VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GENERÁTOR KONEČNÝCH AUTOMATŮ Z GRAFICKÉHO POPISU PRO JAZYK VHDL

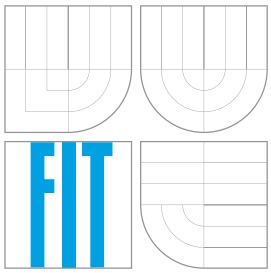
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN JANYŠ

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GENERÁTOR KONEČNÝCH AUTOMATŮ Z GRAFICKÉHO POPISU PRO JAZYK VHDL

FINITE STATE MACHINES GENERATOR BASED ON GRAPHICS DEFINITION FOR VHDL LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN JANYŠ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ŠIMEK VÁCLAV
SUPERVISOR

BRNO 2013

Abstrakt

Práce seznamuje čtenáře s možnostmi návrhu a tvorby konečných stavových automatů s důrazem na reprezentaci pomocí jazyka VHDL. Hlavním tématem je aplikace, která implementuje generátor VHDL kódu na základě grafického popisu, který je možné v něm vytvořit. Popsány jsou klíčové oblasti aplikace. Zejména jejich použití a implementace, která realizuje samotný převod stavového diagramu do VHDL.

Abstract

The work introduces the reader to the possibilities of design and creation of finite state machines with focus on representation using VHDL. The main topic is the application that implements the VHDL code generator based on graphic description which can be create. The key application areas are described. In particular, their use and implementation that implements the actual transformation of the state diagram into VHDL.

Klíčová slova

konečný automat, FSM, VHDL, generátor konečných automatů, sekvenční logika, stavový diagram, graf přechodu

Keywords

finite state machine, FSM, VHDL, generator of finite machine, sequential logic, state diagram, transition diagram

Citace

Martin Janyš: Generátor konečných automatů
z grafického popisu pro jazyk VHDL, bakalářská práce, Brno, FIT VUT v Brně, 2013

Generátor konečných automatů z grafického popisu pro jazyk VHDL

Prohlášení

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

.....
Martin Janyš
14. května 2013

Poděkování

Zde chci poděkovat Ing. Václavovi Šimkovi za odborné vedení práce a konzultace. Dík patří také vyučujícím příbuzných kurzů, kteří se na této práci také nepřímo podíleli.

Obsah

Kapitola 1

Úvod

Následující text bakalářské práce se bude věnovat vytvořené aplikaci *Generátor konečných automatů z grafického popisu pro jazyk VHDL* a teorii, která s tímto tématem souvisí.

Budou stručně vysvětleny vlastnosti jazyka VHDL. Podrobněji budou popsány konečné automaty a jejich varianty popisu. Zejména grafický popis a popis v jazyce VHDL. Ukážeme si několik způsobů a technik, jak navrhnout konečný automat a jakým stylem ho implementovat. To budou základní předpoklady pro návrh a realizaci, jak s těmito druhy popisu sekvenčních obvodů a také se samotným generátorem konečných automatů pracovat.

Dále se budu věnovat tomu, jak byl navržen a implementován program, který s konečnými automaty pracuje, a jaké jsou možnosti při vytváření automatu. Jedna kapitola bude věnována samotnému generátoru 6. Prvním bodem budou požadavky na prostředí, použité technologie. Stručně si popíšeme prvky grafického rozhraní, u kterých budou vysvětleny jejich hlavní funkce a případné implementační zajímavosti. Zmíněna bude i použitá knihovna pro vykreslování grafů a její integrace do výsledné aplikace. Podrobněji se budu věnovat algoritmu, který provádí převod grafického popisu do VHDL pomocí knihovny jVHDL. Tato část je klíčová, protože se jedná o jádro programu a její pochopení uživateli objasní, co se děje na pozadí aplikace. V dnešní době již existují podobné softwarové nástroje, které provádějí analogickou činnost. O některých z nich se zmíním v sekci 4, kde bude uvedeno, co považuji za jejich klady, zápory a jakým způsobem se je navržená aplikace pokusí předčít.

K realizaci jsou využity dvě knihovny. O první z nich se zmíním v kapitole 6.1. Jedná se o knihovnu na vizualizaci a práci s grafy. Druhá, zajišťující výstup do VHDL, byla navržena a implementována hlavně pro tento projekt s důrazem na možnost znovupoužití v jiných projektech, proto je realizována jako samostatná část. Jedná se tedy také o produkt této práce, proto jí bude věnována kapitola 8. V ní bude popsána implementace, vlastnosti a její rozsah ve srovnání s jazykem VHDL, který je úkolem výstupu této části. Pro možné znovupoužití bude uvedena praktická část s ukázkami funkcí knihovny a jejich výstup.

Kapitola 2

Jazyk VHDL

VHDL je vysokoúrovňový programovací jazyk. Pod prvním písmenem V se skrývá zkratka VHSIC. Celým názvem je tedy VHDL *Very High Speed Integrated Circuit Hardware Description Language*. Jedná se o jazyk pro popis hardware. Slouží k syntéze nebo simulaci jak jednoduchých, tak rozsáhlých počítačových systémů [7]. Přičemž kód je nezávislý na cílové architektuře. Závislost na architektuře vytváří až kompilátor VHDL, který je zpravidla dostupný od výrobce čipu.

Při práci s VHDL je nutno brát v úvahu, že pracujeme s číslicovým systémem. Na tento fakt poukazuje věta: „Když pracujeme s VHDL, neprogramujeme, navrhujeme hardware“ ([5] str. 8). VHDL umožňuje jednoduchým způsobem zapisovat paralelismus, specifikovat datové struktury a modelovat elektrické vlastnosti, jako je například zpoždění.
Existují 3 základní způsoby popisu ve VHDL:

1. Data-flow popis:
   Specifikace obvodu data-flow popisem zachycuje, jak data procházejí obvodem. Jedná se o definování vztahu mezi vstupem a výstupem pomocí operací jako AND a OR.

   Jde o dobrý způsob pro popis menších prvků, kde je dobře vidět vztah vstupu a výstupu. Popis se podobá výslednému obvodu. Pro popsání složitých obvodů tento způsob není ideální. ([5] str. 52)

2. Strukturní popis:
   Zachycuje propojení komponent pomocí signálů. Propojení odpovídá schématu zapojení. Často se vyplácí typy popisu kombinovat. Strukturním popisem lze popisovat hierarchické propojení komponent. Komponenty na nižší úrovni mohou být popsány behaviorálně a propojeny.

3. Behaviorální popis:
   Behaviorálním popisem je možné relativně snadno popsat složité číslicové obvody. Při takovém návrhu jsme odstínění od hardware a pracujeme na vyšší abstrakci. Popis algoritmu nemusí odpovídat jeho realizaci v obvodu. Předpis definuje, jak prvek reaguje na odpovídající vstupy. Ve srovnání s data-flow můžeme říci, že data-flow popisuje, jak obvod může vypadat pomocí hradel, a behaviorální popis udává, jak se bude obvod chovat ([5] str. 53).

   Prostředkem pro tuto formu popisu je proces. Procesy jsou vykonávány paralelně. Kód procesu je proveden sekvenčně.

Zdrojem dalších informací o VHDL mohou být knihy [7, 5], ze kterých jsem také čerpal.

Kapitola 3

Konečné automaty

Pod pojmem konečný stavový automat je teoretický aparát. Jedná se o matematickou abstrakci, která popisuje chování systému. Systémem je zpravidla sekvenční obvod. Tento popis je převzat z ([5] kap. 7). Sekvenční obvody řízené konečnými automaty se používají jako řídící jednotky v číslicových obvodech, jakými mohou být například řadiče. Jelikož konečným automatem můžeme popsat i komplexní chování, mají konečné automaty široké možnosti využití.

Zatímco konečný automat z formálních jazyků je definován uspořádanou pěticí (Q, Σ, R, s, F), v oblasti hardware a popisu chování obvodů pracujeme s jinou definici. Konečný automat je definován pěti vstupy: symbolickými stavy, vstupními signály, výstupními signály, funkcí určující další stav a výstupní funkcí. Stavy specifikují unikátní vnitřní stav obvodu. Postup z jednoho stavu do druhého je definován přechodem [3].

3.1 Typy konečných automatů

Popis chování konečného automatu v hardware se realizuje v sekvenčním obvodu. V sekvenčních obvodech tomu není jako u kombinačního obvodu, kde je vstup ihned veden přes logiku obvodu na výstup. Sekvenční obvody mají vstupy, výstupy a hlavně svoji vnitřní paměť, která uchovává aktuální stav, ve kterém se právě nalézá.

V oblasti sekvenčních obvodů mluvíme o dvou typech konečných automatů, které je realizují. Označujeme je jako Moorův automat a Mealyho automat. Toto rozdělení vyplývá z typů výstupu, které obsahuje. Tyto dva typy výstupu lze kombinovat v jednom obecném automatu. Ve speciálních případech je dokonce potřeba využít jeden výstup jak v Mealyho formě tak jako Moorův výstup. Oba typy jsou vzájemně převoditelné, ale může to vést k nárůstu stavů.

- **Moore**
  Automat obsahuje výstup, který je závislý pouze na aktuálním stavu. V grafickém popisu tento typ poznáme tak, že se hodnota výstupu nalézá ve stavu.

- **Mealy**
  Automat obsahuje výstup, který je závislý na aktuálním stavu a zároveň na vstupu. V grafickém popisu tento typ poznáme tak, že se hodnota výstupu nalézá na hraně spolu s podmínkou přechodu. Tento typ obvykle zabírá méně zdrojů. Pro realizaci svého chování nevyžaduje takový počet stavů.

Zdrojem posledních odstavců této části o typech automatu a rozdílech mezi nimi je kapitola [4], ze které jsem čerpal.

Vybraný typ automatu bude mít při syntéze vliv na jeho velikost, a tím i na jeho kódování a rychlost. Při použití automatu jako řídící jednotky je signál generován pomocí Moorova výstupu. Oba dva typy automatů jsou považovány za stejně výpočetně schopné (oba jsou schopné rozpoznávat regulární výrazy) a jsou mezi sebou převoditelné.

Na stejnou úlohu má Mealyho automat z pravidla méně stavů. Je to dáno tím, že výstupní funkce závisí na vstupu. Několik možných výstupů je možné definovat prostřednictvím jednoho stavu.

Druhým rozdílem je, že Mealyho automat může generovat rychlejší odezvu. Jelikož Mealyho výstup je funkcí vstupu, je změněn kdykoliv dojde ke změně vstupního signálu. Naproti tomu Moorův výstup reaguje nepřímo na změny vstupu, protože výstup je proveden po provedení přechodu. V synchronních systémech to znamená, že Moorův výstup je o jeden hodinový cyklus zpožděn za Mealyho.

Třetí rozdíl spočívá v řízení šířky a načasování výstupního signálu. Vstupní signál je generován a normálně bývá deaktivován při vstupu do dalšího stavu. Mealyho výstup je generován při změně vstupu, a proto je náchylnější k chybám, které přenáší ze vstupu na výstup. Moore je synchronizován hodinovým signálem, proto není na tyto chyby náchylný [4].

