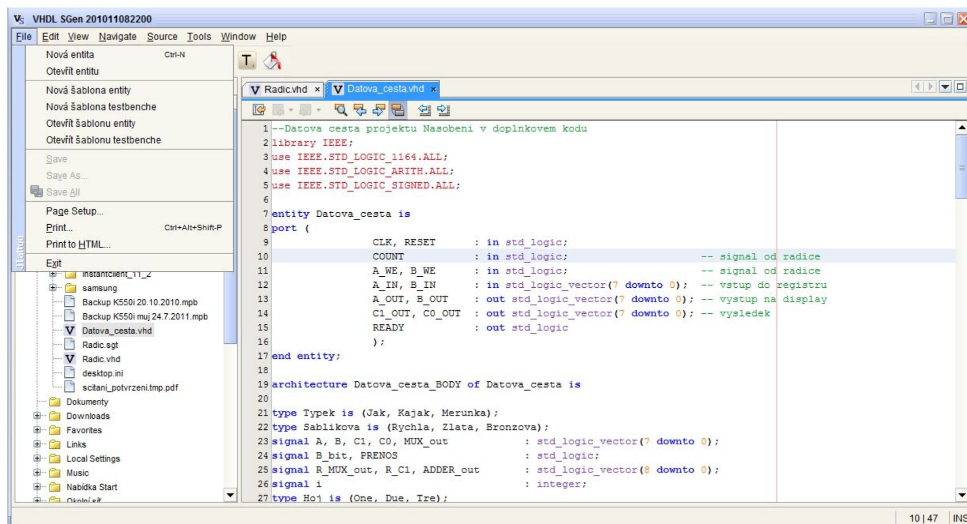
Obrázek B.5: Lo-Fi - Kontextové menu na uzlu VHDL souboru



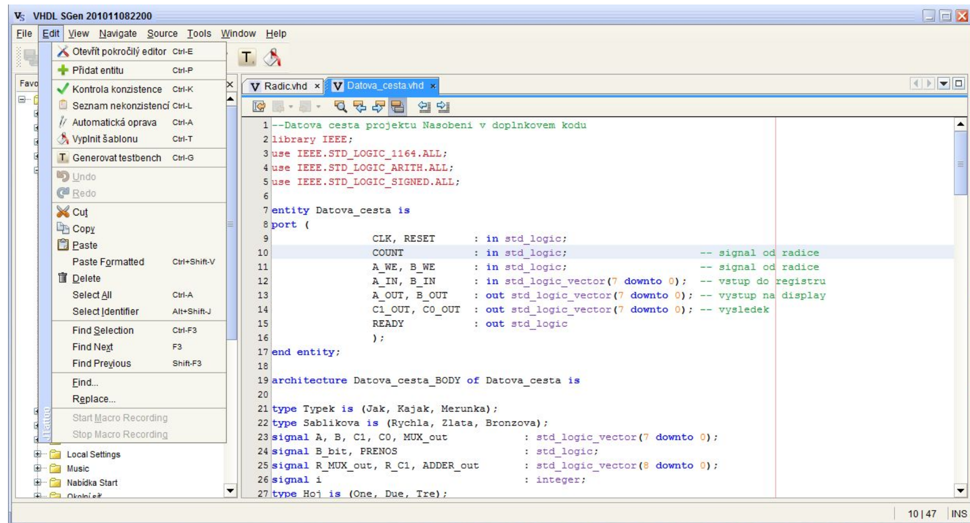
Obrázek B.6: Lo-Fi - Dialogové okno pro správu nekonzistencí



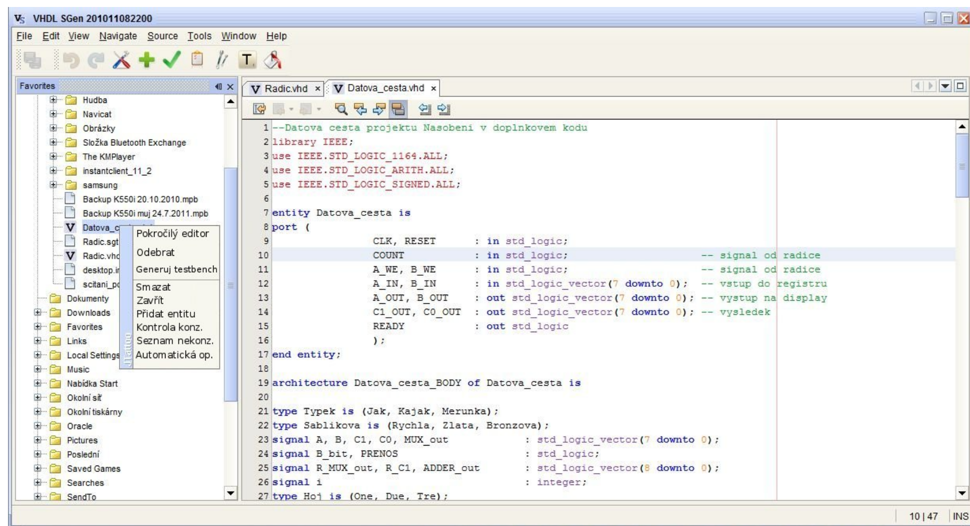
Obrázek B.7: Hi-Fi - Hlavní okno programu