3.2 Formy popisu

Vstupem konečného automatu může být nekonečná posloupnost, je nutné použít vhodný prostředek pro jeho popis. Nelze použít předpis pro každou konfiguraci samostatně. Konečný automat je abstrakcí a existuje několik způsobů, jak je možné ho popsat. Mezi nejběžnější způsoby patří popis pomocí soustavy rovnic, tabulková reprezentace, grafický popis nebo popis v některém programovacím jazyce, například VHDL.

3.2.1 Soustava rovnic

Soustava rovnic je jednoduchý strukturovaný popis, který řádkově zaznamenává přechodové funkce konečného automatu. Zápis je ve tvaru aktuální stav, vstup, šipka a následující stav. Výstupní funkce je zapsána v druhé soustavě v pořadí stav, vstup, výstup. Případně stav, výstup pro Moorův typ.

Tabulka 3.1: **Příklad reprezentace pomocí rovnic na klopném obvodu RS**

aktuální stav	*S R*		*následující stav*
q0	0X	→	q0
q0	10	→	q1
q1	01	→	q0
q1	X0	→	q1

aktuální stav:	*Moore*
q0:	0
q1:	1

3.2.2 Tabulka

Pro tabulkovou reprezentaci je důležité si stanovit, kde uvedeme stavy a kde vstupy. Jestliže se bude jednat o řádky nebo sloupce, v příkladu bude ukázána varianta, kdy řádek je stav a vstup sloupec.

Tabulka 3.2: **Příklad tabulkové reprezentace klopného obvodu RS**

State \ R S	00	01	10
q0	q0	q0	q1
q1	q1	q0	q1

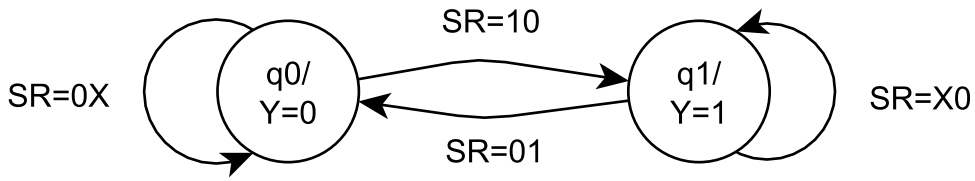
State \ Output	Y
q0	0
q1	1

3.2.3 Graf

Reprezentace pomocí orientovaného grafu je velmi komplexní. Je možné zaznamenat všechny stavy a přechody přehledně na jednom místě (přechodová i výstupní funkce v jedné reprezentaci). Graf, který je použit pro popis chování hardware, označujeme stavový diagram nebo graf přechodu. Ve stavech se nalézá jejich pojmenování a případný výstup typu Moore. Na hranách se zapisuje podmínka přechodu a Mealyho výstup. Jde však pouze o zvolenou konvenci a značení lze zvolit i jinak. Důležitá je pouze jednoznačnost a přehlednost.

Graf přechodu je jednou z grafických forem popisu. Existují i další metody grafického popisu, jako například schéma, algoritmické automaty[1] a další.

Obrázek 3.1: **Příklad grafické reprezentace na klopném obvodu RS:**



3.3 Popis ve VHDL

Popis v některém programovacím jazyce je nejběžnějším použitím. Je to logické, protože předešlé formy popisu jsou pouze návrhem. Teprve pomocí implementace v HDL[2] je možné ho realizovat.

V předešlých formách popisu existuje více způsobů, jak jeden konečný automat popsat. Není tomu jinak ani u popisu v programovacím jazyce. Nicméně je dobré dodržovat některé zásady. Případně vědět jakou variantu zvolit.

[1] Jedná se o reprezentaci podobnou vývojovým diagramům.
[2] Hardware Description Language

Možnosti způsobů, jaké lze použít při návrhu konečného automatu, jsou popsány v [3]. Doporučení stejného typu se nalézají v textu, který je v prostředí Xilinx[3] ISE, v sekci se šablonami[4].

1. Typu výstupu:
   Volba mezi Moorovým nebo Mealyho automatem (viz. 3.1).

2. Kódování stavů:
   Kódování přímo ovlivňuje rychlost výsledného automatu. Důvodem je nutnost stav dekódovat. Většina moderních syntetizačních nástrojů disponuje algoritmy pro volbu optimálního kódování v závislosti na architektuře a počtu stavů automatu. Je však i možnost kódování explicitně určit.

   Existuje více způsobů kódování. Mezi nejběžnější patří binární a 1 z N (one-hot). Pomocí binárního kódování je možné zakódovat více stavů. Nevýhodou je, že dodatečná logika pro kódování a dekódovaní obvod zpomaluje, lze však na n bitech reprezentovat 2^n stavů. 1 z N je rychlejší způsob a také se snáze implementuje, ale s každým dalším stavem roste i šířka registru uchovávající stavy.

3. Počet procesů:
   Chování automatu lze popsat různým počtem procesů. Může se jednat o jeden, dva, tři nebo pro každý výstup samostatný proces.

   V případě jednoho procesu provedeme veškerou logiku popisující chování v jednom procesu. Dvouprocesová alternativa rozdělí předchozí jednu sekvenci na samostatný proces pro logiku následujícího stavu („next-state logic“) a proces aktuálního stavu („current-state logic“). Při použití tří stavů máme samostatný proces aktuálního stavu, následujícího stavu a výstupní logiku.

4. Jazykové konstrukce:
   Podmíněný výraz pro výběr stavů může být realizován pomocí řídící struktury if-elsif-else nebo pomocí switch (case). Obě varianty jsou ekvivalentní. Osobně považuji switch za přehlednější variantu.

5. Bezpečný automat:
   V prostředí Xilinx je zmíněno dělení na automaty bezpečné nebo rychlé (Safe vs. Fast). Automat, který bez uvažování o jeho bezpečnosti vytvoříme, nemusí být nutně bezpečná forma implementace. Bezpečnost stavů závisí na kódování stavů. V případě použití binárního kódování a počtu stavů, který je mocnina dvou, bude automat považován za bezpečný. To znamená, že pokud platí $N = 2^M$, kde N je počet stavů a M celé číslo, bude se jednat o bezpečnou implementaci, protože všechny varianty kódu stavu, které mohou nastat, jsou pokryty určitou reprezentací stavu. Jestliže máme počet stavů, který není mocninou dva, nebo je použito jiné kódovaní, jakým je 1 z N, bude se jednat o automat, který není bezpečný.

[3] Xilinx je jedním z největších světových výrobců logických obvodů.
[4] VHDL > Synthesis Constructs > Coding Examples > State Machines.

Kódování 1 z N je nebezpečné implementaci náchlnější. Při čtyřech stavech s kódováním 1 z N, máme obsazena 4 kódová slova, 12 kombinací není reprezentace stavu. Není jím tedy přiřazen žádný stav a ani chování.

Bezpečný automat se vyznačuje tím, že má ve své implementaci dodatečnou logiku pro neočekávaný stav. Ta ho obnoví v příštím hodinovém taktu zpět do korektního stavu. Bez této logiky je samozřejmě automat rychlejší.

Zdrojem informací pro tuto podkapitolu byl text z prostředí Xilinx a [8].

V implementaci generátoru je možno definovat obecný automat, kde je možné používat Mealyho i Moorovi výstupy. Kódování je ponecháno na syntetizátoru a výstupní kód obsahuje dva procesy. Výběr stavu provádí řídící struktura switch (case).

Kapitola 4

Nástroje pro generování konečných automatů

Existuje několik softwarových nástrojů, které se zaměřují na převod grafického popisu konečného automatu do jazyků pro popis hardware. Některé jsou součástí velkých vývojových prostředí, které slouží pro práci s návrhem číslicových obvodů. Představím jejich vlastnosti a zhodnotím co považuji za výhody a nevýhody. Některé nástroje jsou komerční a dostupné v demo nebo trial verzi. Toto považuji za jejich největší nevýhodu. Ve třech případech se jedná o editory pracující s grafem přechodu. V jednom o práci se schématem zapojení. Existuje v nich možnost definovat řadu atributů od volby výrobce cílového čipu přes jazyk až po komentáře kódu.

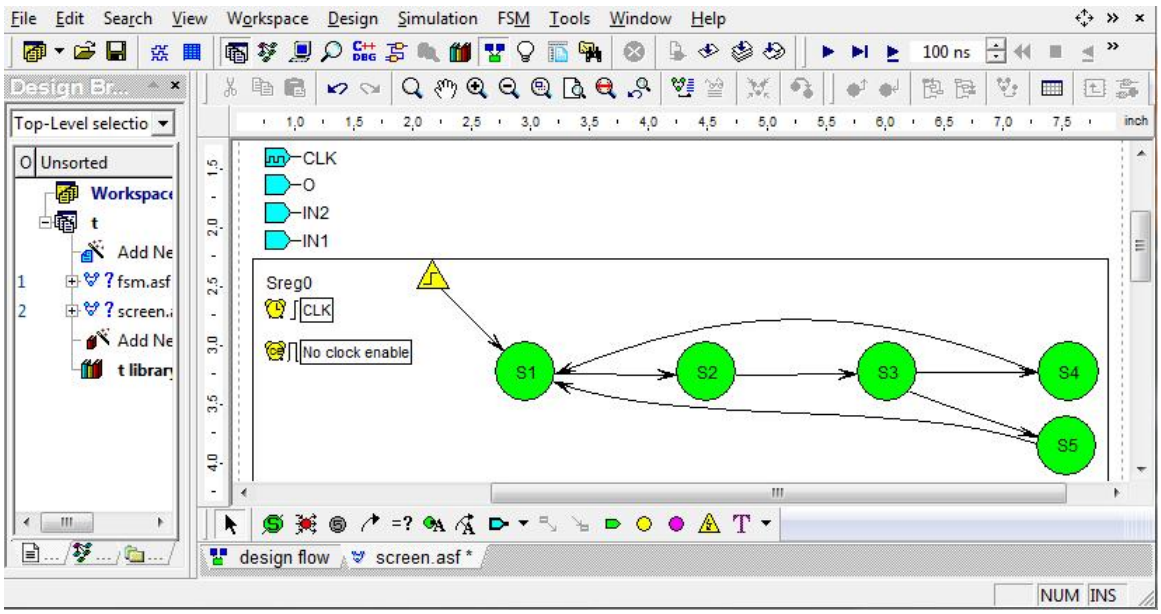
4.1 Active HDL

Active HDL je vývojové prostředí pro VHDL, Verilog, C, SystemC a další jazyky, které se vyskytují v souvislosti s popisem hardware. Na první pohled je vidět, že se jedná o komplexní nástroj. Je plný panelů a nabídek. Velice rychle lze nalézt cestu, jak vytvořit konečný automat. Pod touto volbou se skrývá průvodce, pomocí kterého lze automat definovat. Řetězec dialogových oken je doprovázen obrázkem, který představuje schématickou značku čipu a mění se dle zadávaných údajů. Průvodce po nás bude vyžadovat definici vstupních a výstupních signálů, výčet knihoven, odhadovaný počet stavů a další. Dodatečná definice signálů a entity se provádí graficky, nad definicí automatu, kde jsou také zadefinované signály přehledně k dispozici. Veškerá konfigurace stavů a přechodů je dostupná pomocí dialogových oken. V tomto nastavení se nalézají vyčerpávající možnosti prvku. Lze nastavit jméno, grafickou podobu, komentáře a akce. Každá tato položka představuje možnost volby několika variant.

V případě, že chceme zadat podmínku přechodu nebo výstup, použijeme prvek podobný textovému poli. Spojení s cílovým stavem nebo hranou je provedeno pouze v závislosti na pozici. Stavy mohou mít definovány tři druhy událostí. Jde o akci při vstupu do stavu, ve stavu a při opouštění stavu. Nechybí zde ani možnost uživatelem definovat kódování stavů. Lze zadat i zpoždění pro simulaci.

Výstup může být proveden několika způsoby. Můžeme si vybrat od možnosti generovat komentáře až po volbu stylu výstupu. Tím je myšleno, kolika procesovou logiku požadujeme a zda výběr stavů bude proveden pomocí if nebo case. Je možné vepsat knihovny. Případně měnit jméno entity a podobně.

Obrázek 4.1: Prostředí Active HDL



4.2 StateCAD

Jedná se o nástroj, který byl dříve součástí vývojového prostředí Xilinx. Z pohledu vývojářů Xilinx se stal od verze 10.1 zastaralým. Rozhodli se ho nadále nevylepšovat a neopravovat. Z nástroje se poté stala samostatná aplikace.

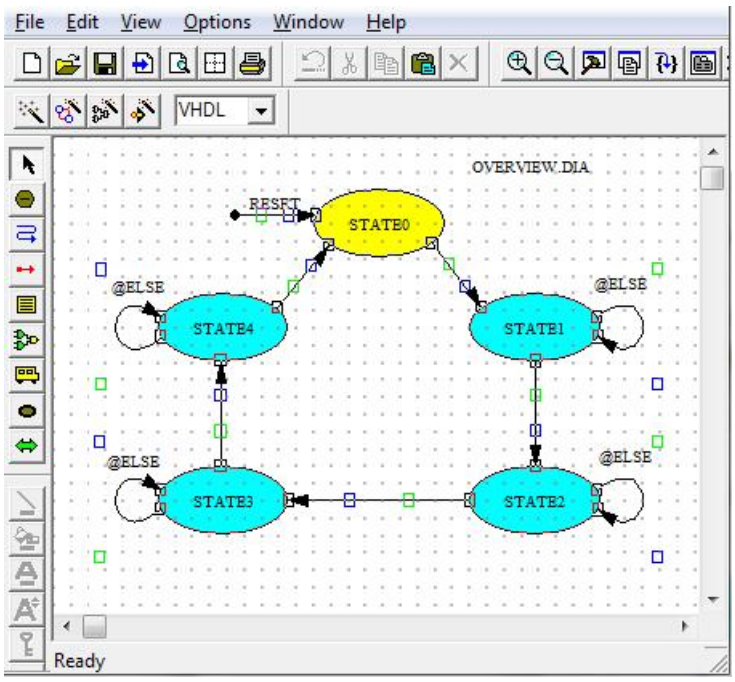
Vytvoření konečného automatu je možné realizovat pomocí průvodce, ve kterém lze definovat všechny potřebné parametry. Na výběr je synchronní či asynchronní reset, forma konečného automatu, který bude vytvořen po konci průvodce jako vzor. Zda bude cílem simulace nebo syntéza na FPGA, cílový jazyk a další. Z cílových jazyků máme na výběr poměrně velké množství. K dispozici je Abel HDL, Altera AHDL, C, Verilog a VHDL. V případě VHDL je potom i možnost výběru prodejce čipu.

V programu na nás čeká plátno, do kterého můžeme jednoduše přidávat místa a přechody. Veškerá data, která do konečného automatu potřebujeme zapsat, je nutné zadat do dialogového okna, do kterého se dostaneme dvojklikem na objekt. V dialogových oknech je možné pomocí průvodce definovat složitější logiku výstupu. Na první pohled je zřejmé, že předchozí aplikace ActiveHDL poskytuje více možností.

4.3 Simulink

Simulink je součástí aplikace Matlab. Prostředí Simulink poskytuje komplexnější možnosti než předchozí programy. K dispozici jsou desítky součástek ze všech oblastí elektrotechniky, které je možné hned použít. V Simulinku lze definovat propojení součástek. Je tedy možné za sebe skládat více prvků a využívat zde konečný automat jako řídící jednotku. Automaty a také ostatní součástky lze simulovat a generovat podle nich HDL. Na obrázku 4.3 je vyobrazen graf, který byl vytvořen v prostředí Simulink. Nemusí se však jednat o graf

Obrázek 4.2: Prostředí StateCAD



přechodu, jedná se o tzv. Stateflow.

Stateflow je prostředí pro modelování a simulaci kombinační a sekvenční logiky. Vychází z diagramů přechodu a vývojových diagramů. Stateflow umožňuje kombinovat grafické a tabulkové znázornění jako stavové diagramy, vývojové diagramy, tabulky přechodu a pravdivostní tabulky. Lze modelovat, jak systém reaguje na události, v závislosti na časových podmínkách a externích vstupních signálech. Se Stateflow lze navrhnout logiku pro dispečerské řízení, plánování úloh atd. Stateflow zahrnuje animace stavového automatu, statické a run-time kontroly pro testování konstrukce souladu a úplnosti před jejich zavedením.[1]

4.4 LabView

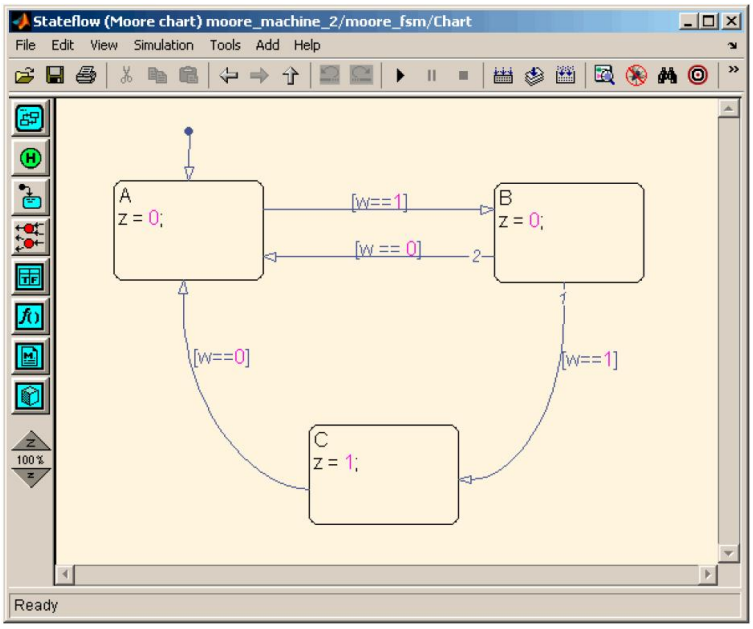
Labview je nástroj, který také umožňuje tvořit konečné automaty pomocí grafického popisu. Nedá se však říci, že spadá do stejné kategorie. Editor realizuje úplně jiný přístup než předešlé aplikace. Konečné automaty jsou definovány ze schématického pohledu pomocí blokových součástek. Dále tento systém neumožňuje generovat VHDL kód, je možné pouze projekty nahrávat na FPGA.

4.5 Shrnutí

Všechny zmíněné aplikace jsou více či méně schopné poskytnou komfortní prostředí pro tvorbu konečných automatů. Každý z uvedených nástrojů disponuje velkým množstvím nabídek a dialogovými či normálními okny, které zobrazují možnosti editace a konfigurace.

[1] http://www.mathworks.com/products/stateflow/

Obrázek 4.3: Prostředí Simulink. Zdroj: [2]



V Generátoru konečných automatů se snažím tento jev eliminovat, zejména používání pomocných oken. Možnosti, které například Simulink poskytuje mají za následek zhoršení přehlednosti aplikace.

Generátor konečných automatů používá podobný koncept náhledů na úrovni blokového schématu a automatu podobně jako Simulink, není však použito pomocné okno. Funkci dialogových oken v Generátoru zastávají postranní panely, které lze skrýt. K realizaci uživatelského rozhraní generátoru se vrátíme v 6.3.

K samotnému algoritmu převodu grafické reprezentace do VHDL jsem bohužel neměl přístup. V jednom případě jsem se inspiroval výstupem pomocí StateCAD, který automaticky doplňoval nedefinované proměnné. Jestliže se totiž vyskytne identifikátor, nadeklaruje ho jako nový signál. Použitím se definují všechny proměnné včetně samotné entity. Aplikace si v případě potřeby (např. použitím výrazů ve výstupní logice) dodefinuje pomocné signály.

Kapitola 5

Aplikace

5.1 Návrh

Navrhnutá aplikace rozšiřuje možnosti základního modulu knihovny JGraphX (viz. 6.1). Cílem bylo vytvořit jednoduchou aplikaci (snadno použitelnou), která bude realizovat převod grafické podoby stavového diagramu na VHDL. Aplikace by měla být co nejjednodušeji rozšiřitelná, jak na úrovni zdrojového kódu, tak umožnit uživateli vkládat již vytvořené komponenty. Navržené rozhraní umožňuje vytvářet grafy a vypisovat k jeho prvkům hodnoty, které jsou spojeny s výstupním VHDL kódem. V aplikaci by mělo být možné definovat vlastní části kódu, takto je vytvořen prostor pro uživatelský vstup. Převod by měl počítat se všemi korektními možnostmi, které mohou být použity v grafickém popisu (hrany bez podmínek, hrany pouze s Mealyho výstupem, stavy bez Moorva výstupu, kombinace předchozích a další). Rozhraní je také možné uvést do kompaktního režimu, kdy je viditelné pouze plátno s grafem. To lze pomocí uzavření postranních panelů, které je možné takto minimalizovat. Návrh prvků grafu je doplněn o UML[1].

5.2 Požadavky

Program je implementován v jazyce Java SE verze 7 a zkompilován do jar archívu. Proto je nutné, aby počítač, na němž je aplikace spuštěna, obsahoval nainstalovanou platformu Java.

Aplikace je podobná grafickým editorům, které pracují s grafy. Předpokládá se, že uživatel je obeznámen s nutnými základy jazyka VHDL a s možnostmi grafického popisu konečných automatů, jak bylo uvedeno v kapitolách 2 a 3.

5.3 Kompilace a spuštění

Projekt je vytvořen dle konvencí nástroje Maven. Maven je nástroj pro automatizaci překladu a správy projektů, který pracuje na příkazové řádce. Pomocí tohoto nástroje je možné aplikaci překládat, spouštět a vytvářet jar archívy.