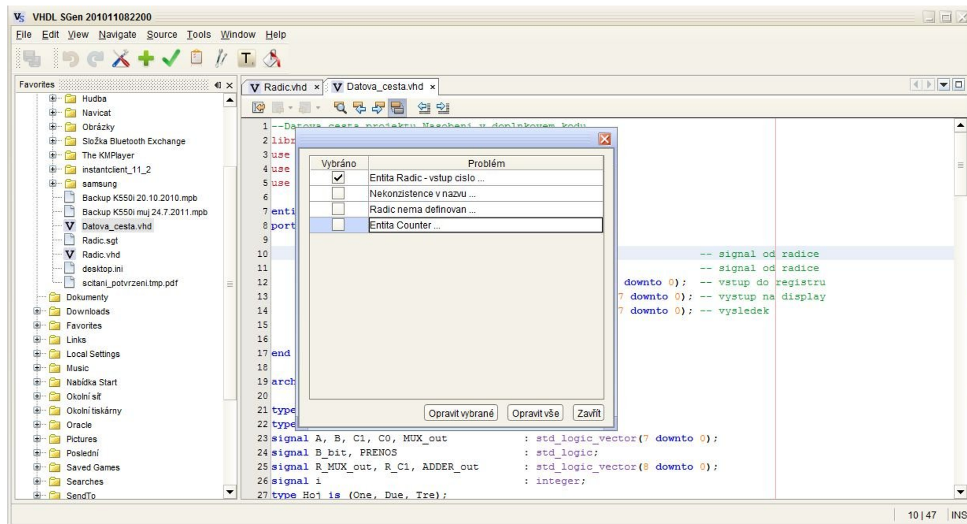
Obrázek B.8: Hi-Fi - Menu Soubor



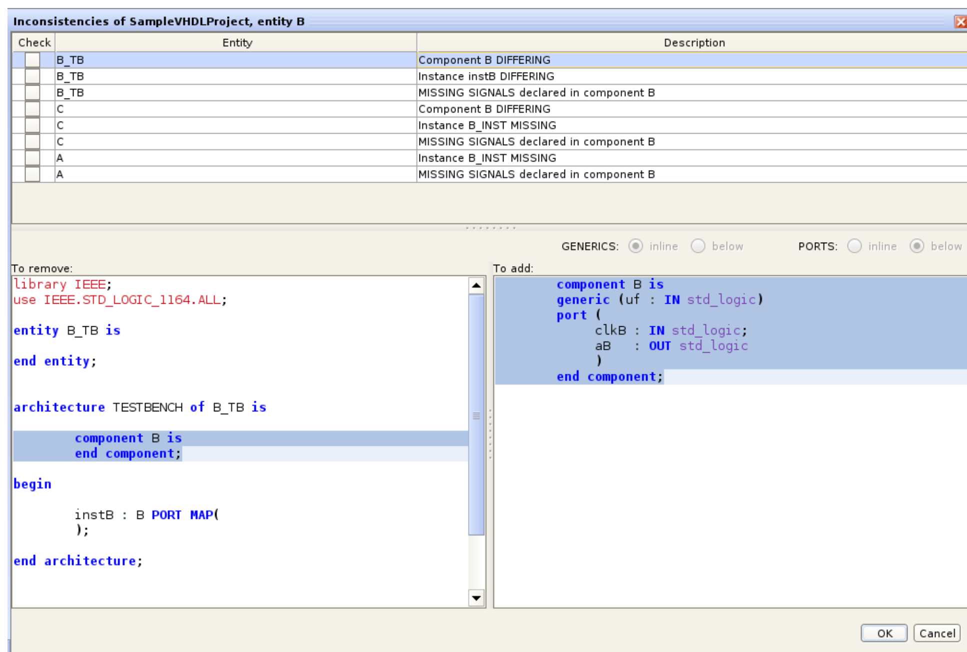
Obrázek B.9: Hi-Fi - Menu Edit



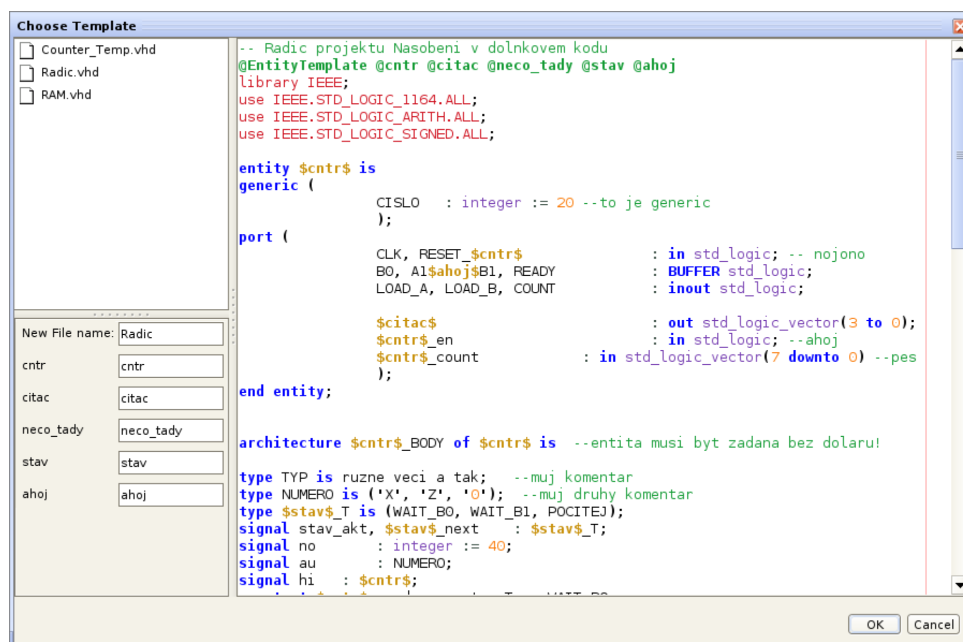
Obrázek B.10: Hi-Fi - Kontextové menu na projektovém uzlu



Obrázek B.11: Hi-Fi - Dialogové okno pro správu nekonzistencí



Obrázek B.12: VHDT - Dialog nekonzistencí



Obrázek B.13: VHDT - Dialog pro výběr šablony

Příloha C

Instalační a uživatelská příručka

C.1 Požadavky

Ke spuštění aplikace je nutné mít v počítači nainstalované běhové prostředí Java. Doporučena je verze Oracle JRE 6 nebo 7.

C.2 Instalace a spouštění

Soubor *vhdt.zip* si rozbalte do libovolné složky v počítači. V adresáři *bin* se nachází soubory *vhdt.exe* a *vhdt*.

V systému Windows stačí kliknutím na soubor *vhdt.exe* spustit aplikaci. Doporučuji zkopírovat zástupce spouštěče například na plochu či do systémové lišty.