Během těchto akcí je nutné připojení k internetu, odkud se stahují pomocné repositáře a knihovna JGraphX.

[1] Diagram tříd se nalézá na přiloženém mediu A.

Kompilace

Adresář `jVHDL`:

- `mvn install` — Příkaz provede kompilaci zdrojových kódů knihovny jVHDL a nahraje její archív do lokálního repositáře, kde bude k dispozici.

Adresář `FSM genenerator`:

- `mvn package` — Příkazem zkompilujeme zdrojové kódy FSM generátoru a vytvoříme archív[2].

Jar archív

- `mvn package`
- `mvn compile assembly:single` — Provedením příkazu je vytvořen kompletní archív (včetně knihoven).

Spuštění

- `mvn exec:java` — Spuštění pomocí Mavenu.
- `java -jar FSMGenerator-1.0-jar-with-dependencies.jar` — Spuštění pomocí Javy (archív je vytvořen s parametrem `assembly:single`).

5.4 Funkčnost

Při správném použití aplikace generuje syntetizovatelný kód. Na více místech je ale možné definovat celý kód nebo část kódu uživatelsky, může se jednat o zdroj chyb. Ověření, zda převod funguje korektně, bylo provedeno na většině příkladů z knih [5, 4]. V této literatuře se nalézají příklady na různé typy automatů, jak z pohledu syntaxe ve VHDL, tak z grafického návrhu. Převodní algoritmus byl postupně vylepšován dle těchto příkladů. Po dokončení úprav byli vyzkoušeny další umělé i praktické příklady, které potvrzovali očekávaný výstup. Jeden příklad vytvořený pomocí Generátoru, který byl obsahem školního projektu, byl nahrán do FPGA, kde byla ověřena jeho funkčnost. Jednalo se o kódový zámek (přístupový terminál).

[2] Jde však pouze o balík obsahující aktuální projekt (bez knihoven) a je ho možné spouštět pomocí Mavenu.

Kapitola 6

Uživatelské rozhraní

Generátor konečných automatů z grafického popisu pro jazyk VHDL je hlavní aplikace této práce. Aplikaci tvoří grafický editor, který umožňuje definovat blokové součástky a jejich vstupy a výstupy. Do takto vytvořených bloků je možné vytvořit grafický popis konečného automatu s výstupy Moorova, Mealyho či obecného typu. Vytvořenou grafickou reprezentaci lze převést na odpovídající VHDL kód. Všechny části a komponenty uvažují znovupoužitelnost a jednoduché rozšíření. Příručka, jak s aplikací pracovat, zdrojové kódy a dokumentace je na přiloženém mediu (viz. A).

Rendrování grafů a část grafického rozhraní zajišťuje Open Source knihovna JGraphX.

6.1 JGraphX

JGraphX je Open Source knihovna pod BSD licencí pro vizualizaci grafů. Vývoj probíhá i na jiných platformách než je Java. Jde hlavně o JavaScript, dále ActionScript a .NET, kde dnes existují komerční verze nebo alfa, beta verze [1]. Zdrojové kódy jsou k dispozici na GitHub[1] nebo na stránkách projektu[2]. JGraph je softwarový produkt, který je zaměřen na interaktivní diagramy a grafy. Grafy v matematickém slova smyslu, nejedná se o sloupcové grafy apod. Příklady použití, které jsou součástí balíku, jsou velmi praktické a mnohé z nich jsou použité v generátoru konečných automatů. Pro práci s knihovou je vhodné mít znalosti jazyka Java a dobrý přehled o knihovně Swing (knihovna pro uživatelské rozhraní v Javě).

Knihovna poskytuje běžnou funkcionalitu pro tvorbu, editování, manipulaci a zobrazení grafů. Ve zvoleném přístupu v generátoru konečných automatů nefigurují pouze grafy, ale i bloková schémata. Jelikož je knihovna určena na grafy a diagramy, bylo potřeba všechny třídy, které jsou pro realizaci generátoru nutné, rozšířit a přepsat jejich chování.

Jen díky této knihovně bylo možné vytvořit uživatelsky použitelné rozhraní pro práci s konečnými automaty.

6.2 JGraphX a Generátor konečných automatů

Knihovna JGraphX pro Javu poskytuje v rámci svých příkladů, jak s tímto nástrojem pracovat, bázovou třídu implementující editor a ukázku jak ji rozšířit. Dále jsou k dispozici základní funkce nad editorem a grafem, který se v něm nalézá. Kromě samotné implementace kostry editoru jsou k dispozici i obrázky (ikony), které jsou v aplikaci použity.

[1] https://github.com/jgraph/jgraphx
[2] http://www.jgraph.com/jgraphdownload.html

Jelikož zmíněná ukázka uvažuje obecné použití a požadavek na bloková schémata a stavový diagram je specifická úloha, bylo nutné provést úpravy a kostru „ohnout na míru“.

Na úrovni grafického rozhraní editoru bylo nutné přidat informační dialogová okna a metody pro vytváření, získávání a testu existence složek a souborů. To jsou běžné funkce, které byly jednoduše doplněny. Uživatelské rozhraní obsahuje palety komponent, které má uživatel k dispozici, ty bylo nutné naplnit vlastními daty. V případě palety se styly hran poté bylo nutné implementovat logiku na výměnu stylů vybrané hrany.

Úprava chování grafu představovala pokročilejší problém. Pro potřeby grafického editoru bylo nutné přepsat chování metod, z nichž některé jsou použity jako základ grafického zobrazení na plátně. Z drobných úprav se jedná o modifikaci metod pro klonování (duplikaci) nových buněk, vytváření buněk a hran. Poté byly nutné drobné úpravy metod na testování stavu buněk např.: zda je možné buňku smazat.

Komplexnější pochopení problematiky a práce knihovny potřebovaly další úpravy. Jedná se především o zanořování mezi úrovněmi a zobrazení blokového schématu. Základní idea knihovny podporuje rekurzivně se opakující prvky. Nejedná se však o oddělené úrovně, ale o vyobrazení vnitřku, kde rodič figuruje jako obálka a jeho obsah je viditelný. Takové buňky lze i minimalizovat, touto cestou bychom se však připravili i o objekty bloku, které představují porty entity. V aplikaci jsou dvě úrovně náhledu, kde jsou programově vybrány prvky, které se zobrazí a které ne. S tímto souvisí i kompletní přepsání funkcí pro zanoření a vynoření. Jak bylo uvedeno, knihovna počítá s rodičem jako s obálkou. Tuto obálku potomci automaticky zvětšují, aby byli její součástí. Tento nežádoucí jev bylo také třeba eliminovat.

6.3 Popis rozhraní

Možnosti, které v současné době editor nabízí, jsou možné hlavně díky velkému množství existujících akcí, komponent a voleb z JGraphX. Mohl jsem se tak soustředit na činnost přímo související s konečnými automaty a VHDL.

Uživatelské rozhraní je rozděleno do několika částí, jak je možné vidět na obrázku 6.1. V horní části se nalézá menu a panel nástrojů. Panel obsahuje nejužitečnější nabídky a volby z menu. Hlavní oblast je složena ze tří sloupců. V levém sloupci se nalézá paleta s bloky, místy automatu a hranami automatu. Paleta bude zmíněna v 6.3.7. Pod ní je volitelný náhled na celý graf, který umožňuje i zoomování. Pravá část patří panelu s volbami (viz. 6.3.4). Prostřední a největší část editoru je plátno pro tvorbu schématu nebo grafu.

6.3.1 Práce s editorem

Na přiloženém mediu A je nahrán manuál, který se věnuje používání hlavní aplikace. Obsahuje příklady, jak je možné vytvořit konečný automat v několika krocích.

Uživatelské rozhraní je navrženo účelně a jednoduše. Snaha byla především o možnost rychle a snadno vytvořit komplexní popis. Důraz je kladen také na minimalizaci chyb ve výstupu. Uživatel je možnostmi rozhraní veden k vyplňování jen relevantních dat. I přesto je konečná hodnota, která se odrazí na výstupu ve VHDL, ponechána na uživateli. V aplikaci se vyskytuje minimum dialogových oken, které při častém používání mohou působit nepříjemně. Jedná se o opak aplikací, které byly popsány v 4, kde se provádí konfigurace stavů právě pomocí dialogových oken.

Aplikace pracuje s hierarchickým pohledem na definici automatu. Definice konečného automatu je rozdělena na dvě části. První je kontext vytváření bloku či blokového schématu

z více bloků. Po nadefinování údajů se můžeme „zanořit“. Tato možnost je v pravém horním rohu a pomocí tlačítka **FSM** se šipkou dolů je možné vstoupit do daného bloku.

Pokud se nalézáme v bloku, jedná se o kontext konečného automatu, kde již není možnost vytvářet bloky. Je zde možné vytvářet stavy, přechody a definovat jejich jména, vstupy a výstupy.

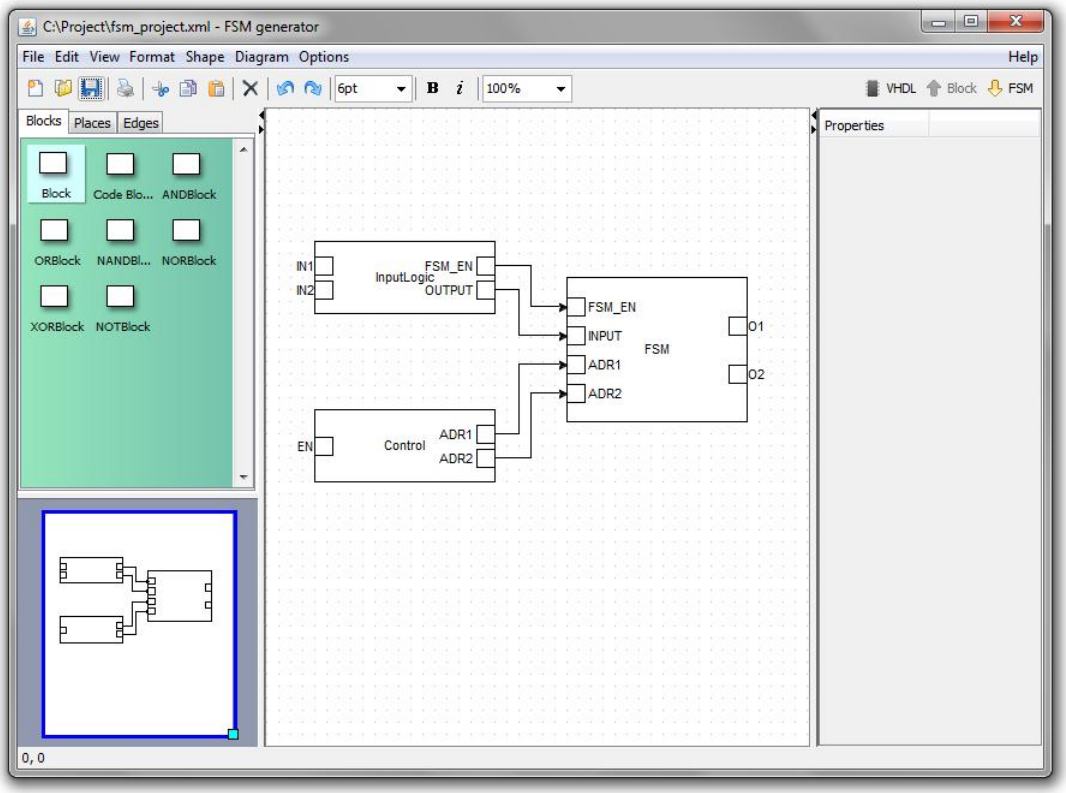
Systém, kdy vstupujeme do bloku je vhodný způsob práce s tímto problémem. Souvisí to i s faktem, že si jednotlivé součástky představujeme jako bloky se vstupy a výstupu a uvnitř nich se nalézá jejich realizace. V našem případě popis chování pomocí grafu konečného automatu. Zároveň je možné brát existující komponenty jako tzv. „black box“ a přehlížet jejich vnitřní implementaci.