V linuxových systémech se aplikace spouští pomocí skriptu *vhdl*. Skript je třeba před spuštění označit jako spustitelný. To se dá provést například příkazem `chmod +x vhdt`. Po označení souboru jako spustitelný se dá aplikace spustit příkazem `sh vhdt`.

C.3 Popis práce s aplikací

C.3.1 Práce s projektem

Pomocí menu či toolbaru *File* lze vyvářet nové projekty (***New Project***) a importovat (***Import Project***) či otevírat (***Open Project***) existující projekty. Otevřít lze projekty, které byli vytvořeny nebo importovány do aplikace VHDT. Pokud byl projekt vytvořen mimo aplikaci, je třeba ho importovat pomocí akce ***Import Project***. Jednoduše vyberte složku, v jejímž adresářovém podstromu se projekt nachází.

Při vytvoření či importu projektu bude v jeho kořenovém adresáři vytvořena složka ***vhdtproject***, pomocí které aplikace později indikuje, že se jedná o VHDL projekt. Tato složka může být vytvořena i manuálně mimo aplikaci.

Je-li vybrán projektový uzel, pak lze v menu *Edit* či v nabídce vyvolané pravým tlačítkem myši na projektovém uzlu (dále *akce na uzlu*) projekt zavřít (***Close***) či úplně vymazat z disku (***Delete***).

Pomocí menu *File/New File/Other* či skrze akci na uzlu *New/Other* lze do projektu přidat novou entitu.

Akce *Project/**Refresh*** a *Project/**Force Refresh*** (dostupné i z toolbaru) slouží k obnovení náhledu struktury projektu. *Refresh* je volán automaticky po každém uložení souboru. *Force Refresh* se hodí především v případech, kdy je projekt měněn mimo aplikaci VHDT. V takovém případě nemusí na obnovení struktury projektu stačit obyčejný *Refresh*. Pozor - u velkých projektů může *Force Refresh* trvat i několik vteřin, neboť jsou znovu analyzovány všechny VHDL soubory v projektu.

Akce *Project/**Find Entity*** zobrazí textové pole, do kterého lze zadat název hledané entity. Aplikace se poté pokusí entitu vyhledat a otevřít v editoru. Vyhledávání není citlivé na velká a malá písmena.

Všechny akce jsou dostupné také pomocí klávesových zkratek. Ty si lze prohlížet či upravovat v dialogu nastavení (menu *Tools/Options/KeyMap*).

Akce ***Generate New Entity***, ***Insert Code***, ***Generate Testbench*** a ***Check Consistency*** jsou rozebrány v následujících sekcích manuálu.

C.3.2 Kontrola konzistence

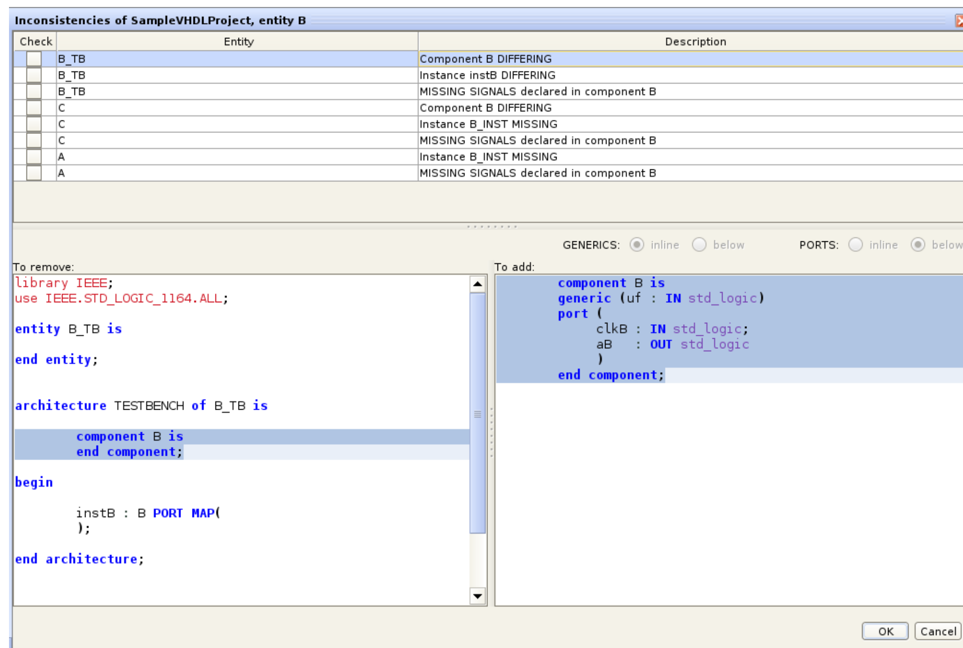
Kontrola konzistence lze vyvolat z menu či toolbaru *Entity/**Check Consistency*** (nebo také prostřednictvím akce na uzlu konkrétní entity). Při vyvolávání akce je třeba mít focus na uzlu konkrétní entity nebo na záložce konkrétní entity v editoru kódu. To aby aplikace mohla určit, pro kterou entitu se bude kontrolovat její konzistence.