Návrat do nadřazené úrovně je možný pomocí tlačítka vpravo nahoře s titulkem **Block**. Vedle tohoto tlačítka se nalézá klíčový prvek této aplikace. Jedná se o volbu s názvem **VHDL**. Jak už název napovídá bude se jednat o možnost výstupu. Výstup je VHDL kód, který blok nebo bloky a jejich popis pomocí konečného automatu realizují.

6.3.2 Bloky

Na obrázku 6.1 vidíme, jak aplikace vypadá v kontextu blokového schématu. Máme zde možnost vytvářet několik různých bloků a definovat propojení mezi nimi. Propojovat je možné pouze porty. Pokud nadefinujeme více než jeden blok a alespoň jeden spoj mezi nimi, je při generování VHDL vytvořen soubor `top_level`, který toto propojení popisuje. Porty, které nejsou propojeny jsou chápany jako vstup/výstup pouzdra celé komponenty, kterou projekt v aplikaci představuje.

Obrázek 6.1: Ukázka blokového schématu v editoru



V základu je možné definovat dva typy bloků:

1. Blok (klasický):
   Jedná se o blok, jehož schránka ma určitý *počet vstupů a výstupů*[3] definovaných uživatelem. Blok má také své *pojmenování*. U jednotlivých vstupů a výstupů, dále jen portů, je možné definovat hodnoty: *jméno, datový typ a výchozí hodnota*. V případě datového typu STD_LOGIC_VECTOR potom dále *směr a hranice od-do*.

   Jméno bloku se odrazí na jméně entity ve VHDL. Jména portu a datové typy se přenesou do jmen vstupů této entity. Výchozí hodnota, pokud je zadána, je použita v behaviorálním popisu tam, kde není explicitně určena.

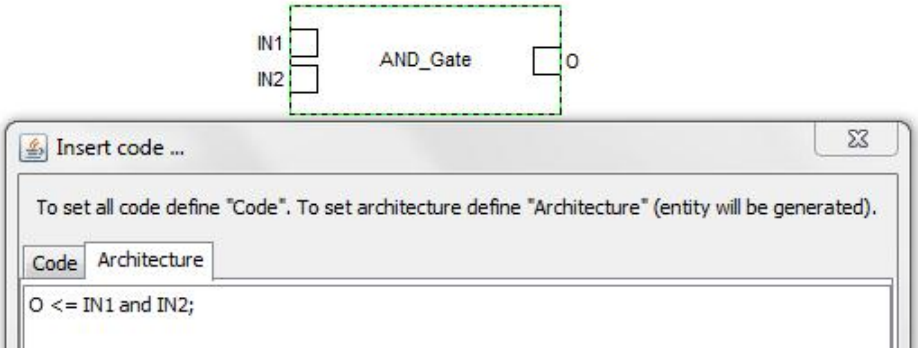
   Po zanoření máme možnost nadefinovat konečný automat.

2. Blok s kódem:
   Od klasického bloku se nastavením neliší. Rozdíl je v zanoření. Při zanoření se zobrazí dialogové okno, které očekává VHDL kód.

   Je zde možnost zadat celý kód celého dokumentu nebo pouze architektury. Při zadání obou má kód vyšší prioritu. Při vyplnění pouze architektury je entita vygenerována z aktuálního bloku. Vstup je ponechán zcela v rukou uživatele, a co zadá, se objeví na výstupu (v případě architektury doplněn o odsazení). Na případnou chybu upozorní až překladač VHDL.

   Vhodné použití tohoto bloku je například pro definici součástek se strukturním popisem, jak je vidět na obrázku 6.2. V ukázce kódu 6.1 je poté vidět přenesení na výstup a přidání odsazení tak, aby byl výstup správně strukturován.

Obrázek 6.2: Použití bloku s kódem



[3] vstupy na levé straně, výstupy na pravé

Ukázka kódu 6.1: VHDL výstup dle stavu editoru na obrázku 6.2

```
library ieee;
use ieee.std_logic_1164.all;

entity AND_Gate is
    port (
        IN1 : IN STD_LOGIC;
        IN2 : IN STD_LOGIC;
        O : OUT STD_LOGIC
    );
end AND_Gate;

architecture description of AND_Gate is
begin
    O <= IN1 and IN2;
end description;
```

6.3.3 Automat

Druhým stavem zobrazení, ve kterém se editor může nalézat je definice konečného automatu. Editor v tomto stavu je vyobrazen na obrázku 6.3. Kontext je přístupný pomocí zanoření do bloku. V tomto kontextu je možné definovat počáteční stav a ostatní stavy automatu. Nejrychlejším způsobem, jak lze grafy vytvořit, je přetáhnout výchozí stav na plátno a následně z tohoto stavu vytvářet nové pomocí vytažení šipky, tímto způsobem je také zajištěno vytvoření automatu bez nedostupných stavů. Po definici stavů a přechodů můžeme definovat jejich konfiguraci. Po označení jedné buňky se v postranním panelu zobrazí nastavení pro vybranou buňku.

V paletě je vidět přítomnost ukončujícího stavu „Final“. Ten je zde pouze pro úplnost a z hlediska návrhu konečných automatů pro popis chování sekvenčních obvodů nemá takový význam jako v oblasti regulárních jazyku. Je ho však nutné použít v případě, že bychom chtěli využít analýzu neukončujících stavů automatu.

6.3.4 Panel možností

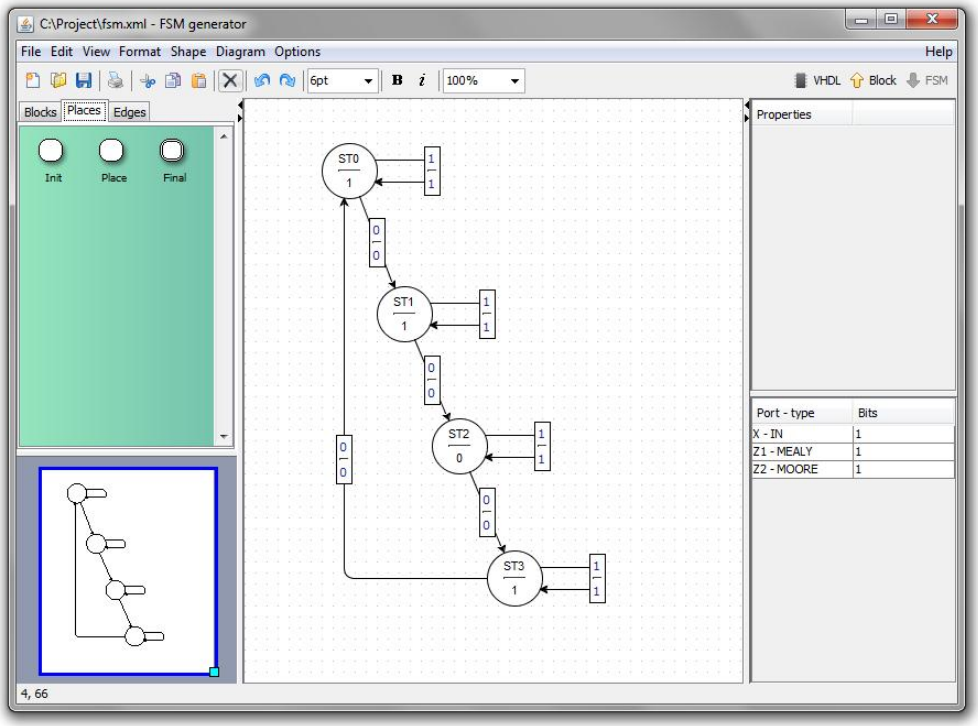
Před dalším výkladem možností tvorby konečného automatu je nutné zmínit část aplikace zajištující správu možností jednotlivých prvků.

Z uživatelského rozhraní je veškerá konfigurace stavů, přechodu a bloků dostupna pomocí postranního panelu vpravo. V panelu je tabulka, která představuje dvojice klíč – hodnota. Tato dvojice přímo odpovídá atributům označeného objektu, které jsou deklarovány jako pole s možnostmi (`propertiesFields`). Při výběru buňky kliknutím se do tabulky vloží aktuální konfigurace označené buňky. Po potvrzení nebo opuštění buňky se hodnota aktualizuje v grafu a buňce.

Jak bylo v úvodu zmíněno, bylo myšleno i na znovupoužitelnost a rozšiřitelnost aplikace. Za tímto účelem jsem vytvořil systém na správu konfigurace libovolné buňky. Řešení spočívá v tom, že každá buňka musí implementovat rozhraní `Properties`, které je obaleno abstraktní třídou `CellWithProperties`. Snazší způsob je tedy rozšířit tuto třídu a definovat jednu metodu.

Způsobem uvedeným v ukázce kódu 6.2 získáme třídu, která nám umožňuje tvořit buňky s možností nastavení. Nesmí zde chybět getter a setter. Jméno v přepsané funkci je vázáno právě na getter a setter. Další proměnné by bylo možné přidat zařazením dalších identifikátorů do pole a přítomností getteru a setteru.

Obrázek 6.3: Ukázka konečného automatu v editoru



Objekty této třídy budou mít proměnné v poli `propertiesFields` přístupné pomocí panelu možností. Kromě možnosti měnit hodnoty těchto proměnných, získáme zobrazení těchto proměnných v tooltipu a automatickou serializaci a deserializaci dané třídy pomocí XML. Vše pouze rozšířením správné třídy a výčtem jmen proměnných.

Tento panel je také rozšířen a použit jako informační. Jde o prostor vpravo dole v režimu konečných automatů (obrázek 6.3). Jsou zde viditelné porty entity, se kterou pracujeme, značení, zda se jedná o vstup, Mealyho nebo Moorův výstup a počet bitů tohoto portu. Panel slouží k tomu, aby byly viditelné proměnné, které máme k dispozici, a uživatel se nedopouštěl zbytečných syntaktických a sémantických chyb.

Ukázka kódu 6.2: Definování nové buňky s možností konfigurace

```
1  public class Use extends CellWithProperties {
       private String var;
       public String getVar() { return var; }
       public void setVar(Object o) { var = o.toString(); }
5
       @Override
       public void initProperties() {
           propertiesFields = new String[]{"var"};
       }
10 }
```

6.3.5 Stavy automatu

Stavy automatu mají jako výchozí styl bílé kruhy s černým okrajem. Jejich vzhled, stejně jako vzhled ostatních prvků, lze měnit. Každý stav musí obsahovat název. Pokud nadřazený

blok obsahuje Moorovy výstupy, je možné je zadat pomocí panelu možností. Tak je částečně kontrolován vstup (je možné zadat výstup jen portům, které jsou definovány jako Moorovy). Zadaná hodnota je přebírána tak, jak je zadána, a správnost je nechána na uživateli. Pouze v případě výskytu hodnoty 1 nebo 0 budou k číslu přidány apostrofy, jakožto k bitové hodnotě.

Ve stavu se nalézá text s názvem. Poté název, horizontální čára a pod ní všechny výstupy oddělené středníkem zleva doprava odpovídají seřazení v nadřazeném bloku shora dolů. Tyto informace jsou viditelné v kontextu tvorby konečného automatu na informačním panelu vpravo. Nezadané výstupy jsou označeny pomocí `X`.

Jména stavů jsou použity k definici signálu, který uchovává aktuální a následující stav. Pokud jsou definovány Moorovy výstupy, jsou tyto hodnoty v tomto stavu přiřazeny.

6.3.6 Přechody automatu

Přechody jsou orientované šipky. Přechod automatu má své hodnoty, které můžeme nastavit. Jedná se o podmínku a Mealyho výstupy.

Nenastavené vstupy jsou značeny symbolem `X` a ve výstupním VHDL se neuvažují.

V možnostech přechodu jsou přístupné všechny vstupy Mealyho výstupy. Existuje více možností jak podmínku přechodu definovat. Všechny varianty budou popsány a uvedeny příklady, jak by vypadala konfigurace pomocí panelu možností (vlevo) a výstupní VHDL (vpravo).

1. Běžný vstup — 6.1:
   Pod běžným vstupem je myšlena varianta, kdy zadáváme například číslo nebo proměnnou. V takovém případě se všechny vstupy a hodnoty se otestují na rovnost a spojí logickým součinem and. Odpovídá-li vstupní řetězec identifikátoru, bude tato proměnná dodefinována v deklaraci architektury.

2. OR — 6.2:
   Pokud hodnota podmínky končí klíčovým slovem „or“ nebo „OR“, je aktuální porovnání spojeno s následujícím pomocí logického or, nikoli and. V předchozím případě byly uvedeny pouze jedničky bez apostrofů. Jelikož se jednalo o 1 bitový vstup hodnoty 0 nebo 1 knihovna jVHDL provedla korekci. Nyní zde máme přidáno or, a proto musíme zadat vstup kompletně.

3. Relační operátory — 6.3:
   Výchozím relačním operátorem je rovnost. Je možné explicitně uvést před testovanou hodnotu jiný relační operátor. Ten pak bude použit i ve VHDL výstupu. Je možné tuto variantu kombinovat s předchozími.

4. Uživatelský vstup — 6.4:
   Tato varianta nechává volbu hodnoty v podmínce zcela v kompetenci uživatele. Za-

Tabulka 6.1: Konfigurace přechodu

Properties	
CE	1
FSM_CE	1
INPUT	1
MEALY	1

```
1  if (CE = '1' and FSM_CE = '1' and INPUT = '1') then
       MEALY <= '1';
       nstate <= State1;
   end if;
```

Tabulka 6.2: Konfigurace přechodu pomocí OR

Properties	
CE	’1’ or
FSM_CE	’1’ or
INPUT	’1’ or
MEALY	1

```
1  if (CE = '1' or FSM_CE = '1' or INPUT = '1') then
       MEALY <= '1';
       nstate <= S;
   end if;
```

Tabulka 6.3: Konfigurace přechodu pomocí relačních operátorů

Properties	
CE	/= ’0’ or
FSM_CE	/= ’0’
INPUT	= ’1’
MEALY	1

```
1  if (CE /= '0' or FSM_CE /= '0' and INPUT = '1') then
       MEALY <= '1';
       nstate <= State1;
   end if;
```

Tabulka 6.4: Konfigurace přechodu pomocí uživatelského vstupu

Properties	
CE	"not CE = ’0’"or
FSM_CE	"FSM_CE = ’X’"
INPUT	"INPUT /= ’X’"
MEALY	1

```
1  if (not CE = '0' or FSM_CE = 'X' and INPUT /= 'X') then
       MEALY <= '1';
       nstate <= State1;
   end if;
```

daná hodnota se pouze vloží na výstup. Tuto variantu využijeme uvedením řetězce podmínky v uvozovkách.

6.3.7 Paleta komponent

Paleta komponent se nalézá v levém postranním panelu. V této nabídce jsou komponenty, které lze přetažením vkládat do grafu.

Zde je uplatněna možnost znovu použít vytvořené blokové entity. Jakýkoliv projekt můžeme po uložení do XML a pomocí nabídky Menu > Soubor > Importovat bloky přidat do palety a vytvářet je opakovaně. Po spuštění editoru se aplikace nahraje soubor `components.xml`, který obsahuje základní bloky a je součástí aplikace. Jedná se o popis hradel.
Paleta nabízí 3 záložky:

- Bloky:
  Aplikace obsahuje dva základní typy bloků, kterým je věnována kapitola 6.3.2.
- Místa konečného automatu viz. 6.3.5
- Hrany (přechody):
  Prvky v této záložce udávají, jaký styl bude použit pro grafickou podobu šipky přechodu. Volba je provedena kliknutím na daný prvek. Výběrem budou nové šipky vytvořeny pod vybraným stylem. Je možné aktualizovat existující označený přechod.

Kapitola 7

Zpracování automatu

7.1 Analýza konečného automatu

Aplikace poskytuje několik možností jak analyzovat graf konečného automatu. Jedná se o analýzu nedostupných a neukončujících stavů, dále o detekci nedeterministických a prázdných přechodů. Všechny algoritmy jsou velmi zjednodušené oproti svým ekvivalentů z formálních jazyků. Důvodem je, že analýza nemění strukturu grafu, pouze identifikuje hrany či místa, které obsahují danou nežádoucí vlastnost. Tyto akce jsou dostupné v nabídce `Diagram` > `Analyze` a mají efekt pouze pokud je aktuální pohled nastaven na konečný automat, nad kterým chceme akci provést. Výstupem všech akcí je varování ve formě žlutého výstražného trojúhelníku u objektů, které analýza vyhodnotila jako nevyhovující. Tato výstraha obsahuje i nápovědu přístupnou po najetí na výstražný trojúhelník pomocí myši.

Analýza nedostupných stavů je provedena pomocí rekurzivního volání nad stavy, přes které algoritmus postupuje ve směru přechodů na další stav a vstupním bodem je počáteční stav. Můžeme si to přestavit jako rozšíření množiny dostupných stavů v grafů ve směru přechodů. Postup grafem je omezen tak, aby se v rekurzi každý stav objevil pouze jednou. Stavy, které nejsou součástí průchodu grafem jsou nedostupné.

Detekce neukončujících stavů využívá stejný princip. Rozdíl je v tom, že vstupními body jsou koncové stavy a postup grafem se provádí proti směru přechodu. Tato analýza tedy vyžaduje, aby graf obsahoval alespoň jeden ukončující stav.

Předposlední akcí pro analýzu je možnost nechat zvýraznit přechody, které nemají podmínku. Často však můžeme tyto přechody a stavy využívat jako čekací.

Poslední je detekce nedeterministických přechodů tedy takových, kde z jednoho stavu vychází více hran se stejnou podmínkou.

7.2 Graf na VHDL

V této sekci bude podrobě popsán způsob generování VHDL z grafického popisu. Pochopení následujícího textu zároveň objasní, co je možné na výstupu očekávat a jak s editorem korektně pracovat. Bude se jednat o postup převodu nikoli o samotný výstup do VHDL. Knihovně, která se stará o VHDL, je vyčleněna samostatná kapitola 8. Aplikace je viditelně rozdělena do dvou samostatných částí a v editoru jsou propojeny třídami z balíku `jvhd.utils`.

Generování probíhá do souborů v uživatelem vybraném adresáři.

1. Celý proces odstartuje průchodem úrovně blokového schématu. Všechny blokové prvky na této úrovni jsou ihned převáděny do VHDL a odkazy na ně uloženy.

2. Pokud se jedná o blok s celým kódem nemůže být mezi ostatní odkazy entit přidán, protože v aplikaci není definováno jeho rozhraní. Když je kódem určena pouze architektura, je rozhraní definováno, a tím i entita.

3. Při převodu bloku do VHDL se již dostáváme na úroveň konečného automatu.

   (a) V prvním kroku jsou přidány knihovny a nastaveno automatické dodefinování signálu **CLK** a **RESET**, pokud již neexistují. Tyto dva vstupy totiž nejsou pro uživatele z hlediska návrhu konečného automatu většinou klíčové, pro samotný automat však ano.

   (b) Následně je vytvořena entita podle struktury bloku. Vytvoření entity spočívá v průchodu objekty, které blok obsahuje, nalezení portů, které ji definují. Během tohoto průchodu jsou ukládány reference na místa (stavy) konečného automatu.

   (c) Nejsložitější je definice architektury. Jedná se o behaviorální popis. Před vytvořením procesu získáme seznamy proměnných Mealyho a Moorových výstupů, které budou později potřeba. Oba abecedně setřídíme.

   (d) Následuje první proces architektury, ve kterém je přiřazen počáteční stav do proměnné současného stavu. Jedná se o tzv. „present state process“.

   (e) Ze vstupních portů lze vytvořit sensitivity list. Ten použijeme do argumentu druhého procesu.

   (f) První řádky procesu patří inicializaci výstupních proměnných a registru aktuálního stavu. V procesu je definována konstrukce switch (case). Setříděný seznam[1] stavů použijeme pro větve této řídící konstrukce.

   (g) Každá větev je podmíněna názvem stavu. Případ, kdy není jméno definováno, je závažnou chybou a uživatel je upozorněn a stav je v editoru označen. V těle je nutné definovat výstupy a přechod do dalšího stavu (často v závislosti na podmínce).

   Přechody, které jsou pro tento stav relevantní, jsou pouze ty, kde aktuální stav figuruje jako zdroj. Pokud se na hraně nalézá podmínka, je vsazena do konstrukce if nebo elseif, dle pravidel z 6.3.6. Do těla podmínky jsou vloženy Mealyho výstupy a přiřazení následujícího stavu. Pokud podmínka přechodu není a je definován výstup, je Mealy přiřazen ve větvi else, pokud již existuje if, jinak je pro něj vytvořen samostatný if s podmínkou true[2]. Je to dáno pořadím vytvoření hran přechodů.