Kontroluje se konzistence vybrané entity v rámci celého projektu. Kontrola se provede ve všech entitách, v jejichž architekturách je kontrolovaná entita použita jako komponenta nebo instance.

Nalezené nekonzistence se zobrazí v ***dialogu nekonzistencí*** (viz obrázek C.1). Dialog se skládá ze tří částí:

- **seznam nalezených nekonzistencí** - V seznamu lze zvolit, které nekonzistence budou opraveny, a které nikoliv.
- **náhled na stávající část kódu** - Zvýrazněn je kus kódu, který bude nahrazen. Pokud se žádných kód nahrazovat nebude, je zvýrazněn pouze řádek, kam bude vložen nový kód.
- **náhled na nově vygenerovanou část kódu** - Zde má uživatel možnost provést před vložením nového kusu kódu libovolné úpravy.

Po stisknutí tlačítka *OK* je provedena oprava zvolených nekonzistencí.



Obrázek C.1: VHDT - Dialog nekonzistencí

C.3.3 Práce se šablonami

Správa šablon probíhá přes záložku ***VHDL Templates***. Kliknutím pravého tlačítka myši v prostoru okna lze přidávat existující šablony nebo celé složky šablon. V konkrétní složce lze vytvořit novou šablonu pomocí menu vyvolaného kliknutím pravého tlačítka myši a poté *New/Empty File*.

Při akcích *Generate New Entity*, *Generate Testbench* a *Insert Code* aplikace nabízí šablony právě ze záložky *VHDL Templates*, přičemž filtruje pouze ty, které jsou relevantní dané akci (např. pro generování nového testbenche nabídne aplikace pouze šablony označené jako `@TestbenchTemplate`).

Šablona entity a fragmentu kódu

Šablonu je třeba označit jednou z následujících anotací:

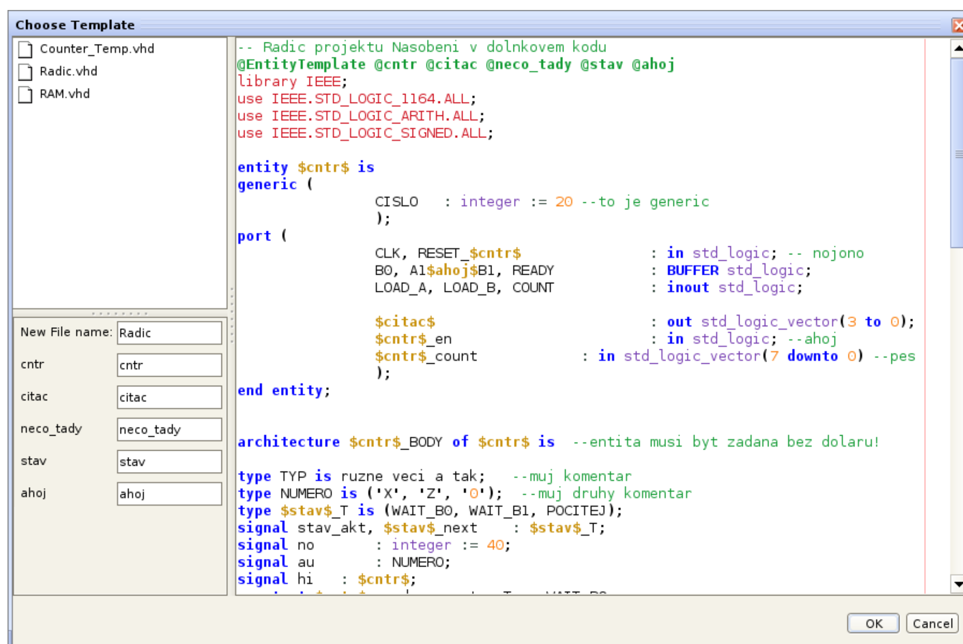
- `@EntityTemplate` ... pro šablonu celé entity
- `@CodeTemplate` ... pro šablonu fragmentu kódu