   Pokud existují přechody bez podmínky a výstupu, je přiřazení následujícího stavu provedeno nepodmíněně. Je-li takovýchto hran více, jedná se o sémantickou chybu a v kódu se objeví více přiřazení nového stavu.

   (h) Před koncem procesu jsou uzavřeny všechny větve if. Switch je ukončen větví `when others => null`, poté i samotný proces.

   (i) Po procesu je nutné definovat signály, které jsme použili. Deklarace po použití je vlastnost, kterou má knihovna jVHDL. Během vytváření konečného automatu

[1] Abecedně a na prvním místě vždy počáteční

[2] Tato větev je vytvořena pouze proto, aby byly Mealyho výstupy odděleny od Moorových.

byly také zaznamenány řetězce se vzorem identifikátoru, které nebyly definovány. Tyto proměnné se nadefinují jako signály.

Zde je konec architektury.

4. Ve finální fázi kontrolujeme počet odkazů na blokové prvky, které se ukládají během kroku 1. Pokud je počet pouze jedna, není potřeba definovat top level soubor. Pokud existuje více entit a nejsou spojeny, top level soubor také není potřeba.

5. Nad propojenými entitami bude vytvořen top level soubor. Architektura tohoto souboru bude obsahovat propojení entit. Nejprve jsou nadeklarovány signály všech portů každého bloku, který je součástí spoje. Porty, které nejsou součástí spoje, jsou vloženy do entity, kterou realizuje tato architektura.

   Následně jsou všechny signály namapovány do entit a poté jsou kombinačně propojeny pomocí přiřazení.

7.2.1 Shrnutí

Generování samotného konečného automatu z jednoho bloku je de-facto provedeno na jeden průchod[3]. Je to možné i díky návrhu jVHDL. Entita odpovídá grafické realizaci bloku. Konečný automat ve VHDL má podobu dvouprocesové architektury.

7.3 Graf na XML

Aplikace umožňuje ukládat a znovu načítat data celého projektu do XML. Soubor musí mít validní kořenovou značku, která musí obsahovat výchozího rodiče všech prvků. Chybějící nebo chybné údaje tohoto typu vedou na chybu při načítání souboru. Ostatní chyby způsobí ignorování chybné buňky nebo vlastnosti.

V ukázce v příloze B je vidět část XML formátu. Buňky mají v atributech svojí třídu. Tento identifikátor je poté použit pro konstruktor rekonstruovaného objektu. Dále unikátní identifikátor (`id`) a hodnotu (`value`), která je zobrazena v dané buňce. Element `geo` obsahuje informace o poloze buňky a element `properties` konfiguraci možností, které jsou nastavovány pomocí panelu možností. Případně se může vyskytovat element `points` s údaji o dodatečných kotvících bodech.

Celá struktura se rekurzivně opakuje a kopírují buňky dle grafické reprezentace.

Do XML lze ukládat a načítat z něj celé projekty vytvořené v aplikaci. Lze také načíst existující projekt a přidat jeho obsah do palety s bloky.

[3]Hlavní převod je jedním průchodem, některé jeho dodatečné kontroly obsahují iterace nad seznamy s odkazy na potomky

Kapitola 8

Modul jVHDL - Generování VHDL

Pod názvem jVHDL se skrývá modul ve formě knihovny v jazyce Java SE verze 7, která má kompletní objektově orientovanou stavbu a objektový návrh.

8.1 Návrh

Modul byl tvořen jako nástroj zajišťující výstupní logiku aplikace, kde je potřeba generovat VHDL. Jedná se o základní konstrukce, jako je entita, proces, architektura a základní datové typy. Cílem byla také možnost jednoduše přidávat další vlastní elementy, které modul neobsahuje, rozšířením existujících. Přepsáním funkcí pro převod objektů do řetězců je možné modul změnit z VHDL na Verilog.

Použití knihovny bude zprostředkováno uživateli prostřednictvím rozšíření jedné třídy, která bude umožňovat veškeré operace nad generováním kódu. Do těchto funkcí jsou přidány kontroly chyb, které se mohou vyskytnout.

Návrh je doplněn o UML[1].

Ke zdrojovým kódům jsou přiloženy testy (`JUnit`), takže je po rozšíření možné kontrolovat bezchybnost původní logiky. Zároveň několik z nich slouží jako příklad, jak modul používat.

8.2 Vlastnosti jVHDL

V této kapitole budou zmíněny vlastnosti modulu a jazykové konstrukce, které jVHDL podporuje. Závěrem této kapitoly bude předveden způsob jak s modulem snadno pracovat v sekci 8.5.

Zdrojem informací, jak lze, vytvořit dané syntaktické konstrukce je [6]. Některé z forem, které jsou uvedeny v tomto zdroji, jsou součástí programové dokumentace.

- Automatické odsazování
- Syntaktická kontrola v době vytváření
- Kontrola klíčových slov v názvech
- Deklarace na konci bloku

[1] Diagram tříd se nalézá na přiloženém mediu A.

Možnosti jVHDL pokrývají podmnožinu syntetizovatelného jazyka VHDL, zejména pak konstrukce a prostředky nutné pro vytvoření konečného automatu, ale i další.

Pro přehlednost je výstup automaticky odsazován. Každý prvek, který se nalézá v procesu generování obsahuje svého rodiče. Pomocí rekurzivního volání funkce, která k aktuálnímu odsazení přidává odsazení předka, je možné zjistit, kolik odsazení je nutné provést. V základní implementaci je použito odsazení 4 znaky na jednu úroveň.

Do modulu jsou zabudovány mechanizmy pro eliminaci chyby, které při používání mohou vznikat. Jedná se o kontroly ukončovaní větve `if` nebo `case` a kontext jazyka, do kterého provádíme sazbu. Tyto kontroly jsou na místě hlavně při používání cyklů a větvení při generování kódu. Kontrola ukončování větví řídících struktur i kontext jazyka se provádí pomocí zásobníku, který spravuje rekurzivně zanořovatelné struktury. Tak je možné hlídat uzavírání příslušných řídících konstrukcí a také kontext, do kterého chceme aktuální příkaz přidat. Jedná se hlavně o kontrolu, zda danou sekvenční konstrukci nechce uživatel přidat mimo kontext procesu.

Díky návrhu s pomocným zásobníkem je možné využít i jednu velice zajímavou vlastnost, která se později ukázala jako praktická. Je možné deklarovat datové typy i na konci daného kontextu. Jinými slovy na konci zanoření, ve kterém se nalézáme. Tato vlastnost nám dovoluje provést analýzu, generování a definování typů v rámci jednoho cyklu. Jako příklad uvedu generování konečného automatu, jedním cyklem procházíme všechny stavy a jejich identifikátory si uložíme. Nakonec sebrané údaje použijeme pro deklaraci datového typu enum.

Generátor dále kontroluje na některých místech, jako je třeba název entity, zda se nejedná o název, jež je v klíčových slovech VHDL. Případně za toto klíčové slovo připojí podtržítko a číslici 1. Předpokládá se, že to nemůže způsobit chybu nebo změnu původního návrhu generování, protože změna bude provedena na všech místech.

Posledním mechanizmem pro eliminaci chyb je automatické uzavíraní hodnoty 1 nebo 0 do apostrofů. Například po provedení přiřazení bitové hodnoty do proměnné.

Stále ale existuje možnost vytvořit syntakticky nevalidní kód. V místech, kde jsou údaje přes funkce jVHDL definovány pomocí řetězců, je bezchybnost výstupu ponechána na uživateli a odhalení chyb na kompilátoru VHDL. Je možné také vkládat čistě vlastní kód, u kterého by kontrola představovala úlohu ekvivalentní funkce předkladače VHDL.

8.3 Datové typy jVHLD

Uživatel může pomocí modulu definovat různé datové typy. Jedná se o několik nejběžnějších datových typů.

Základní třídou pro veškeré datové typy je třída `DataTypeVhdl` (viz. UML na mediu A). Z této třídy jsou pak vytvářeny skalární a kompozitní datové typy. Třídu datový typ pak lze přiřazovat do deklarací. Jak již bylo zmíněno, toto lze provést na začátku i na konci daného bloku.

Signály a proměnné mají tu vlastnost, že pokud jsou l-hodnotou přiřazení, modul automaticky zvolí správný přiřazovací operátor.

Podporované datové typy jsou: Enum, Std_logic, Bit, Skaláry (Float, Integer), Kompoziční (Array, Std_logic_vector, a další ...

8.4 Základní stavební jednotky v jVHDL

Základními stavebními jednotkami jsou myšlené tzv. Design Units, jak je popsáno v kap. 3 [5]. Jedná se tedy o *entity*, *architektury*, *knihovny* a *signály* a *proměnné*. Z architektur pak zejména sekvenční popis chování pomocí procesu.

Práce s těmito jednotkami spočívá ve volání daných funkcí, které vytvářejí strukturu podobnou reálnému kódu VHDL. Při používání je dobré volání strukturovat, abychom si byli jisti, že další příkazy přidáváme na správnou úroveň.

- **Kombinační logika**
  Z popisu kombinační logiky jsou možné dvě základní konstrukce. První je klasické přiřazení, druhou pak konstrukce *with/select*.

- **Sekvenční logika**
  Základním prostředkem pro tvorbu sekvenční logiky ve VHDL je proces. Modul jVHDL umožňuje definovat proces pomocí počátečního a ukončujícího volání příslušné funkce. V rámci procesu je možné používat řídící konstrukce *if, elsif, else, switch (case)*. Pokud jsou volány mimo vytvořený proces, je vyvolána výjimka.

8.5 Použití jVHDL

Modul je navržen tak, aby se s ním pracovalo co nejjednodušeji. Proto je veškerá logika kontrol, odsazovaní a další uživateli skryta. Napsání generátoru VHDL kódu v Javě se pak velice podobá svému výstupu, ale je zde možné použít cykly a vše, co programovací jazyk Java nabízí. Principem je rozšíření hlavní třídy modulu `jvhdl.VHDL`. Prostřednictvím této třídy jsou k dispozici veškeré metody pro generování kódu a práci s modulem. Je nutné přistupovat k modulu pouze přes tomu určené metody, protože se v nich nalézá logika kontrol a automatického odsazování. Většinu metod lze využívat různým způsobem, příkladem může být vytvoření výčtového typu, kde lze data předat více způsoby. Metoda pro přiřazení je také ve více variantách, etc. Existuje možnost ponechat kód na uživateli.

Kompletní API pro tuto třídu a ostatní třídy modulu jsou dostupné na přiloženém médiu A v dokumentaci.

Nyní se dostáváme k příkladu kódu 8.1, jak s modulem generovat VHDL. V ukázce je uveden celý funkční kód a jeho výstup do souboru. Všechny složené závorky uvnitř konstruktoru jsou nepovinné.