V šabloně entity či fragmentu kódu se dají použít následující značky a příkazy:

- `@jmeno` ... definice globální proměnné

- `$jmeno$` ... odkaz na globální proměnnou

Vytváření nové entity ze šablony se spustí pomocí akce ***Generate New Entity***, dostupné z menu či toolbaru *Project*. Po kliknutí se zobrazí ***dialog pro výběr šablony*** (viz obrázek C.2). Při zvolení konkrétní šablony se zobrazí náhled kódu a dialog s editovatelným seznamem globálních proměnných. Ty můžete dle libosti přejmenovávat. Po stisknutí tlačítka *OK* se v hlavním adresáři projektu vytvoří nově vygenerovaná entita. Zároveň se zobrazí také v editoru kódu.



Obrázek C.2: VHDT - Dialog pro výběr šablony

Vkládání generovaného kusu kódu se vyvolává v editoru. Na místě, na které má být nový kód vložen, klikněte pravým tlačítkem a vyberte možnost ***Insert Code***. Po kliknutí se zobrazí dialog pro výběr šablony (viz obrázek C.2). Při zvolení konkrétní šablony se zobrazí náhled kódu a dialog s editovatelným seznamem globálních proměnných. Ty můžete dle libosti přejmenovávat. Po stisknutí tlačítka *OK* se v editoru na místě kurzoru objeví nově vygenerovaný kus kódu.

Šablona testbenche

Šablonu testbenche je třeba označit anotací `@TestbenchTemplate`.

V šabloně VHDL testbenche se dají použít následující příkazy:

- `$entityName$` ... Na všechna místa, na kterých se vyskytuje tento příkaz, bude vloženo jméno testované entity (i do komentářů).
- `$tbName$` ... Na všechna místa, na kterých se vyskytuje tento příkaz, bude vloženo jméno testbenche (i do komentářů).
- `$component$` ... Na místě této značky se v testbenchi vytvoří celá komponenta, obsahující porty a generiky testované entity (pokud jsou).
- `$signals$` ... Příkaz pro vložení signálů.
- `$instance$` ... Příkaz pro vložení celé instance.
- `$port_map$` ... Namapování portů na signály.
- `$port_map_inline$` ... Namapování portů na signály (stylem „v řadě").
- `$generic_map$` ... Namapování generiků na signály.
- `$generic_map_inline$` ... Namapování generiků na signály (stylem „v řadě").

Vytváření testbenche se spouští pomocí akce ***Generate Testbench***, která je dostupná z menu či toolbaru *Entity* a nebo ze seznamu akcí na uzlu konkrétní entity. Po kliknutí se zobrazí dialog pro výběr šablony (viz obrázek C.2). Lze volit mezi vytvořením nového souboru s testbenchem (*Generate*) a vložením vygenerovaného kódu do systémové schránky (*Clipboard*).

Kopírování do schránky se dá využít například v případech, kdy je třeba vytvořit komponentu z konkrétní entity. Stačí vytvořit šablonu označenou jako šablona testbenche, která obsahuje pouze příkaz pro vytvoření komponenty.

C.3.4 Nastavení prostředí

Veškerá nastavení probíhají pomocí dialogu přístupného z menu *Tools/**Options***. Zmíním například možnost nastavení velikosti a chování tabulátoru v záložce *Editor/Formatting* či možnost nastavení obarvování VHDL syntaxe v záložce *Fonts And Colors/Syntax* (v nabídce *Language* vybrat VHDL). Klávesové zkratky se nastavují v záložce *Keymap*.

Příloha D

Příručka programátora

D.1 Požadavky

D.1.1 Verze JDK

Při vývoji je nutné používat JDK 6 a vyšší. Aplikace jde bez problémů kompilovat i pomocí JDK 7. Doporučuji verzi od společnosti Oracle (kvůli nástroji NetBeans IDE - viz dále).

D.1.2 Vývojové prostředí

Vzhledem ke skutečnosti, že je aplikace postavena na platformě NetBeans, doporučuji používat k jejímu vývoji prostředí NetBeans IDE. To v sobě rovnou obsahuje platformní knihovny a umí modulární aplikace postavené na těchto knihovnách sestavovat. Obsahuje také množství průvodců, kteří programátorovi pomáhají s vytvářením nejčastěji používaných kusů kódu, jakými jsou například akce.

K vývoji je třeba používat NetBeans IDE 7.0 a vyšší. Předchozí verze NetBeans neobsahují kompletní podporu anotací. Pomocí anotací se dá nastavovat například jméno, viditelnost a umístění nově vytvářených akcí. Ve starších verzích platformy byl uživatel nucen toto nastavení provádět pomocí souborů *layer.xml*.

NetBeans doporučuji spouštět pouze pomocí Oracle JDK. Programátor se tím vyvaruje zbytečným problémům, které mohou nastat při použití OpenJDK. NetBeans IDE a celá platforma NetBeans jsou vyvíjeny a testovány pod Oracle JDK, je to tedy „sázka na jistotu“.

V prostředí Windows by neměl nastat problém. V linuxových distribucí doporučuji stáhnout a nainstalovat nejprve Oracle JDK a až poté NetBeans IDE. V distribucích založených na Debianu stačí stáhnout a nainstalovat *.bin* soubor s JDK do domovského adresáře na disku. Při instalaci NetBeans IDE použijte verzi ze stránek Oracle (soubor *.sh*). Instalátor si sám najde a přednastaví Oracle JDK jako výchozí běhové a sestavovací prostředí.

D.2 Rozšiřování aplikace

Před započetím prací doporučuji projít alespoň základní tutoriály na stránkách platformy NetBeans(12). Ideální je pročíst také výbornou knihu o platformě NetBeans(3). Na jejím překladu se podílel také Jaroslav Tulach, původní zakladatel platformy NetBeans.

Při vývoji je nutné nahlížet do platformní dokumentace(11). Doporučuji doinstalovat platformní javadoc přímo do prostředí NetBeans. Dokumentace se poté zobrazuje v podobě rychlé nápovědy v kódu. Instalace se provádí jednoduše přímo v aplikaci přes menu dostupných pluginů.

Struktura aplikace VHDT je poměrně podrobně popsána v kapitole 4. Kód je dobře dokumentován, další informace tedy mohou být čerpány z dokumentace k aplikaci.

Příloha E

Ukázky šablon

Šablona VHDL struktury - automat

Šablona automatu

```
@EntityTemplate
-- Ukazkova sablona
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
-- Definice globalni promenne
@cntr

-- Odkazy na globalni promennou
entity $cntr$ is
port (
        $cntr$_out    : out std_logic_vector(3 to 0);
        $cntr$_en     : in std_logic;
        $cntr$_count  : in std_logic_vector(7 downto 0)
);
end entity;


architecture $cntr$_BODY of cntr is
...
end architecture;
```

Vygenerovaná struktura

```
-- Ukazkova sablona
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
-- Definice globalni promenne


-- Odkazy na globalni promennou
entity citac is
port (
        citac_out   : out std_logic_vector(3 to 0);
        citac_en    : in std_logic;
        citac_count : in std_logic_vector(7 downto 0)
);
end entity;


architecture citac_BODY of citac is
...
end architecture;
```