Na výstupu vidíme automatické odsazování pomocí dvou bílých znaků. Pokud bychom na řádku 35 vynechali volání metody „konec procesu“, byla by vyvolána výjimka, která nás na chybu upozorní (`SyntaxErrorVhdl: Missing end of block`). Detekována by byla díky ukončení architektury, které by předcházelo neukončenému procesu. V případě chybějícího konce větve if (tj. chybějící EndIf()) na řádku 33 by bylo znění výjimky `Bad using of if/endif, when/endWhen or cases/endCases`. Opět by se v tomto případě jednalo o analogii s příkladem předchozí chyby, kdy by přišlo volání konce rodiče na řadu dříve, než jsou uzavřeny všechny synovské struktury.

Sekvenční příkazy lze zapisovat pouze do procesů. V případě pokusu o zapsání sekvence if-else-endif po ukončení procesu obdržíme chybové hlášení `There is not process context (class jvhdl.ArchitectureVhdl)`, které nám říká, že se snažíme používat sekvenční příkazy mimo proces, v závorce je název třídy, do které jsme se pokusili tuto sekvenci vložit.

Pokud si pozorně prohlédneme výstup, zjistíme, že se nám identifikátor výstupu změnil z „out“ na „out_1“. Jedná se o změnu, kterou provedl generátor VHDL. Ten se snaží o eliminaci chybových hlášení, a tím i komfortnější používání uživatelem. Je tedy provedeno nahrazení, které nezmění logiku, protože je substituce aplikována na všech místech výskytu. Může však dojít na chybu v případě deklarace identifikátoru „out“ a „out_1“ v jednom souboru. Proto je nutné na tento fakt brát zřetel.

V příkladu je na řádku 36 vyznačeno místo, kde je možné deklarovat proměnné pro danou úroveň. Tato deklarace by se ve výstupu dostala za deklaraci signálu `pstate`.

Ukázka kódu 8.1: Příklad generování pomocí jVHDL

```
1  package example;

   import jvhdl.*;
   import jvhdl.exception.*;
5  import jvhdl.datatypes.*;
   import jvhdl.datatypes.std.*;
   import static jvhdl.datatypes.DataTypeVhdl.Type.*;

   public class Use extends VHDL { // rozšíření hlavní třídy modulu
10
       Use() throws SyntaxErrorVhdl, InvalidVhdlTypeException {
           addLib(UsesAndLibs.IEEE); // library ieee;
           addLib(UsesAndLibs.IEEE_STD_LOGIC); // ieee.std_logic_1164.all;
           EntityVhdl entity = Entity("gate"); // entita gate
15         // deklarace vstupů a výstupů
           entity.put("in1", new StdLogicVhdl(IN));
           entity.put("in2", new StdLogicVhdl(IN));
           entity.put("out", new StdLogicVhdl(OUT));
           // definice architektury
20         Architecture("arch", entity);
           {   // je možné přidat pomocné bloky
               // deklarace výčtového typu
               EnumVhdl states = Enum(new EnumVhdl("states", "Init, Fin"));
               // použití tohoto typu jako signál
25             SignalVhdl pstate = Signal("pstate", states, "Init");
               // začátek procesu a vytvoření sensitivity listu
               Process("logic", new SignalVhdl[]{pstate});
               {
                   If(new ConditionVhdl(new ExprVhdl("pstate = Init")));// podmínka
30                 {
                       Assignment(pstate, "Fin"); // přiřazení
                   }
                   EndIf(); // konec if
               }
35             EndProcess(); // konec procesu
               // na této úrovni lze vytvářet proměnné architektury
           }
           EndArchitecture(); // konec architektury
       }
40
       public static void main(String[] args) throws Exception {
           new Use().write("use");
       }
   }
```

Ukázka kódu 8.2: Výstup Use.java

```
library ieee;
use ieee.std_logic_1164.all;

entity gate is
    port (
      in1 : IN STD_LOGIC;
      in2 : IN STD_LOGIC;
      out_1 : OUT STD_LOGIC
    );
end gate;

architecture arch of gate is

    type states is (Init, Fin);
    signal pstate : states := Init;

begin
    -- process logic
    logic: process(pstate)
    begin
      if (pstate = Init) then
        pstate <= Fin;
      end if;
    end process;
end arch;
```

Kapitola 9

Závěr

V úvodu dokumentu byly stručně uvedeny informace o jazyku VHDL. Následující kapitoly se věnovaly problematice návrhu konečných automatů. Byly představeny možnosti, jak konečný automat popsat a vybraný způsob práce s nimi. Z oblasti popisu automatů byla práce zaměřena zejména na možnosti popisu prostřednictvím jazyka VHDL a na problémy, které mohou vznikat při takové realizaci.

Aplikace jakožto grafický editor poskytuje náležitý uživatelský komfort zejména díky použité knihovně na vykreslování grafů.

Při správném použití aplikace generuje syntetizovatelný kód. Na více místech je ale možné definovat celý kód nebo část kódu uživatelsky, může se jednat o zdroj chyb. Ověření, zda převod funguje korektně, bylo provedeno na většině příkladů z knih [5, 4]. V této literatuře se nalézá grafický popis automatu a očekávaný výstup. Jeden příklad, který byl obsahem školního projektu, byl nahrán do FPGA, kde byla ověřena jeho funkčnost. Jednalo se o kódový zámek (přístupový terminál).

Podrobně byl popsán postup při generování VHDL. Jedná se o jádro aplikace a jeho pochopení může uživateli objasnit klíčové chování celé aplikace.

Při psaní kódu, který popisuje konečný automat, je často zapotřebí využít předlohu ve formě stavového diagramu. Z tohoto pohledu je tato aplikace velice užitečná, protože vytvoření diagramu nepředstavuje mezi krok, ale koncový bod. Při ručním psaní je samozřejmě možné navrhovat a realizovat specifičtější úlohy a optimalizace, jako například samostatné procesy pro výstup nebo některé akce provádět pomocí kombinační logiky. To generátor neuvažuje a jeho postup převodu je uniformní.

V projektu byla od počátku uvažována znovupoužitelnost. Proto by jeho rozšíření o další funkce a komponenty nemělo představovat problém. Jeho největší slabinou jsou zmíněné možnosti volby při generování kódu apod. Jeho předností ve srovnání s ostatními produkty podobného typu je pak jednoduchost ovládání, možnost definovat propojení na úrovni entit a možnost vkládat do palety již vytvořené bloky v jiných projektech touto aplikací. Další předností je zcela určitě licence, pod kterou bude šířen, a implementace v multiplatformním jazyku Java. Nejedná se o placený software.

Dalšími přínosnými funkcemi aplikace by mohla pokročilejší ochrana před sémantickými chybami, které uživatel může do grafu zanést, a vlastní simulátor s krokováním na kódu s možností volby následujícího stavu.

Zdrojové kódy se po odevzdání této práce budou nalézat na serveru GitHub, kde jsou vedeny repositáře pod mým jménem.

V projektu jsem vytvořil jednu knihovnu pro práci s VHDL a nástroj pro práci s konečnými automaty, který by mohl zaujmout místo mezi ostatními existujícími nástroji díky svému konceptu práce a hlavně dostupnosti.

Literatura

[1] JGraphX (JGraph 6) User Manual. [online]. 2013 [cit. 2013-3-30].
URL http://jgraph.github.io/mxgraph/docs/manual_javavis.html

[2] *Simulink HDL Coder, User's Guide*, kapitola Using Mealy and Moore Machine Types in HDL Code Generation. 2010, [online]. 2010 [cit. 2013-5-5].
URL http:
//www.manualslib.com/manual/392939/Matlab-Simulink-Hdl-Coder-1.html

[3] Chiuchisan, I.; Potorac, A. D.; Graur, A.: Finite State Machine Design and VHDL Coding Techniques. [online]. 2010 [cit. 2013-4-10].
URL http://www.dasconference.ro/cd2010/data/papers/C80.pdf

[4] Chu, Pong P.: *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*, kapitola Finite State Machine: Principle and Practice. 2006, ISBN 0-471-72092-5.

[5] Mealy, B.; Tappero, F.: *Free Range VHDL*. [online]. 2012 [cit. 2013-3-18].
URL http://www.freerangefactory.org/dl/free_range_vhdl.pdf

[6] Mehta, P.: VHDL Syntax Reference. [online]. 2013 [cit. 2013-3-21].
URL http:
//webdocs.cs.ualberta.ca/~amaral/courses/329/labs/VHDL_Reference.html

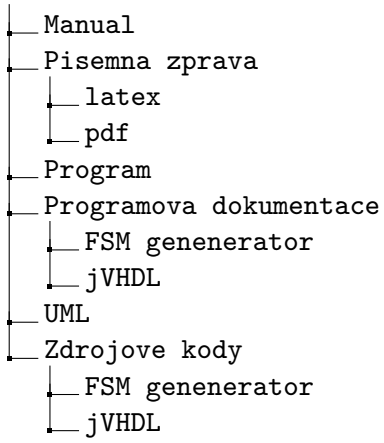
[7] Pinker, Jiří a Martin Poupa: *Číslicové systémy a jazyk VHDL*. BEN, 2006, ISBN 80-7300-198-5.

[8] Zhang, S. Z.: Creating Safe State Machines. [online]. 2002 [cit. 2013-3-25].
URL http://www.fpga.com.cn/application/safe_sm_12157.pdf

Příloha A

Obsah media

Přiložené medium obsahuje následující adresářovou strukturu.



- Manual — Příručka práce s editorem.
- Program — Spustitelná verze programu. Podadresář `examples` obsahuje XML soubory, které jdou aplikací otevřít a obsahují příklady konečných automatů.
- Programova dokumentace — Dokumentace ve formátu javadoc.
- Pisemna zprava — Zdrojový text a soubor PDF
- UML — Vektorové a bitmapové class diagramy.

Příloha B

XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<fsmgenGraph>
  <cell defaultParent="1">
    <cell class="cz.fsmgen.gui.cells.BlockCell" id="8" value="FsmExample">
      <geo height="60.0" width="150.0" x="150.0" y="80.0"/>
      <properties inputsCount="1" outputsCount="2" value="FsmExample"/>
      <cell class="cz.fsmgen.gui.cells.BlocksInputPort" id="9" value="X">
        <geo height="15.0" width="15.0" x="0.0" y="0.5"/>
        <properties name="X" type="STD_LOGIC"/>
      </cell>
      ...
      <cell class="cz.fsmgen.gui.cells.FsmEdgeCell" edge="1" id="22"
          value="1&lt;hr&gt;1">
        <edge source="15" style="..." target="15"/>
        <geo height="120.0" width="120.0" x="0.0" y="10.0"/>
        <properties inputs="X=1;" outputs="Z1=1;Z2=;"/>
        <points>
          <point x="220.0" y="200.0"/>
        </points>
      </cell>
      <cell class="cz.fsmgen.gui.cells.FsmEdgeCell" edge="1" id="23"
          value="1&lt;hr&gt;1">
        <edge source="17" style="..." target="17"/>
        <geo height="120.0" width="120.0" x="0.0" y="10.0"/>
        <properties inputs="X=1;" outputs="Z1=1;Z2=;"/>
        <points>
          <point x="270.0" y="320.0"/>
        </points>
      </cell>
      ...
    </cell>
  </cell>
</fsmgenGraph>
```