Šablona VHDL testbenche

Testovaná entita

```
-- Priklad jednoduche testovane entity
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity muj_counter is
port (
           clk              : in std_logic;
           counter_reset    : in std_logic;
           counter_enable   : in std_logic;
           counter_done     : out std_logic;
        );
end muj_counter;

architecture arch of muj_counter is
...
end arch;
```

Šablona, podle které se bude tvořit testbench

```
@TestbenchTemplate
-- Ukazka sablony bezneho testbenche
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

-- Na miste znacky $tbName$ bude jmeno testbenche
entity $tbName$ is
end entity;

architecture TESTBENCH of $tbName$ is

    -- Na toto misto bude vlozena cela komponenta
    $component$

    -- Zde budou vlozeny signaly
    $signals$

begin

    -- Nazev testovane entity
    mojePortMap1 : $entityName$ PORT MAP(
          -- Namapovani portu na signaly
          -- Zde by mohl byt jeste prikaz $generic_map$ slouzici
          -- k namapovani generiku
          $port_map$
    );

    mojePortMap2 : $entityName$ PORT MAP(
          -- Namapovani portu na signaly stylem "v rade"
          -- Zde by mohl byt jeste prikaz $generic_map_inline$
          $port_map_inline$
    );

    -- Vlozeni cele instance
    $instance$

    -- hodinovy signal
    clk_sig : process
    begin
        CLK<= '1';
        wait for 50 ns;
```

```
        CLK <= '0';
        wait for 50 ns;
    end process;

end architecture;
```

Výsledný testbench

```
-- Ukazka sablony bezneho testbenche
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

-- Na miste znacky muj_counter_tb bude jmeno testbenche
entity muj_counter_tb is
end entity;

architecture TESTBENCH of muj_counter_tb is

    -- Na toto misto bude vlozena cela komponenta
    component muj_counter is
    port (
               clk              : in std_logic;
               counter_reset    : in std_logic;
               counter_enable   : in std_logic;
               counter_done     : out std_logic;
            );
    end component;

    -- Zde budou vlozeny signaly
    signal clk : std_logic;
    signal counter_reset : std_logic;
    signal counter_enable : std_logic;
    signal counter_done : std_logic;

begin

    -- Nazev testovane entity
    mojePortMap1 : muj_counter PORT MAP(
          -- Namapovani portu na signaly
          -- Zde by mohl byt jeste prikaz $generic_map$ slouzici
          -- k namapovani generiku
          port map(
            clk => clk,
```

```
            counter_reset => counter_reset,
            counter_enable => counter_enable,
            counter_done => counter_done
          );
    );

    mojePortMap2 : muj_counter PORT MAP(
          -- Namapovani portu na signaly stylem "v rade"
          -- Zde by mohl byt jeste prikaz $generic_map_inline$
          port map(clk => clk, counter_reset => counter_reset,...
    );

    -- Vlozeni cele instance
    i_muj_counter : muj_counter
    port map(
            clk => clk,
            counter_reset => counter_reset,
            counter_enable => counter_enable,
            counter_done => counter_done
    );

    -- hodinovy signal
    clk_sig : process
    begin
        CLK<= '1';
        wait for 50 ns;
        CLK <= '0';
        wait for 50 ns;
    end process;

end architecture;
```

Příloha F

Obsah přiloženého CD

```
readme.txt
src
  javadoc ........................ dokumentace ke zdrojovým souborům
  VHDT ........................... zdrojové soubory programu
text
  DPMateju.pdf ................... text práce ve formátu pdf
  DPsource ....................... zdrojové soubory textu práce
vhdt.zip ......................... archiv se spustitelnou formou implementace
  bin ............................ adresář se spouštěči pro Linux a Windows
    vhdt ......................... spouštěcí skript pro Linux
    vhdt.exe ..................... spouštěč pro Windows
  etc
  ide
  platform
  vhdt
plugin ........................... adresář s pluginem pro NetBeans IDE
```